**Софийски университет „Св. Кл. Охридски”**

Факултет по математика и информатика

*Магистърска програма*

*„Софтуерни технологии”*

**Предмет: Обектно-ориентиран анализ и проектиране на софтуерни системи**

***Зимен семестър, 2020/2021 год.***

**Тема 44: Недостатъци на ООП парадигмата в ерата на гъвкавото проектиране. Алтернативи.**

**Есе**

*Автор:*

*Тодор Андонов, фак. номер 26490*

ноември, 2020

София

# 

[**1. Въведение**](#_5qpgeev7dhd6) **3**

[**2. История на Обектно-Ориентираното програмиране**](#_l05c8p6yv1ub) **3**

[2.1. Началото](#_cmy22awhn2tp) 3

[2.2. Object-Orientation in a nutshell](#_i4r9rc4zp81c) 3

[2.3. Принципи на Обектно-Ориентираното Програмиране](#_ridfflm93p15) 4

[2.3.1. Унаследяване](#_w3lggwef96pv) 4

[2.3.2. Енкапсулация](#_hdcjrhdsfh31) 4

[2.3.3. Полиморфизъм](#_jlnh5imbad9y) 4

[2.4. Заключение](#_pzknmlz90zj5) 5

[**3. Гъвкави методологии за разработване на софтуер**](#_jwecp03t1nkx) **5**

[3.1. История](#_7uk89c33qqzm) 5

[3.2. Манифест на гъвкавата методология](#_y9q39enwfdgt) 6

[3.3. Видове гъвкави методологии](#_wygvjf9gjih5) 7

[3.3.1. Scrum](#_nrhpsl7q9mn5) 7

[3.3.2. Kanban](#_l72oxwl1hy14) 7

[3.3.3. Екстремно програмиране](#_6esst2ibvq1a) 7

[**4. Обещанията на ООП във времето на гъвкавите методологии**](#_xij6d48ilwzi) **8**

[4.1 Преизползване на код (Reuse)](#_hw32274gwmqt) 8

[4.1.1. Проблемът](#_hpji47b1yz5d) 8

[4.1.2. Решението](#_byigyj3a5dq) 10

[4.2. Моделиране](#_46wahgfvltev) 11

[4.2.1. Екстремно Програмиране](#_et8awr58buh) 11

[4.2.2. Екстремно Програмиране и UML](#_905mzk91l6bs) 12

[**5. Заключение**](#_vlw0stuqbmyy) **13**

# 1. Въведение

Настоящата работа разглежда какви са недостатъците при Обектно-Ориентирания подход за разработка на софтуерни решения. Също така, какви са проблемите, които възникват, когато се използва гъвкав подход за управление на проекти, които използват Обектно-Ориентираната парадигма за програмиране. Накрая са представени възможните решения и алтернативи.

# 2. История на Обектно-Ориентираното програмиране

## 2.1. Началото

Първите програмни езици са били поредица от директни инструкции към компютъра. Една от най-честите операции е присвояването на стойност към променлива, като това се повтаря докато програмата не завърши изпълнението си. “Дистанцията” между разработчика и компютъра е малка, следователно възможността за грешка нараства. С появата на Simula, считан за първият обектно-ориентиран език за програмиране, в средите на програмистите се появява нещо ново, което привидно скъсява тази дистанция и прави разработката по-лесна.

## 2.2. Object-Orientation in a nutshell

Вместо да разделят действията и данните на програмата, езиците, които следват обектно-ориентираната парадигма за програмиране правят точно обратното. Те комбинират смислово свързани данни и поведение в една единица, наречена клас. Целта на класовете е да опишат предметите, които ни заобикалят и които трябва да се автоматизират чрез софтуера, който създаваме. Докато класовете служат да моделират, обектите са тези, които представят действителни обекти от заобикалящият ни свят. От един клас може да създадем много обекти. Например ако имаме клас, които моделира къща, то той е в ролята на чертежа на къщата. С един чертеж може да построим много къщи. Други отличителни черти на ООП са Унаследяването (Inheritance), Полиморфизъм и Енкапсулация.

## 2.3. Принципи на Обектно-Ориентираното Програмиране

### 2.3.1. Унаследяване

Най-голямата отличителна черта и това, което често се изтъква като основната причина при изграждането на едно софтуерно решение да се използва ОО парадигмата, е Унаследяването. Това е похват при които разработчиците могат да споделят и преизползват общи части на един клас. Парадоксално, докато преди на Унаследяването се е гледало като способ чрез които може да се улесни разработката на софтуер, в наши дни то не се препоръчва и предизвиква редица полемики[[1]](#rscjf1q6lsoa) в средите на разработчиците.

### 2.3.2. Енкапсулация

Енкапсулацията е метод за групиране на свързани по смисъл части от програмата и контрол над данните, които влизат и излизат от един клас.

### 2.3.3. Полиморфизъм

Свойството на един език да третира обектите различно на база техният тип се наричи Полиморфизъм. Това позволява много различни обекти да бъдат взаимозаменяеми. Например, ако една кола получи инструкция “движи се” то тя ще включи двигателя и ще тръгне. Ако един самолет получи същата инструкция, той ще включи двигателя и ще полети. И двата обекта са превозни средства, но в различни случаи се третират по различен начин.

## 2.4. Заключение

С появата на езика Smalltalk, обектно-ориентираният подход добива още повече популярност. След като разработчиците се убеждават във всички негови предимства се появяват още стотици ОО езици. Всички те са приветствани със всички техни “обещания” за улеснение на разработването, но само някои успяват да ги изпълнят. Сред езиците от тази категория, които все още се използват днес са Java, C# и C++.

# 3. Гъвкави методологии за разработване на софтуер

## 3.1. История

Така наречените леки или гъвкави методи се развиват в средата на 90-те години в отговор на смятаните за “тежки” традиционни методи, един от които е “методът на водопада” (waterfall). Като главен недостатък при тях се изтъква употребата на силен контрол, правила и микроуправление. Също така за неефикасен се счита и прекалено инкременталният подход за разработване.

Привържениците на гъвкавите методологии твърдят, че те връщат разработката на софтуер към практики за управлението на проекти от началото на историята на разработване на софтуер. Някои от ранните реализации на гъвкавите методологии включват Rational Unified Process (1994), Scrum (1995), Екстремно програмиране (1996) и други. Всички те се класифицират като “гъвкави”, след публикуването на Манифеста на гъвкавите методологии (Agile Manifesto) през 2001г.

## 3.2. Манифест на гъвкавата методология

През февруари 2001г. е публикуван Манифестът на гъвкавите методологии[[2]](#e9yij68blcva). В него се дефинират подходите, сега известни като гъвкави методи за разработка на софтуер. В него се казва:

“*Откриваме по-добри методи за разработване на софтуер, като ги практикуване и помагаме на другите да го правят. В процеса на работа стигнахме до следните важни изводи:*

* *Хората и комуникацията стоят над процесите и инструментите*
* *Работещият софтуер е над подробната документация*
* *Сътрудничеството с клиента е над преговорите по време на сключване на договора*
* *Адресирането на промените стои над следването на плана*

*Това означава, че въпреки че десните елементи имат своята стойност, ние ценим точките отляво повече.*”

Манифестът очертава рамката за успешното и ефективно разработване на софтуер и всички процеси около него. Той измества фокуса върху хората и екипите, като се отбелязва, че мотивацията и самоорганизацията са важни качества за един успешен разработчик. Насърчава се работата в съвместно помещение, по екипи и програмирането по двойки (pair programming).

Сътрудничеството с клиента също има важна роля в процеса. Клиентът взима активно участие в разясняването на изискванията, тестване и даване на обратна връзка за продукта. Тази черта представя своите препятствия, като например времето, което заинтересованото лице трябва да посвети на проекта, но несъмнено е от голямо значение за успешното имплементиране на изискванията.

Гъвкавите методологии целят да насърчат работата в екип, сътрудничеството, както и адаптивността през целият жизнен цикъл на проекта. Това се постига чрез разбиване на задачите на малки стъпки с минимално планиране, без да се засяга дългосрочното планиране на проекта.

## 3.3. Видове гъвкави методологии

### 3.3.1. Scrum

Скръм е процес, използван при изготвяне и управление на големи проекти. Той е разработен с цел дългосрочното планиране на разработването на даден продукт да бъде улеснено значимо. За разлика от типичния мениджмънт чрез контрол и командване, при Скръм процесите се набляга на обратна връзка и се дава повече власт на хората, вършещи “черната работа”.

### 3.3.2. Kanban

Канбан е метод за управление на интелектуални дейности с наблягане на доставката точно на време, без същевременно членовете на екипа да са претоварени. Името произлиза от японски и буквално преведено означава „сигнална карта“. Методът е подход към постепенен, еволюционен процес. Използва система за ограничаване на текущата работа като основен механизъм, за да разкрива системно операционни проблеми и да стимулира взаимопомощта, с цел продължително подобряване на системата.

### 3.3.3. Екстремно програмиране

Екстремно програмиране (Extreme Programming – XP) e друг вид методология за създаване на софтуер. Основната цел на XP е да редуцира цената на проект, ако се наложи дадена промяна. Оттук се вади извода, че XP е методология, подходяща да се използва при проекти, които имат често променящи се изисквания и при които по-стандартни методологии (като Waterfall модела например) не са оптимални за постигане на голяма продуктивност; подходяща е при проекти, които включват нови технологии или непредвидими проблеми, свързани с имплементацията; използва се също така при по-малки и по-лесни за реализация проекти с неофициални методи; добра технология за проекти, изискващи изследване.

# 4. Обещанията на ООП във времето на гъвкавите методологии

През последното десетилетие, обектно-ориентираните езици са приемани като решението на сложни софтуерни проблеми. Способността за лесно разширяване и надграждане на съществуващия код е ключово в израстването на ОО парадигма. Това е една от причините толкова много организации да използват обектно-ориентиран език за своите системи. Това за което преди отговаряха процедурните езици, сега е задача на обектно-ориентираните. Въпреки това, ОО езиците не успяват да усвоят и приложат ефикасно всички аспекти, които са успешни в процедурните езици.

## 4.1 Преизползване на код (Reuse)

### 4.1.1. Проблемът

Николо Пигнатели кръщава една своя статия с гръмкото “Inheritance Is Evil. Stop Using It.” (Унаследяването е зло. Спрете да го използвате.)[[3]](#urfoy23cz52z). В нея той разглежда до колко е вярно твърдението, че унаследяването ни позволява да разширяваме и надграждаме класовете на едно приложение. По-конкретно, при използването на унаследяване, ние обвързваме (couple) наследения и наследяващия клас. На пръв поглед това не е толкова голям проблем, но всеки софтуер се развива и променя. Ако имаме един клас с няколко метода и полета и друг клас, които го наследява и надгражда поведението му, какво ще стане, ако се наложи да променим наследявания клас? Как тази промяна ще се отрази на наследяващия клас? Още повече, какво би станало ако главният клас се наследи от друг клас, който на свой ред се наследява от трети клас? Една промяна в тази верига може да е пагубна. Това, което сме направили е, че сме създали четири пряко обвързани класа и сме поставени в безизходица, ако трябва да променим или поправим нещо.

В контекста на гъвкавите методологии, нуждата да се направи поправка в тази йерархия може да е катастрофална. Тъй като методологиите се осланят на малки и поетапни промени, в кратък период (sprint), обикновено 1 или 2 седмици, проблеми от този тип могат да костват много ресурс. Обикновено поправянето става чрез пренаписване и в зависимост от това от колко време се разработва проектът (колкото повече време, толкова повече свързаности между класовете може да има), това пренаписване може да отнеме много време. Ръка за ръка с пренаписването на кода върви и пренаписването на тестовете към него, също нещо важно за качеството на един софтуерен продукт.

За да бъде използвано по най-добрият начин, преизползването на код трябва да се наложи от самото начало на създаването на продукта и да бъде наложено неговото правилно прилагане по време на живота му. Изработването на компоненти, които могат да се преизползват изисква сериозна координация и мениджърски ресурс. Съпътствано е с много срещи и срокове, с цел на заинтересованите лица да се предоставят решения, които правят каквото е нужно, но и в същото време, решения, които са готови да устоят на изпитанието на времето. Ако преизползването се наложи след известно време от съществуването на проекта, резултатите биха били непредвидими, заради съпътстващата ги сложност на управлението на разработчиците и техните навици на програмиране.

### 4.1.2. Решението

Има няколко техники за да се избегне наследяване. Някои от основните са:

* Използване на интерфейси - използването на интерфейси ще дефинира строго и ясно “договора”, който класовете, които имплементират този интерфейс ще трябва да спазват. Чрез интерфейсите може да се фокусираме върху това какви съобщения ще си разменят класовете;
* Инжектирането на тези интерфейси в конструктора на класа, който зависи на него - инжектирането на интерфейси като зависимости е изключително полезно в случаите в които се налага да променим нещо по начина на изпълнение на някои от методите. Ще направим промяната в класа, които имплементира интерфейса и няма да предизвикаме проблеми във всички класове, които зависят на този интерфейс;
* Използването на Контейнер за инжектиране на зависимостите (Dependency Injection Container);

Докато наследяването е характерно за ООП, някои смятат, че ползите му могат да са по-видими, ако се използва композиция. При нея един клас не наследява свойствата на друг, а по-скоро един клас съдържа в себе си свойствата на друг. Например класът Кола не наследява класът Двигател, а има в себе си поле от тип Двигател, което дава достъп до свойствата и способности на двигателя.

## 4.2. Моделиране

### 4.2.1. Екстремно Програмиране

За моделирането на софртуерни системи, изградени чрез, обектно-ориентиран език за програмиране се използват Обектно-Ориентиран Анализ и Обектно-Ориентиран Дизайн. Това са похвати за анализиране и проектиране на приложение или цяла система, чрез прилагането на принципите на обектно-ориентираното програмиране и визуално моделиране. Езиците за моделиране, какъвто е UML (Unified Modeling Language), служат да направляват и улеснят комуникацията между разработчиците и заинтересованите лица по време на разработване на продукта.

В своята статия “Is Design Dead?”[[4]](#ubw54u7xcvuo), Мартин Фаулър, автор на книга за UML и явен проповедник на ХР (eXtreme Programming), прави много интересна аналогия - Ако искаме да построим къща за кучета можем да купим няколко дъски и да ги заковен заедно, но ако искаме да построим небостъргач, този подход няма да доведе далеч. Преди да започне строежът, архитектът прави проект посредством изпитани и утвърдени практики в строителството. Проектира се и се избира всяка една част, която ще се използва, до последният винт и едва след това готовият дизайн се предава към строителите, които започват изпълнението му.

Една от най-спорните характеристики на Екстремното Програмиране е еволюционен вместо планов дизайн. С други думи - дизайна на системата се определя по време на създаването ѝ, а не преди него. Ако продължим аналогията със строителството, това би било да започнем строежа на небостъргача, но да не сме сигурно колко точно етажа ще има. Причината еволюционният дизайн да е смятан за неефективен от много е постоянната промяна в софтуера. Цената на нова функционалност или поправяне на стара нараства с нарастването на възрастта на софтуерният продукт. Тази цена може да се намали значително чрез плановия дизайн.

В Екстремното Програмиране, обаче, има и практики, които биха позволили практикуването на еволюционен дизайн. Такива са Refactoring, Continuous Integration и тестването. Това “безразборно” програмиране е съпътствано с постоянно рефакториране и без сигурността, която тестването ни предоставя ХР би било невъзможно. Continuous Integration, от своя страна, позволява целият екип от разработчици да работи по една и съща версия на сорс кода.

### 4.2.2. Екстремно Програмиране и UML

Връщайки се към статията на Мартин Фоулър, изглежда странно, че човек, който проповядва използването на Екстремно Програмиране и е написал книга за UML. Как двете се съвместяват, ако изобщо се?

В идеологията на ХР е своеобразното отричане на визуалните модели. Това, най-вероятно, произлиза от факта, че причината да се практикува Екстремно Програмиране е скоростта с която се разработва продукта. Хората, които го практикуват по-скоро чувстват, че отделянето на време за първоначалното проектиране, макар и полезно, ги забавя значително. Друг фактор е, че доста хора асоциират с тежки и времеемки процеси изготвянето на тези диаграми.

Мартин Фоулър има съвет за това как да намерим пресечната точка между UML и XP. На първо място той препоръчва да се замислим за какво служат тези диаграми. Главната им цел е ефективна комуникация на изискванията и това как да бъдат имплементирани те по най-оптималният начин. Това означава да се отсеят важните и съществени части от второстепенните. Нещо друго, което той препоръчва е да гледаме на диаграмите като скица, а не финаният чертеж. Колкото и време да се отдели на планирането, при имплантирането неизбежно се появяват пропуски. С оглед на това е напълно валидно да се направят диаграми, които да “очертаят” структурата и след това да спрат да се поддържат. Евентуалната им поддръжка би означавала още повече тежест в работата на разработчиците. Именно тук е пресечната точка между двата привидно взаимноотричащи се подхода. Баланс между най-добрите черти на UML и ХР са напълно способни да работят заедно и да си помагат.

# 5. Заключение

Има ли място Обектно-Ориентираният Дизайн и Анализ в ерата на гъвкавите методологии? Определено. Дизайнът е далеч от мъртъв, по-скоро се е променил. В себе си той включва постоянното желание да поддържаме кода колкото се може по-чист. За чистотата помага рефакторирането, а за спокойствие, че всичко ще е наред след рефакториране ни дават тестовете. Също така при дизайна трябва да се проектира с поглед към бъдещето. И най-важното е, че умението да се комуникира дизайна към хората, които трябва да го разберат е най-същественото умение. Това става чрез код, диаграми и най-вече - вербална комуникация.

1. [Inheritance, Issues and alternatives](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Issues_and_alternatives)

1. [Манифест за Agile разработка на софтуер](https://agilemanifesto.org/iso/bg/manifesto.html)

1. [Inheritance Is Evil. Stop Using It. | by Nicolò Pignatelli](https://codeburst.io/inheritance-is-evil-stop-using-it-6c4f1caf5117)
2. [MartinFowler - Is Design Dead?](https://martinfowler.com/articles/designDead.html)
3. [Agile Modeling with the UML](https://arxiv.org/ftp/arxiv/papers/1409/1409.6767.pdf)
4. [Has the Object-Oriented Paradigm Kept Its promise?](http://ddi.cs.uni-potsdam.de/HyFISCH/Informieren/Programmiersprachen/OOPromisesAndReality.pdf)