



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ




Душан Рађеновић

Примена одабраних дизајн шаблона у реалистичним софтверским апликацијама

ДИПЛОМСКИ РАД
- Основне академске студије -

Нови Сад, 2020.

	Универзитет у Новом Саду, Факултет техничких наука 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Датум:
	ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА	Лист/Листова:

(Податке уноси предметни наставник - ментор)

Врста студија:	<input type="checkbox"/> Основне академске студије
Студијски програм:	Рачунарство и аутоматика
Руководилац студијског програма:	проф. др Милан Видаковић

Студент:	Душан Рађеновић	Број индекса:	RA 33/2015
Област:	Пословна информатика		
Ментор:	проф. Гордана Милосављевић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ (Bachelor) РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА: <ul style="list-style-type: none"> - проблем – тема рада; - начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; - литература 			

НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:

Примена одабраних дизајн шаблона у реалистичним софтверским апликацијама

ТЕКСТ ЗАДАТКА:

Задатак овог рада је спецификација, имплементација и верификација софтверског решења које испољава структуру и својства погодна за примену софтверских дизајн шаблона. Над датим апликацијама је потребно применити одабране дизајн шаблоне и истаћи предности њихове употребе.

Руководилац студијског програма:	Ментор рада:

Примерак за: <input type="checkbox"/> - Студента; <input type="checkbox"/> - Ментора
--

САДРЖАЈ

1.	УВОД	9
2.	Одабрани шаблони	11
2.1	<i>Adapter</i>	11
2.1.1	Општи опис проблема	11
2.1.2	Решење	11
2.1.3	Структура.....	12
2.1.4	Када га користити	12
2.2	<i>Template method</i>	12
2.2.1	Општи опис проблема	13
2.2.2	Решење	13
2.2.3	Структура.....	13
2.2.4	Када га користити	14
2.3	<i>Strategy</i>	14
2.3.1	Општи опис проблема	14
2.3.2	Решење	15
2.3.3	Структура.....	16
2.3.4	Када га користити	16
2.4	<i>State</i>	16
2.4.1	Општи опис проблема	17
2.4.2	Решење	17
2.4.3	Структура.....	18
2.4.4	Када га користити	18
2.5	<i>Singleton</i>	18
2.5.1	Општи опис проблема	19
2.5.2	Решење	19
2.5.3	Структура.....	19
2.5.4	Када га користити	20
2.6	<i>Chain of responsibility</i>	20
2.6.1	Општи опис проблема	20
2.6.2	Решење	20
2.6.3	Структура.....	21
2.6.4	Када га користити	22
2.7	<i>Bridge</i>	22
2.7.1	Општи опис проблема	22
2.7.2	Решење	23
2.7.3	Структура.....	23
2.7.4	Када га користити	24
2.8	<i>Observer</i>	24

2.8.1	Општи опис проблема.....	24
2.8.2	Решење	25
2.8.3	Структура	25
2.8.4	Када га користити.....	25
2.9	<i>Decorator</i>	26
2.9.1	Општи опис проблема.....	26
2.9.2	Решење	26
2.9.3	Структура	27
2.9.4	Када га користити.....	27
2.10	<i>Factory method</i>	28
2.10.1	Општи опис проблема.....	28
2.10.2	Решење	28
2.10.3	Структура	29
2.10.4	Када га користити.....	29
3.	Реална примена / пројекат	31
3.1	Опште напомене о пројекту	31
3.1.1	<i>Singleton</i>	31
3.1.1.1	Опис проблема.....	32
3.1.1.2	Решење	33
3.2	Музички плејер.....	34
3.2.1	<i>State</i>	35
3.2.1.1	Опис проблема.....	35
3.2.1.2	Решење	36
3.2.2	<i>Adapter</i>	39
3.2.2.1	Опис проблема.....	39
3.2.2.2	Решење	40
3.3	<i>YouTube</i> налог	42
3.3.1	<i>Strategy</i>	42
3.3.1.1	Опис проблема.....	42
3.3.1.2	Решење	44
3.3.2	<i>Observer</i>	47
3.3.2.1	Опис проблема.....	47
3.3.2.2	Решење	48
3.4	<i>Google</i> налог	51
3.4.1	<i>Observer</i>	51
3.4.1.1	Опис проблема.....	51
3.4.1.2	Решење	53
3.5	Пицерија.....	54
3.5.1	<i>Template method</i>	54
3.5.1.1	Опис проблема.....	54

3.5.1.2	Решење.....	56
3.5.2	<i>Chain of responsibility</i>	59
3.5.2.1	Опис проблема	59
3.5.2.2	Решење.....	62
3.5.3	<i>Decorator</i>	64
3.5.3.1	Опис проблема	65
3.5.3.2	Решење.....	67
3.5.4	<i>Bridge</i>	68
3.5.4.1	Опис проблема	68
3.5.4.2	Решење.....	71
3.5.5	<i>Factory method</i>	74
3.5.5.1	Опис проблема	74
3.5.5.2	Решење.....	76
4.	ЗАКЉУЧАК	79
	ЛИТЕРАТУРА	81
	БИОГРАФИЈА	83
	КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА	85
	KEYWORDS DOCUMENTATION.....	87

1. УВОД

Кроз историју развоја софтвера, инжењери су током формирања кода наилазили на сличне проблеме за које су производили разна решења. Нека решења су била више квалитетна од других и временом се искристалисао скуп шаблонских решења за наведене честе проблеме. Посматрајући корисна искуства научена кроз праксу, као и анализирајући доказано валидна решења, дошло је до појављивања и изучавања дизајн шаблона.

Први рад на ову тему објављен је 1995. године и потписан је од стране четири аутора, а то су: Ерих Гама, Ричард Хелм, Ралф Џонсон и Џон Вилсидис. Назив рада је „Дизајн шаблони: Елементи поновно искористивог објектно-оријентисаног софтвера“[1]. У овом раду је први пут уведено 23 дизајн шаблона. Временом, рад је добио популаран назив „*The gang of four book (The GoF book)*“.

Дизајн шаблони представљају типична решења за понављајуће проблеме у развоју софтвера[2]. Они не представљају конкретну имплементацију решења за конкретан проблем, него генерализују проблем и његов узрок, тиме дајући исто тако генерализовано решење. Дакле, дизајн шаблони представљају решење проблема на вишем нивоу апстракције од конкретног решења, тако да представљају упутство како неки проблем решити. Сваки шаблон, иако неки структурално међусобно слични или исти, је својствен по својој намени, тј. проблему који решава, као и начину на који га решава.

Дизајн шаблони се могу поделити у три групе: креациони, структурални и шаблони понашања. Креациони шаблони дају решења у виду различитих начина креирања објеката. Структурални шаблони се баве различитим начинима на који се класе и објекти састављају у сложеније структуре. Шаблони понашања се баве алгоритмима и доделом одговорности објектима у зависности од њихове улоге.

У овом раду ће се кроз реалистичан софтверски пројекат, приказати типични проблеми који се ефикасно решавају применом одговарајућих дизајн шаблона. Сваки проблем ће поседовати два решења, где ће прво избегавати употребу одговарајућег дизајн шаблона, док ће га друго применити. Кроз анализу ових решења, представиће се предности употребе шаблона и њихов утицај на даљи развој решења.

Софтверски пројекат који је имплементиран се састоји од четири подсистема. То су подсистеми за пицерију, *YouTube* налог, *Google* налог, као и музички плејер. Подсистем за пицерију је дизајниран тако да омогући увид у примену шаблона *Decorator*, *Chain of responsibility*, *Bridge*, *Template method* и *Factory method*. Подсистем за *YouTube* налог је дизајниран да омогући увид у примену шаблона *Observer* и *Strategy*. Подсистем за *Google* налог додатно пружа увид у предности примене *Observer* шаблона. Подсистем за музички плејер омогућава увид у примену *State* и *Adapter* шаблона. Повезани подсистеми, чинећи заједно једну апликацију, додатно пружају увид у имплементацију *Singleton* шаблона.

У поглављу 2 ће бити речи о одабраним шаблонима који ће касније бити коришћени у софтверском пројекту. За сваки одабрани шаблон биће пружен увид у његову дефиницију и намену, општи опис проблема, решење истог, структуру шаблона као и када га је потребно користити. Поглавље 3 ће пружити увид у имплементацију софтверског пројекта, тако што ће пружити увид у имплементацију његових подсистема. За сваки подсистем, биће објашњени проблеми који настају у потреби да се реализује одређена функционалност. Након тога, биће дато решење проблема не користећи одговарајући дизајн шаблон и проблем таквог приступа. На крају биће пружен увид у имплементацију решења користећи одговарајући дизајн шаблон и предности његове примене за дати сценарио.

2. Одабрани шаблони

У овом поглављу биће приказани одабрани дизајн шаблони. За сваки шаблон, биће дат општи опис проблема када се шаблон не користи, решење које он уводи, његова структура дата дијаграмом класа користећи *UML (Unified Modeling Language)*[3], као и када га је потребно користити.

2.1 Adapter

Adapter је структурални дизајн шаблон који интерфејс једне класе претвара у интерфејс какав клијент очекује. *Adapter* омогућава сарадњу класа која иначе не би била могућа због некомпатибилних интерфејса[1].

2.1.1 Општи опис проблема

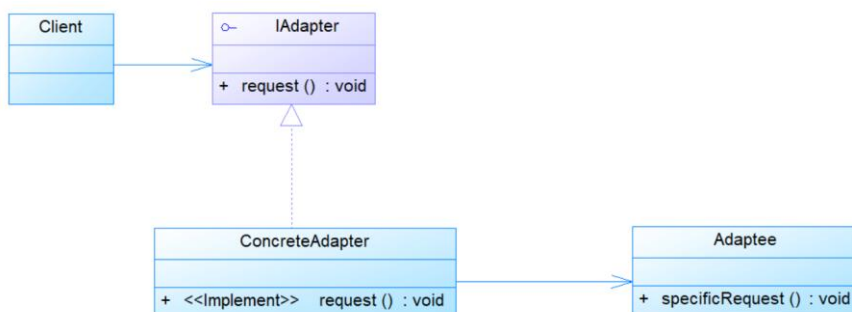
Замислити сценарио где је потребно искористити функционалност неке постојеће класе (или *3rd party* библиотеке) чији програмски код није могуће мењати. Потенцијални проблем би могао бити да жељену функционалност те класе није могуће искористити јер се она позива на начин који не одговара клијенту. Конкретно, интерфејси тих класа су некомпатибилни са потребама клијента. У другачијем сценарију, где се функционалност позива на очекиван начин, временом можда буде потребе да се класа или библиотека која се користи за реализацију те функционалности промени (јер постојећа библиотека престане да испуњава потребе клијента). Таква промена би проузроковала потребу мењања програмског кода, због прилагођавања новој класи или библиотеци. Разлог ових проблема је што оба сценарија представљају дизајн који крши *dependency inversion* принцип[4] тиме што се клијент везује за конкретну имплементацију класе или библиотеке.

2.1.2 Решење

Горе поменуте проблеме је могуће решити применом *Adapter* шаблона. *Adapter* шаблон омогућава да класе које претходно нису могле да сарађују због некомпатибилних интерфејса, посредством *Adapter*-а, сада могу сарађивати. Клијент не мора да зна тачан начин позивања конкретне функционалности класе или библиотеке коју жели да искористи, већ је довољно да само упути захтев *Adapter*-у за ту функционалност, а он ће позив проследити конкретној класи, онако како она очекује. Тиме се у фокус ставља оно што је потребно постићи. Уколико дође до потребе за променом конкретне класе или библиотеке, као и потребом за коришћењем неке друге, само је потребно заменити *Adapter* који се користи, док програмски код који га користи остаје исти.

2.1.3 Структура

На слици 2.1 је приказан *UML* дијаграм *Adapter* шаблона, док су у табели 2.1 објашњени чиниоци овог дијаграма.



Слика 2.1 *UML* дијаграм *Adapter* шаблона

Назив	Опис
Client	Класа или део програмског кода из ког се позива потребна функционалност.
Adapter	Интерфејс који апстрахује функционалност. Дефинише начин позивања функционалности који клијенту одговара. Позивањем функционалности преко овог интерфејса, клијент се не везује за конкретан <i>Adapter</i> .
ConcreteAdapter	Конкретна <i>Adapter</i> класа која прима захтев од клијента и прослеђује га онако како класа чију функционалност је потребно искористити очекује.
Adaptee	Класа чију функционалност је потребно искористити.

Табела 2.2 Опис чиниоца *Adapter* шаблона

2.1.4 Када га користити

Adapter шаблон је погодно користити када је потребно искористити функционалност постојеће класе, али њен интерфејс није компатабилан интерфејсу клијента. Такође, погодно га је користити када је потребно модификовати такву функционалност без мењања програмског кода класе чију функционалност је потребно модификовати. Када је неопходно не везивати се за конкретну класу или библиотеку, због могуће замене у будућности, *Adapter* шаблон такође може послужити као одговарајуће решење таквог проблема. Типична примена *Adapter* шаблона је интеграција са 3rd party програмским кодом – библиотеком или спољашњом апликацијом.

2.2 Template method

Template method је шаблон понашања који дефинише костур алгоритма у родитељској класи, али препушта класама наследницима да

дефинишу специфичне кораке алгоритма, без да мењају његову структуру[1].

2.2.1 Општи опис проблема

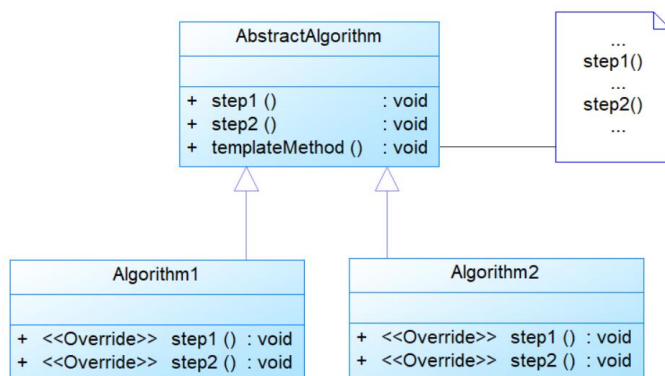
Замислити сценарио где је потребно имплементирати варијације неког алгоритма, где се алгоритам састоји из више корака, који се извршавају у тачно одређеном редоследу. У свакој варијацији алгоритма, постоје кораци који су увек међусобно исти, док се по неким корацима међусобно разликују. Због постојања корака који су увек исти, долази до појављивања истог програмског кода у свакој имплементацији варијације алгоритма, што нарушава један од основних дизајн принципа, а то је *DRY*[5] (енгл. *Don't repeat yourself*). Самим тим, са повећањем броја варијација алгоритма, долази до повећања броја понављања програмског кода који је увек исти.

2.2.2 Решење

Решење поменутог проблема је дато шаблоном под називом *Template method*. Овај шаблон говори да алгоритам треба раздвојити у кораке, тако да се за сваки корак зна да ли је исти за све варијације алгоритма, или је ипак специфичан за сваку варијацију. Тако одвојени кораци се декларишу у родитељској класи. Затим, у родитељској класи се такође додаје још једна метода специфична за овај шаблон, а то је *templateMethod()* метода. У телу ове методе је потребно позвати специфициране кораке у редоследу тако да дају коначан алгоритам. Корацима који су исти за сваку варијацију алгоритма, потребно је дати имплементацију у родитељској класи и уз то забранити да се они мењају у класама наследницама. Кораке који су специфични за сваку варијацију, треба прогласити апстрактним у родитељској класи, и самим тим препустити њихову имплементацију класама наследницама, које представљају варијације алгоритма. Дефинисањем и имплементацијом *templateMethod()* методе, у којој се позивају појединачни кораци алгоритма, добија се позивање заједничког програмског кода који је сада у родитељској класи на једном месту, као и позивање програмског кода чија је имплементација дата у класи наследници, и тиме нема дуплирања кода.

2.2.3 Структура

На слици 2.2 је приказан *UML* дијаграм *Template method* шаблона, док су у табели 2.2 објашњени чиниоци овог дијаграма.



Слика 2.2 UML дијаграм Template method шаблона

Назив	Опис
AbstractAlgorithm	Дефинише кораке алгорита као методе и садржи <i>template</i> методу која саставља кораке алгорита у смислену целину, позивајући их једног по једног. <i>Template</i> метода се не сме прегазити.
Algorithm	Класа која представља конкретну варијацију алгорита. Наслеђује класу <i>AbstractAlgorithm</i> и самим тим њене кораке за које треба да пружи имплементацију. Нема могућност да мења <i>template</i> методу. Наслеђивањем добија и кораке који имају заједничку имплементацију дату у родитељској класи.

Табела 2.2 Опис чиниоца Template method шаблона

2.2.4 Када га користити

Template method шаблон је погодно користити када је потребно имплементирати варијације неког алгорита, где долази до понављања истих делова кода. Погодно га је користити и када је потребно оставити слободу у даљој имплементацији неких делова алгорита, док је неке делове потребно забранити за сваку измену. Типичне примене овог шаблона су код различитих радних оквира и генератора кода.

2.3 Strategy

Strategy је шаблон понашања који дефинише фамилију алгоритама, енкапсулира сваки од њих у засебну класу, и омогућује њихову замену у току извршавања програма. *Strategy* шаблон дозвољава да алгоритам варира независно од клијента који га користи[1].

2.3.1 Општи опис проблема

Наслеђивање је један од основних принципа објектно оријентисаног програмирања, али његова прекомерна употреба може произвести нефлексибилан програмски код. Замислити сценарио где је за дату класу и њена понашања, потребно имплементирати њене

варијације, самим тим и пружити имплементацију њених понашања. Уколико се наслеђивањем покуша решити овај проблем, потребно би било направити родитељску класу. У њу би затим било потребно сместити сва понашања која су заједничка за све класе наследнице, односно варијације. Након тога, било би потребно пружити имплементацију оним понашањима у родитељској класи, која су заједничка за сваку класу наследницу. Замислити даље, да се после неког времена појави захтев за новом класом наследницом. Јавља се проблем ако она треба да има другачије понашање од оног које је специфицирано и имплементирано у родитељској класи, као заједничко за тада све варијације. У том случају потребно би било прегазити то понашање. Сваки такав случај би захтевао измену методе која представља одређено заједничко понашање. Уколико се појави захтев за новим понашањем потребним за неке од класа наследница, једно од могућих решења би било додати га у родитељску класу. У том случају би све класе наследнице добиле ново понашање иако им не треба. Наведене наследнице би истакле да не имплементирају дато понашање родитеља, чиме се крши *Liskov Substitution* принцип[4]. Друго решење би било копирати такво понашање у сваку класу наследницу која треба да га садржи. У овом случају, долази до дуплирања истог програмског кода, што опет ствара проблем и кршење *Don't Repeat Yourself* принципа чистог кода[5]. Потребно је приметити да оваквим приступом, долази до великог броја ситуација где је потребно прегазити методу да би добили неко другачије понашање од специфицираног. Због тог разлога, нема велике поновне искористивости кода, која се иначе добија наслеђивањем. Свака мања промена у родитељској класи доводи до великих промена у класама наследницама. Додатни проблем који се јавља је немогућност промене понашања класе наследнице у реалном времену, јер је свакој класи наследници дата имплементација понашања у *compile time-y*.

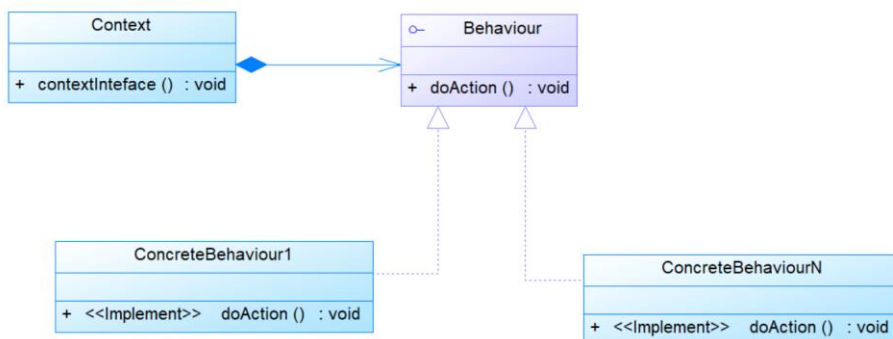
2.3.2 Решење

Решење горепомнутих проблема је дато *Strategy* шаблоном. У првом кораку потребно је сваки тип понашања представити интерфејсом. Све различите варијације тог типа понашања имплементирају тај интерфејс на одговарајући начин. Родитељска класа затим композицијом добија референце ка интерфејсима за сваки тип понашања који је потребно да садржи. Оваквим приступом постигнуто је да свака класа наследница може да има различиту имплементацију за одређени тип понашања. Такође, могуће је и да промени конкретну имплементацију одређеног типа понашања у реалном времену. Уколико је потребно променити имплементацију методе одређеног типа понашања, довољно је само променити класу која имплементира интерфејс тог типа понашања. Програмски код остаје исти. Уместо ослањања на пружање

конкретне имплементације методе која представља понашање, применом *Strategy* шаблона, долази до ослањања на дизајн принцип „Дати предност композицији над наслеђивањем” (енгл. *Favor composition over inheritance*)[1].

2.3.3 Структура

На слици 2.3 је приказан *UML* дијаграм *Strategy* шаблона, док су у табели 2.3 објашњени чиниоци овог дијаграма.



Слика 2.3 *UML* дијаграм *Strategy* шаблона

Назив	Опис
Context	Садржи референцу ка једној од конкретних имплементација понашања и комуницира са њом преко <i>Behaviour</i> интерфејса.
Behaviour	Представља тип одређеног понашања. Све различите имплементације овог типа понашања морају имплементирати овај интерфејс. Декларише методу коју <i>Context</i> користи да би извршио одређено понашање.
ConcreteBehaviour	Конкретна имплементација типа понашања.

Табела 2.3 Опис чиниоца *Strategy* шаблона

2.3.4 Када га користити

Користити *Strategy* шаблон када је потребно имати различите имплементације понашања неке класе, а метода која представља такво понашање може временом променити имплементацију. Такође, користити овај шаблон када је потребно омогућити динамичку промену понашања неке класе.

2.4 State

State шаблон је шаблон понашања, који дозвољава објекту да промени своје понашање када се његово унутрашње стање промени. У таквом сценарију, чиниће се да објекат мења своју класу[1].

2.4.1 Општи опис проблема

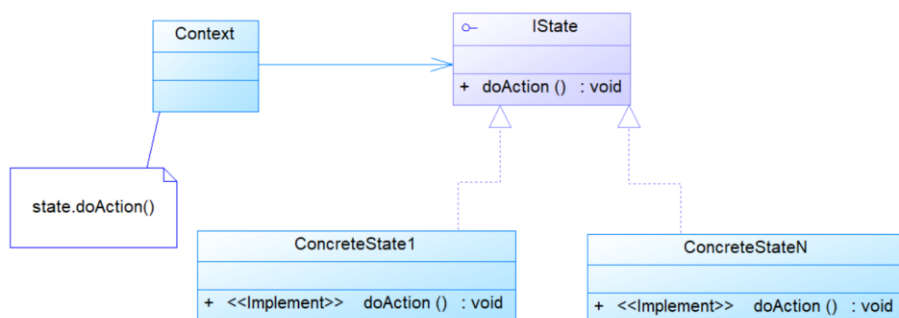
Замислити сценарио где је потребно имплементирати аутомат стања. Аутомат стања, дефинише скуп стања, као и акције које могу узроковати прелазе из једног стања у друго. У зависности од тренутног стања, објекат који представља аутомат стања се понаша другачије. Такође, није увек могућ прелаз из тренутног стања у било које друго. Због те чињенице, за сваку акцију, потребно је проверавати које је тренутно стање и да ли је ту акцију у тренутном стању могуће извршити, и на који начин. Проблеми се јављају приликом увођења новог стања или нове акције за прелаз стања. Увођењем новог стања, потребно је модификовати условљавање за сваку акцију, провером да ли је могуће извршити исту за новонастало стање, на који начин, као и да ли она проузрокује прелаз стања. Ако се јави нова акција, потребно је извршити условљену проверу тренутног стања, и пружити имплементацију функционалности те акције, у зависности од тренутног стања. Тиме се нарушава један од основних дизајн принципа, а то је да класа треба да буде отворена за проширивање, али затворена за модификацију (енгл. *Open-closed principle*)[6].

2.4.2 Решење

Решење овог проблема је дато *State* шаблоном. Уместо да класа која представља машину стања пружи имплементацију сваке акције која може условити прелаз стања из тренутног у неко друго, вршећи условљену проверу тренутног стања, потребно је представити стања помоћу класа. Стања су класе које представљају различита стања аутомата, и свака за себе имплементира акцију онако како би се она извршила ако је у тренутном стању које је дато том класом. Класа која представља аутомат стања има референцу на објекат тренутног стања (објекат који је инстанца класе стања). Позивом било које акције над аутоматом стања, извршење акције препушта се објекту који представља тренутно стање. Када дође до потребе да се промени тренутно стање, само је потребно променити референцу тренутног стања на други објекат дат класом стања. Ово је могуће само ако објекти стања дати класом стања, имају заједнички интерфејс. Давајући им заједнички интерфејс, и пратећи дизајн принцип „Дати предност композицији над наслеђивањем“, *State* шаблон подсећа на *Strategy* шаблон. Међутим, кључна разлика лежи у њиховој намени. *State* шаблон се користи када је потребно имати различита стања неке класе, док се *Strategy* шаблон користи када је потребно динамички мењати понашање неке класе. Оваквим приступом, уколико дође до појава новог стања, потребно је само направити нову класу која репрезентује то стање, и у њој пружити имплементацију за сваку акцију над тим стањем.

2.4.3 Структура

На слици 2.4 је приказан *UML* дијаграм *State* шаблона, док су у табели 2.4 објашњени чиниоци овог дијаграма.



Слика 2.4 *UML* дијаграм *State* шаблона

Назив	Опис
Context	Представља аутомат стања. Дефинише интерфејс ка клијентима. Садржи референцу ка неком стању, означавајући га као тренутно стање. Кад год се пошаље захтев ка <i>Context</i> -у, он га прослеђује објекту који представља тренутно стање.
State	Дефинише заједнички интерфејс који све конкретне класе стања морају имплементирати. Методе овог интерфејса представљају акције које могу проузроковати прелазе стања.
ConcreteState	Конкретно стање које пружа своје специфично понашање за сваку акцију, гледајући своје стање као тренутно.

Табела 2.4 *Опис чиниоца State шаблона*

2.4.4 Када га користити

Користити *State* шаблон када је потребно имати различито понашање објекта, у зависности од његовог тренутног стања. Када је број стања велики, као и када долази до промена у броју стања и акцијама над стањима, погодно га је користити јер окупља сва специфична понашања за конкретно стање на једно место. Самим тим, са тог места је лако манипулисати њиховим извршавањем, као и уколико дође до потребе за промену у имплементацији акција. Типична примена овог шаблона је код различитих апликација за репродуковање аудио и видео садржаја, као и код практично сваког система који представља аутомат стања.

2.5 Singleton

Singleton је креациони дизајн шаблон који осигурава да класа има само једну инстанцу и обезбеђује глобалну тачку приступа истој[1].

2.5.1 Општи опис проблема

Замислити да је потребно пружити имплементацију дељених ресурса као што су објекти за логовање, кориснички интерфејси у виду модалних дијалога, било чега где је потребно осигурати постојање само једног таквог ресурса. Постоји доста ресурса за које је потребно да их представља само један објекат. Због потенцијалне неконзистентности система, као и због потенцијалног пада целог система, уколико их има више од једног. У таквим случајевима, постојање тачно једног објекта за такве ресурсе је неопходно. Једно решење је да се креира један такав објекат, који ће се онда користити. Проблем у таквом решењу је да ништа не спречава да се још један објекат те класе не креира негде другде. Нема никаквог осигурања да ће постојати само тај један објекат, и да није могуће креирати више њих обичним инстанцирањем класе његовог типа. Потребно је на неки начин забранити инстанцирање објеката, уколико објекат класе која представља дељени ресурс већ постоји.

2.5.2 Решење

Решење датог проблема је дато *Singleton* шаблоном и реализује се у два корака. У првом кораку се осигурава да не може било ко да инстанцира објекат *Singleton* класе (класе за коју је потребан максимално један објекат). Да би то било обезбеђено, потребно је конструктор *Singleton* класе прогласити приватним. У другом кораку је потребно креирати статичку методу у *Singleton* класи, која ће обезбеђивати да класа има максимално једну инстанцу и да постоји глобална тачка приступа истој. Та метода, уколико не постоји тренутно ни једна инстанца *Singleton* класе, креира инстанцу позивајући приватни конструктор. Новокреирану инстанцу затим чува у статичком пољу те класе и враћа новокреирану инстанцу као повратну вредност методе. Уколико постоји већ креирана инстанца, метода онда враћа ту инстанцу. На овај начин је обезбеђено да класа не може имати више од једне инстанце. Такође, статичком методом *Singleton* класе, обезбеђено је да је увек могуће имати глобални приступ тој инстанци.

2.5.3 Структура

На слици 2.5 је приказан *UML* дијаграм *Singleton* шаблона, док су у табели 2.5 објашњени чиниоци овог дијаграма.



Слика 2.5 *UML* дијаграм *Singleton* шаблона

Назив	Опис
Singleton	<i>Singleton</i> класа за коју је потребно обезбедити постојање највише једне инстанце. Мора имати приватни конструктор како други објекат не би могао да је инстанцира. Статичком методом <i>getInstance()</i> , добија се инстанца уколико већ постоји. Уколико не постоји, креира се инстанца и враћа као повратна вредност ове методе.

Табела 2.5 Опис чиниоца Singleton шаблона

2.5.4 Када га користити

Користити *Singleton* шаблон када је потребно осигурати да класа има само једну инстанцу и она мора бити доступна преко глобалне тачке приступа. Типичне примене овог шаблона су код лог система, *connection pools* и *thread pools*.

2.6 Chain of responsibility

Chain of responsibility је шаблон понашања који избегава чврсто везивање пошиљаоца захтева за неком акцијом, од његовог примаоца, тако што даје шансу више од једном објекту да обради захтев. Везује објекте који примaju захтев у ланац, и шаље захтев кроз такав ланац док га један објекат не обради[1].

2.6.1 Општи опис проблема

Замислити сценарио где је потребно имплементирати такав систем, где је неопходно да обраду одређеног захтева може извршити више потенцијалних примаоца. Проблем би био ако није познато унапред који прималац треба да изврши обраду. Уколико се обичним условљавањем покуша решити овај проблем, долази до новог проблема. Проблем се јавља уколико је потребно додати новог потенцијалног примаоца за обраду захтева. Због претходно поменутог примера, као и сваком даљом изменом у приоритету примаоца за вршењем обраде захтева, морао би се мењати програмски код, а то доводи до нарушавања дизајн принципа да класа треба да буде отворена за проширивање, али затворена за модификацију[6]. Такође, проблем се јавља уколико је потребно у реалном времену додати или избацити одређени прималац за обраду захтева.

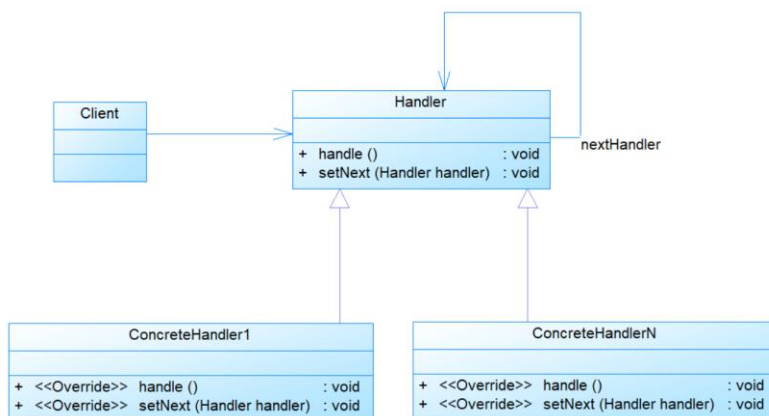
2.6.2 Решење

Решење горепоменутих проблема је дато *Chain of responsibility* шаблоном. Шаблон предлаже да се сваки објекат који је потенцијални прималац за обраду захтева, представи својом посебном класом, тзв. *handler* класом. Такође, потребно их је повезати у ланац, тако што сваки од потенцијалних прималаца захтева има поље које говори који је следећи *handler* у ланцу. Да би ово било могуће, оваквим *handler*-има је

потребно дати исти интерфејс. Захтев за обраду се онда пропушта кроз овако формиран ланац *handler*-а помоћу одговарајуће методе *handler* класе. Сваки *handler* у ланцу врши проверу да ли је у стању да обради захтев. Ако може, обрађује га. Уколико не може да га обради, захтев препушта следећем у ланцу. Уколико је потребно да тачно један *handler* обради захтев, наилазком на такав *handler*, захтев се обрађује и не пропушта даље кроз ланац. У супротном случају, ако је могуће да захтев буде обрађен од стране више *handler*-а, захтев се, без обзира да ли је обрађен или не, увек шаље следећем у ланцу на обраду, док се не стигне до краја ланца. Потребно је приметити да овај шаблон не гарантује сигурну обраду захтева од стране бар једног *handler*-а. Уколико је потребно додати нови потенцијални прималац захтева, потребно је само направити нову класу која ће имплементирати *handler* интерфејс, и објекат те класе додати у ланац *handler*-а. Из разлога што су *handler*-и представљени као класе које су увезане у ланац, могуће је динамички мењати његову структуру, тј. увезивати и извезивати *handler*-е из ланца по потреби.

2.6.3 Структура

На слици 2.6 је приказан *UML* дијаграм *Chain of responsibility* шаблона, док су у табели 2.6 објашњени чиниоци овог дијаграма.



Слика 2.6 *UML* дијаграм *Chain of responsibility* шаблона

Назив	Опис
Client	Шаље захтев који је потребно обрадити ка ланцу потенцијалних прималаца.
Handler	Дефинише интерфејс за обраду захтева који сваки прималац мора имплементирати. Садржи методу <i>setNext()</i> за додавање новог примаоца у ланац. Метода <i>handle()</i> представља методу која служи за обраду захтева.
ConcreteHandler	Потенцијални прималац захтева. Уколико може да обави обраду захтева, обавља је. У супротном, обаву захтева прослеђује наредном <i>handler</i> -у у ланцу.

Табела 2.6 Опис чиниоца Chain of responsibility шаблона

2.6.4 Када га користити

Користити *Chain of responsibility* шаблон када објекат који треба да обради захтев није познат унапред. Такође, користити га када више објеката може обрадити захтев. Уколико је потребно динамички додати објекат за потенцијалну обраду захтева, *Chain of responsibility* шаблон, због његове структуре ланца, може обезбедити реализацију потребе за таквим захтевом.

2.7 Bridge

Bridge је структурални дизајн шаблон који одваја апстракцију од њене имплементације и допушта да то двоје независно варирају[1].

2.7.1 Општи опис проблема

Замислити сценарио где је потребно имати различите типове класе А по некој основи. Нека та основа представља одређено својство С, класе А. За сваку различиту вредност својства С, постојаће различити тип класе А. Приступ решавању овог проблема би био дефинисати класу А, а наслеђивањем створити њене различите типове по тој основи. На пример, подтип А₁ и подтип А₂. Класа А₁ има једну вредност својства С, вредност С₁, а класа А₂ неку другу вредност својства С, вредност С₂. Замислити да је после неког времена потребно разликовати класу А по новој основи, односно по вредности неког новог својства, својства М. Класа А може имати две вредности својства М, М₁ и М₂. Потребно би било да се сви досадашњи подтипови разликују и по тој основи. На основу претходног примера, нови подтипови би били С₁М₁, С₁М₂, С₂М₁ и С₂М₂. Класа С₁М₁ има С₁ вредност својства С и М₁ вредност својства М. Класа С₁М₂ има С₁ вредност својства С и М₂ вредност својства М. Класа С₂М₁ има С₂ вредност својства С и М₁ вредност својства М. Класа С₂М₂ има С₂ вредност својства С и М₂ вредност својства М. Потребно је приметити да се број класа наследница удвостручио. Додајући нове

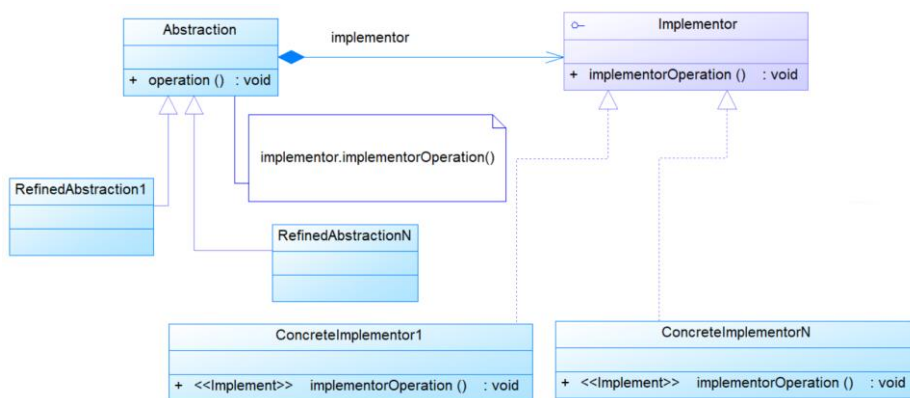
основе, као и нове подтипове класа, долази до непотребног повећавања броја класа.

2.7.2 Решење

До претходног проблема долази због покушаја да се наслеђивањем реши проблем разликовања класа по свакој новој основи. *Bridge* шаблон овај проблем решава пратећи дизајн принцип: „Фаворизуј композицију пре него наслеђивање”[1]. Када је потребно имати различите типове класе по некој основи, тада се наслеђивањем решава овај проблем. Тиме се добија разликовање класа по првобитној намери. Сваку следећу потребу разликовања класа по некој основи је потребно решити композицијом. Потребно је направити нови интерфејс који представља нову основу. Након тога, сваку варијацију по тој основи потребно је решити новом класом која имплементира тај интерфејс. У последњем кораку, потребно је додати референцу на интерфејс те основе у првобитној класи, након чега је могуће да класа добије било коју имплементацију те основе, јер оне имају заједнички интерфејс. Уместо великог броја класа и наслеђивања, класе сада имају референцу ка онолико интерфејса, колико је потребно имати разлика по неким основама. Оваквим приступом, избегнут је велики број класа.

2.7.3 Структура

На слици 2.7 је приказан *UML* дијаграм *Bridge* шаблона, док су у табели 2.7 објашњени чиниоци овог дијаграма.



Слика 2.7 *UML* дијаграм *Bridge* шаблона

Назив	Опис
Abstraction	Класа коју је потребно разликовати по више основа. По првобитној основи, наслеђивањем добија своје подтипове, а за сваку следећу потребу разликовања ове класе по некој основи, потребно је композицијом додати референцу на ту основу (<i>Implementor</i>).
RefinedAbstraction	Наслеђује класу <i>Abstraction</i> . Представља њену одређену варијацију по првобитној основи.
Implementor	Интерфејс који представља основу по којој је потребно разликовати класу <i>Abstraction</i> .
ConcreteImplementor	Конкретна варијација основе дефинисана интерфејсом <i>Implementor</i> .

Табела 2.7 Опис чиниоца *Bridge* шаблона

2.7.4 Када га користити

Користити *Bridge* шаблон када је потребно разликовати неку класу по више основа, где потенцијално може доћи до великог броја варијација класа, само зато што се разликују по тим основама. Предност овог шаблона је да је могуће динамички променити варијацију основе, јер је она представљена композицијом, и сакривена иза интерфејса.

2.8 Observer

Observer је шаблон понашања који дефинише *one-to-many* зависност између објеката, тако да када један објекат промени своје унутрашње стање, тада су сви објекти који зависе од њега аутоматски обавештени и ажурирани[1].

2.8.1 Општи опис проблема

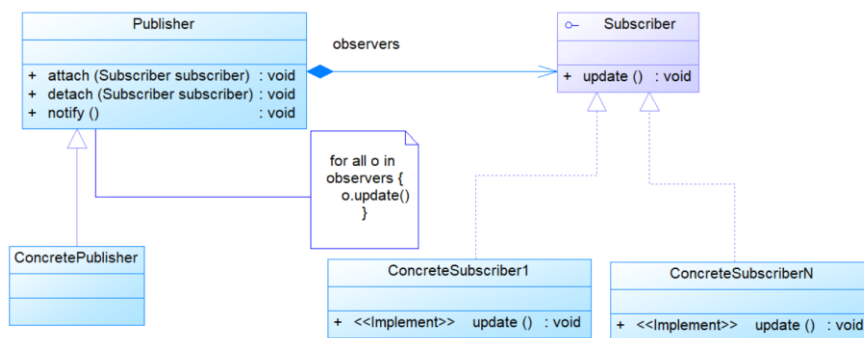
Нека је објекат класе А један од многих објеката који треба да реагује на промену стања објекта класе Б. Класичним приступом, претходни проблем је могуће решити на два начина. Први начин је да објекат класе А у одређеним временским интервалима испитује стање објекта класе Б. Уколико је дошло до промене стања, објекат класе А врши одговарајућу акцију. Ово је лош приступ из разлога што се шаље велики број захтева ка објекту класе Б и врши велики број непотребних провера. Други начин је да објекат класе Б, у тренутку када дође до промене његовог стања, обавештава све објекте који зависе од њега. Овај приступ је бољи од првог, али је лош из разлога што тако долази до чврстог везивања за објекте класе А, што доводи до немогућности обавештавања неког новог типа, уколико дође до потребе за тим [2].

2.8.2 Решење

Решење овог проблема је дато *Observer* шаблоном. *Observer* шаблон дефинише зависност између класа као *Subscriber-Publisher*[7]. *Publisher* је она класа чије промене стања је потребно пратити. *Subscriber* је она класа која жели да буде обавештена о одређеној промени стања у класи *Publisher*. По правилу *Observer* шаблона, потребно је дефинисати механизам за додавање и избацивање *Subscriber*-а у *Publisher*. Сваки пут када се промени стање у *Publisher*-у, он обавештава све своје *Subscriber*-е. *Subscriber*-и не морају бити објекти исте класе, већ је битно да имплементирају исти интерфејс. Оваквим приступом се избегава чврста повезаност *Publisher*-а са типовима *Subscriber*-а.

2.8.3 Структура

На слици 2.8 је приказан *UML* дијаграм *Observer* шаблона, док су у табели 2.8 објашњени чиниоци овог дијаграма.



Слика 2.8 *UML* дијаграм *Observer* шаблона

Назив	Опис
Publisher	Дефинише костур класе чије промене стања је потребно пратити. Дефинише које методе таква класа мора имплементирати. Има знање о својим <i>Subscriber</i> -има, који имплементирају <i>Subscriber</i> интерфејс. По промени стања, обавештава све своје <i>Subscriber</i> -е.
Subscriber	Интерфејс који све класе које желе да се региструју као <i>Subscriber</i> -и морају имплементирати.
ConcretePublisher	Конкретна класа чије промене стања је потребно пратити.
ConcreteSubscriber	Конкретна класа која жели да се региструје као <i>Subscriber</i> , односно, која жели да прати промене стања у класи <i>ConcretePublisher</i> .

Табела 2.8 Опис чиниоца *Observer* шаблона

2.8.4 Када га користити

Користити *Observer* шаблон када промена у стању једног објекта утиче на стање или понашање других објеката, који су зависни од њега.

Такође, користити овај шаблон када нису унапред познате класе објеката које је потребно обавестити о промени стања.

2.9 Decorator

Decorator је структурални дизајн шаблон који дозвољава да се додатне одговорности објекту додају динамички. Пружа флексибилну алтернативу наслеђивању у виду додавања функционалности[1].

2.9.1 Општи опис проблема

Замислити сценарио где је потребно направити класу *C*. Класа *C* треба да има одређену функционалност, и постоји *N* различитих варијације те класе. На прву помисао, овај проблем се може решити наслеђивањем. Сваки од *N* различитих типова ће прегазити функционалност у родитељској класи *C*, тамо где је потребно да се та функционалност промени. Замислити да је након извесног времена потребно наследнице класе *C* разликовати по присуству неког својства у њима. На пример, ако је *A* једно својство варијације класе *C*, потребно је даље разликовати ту варијацију по присуству својства *A*. Додавањем нових својстава по којима је такође потребно разликовати класе наследнице, настао би велики број класа које би се разликовале само по присуству тих својстава у њима. Такође би проблем настао додавањем нове наследнице класе *C*. Тада би за свако својство било потребно даље разликовати додану класу наследницу. Такође, у свим класама наследницама би било потребно прегазити методу која представља жељену функционалност, на сопствен начин, у зависности од присуства одређених својстава у тој класи наследници. Други приступ би био да се уведу логичке променљиве у наследницама класе *C*, које означавају присуство одређеног својства. Овакав приступ је бољи од претходног, али би се и тако за сваку потребну функционалност морала проверавати појава свих својстава у класи наследници. Додавањем новог својства, свуда где се врши провера присуства одређеног својства, морала би се додати и провера присуства новог својства. Долази до нарушавања *Open-Closed* принципа[6].

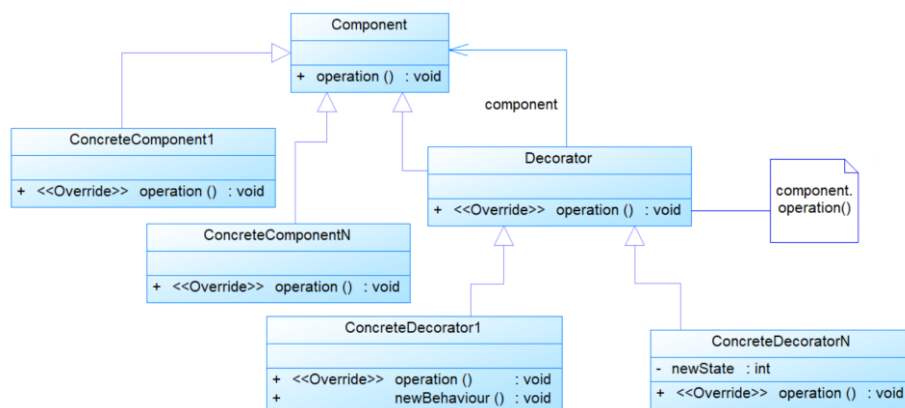
2.9.2 Решење

Решење претходног проблема је дато *Decorator* шаблоном. *Decorator* шаблон уско прати дизајн принцип: „Фаворизуј композицију пре него наслеђивање[1]”. Уместо да се претходно наведени проблем решава класичним наслеђивањем, декоратор пружа флексибилну алтернативу таквом приступу. Сваки различити тип класе *C*, треба наследити класу *C*, чиме се добијају различите варијације класе *C*. Додатна својства која могу, а и не морају бити присутна у класи, се називају декоратори. Осим различитих типова класе *C* који је наслеђују,

још једна класа наслеђује класу *C*, а то је *Decorator* класа. *Decorator* класа представља апстрактну класу, коју конкретни декоратори морају наследити. Конкретни декоратори представљају својства која наследнице класе *C* могу, или не морају имати. Поред тога што *Decorator* класа наслеђује класу *C*, она садржи и референцу ка њој. Оваквом структуром, као и због тога што конкретни декоратори наслеђују овако структурирану *Decorator* класу, могуће је међусобно садржавање конкретних декоратора и наследница класе *C*. Разлог за то је јер су све наследнице класе *C*.

2.9.3 Структура

На слици 2.9 је приказан *UML* дијаграм *Decorator* шаблона, док су у табели 2.9 објашњени чиниоци овог дијаграма.



Слика 2.9 *UML* дијаграм *Decorator* шаблона

Назив	Опис
Component	Класа <i>C</i> за коју је потребно имати више варијација.
ConcreteComponent	Дефинише конкретну варијацију као поткласу класе <i>C</i> , коју је даље потребно разликовати по одређеним својствима.
Decorator	Декоратор класа. Наслеђује класу <i>C</i> , и садржи референцу на класу <i>C</i> , којом се добија структура обавијања једног декоратора другим.
ConcreteDecorator	Класа која представља својство по којем је потребно разликовати поткласе класе <i>C</i> .

Табела 2.9 Опис чиниоца *Decorator* шаблона

2.9.4 Када га користити

Користити *Decorator* шаблон када је потребно имати различите комбинације класа, које није могуће знати унапред. Такође, користити овај шаблон када је потребно одговорности класи додати динамички, као и кад је разликовање класа наслеђивањем непрактично због стварања великог броја сличних класа.

2.10 *Factory method*

Factory method је креациони дизајн шаблон који дефинише интерфејс за креирање објекта, али препушта класама наследницама да одлуче коју класу ће инстанцирати. *Factory method* дозвољава да родитељска класа препусти инстанцирање својим класама наследницама[1].

2.10.1 Општи опис проблема

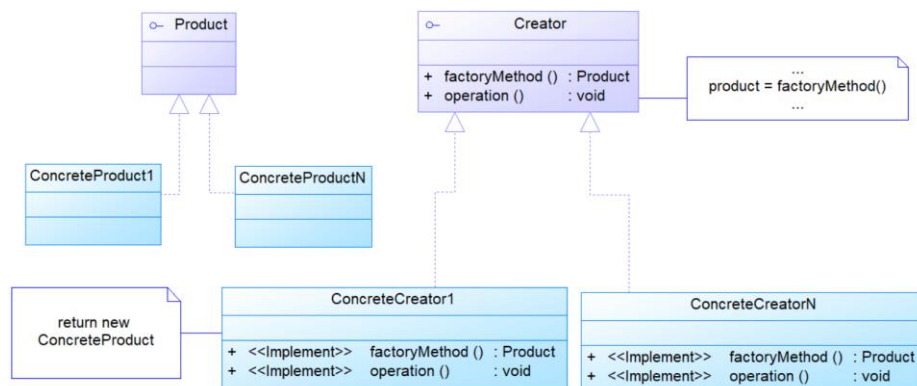
Замислити сценарио где је дата родитељска класа *A*, која има своје класе наследнице. Даље, замислити да је дата нова класа, која садржи методе у којима се извршавају одређене операције над објектом неке класе наследнице класе *A*. Такав објекат, пре коришћења, би било потребно креирати. Проблем се јавља уколико није познато коју конкретно класу наследницу класе *A* инстанцирати. Такође, проблем се јавља и уколико би логика креирања била позната, али је комплексна и понавља се на више места у апликацији. На сваком месту где је потребно креирати објекат, извршавала би се логика креирања, где би дошло до понављања програмског кода. Понављање је веће што је логика креирања комплекснија. Уколико дође до промене логике у креирању објекта, потребно би било на свим местима у апликацији где се налази логика креирања, модификовати програмски код. Решење претходног проблема је *Simple factory method* [8]. Уколико се јави потреба за увођењем нове наследнице класе *A*, *Simple factory method* није довољно решење. Стари програмски код не може да користи објекат нове наследнице класе *A*. Потребно би било прилагодити стари програмски доданој класи.

2.10.2 Решење

Решење датог проблема је дато *Factory method* шаблоном. Шаблон предлаже додавање посебне методе чија је сврха креирање објекта конкретне наследнице класе *A*, у класу где је потребно да такав објекат већ постоји. Таква метода се зове *factory* метода, и њена функција је извршавање одређене логике ради креирања објекта конкретне наследнице класе *A* (класе *Product*). Повратна вредност ове методе је креирани објекат. Класа која садржи *factory* методу се зове *Creator*. С обзиром да није познато који конкретан *Product* треба да буде креиран, класе наследнице које ће наследити класу *Creator* морају пружити имплементацију *factory* методе. Оваквим приступом је цела логика креирања издвојена у посебну методу, *factory* методу. Уколико је потребно мењати логику креирања, потребно је променити је само на једном месту. Класа *Creator* препушта логику инстанцирања својим класама наследницама. Самим тим може да претпостави да је конкретан *Product* успешно креиран, и да настави да ради са њим.

2.10.3 Структура

На слици 2.10 је приказан *UML* дијаграм *Factory method* шаблона, док су у табели 2.10 објашњени чиниоци овог дијаграма.



Слика 2.10 *UML* дијаграм *Factory method* шаблона

Назив	Опис
Product	Дефинише интерфејс класе А, који класе наследнице класе А имплементирају.
ConcreteProduct	Конкретна класа наследница класе А коју је потребно креирати у методи <i>factoryMethod()</i> .
Creator	Дефинише интерфејс класе Б, који све класе наследнице класе Б морају имплементирати. Декларише <i>factoryMethod()</i> , као и остале методе које садрже логику класе. <i>FactoryMethod()</i> враћа одређени <i>ConcreteProduct</i> .
ConcreteCreator	Конкретна класа наследница класе Б којој је препуштено креирање одређене класе наследнице класе А. Потребно је да пружи имплементацију методе <i>factoryMethod()</i> која враћа одговарајућ <i>ConcreteProduct</i> .

Табела 2.10 Опис чиниоца *Factory method* шаблона

2.10.4 Када га користити

Користити *Factory method* шаблон када класа која инстанцира неки објекат не зна унапред конкретну класу тог објекта. Такође, користити овај шаблон када је потребно да родитељска класа препусти креирање објекта класи наследници.

3. Реална примена / пројекат

У овом поглављу биће пружен увид у имплементацију софтверског пројекта и његових подсистема. Сваки подсистем, је описан кроз функционалности које подржава. За сваку потребну функционалност ће постојати два решења. Прво решење ће избегавати употребу одговарајућег дизајн шаблона, уз коментар до каквих проблема долази таквим приступом. Друго решење ће бити дато уз употребу одговарајућег дизајн шаблона, као и уз коментар које су предности таквог приступа.

3.1 Опште напомене о пројекту

Софтверски пројекат који је имплементиран је јавно доступан и његов програмски код се може пронаћи на *GitHub*-у¹. С обзиром на потребу постојања два решења за сваки проблем, биће дате две апликације. Прва, која повезује подсистеме који су имплементирани не користећи дизајн шаблоне, дата класом *WithoutPatternApp*. Друга апликација повезује подсистеме користећи дизајн шаблоне, и дата је класом *PatternApp*. Софтверски пројекат који је имплементиран се састоји од четири подсистема, а то су подсистеми за пицерију, *YouTube* налог, *Google* налог, као и музички плејер. Повезани подсистеми, чинећи заједно једну апликацију, пружиће увид у имплементацију *Singleton* шаблона. Подсистем за пицерију симулира процес поруџбина пице, и пружа увид у примену шаблона *Decorator*, *Chain of responsibility*, *Bridge*, *Template method* и *Factory method*. Подсистем за *YouTube* налог је имплементиран тако да представља реалистичан *YouTube* налог, и пружа увид у примену *Observer* и *Strategy* шаблона. Подистем за *Google* налог је имплементиран тако да представља реалистичан *Google* налог, и додатно пружа увид у примену *Observer* шаблона. Подсистем за музички плејер симулира рад музичког плејера и пружа увид у примену *State* и *Adapter* шаблона.

3.1.1 *Singleton*

Сваки подсистем садржи класу чија је улога да омогући кориснику да управља тим подсистемом. Класа *MusicPlayerApp* представља клијент класу за корисничко управљање музичким плејером. Класа *YouTubeApp* представља клијент класу за управљање *YouTube* налозима. Класа *GoogleAccountApp* представља клијент класу за управљање *Google* налозима. Класа *PizzaStoreApp* представља класу за корисничко управљање пицеријом. Дате класе потребно је повезати заједно да чине једну целину која ће представљати апликацију.

¹ <https://github.com/DusanRadj/DesignPatterns>

3.1.1.1 Опис проблема

Повезивање подсистема је могуће извршити креирањем објекта за сваку класу која представља клијента за неки подсистем. Након тога, потребно је додати референце на такве објекте у класу која ће чинити апликацију. Класа *WithoutPatternApp* која представља апликацију, дата је на листингу 3.1.

```
class WithoutPatternApp
{
    private MusicPlayerApp musicPlayerApp;
    private GoogleAccountApp googleAccountApp;
    private YouTubeApp youtubeApp;
    private PizzaOrderApp pizzaOrderApp;

    public WithoutPatternApp()
    {
        this.musicPlayerApp = new MusicPlayerApp();
        this.googleAccountApp = new GoogleAccountApp();
        this.youtubeApp = new YouTubeApp();
        this.pizzaOrderApp = new PizzaOrderApp();
    }

    public void menu()
    {
        while (true)
        {
            Console.Clear();
            Console.WriteLine("Choose example to run: ");
            Console.WriteLine("1. Music player ");
            Console.WriteLine("2. YouTube ");
            Console.WriteLine("3. Google account");
            Console.WriteLine("4. Pizza order");
            Console.WriteLine("0. Exit");
            Console.WriteLine();
            Console.Write("Command: ");

            String option = Console.ReadLine();

            Console.Clear();

            switch (option)
            {
                case "1":
                {
                    this.musicPlayerApp.startApp();
                    break;
                }
                case "2":
                {
                    this.youtubeApp.startApp();
                    break;
                }
                case "3":
                {
                    this.googleAccountApp.startApp();
                    break;
                }
                case "4":
                {
                    this.pizzaOrderApp.startApp();
                    break;
                }
                case "0":
                {
                    return;
                }
            }
        }
    }
}
```

Листинг 3.1 Класа *WithoutPatternApp*

Након инстанцирања објекта класе *WithoutPatternApp*, кориснику је омогућено управљање повезаним подсистемима. Проблем оваквог приступа је да ништа не забрањује вишеструко инстанцирање класа које представљају клијенте за подсистеме. Такође, ништа не забрањује ни постојање више инстанци апликације *WithoutPatternApp*. Постојање више од једне инстанце ових класа, може довести до неконзистентности система, као и података које се налазе у инстанцама тих класа.

3.1.1.2 Решење

Решење претходног проблема је дато *Singleton* шаблоном. Потребно је представити сваку клијент класу подсистема као *Singleton* класу (листинг 3.2).

```
class MusicPlayerApp
{
    private MusicPlayer musicPlayer;
    private static MusicPlayerApp instance;

    private MusicPlayerApp()
    {
        this.musicPlayer = new MusicPlayer();
        this.initializeMusicPlayerWithSongs();
    }

    public static MusicPlayerApp getInstance()
    {
        if (instance == null)
        {
            instance = new MusicPlayerApp();
        }

        return instance;
    }
}
```

Листинг 3.2 Клијент класа за подсистем музичког плејера

Класу *PatternApp*, која представља апликацију, такође је неопходно представити као *Singleton* класу (листинг 3.3).

```

class PatternApp
{
    public PatternApp() { }

    public void menu()
    {
        while (true)
        {
            Console.Clear();
            Console.WriteLine("Choose example to run: ");
            Console.WriteLine("1. Music player");
            Console.WriteLine("2. YouTube");
            Console.WriteLine("3. Google account");
            Console.WriteLine("4. Pizza ordering");
            Console.WriteLine("0. Exit");
            Console.WriteLine();
            Console.Write("Command: ");

            String option = Console.ReadLine();

            Console.Clear();

            switch (option)
            {
                case "1":
                {
                    MusicPlayerApp.getInstance().startApp();
                    break;
                }
                case "2":
                {
                    YouTubeApp.getInstance().startApp();
                    break;
                }
                case "3":
                {
                    GoogleAccountApp.getInstance().startApp();
                    break;
                }
                case "4":
                {
                    PizzaOrderApp.getInstance().startApp();
                    break;
                }
                case "0":
                {
                    return;
                }
            }
        }
    }
}

```

Листинг 3.3 *PatternApp* класа

Оваквим приступом, обезбеђује се постојање максимално једне инстанце клијент класа, као и класе која представља апликацију. Тиме се онемогућује да било који други објекат инстанцира дате класе, уколико њихове инстанце већ постоје. Пружајући максимално један објекат датим класама, и обезбеђујући глобалну тачку приступа ка њима, избегава се потенцијална неконзистентност система.

3.2 Музички плејер

Музички плејер представља подсистем који симулира рад музичког плејера. Корисник преко различитих функционалности музичког плејера, манипулише репродукцијом аудио садржаја који се на њему налази.

3.2.1 State

Потребно је имплементирати функционалности класе *MusicPlayer* која представља музички плејер. Музички плејер обухвата функционалности укључивања, искључивања, репродукције звука, паузирања репродукције звука, заустављања репродукције звука, премотавања на следећи звучни запис, враћања на претходни звучни запис, као и закључавања и откључавања.

3.2.1.1 Опис проблема

С обзиром на претходно дате функционалности музичког плејера, потребно је дефинисати његове методе тако да одговарају потребним функционалностима. Потребно је приметити да се не може свака од функционалности применити у сваком случају, тј. стању музичког плејера. На пример, ако музички плејер тренутно није укључен, није могуће извршити репродукцију звучног записа. Овај проблем је могуће решити увођењем променљивих стања које ће представљати свако потенцијално стање музичког плејера. Такође, потребно је увести и једну променљиву која представља тренутно стање музичког плејера (листинг 3.4).

```
class MusicPlayer
{
    readonly static int LOCKED = 0;
    readonly static int PAUSED = 1;
    readonly static int PLAYING = 2;
    readonly static int STAND_BY = 3;
    readonly static int TURNED_OFF = 4;

    private int state = TURNED_OFF;
```

Листинг 3.4 Потенцијална стања и тренутно стање музичког плејера

У свакој методи, потребно је извршити проверу тренутног стања. У зависности од тренутног стања, потребно је извршити методу на начин који одговара том стању. Листинг 3.5 приказује имплементацију оваког приступа, на примеру методе за репродукцију звучног записа.

```

public void play()
{
    if (!this.isWorking)
    {
        Console.WriteLine("There is an error in media player!");
        return;
    }

    if (state == TURNED_OFF)
    {
        Console.WriteLine("You need to turn on your device first!");
    }
    else if (state == STAND_BY)
    {
        Song song = this.Songs[CurrentSongIndex];
        Console.WriteLine(song.Artist + " - " + song.Title + " is now playing...");
        this.timer.Enabled = true;
        this.state = PLAYING;
        this.MediaPlayer.FileName = song.Path;
        this.MediaPlayer.Play();
    }
    else if (state == PLAYING)
    {
        this.stopTimerAndSound();
        Song song = this.Songs[this.CurrentSongIndex];
        Console.WriteLine(song.Artist + " - " + song.Title + " is now playing ...");
        this.timer.Enabled = true;
        this.MediaPlayer.FileName = song.Path;
        this.MediaPlayer.Play();
    }
    else if (state == PAUSED)
    {
        Song song = this.Songs[this.CurrentSongIndex];
        Console.WriteLine(song.Artist + " - " + song.Title + " is resuming from " +
            this.PausedAt + " seconds ...");
        this.MediaPlayer.Play();
        this.timer.Enabled = true;
        this.state = PLAYING;
    }
    else if (state == LOCKED)
    {
        Console.WriteLine("You need to unlock your device first!");
    }
}
}

```

Листинг 3.5 *Имплементација методе за репродукцију звучног записа*

Проблем претходног приступа је вршење условљене провере за сваку функционалност. Додавањем нове функционалности, морала би се дефинисати нова метода која би одговарала таквој функционалности. У телу те методе, опет би се морала извршити условљена провера тренутног стања и пружити одговарајућа имплементација те методе за свако стање. Такође, уколико је потребно додати ново стање, свака метода би се морала мењати због вршења додатне провере, а то је да ли је музички плејер у новододаном стању. Ако јесте, неопходно би било и пружити одређену имплементацију сваке методе за ново стање. Овакав програмски код је тешко одржавати и нарушава *Open-Closed* дизајн принцип[6].

3.2.1.2 Решење

Решење претходног проблема је дато *State* шаблоном. Класу *MusicPlayer* је могуће представити као аутомат стања. Стања која музички плејер може да има су *закључан*, *паузиран*, *у репродукцији*, *у приправности* и *угашен*. Класе које репрезентују ова стања су

LockedState, *PausedState*, *PlayingState*, *StandByState* и *TurnedOffState*, респективно. Класа *AbstractState* представља заједнички интерфејс који декларише методе које свако стање мора имплементирати. Методама које имају исту имплементацију у великом броју конкретних стања, може се пружити имплементација у овој класи. Ако је потребно да у неком конкретном стању имплементација такве методе буде другачија, метода се може редефинисати. Пример таквих метода су *turnOn()* и *turnoff()* (листинг 3.6)

```
abstract class AbstractState
{
    protected MediaPlayer musicPlayer;

    protected AbstractState(MusicPlayer musicPlayer)
    {
        this.musicPlayer = musicPlayer;
    }

    public virtual void turnOn()
    {
        Console.WriteLine("Device is already turned on!");
    }

    public virtual void turnOff(){...}

    public abstract void play();
    public abstract void pause();
    public abstract void stop();
    public abstract void lockUnlock();
    public abstract void next();
    public abstract void previous();
}
```

Листинг 3.6 Класа *AbstractState*

Свако стање наслеђујући класу *AbstractState*, имплементира методе онако како би се оне извршиле уколико је музички плејер у том стању. На листингу 3.7 је дат пример класе која представља стање за репродукцију звука, као и имплементација методе *pause()* која врши паузирање репродукције звука.

```
class PlayingState : AbstractState
{
    public PlayingState(MusicPlayer musicPlayer) : base(musicPlayer) { }

    public override void play(){...}

    public override void pause()
    {
        musicPlayer.MyMediaPlayer.pause();
        musicPlayer.Timer.Enabled = false;
        Console.WriteLine("Song paused on " + musicPlayer.PausedAt + " seconds ...");
        musicPlayer.CurrentState = musicPlayer.pausedState;
        Console.ForegroundColor = ConsoleColor.DarkYellow;
        Console.WriteLine("STATE CHANGED : PLAYING_STATE ---> PAUSED_STATE");
        Console.ResetColor();
    }

    public override void stop(){...}
    public override void lockUnlock(){...}
    public override void next(){...}
    public override void previous(){...}
}
```

Листинг 3.7 Класа *PlayingState*

Потребно је да класа *MusicPlayer* садржи референце на сва могућа стања, као и на тренутно стање у ком се налази, које је дато пољем *currentState* (листинг 3.8).

```
class MusicPlayer
{
    public AbstractState turnedOffState;
    public AbstractState standbyState;
    public AbstractState lockedState;
    public AbstractState playingState;
    public AbstractState pausedState;
    private AbstractState currentState;

    public MusicPlayer()
    {
        this.turnedOffState = new TurnedOffState(this);
        this.standbyState = new StandByState(this);
        this.playingState = new PlayingState(this);
        this.pausedState = new PausedState(this);
        this.lockedState = new LockedState(this);
        this.currentState = turnedOffState;
    }
}
```

Листинг 3.8 Класа *MusicPlayer*

Уколико дође до позива било које функционалности музичког плејера, музички плејер извршење препушта тренутном стању, односно објекту *currentState* (листинг 3.9).

```
public void play()
{
    this.currentState.play();
}
```

Листинг 3.9 Препуштање извршења *play()* функционалности

Кад дође до потребе за променом тренутног стања, довољно је променити референцу објекта тренутног стања *MusicPlayer*-а на други објекат стања. На листингу 3.10, приказана је потреба промене тренутног стања, када се током репродукције звука позове функционалност паузирања. Тада је потребно да *MusicPlayer* из стања *PlayingState*, пређе у стање *PausedState*.

```
class PlayingState : AbstractState
{
    public PlayingState(MusicPlayer musicPlayer) : base(musicPlayer) { }

    public override void play(){...}

    public override void pause()
    {
        musicPlayer.MyMediaPlayer.pause();
        musicPlayer.Timer.Enabled = false;
        Console.WriteLine("Song paused on " + musicPlayer.PausedAt + " seconds ...");
        musicPlayer.CurrentState = musicPlayer.pausedState;
        Console.ForegroundColor = ConsoleColor.DarkYellow;
        Console.WriteLine("STATE CHANGED : PLAYING_STATE --> PAUSED_STATE");
        Console.ResetColor();
    }
}
```

Листинг 3.10 Промена тренутног стања

Оваквим приступом, класа *MusicPlayer* не треба да познаје детаље имплементације одређених функционалности, већ је потребно да зна само у ком је тренутно стању. Тренутном стању се препушта извршење функционалности. Уколико је неопходно додати ново стање, потребно је само направити нову класу стања, која наслеђује класу *AbstractState*. Након наслеђивања класе *AbstractState*, потребно је пружити имплементацију свих функционалности музичког плејера, онако како би се те функционалности извршиле ако су у стању које је дато класом стања. Уколико је неопходно додати нову функционалност, потребно је декларисати методу у класи *MusicPlayer* која представља ту функционалност. Такође, потребно је додати такву методу и у класу *AbstractState*, као и пружити јој имплементацију у свакој класи стања. Користећи *State* шаблон за дати проблем, програмски код је лако одржавати, јер се за свако посебно стање налази на једном месту, у класи стања. Такође, нема дуплирања кода, услед изостанка условљене провере тренутног стања за сваку функционалност, као и могуће имплементације метода у класи *AbstractState* (уколико је тело тих метода исто за велики број стања).

3.2.2 Adapter

Потребно је омогућити манипулацију репродукције звука за музички плејер, тако да се звучни запис може репродуковати, паузирати, одпаузирати и зауставити.

3.2.2.1 Опис проблема

Функционалност манипулације репродукције звука је могуће имплементирати користећи класу *MediaPlayer*. Објекат те класе потребно је сместити као поље класе *MusicPlayer* (листинг 3.11).

```
private MediaPlayer mediaPlayer;
```

Листинг 3.11 Објекат класе *MediaPlayer*

Уколико је потребно репродуковати звучни запис, функционалност репродукције звука се препушта објекту *mediaPlayer*. Оваквим приступом долази до везивања за класу *MediaPlayer*. Уколико дође до појаве *bug*-а у тој класи, више није могуће вршити репродукцију звука. Симулација проузроковања *bug*-а дата је на листингу 3.12. Проблем се такође јавља и уколико је после неког времена потребно користити неку другу класу или библиотеку. Услед везивања за конкретну класу код тренутног приступа, неопходно би било мењати програмски код.

```
public void causeBug()
{
    this.isWorking = false;
}
```

Листинг 3.12 Метода која проузрокује bug

3.2.2.2 Решење

Решење претходног проблема је дато *Adapter* шаблоном. Потребно је дефинисати интерфејс који апстрахује жељене функционалности за манипулацију звучним записом, *IMyMediaPlayer* (листинг 3.13).

```
interface IMyMediaPlayer
{
    void play(string path);
    void pause();
    void stop();
    void resume();
}
```

Листинг 3.13 *IMyMediaPlayer* интерфејс

С обзиром да је потребно користити *MediaPlayer* класу, неопходно је направити адаптер класу која имплементира *IMyMediaPlayer* интерфејс и има референцу ка објекту *MediaPlayer* класе. На листингу 3.14 је дата *MediaPlayerAdapter* класа.

```
class MediaPlayerAdapter : IMyMediaPlayer
{
    private MediaPlayer.MediaPlayer mediaPlayer;

    public MediaPlayerAdapter()
    {
        this.mediaPlayer = new MediaPlayer.MediaPlayer();
    }

    public void play(string path)
    {
        this.mediaPlayer.FileName = path;
        this.mediaPlayer.Play();
    }

    public void pause()
    {
        this.mediaPlayer.Pause();
    }

    public void stop()
    {
        this.mediaPlayer.Stop();
    }

    public void resume()
    {
        this.mediaPlayer.Play();
    }
}
```

Листинг 3.14 *MediaPlayerAdapter* класа

Позив жељене функционалности *MediaPlayerAdapter* класе, прослеђује се пољу ове класе, односно, објекту конкретне класе што је у овом случају класа *MediaPlayer*. Оваквим приступом, клијент не

комуницира директно са конкретном класом или библиотеком, већ преко интерфејса. Музички плејер уместо конкретне класе *MediaPlayer*, сада има референцу ка објекту класе која имплементира интерфејс *IMyMediaPlayer* (листинг 3.15).

```
private IMyMediaPlayer myMediaPlayer;
private bool isWorking = true;

public MediaPlayer()
{
    this.myMediaPlayer = new MediaPlayerAdapter();
}
```

Листинг 3.15 *MediaPlayer* класа

Потребно је приметити да класа *MediaPlayer* нема свест о томе која конкретно класа, односно адаптер, се крије иза овог интерфејса. На листингу 3.16 је приказана још једна адаптер класа за манипулацију репродукције звука, која користи *NAudio 3rd party* библиотеку.

```
class NAudioAdapter : IMyMediaPlayer
{
    NAudio.Wave.DirectSoundOut output;

    public NAudioAdapter()
    {
        this.output = new NAudio.Wave.DirectSoundOut();
    }

    public void play(string path)
    {
        NAudio.Wave.WaveStream a = NAudio.Wave.WaveFormatConversionStream.CreatePcmStream(
            new NAudio.Wave.Mp3FileReader(path));
        NAudio.Wave.BlockAlignReductionStream stream =
            new NAudio.Wave.BlockAlignReductionStream(a);
        this.output.Init(stream);
        this.output.Play();
    }

    public void pause()
    {
        this.output.Pause();
    }

    public void stop()
    {
        this.output.Stop();
    }

    public void resume()
    {
        this.output.Play();
    }
}
```

Листинг 3.16 *NAudioAdapter* класа

Уколико се проузрокује *bug*, или је неопходно из неког другог разлога заменити класу или библиотеку која се тренутно користи у музичком плејеру, потребно је само заменити адаптер класу. Пример пребацивања са једне адаптер класе на другу, дат је методом *switchToAnotherPlayer()*, чија се имплементација може видети на листингу 3.17

```

public void switchToAnotherPlayer()
{
    if (this.myMediaPlayer.GetType() == typeof(MediaPlayerAdapter))
    {
        this.myMediaPlayer = new NAudioAdapter();
        this.isWorking = true;
        Console.WriteLine("Switching to NAudio...");
    }
    else
    {
        this.myMediaPlayer = new MediaPlayerAdapter();
        this.isWorking = true;
        Console.WriteLine("Switching to MediaPlayer...");
    }
}

```

Листинг 3.17 *Метода switchToAnotherPlayer()*

Програмски код који користи библиотеку се не мења. Ово је могуће јер се све адаптер класе крију иза јединственог интерфејса. Уколико је потребно променити библиотеку, неопходно је само направити нову адаптер класу за њу, и променити референцу на објекат те класе.

3.3 *YouTube* налог

Подсистем за *YouTube* налог представља реалистичну симулацију *YouTube* налога и његових функционалности. Корисник преко овог подсистема може креирати *YouTube* налог, креирати плејлисте, додавати видео клипове директно на *YouTube* налог или у плејлисту, претплатити се или отказати претплату на други *YouTube* налог, вршити преглед видео клипова и плејлисте.

3.3.1 *Strategy*

Потребно је имплементирати функционалност разликовања типова *YouTube* налога. *YouTube* налог може бити основни или премијум. У зависности од типа, разликује се начин репродукције видео садржаја (репродукција са/без реклама).

3.3.1.1 Опис проблема

Неопходно је дефинисати класу *YouTubeChannel* која представља *YouTube* налог. Поред осталих функционалности, *YouTubeChannel* има две апстрактне методе. Методу за репродукцију видео садржаја, *playVideo()*, и методу за креирање плејлисте, *createPlaylist()*. Функционалност методе *createPlaylist()* је да на основу типа *YouTube* налога, креира одговарајући тип плејлисте. Класа *BasicYouTubeChannel* представља основни тип налога, где се пре репродукције видео садржаја врши репродукција одговарајуће рекламе. *BasicYouTubeChannel* наслеђује класу *YouTubeChannel* и пружа имплементацију њених апстрактних метода (листинг 3.18).

```

class BasicYouTubeChannel : YouTubeChannel
{
    public BasicYouTubeChannel(String name) : base(name) { }

    public override void playVideo()
    {
        Console.Write("Enter video title: ");
        String videoTitle = Console.ReadLine();
        try
        {
            YouTubeVideo video = this.Videos[videoTitle];
            YouTubeApp.getRandomAdvertisement().play();
            video.play();
        }
        catch (KeyNotFoundException)
        {
            Console.WriteLine("This channel does not have video with given title!");
        }
    }

    public override void createPlaylist()
    {
        Console.Write("Enter playlist name: ");
        String playlistName = Console.ReadLine();
        Playlist playlist = new BasicPlaylist(playlistName);
        this.playlists.Add(playlistName, playlist);
    }
}

```

Листинг 3.18 Класа *BasicYouTubeChannel*

Класа *PremiumYouTubeChannel* представља *YouTube* налог са претплатом, где не долази до репродукције рекламе пре репродукције видео садржаја (листинг 3.19).

```

class PremiumYouTubeChannel : YouTubeChannel
{
    public PremiumYouTubeChannel(String name) : base(name) { }

    public override void playVideo()
    {
        Console.Write("Enter video title: ");
        String videoTitle = Console.ReadLine();

        try
        {
            YouTubeVideo video = this.Videos[videoTitle];
            video.play();
        }
        catch (KeyNotFoundException)
        {
            Console.WriteLine("This channel does not have video with given title!");
        }
    }

    public override void createPlaylist()
    {
        Console.Write("Enter playlist name: ");
        String playlistName = Console.ReadLine();
        Playlist playlist = new PremiumPlaylist(playlistName);
        this.playlists.Add(playlistName, playlist);
    }
}

```

Листинг 3.19 Класа *PremiumYouTubeChannel*

Потребно је и имплементирати разликовање плејлиста. Апстрактна класа *Playlist* представља плејлисту, која поред осталих метода, има апстрактну методу *play()*, која представља репродукцију видео клипова који се у плејлисти налазе. Класа *BasicPlaylist* наслеђује класу *Playlist* и

представља плејлисту са рекламама. Пружа имплементацију методе *play()* тако што пре репродукције сваког видео записа из плејлисте, врши репродукцију одређене рекламе (листинг 3.20).

```
class BasicPlaylist : Playlist
{
    public BasicPlaylist(String name) : base(name) { }

    public override void play()
    {
        foreach (KeyValuePair<String, YouTubeVideo> video in videos)
        {
            YouTubeApp.getRandomAdvertisement().play();
            video.Value.play();
        }
    }
}
```

Листинг 3.20 Класа *BasicPlaylist*

Класа *PremiumPlaylist* наслеђује класу *Playlist* и представља плејлисту без реклама. Пружа имплементацију методе *play()* тако што врши репродукцију сваког видео записа из плејлисте, без реклама (листинг 3.21).

```
class PremiumPlaylist : Playlist
{
    public PremiumPlaylist(String name) : base(name) { }

    public override void play()
    {
        foreach (KeyValuePair<String, YouTubeVideo> video in videos)
        {
            video.Value.play();
        }
    }
}
```

Листинг 3.21 Класа *PremiumPlaylist*

Овим приступом, за сваки тип *YouTube* налога, постоји одговарајући тип плејлисте, који је одређен у фази компајлирања. Проблем се јавља ако је потребно променити тип налога у реалном времену. Промена типа није могућа јер при креирању налога долази до чврстог везивања за конкретну класу наследницу која представља тип. Проблем настаје и ако се уведе нови тип репродукције видео садржаја, јер би било неопходно увести нове типове налога и плејлисте за уведени тип репродукције.

3.3.1.2 Решење

Решење претходног проблема је дато *Strategy* шаблоном. Према шаблону, потребно је жељено понашање представити помоћу интерфејса. Тако је понашање које представља репродукцију видео садржаја, представљено интерфејсом *IPlayAlghoritm*, који декларише методу *play()* која прихвата *YouTubeVideo* који је потребно репродуковати. Сваки различити тип репродукције видео садржаја, потребно је представити посебном класом, која имплементира овај

интерфејс и пружа одговарајућу имплементацију методе *play()*. На листингу 3.22 је приказана класа која представља начин репродукције видео садржаја без реклама, *NoAdvertisingPlaying*.

```
class NoAdvertisingPlaying : IPlayAlghoritm
{
    public void play(YouTubeVideo video)
    {
        video.play();
    }
}
```

Листинг 3.22 Класа *NoAdvertisingPlaying*

На листингу 3.23 је приказана класа која представља репродукцију видео садржаја са рекламама, *AdvertisingPlaying*.

```
class AdvertisingPlaying : IPlayAlghoritm
{
    public void play(YouTubeVideo video)
    {
        YouTubeApp.getInstance().getRandomAdvertisement().play();
        video.play();
    }
}
```

Листинг 3.23 Класа *AdvertisingPlaying*

На овај начин је избегнуто постојање посебне класе наследнице *YouTube* налога за сваки различити тип репродукције видео записа. Потребно је само да класа *YouTubeChannel* има референцу на неки тип репродукције видео записа, који имплементира интерфејс *IPlayAlghoritm* (листинг 3.24).

```
class YouTubeChannel : IObservable, IObservable, ISubscribe
{
    private IPlayAlghoritm playAlghoritm;
```

Листинг 3.24 Класа *YouTubeChannel*

Репродукција видео садржаја се сада своди на препуштање репродукције видео садржаја пољу *playAlghoritm*, који сада има улогу и разликовања типова налога (листинг 3.25).

```
public void playVideo()
{
    Console.WriteLine("Enter video title: ");
    String videoTitle = Console.ReadLine();

    try
    {
        YouTubeVideo video = this.videos[videoTitle];
        this.playAlghoritm.play(video);
    }
    catch (KeyNotFoundException)
    {
        Console.WriteLine("This channel does not have video with given title!");
    }
}
```

Листинг 3.25 Препуштање репродукције видео записа

На исти начин се реализује и разликовање типова плејлисти (листинг 3.26).

```
class Playlist
{
    private String name;
    private Dictionary<String, YouTubeVideo> videos;
    private IPlayAlghoritm playAlghoritm;

    public Playlist(String name, IPlayAlghoritm playAlghoritm){...}

    public void addVideo(YouTubeVideo video){...}

    public void removeVideo(){...}

    public void display(){...}

    public void play()
    {
        foreach (KeyValuePair<String,YouTubeVideo> video in videos)
        {
            playAlghoritm.play(video.Value);
        }
    }
}
```

Листинг 3.26 Класа *Playlist*

Оваквим приступом, не долази до чврстог везивања између класе *YouTubeChannel* и имплементације методе за репродукцију видео садржаја, на основу типа *YouTube* налога. У сваком тренутку, уколико дође до додавања новог типа репродукције видео садржаја, потребно је само направити нову класу која имплементира интерфејс *IPlayAlghoritm*. Предност оваквог приступа је и могућа динамичка промена типа налога, односно алгорита репродукције видео садржаја и плејлисти (листинг 3.27).

```
public void switchTypeOfChannel()
{
    if (this.PlayAlghoritm.GetType() == typeof(NoAdvertisingPlaying))
    {
        this.PlayAlghoritm = new AdvertisingPlaying();
        foreach (KeyValuePair<String, Playlist> playlist in this.Playlists)
        {
            playlist.Value.PlayAlghoritm = new AdvertisingPlaying();
        }

        Console.WriteLine("YouTube channel" + this.name + " has switched from no  
advertising, to advertising type");
    }
    else
    {
        this.PlayAlghoritm = new NoAdvertisingPlaying();
        foreach (KeyValuePair<String, Playlist> playlist in this.Playlists)
        {
            playlist.Value.PlayAlghoritm = new NoAdvertisingPlaying();
        }

        Console.WriteLine("YouTube channel" + this.name + " has switched from advertising,  
to no advertising type");
    }
}
```

Листинг 3.27 Динамичка промена типа репродукције видео садржаја

3.3.2 Observer

Потребно је имплементирати функционалност претплате и отказивања претплате на *YouTube* налог. Уколико *YouTube* налог објави видео садржај, сви *YouTube* налози који су претплаћени на тај налог бивају обавештени о објављеном видео садржају.

3.3.2.1 Опис проблема

Сваки *YouTube* налог има своје претплатнике (*subscriber-e*), односно *YouTube* налоге које је потребно обавестити када се објави нови видео садржај. Претплаћени налози су представљени пољем *youtubeSubscribers*. Исти *YouTube* налог може такође бити претплаћен на друге *YouTube* налоге. *YouTube* налози на које је тренутан налог претплаћен, представљени су пољем *subscribedTo* (листинг 3.28).

```
abstract class YouTubeChannel : ISubscribe
{
    private Dictionary<String, YouTubeChannel> subscribedTo;
    private List<YouTubeChannel> youtubeSubscribers;
```

Листинг 3.28 Класа *YouTubeChannel*

Оваквим приступом, доступна је информација о *YouTube* налозима који су претплаћени на тренутан налог, као и о налозима на које је тренутан *YouTube* налог претплаћен. Потребно је имплементирати логику претплаћивања и отказивања претплате на *YouTube* налог. На листингу 3.29 је дата имплементација тих функционалности. Метода *subscribe()* представља претплаћивање тренутног налога на прослеђени налог. У листу претплатника прослеђеног *YouTube* налога додаје се тренутан налог, а у листу *YouTube* налога на који је тренутан налог претплаћен, додаје се прослеђен *YouTube* налог. Метода *unsubscribe()* представља функционалност отказивања претплате на прослеђен налог, и имплементирана је истим принципом као и метода *subscribe()*.

```
public void subscribe(YouTubeChannel toSubscribe)
{
    if (this == toSubscribe)
    {
        Console.WriteLine("You can not subscribe to your own channel!");
        return;
    }
    try
    {
        this.subscribedTo.Add(toSubscribe.Name, toSubscribe);
        toSubscribe.youtubeSubscribers.Add(this);
    }
    catch (ArgumentException)
    {
        Console.WriteLine("You are already subscribed to this channel!");
    }
}

public void unsubscribe(YouTubeChannel toUnsubscribe)
{
    this.subscribedTo.Remove(toUnsubscribe.Name);
    toUnsubscribe.youtubeSubscribers.Remove(this);
}
```

Листинг 3.29 *Memode subscribe() i unsubscribe()*

Потребно је имплементирати логику креирања и објављивања видео садржаја. Након попуњавања података потребних за креирање видео садржаја, долази до његовог креирања и додавања у листу видео садржаја *YouTube* налога (листинг 3.30). Логика обавештавања о управо објављеном видео садржају, је да се пролази кроз листу свих претплатника на *YouTube* налог, *youtubeSubscribers*, и сваком посебно шаље нотификација. Проблем оваквог приступа је да класа *YouTube* налог мора да има информацију о класама свих претплатника, као и њиховим методама преко којих ће да их обавести о новом садржају. Проблем настаје и уколико је потребно увести нове типове који могу да се претплате на *YouTube* налог. Потребно би било посебно водити рачуна и о њиховим типовима, као и обавестити их о објављеном видео садржају на начин како они то очекују. Долази до чврстог везивања класе која обавештава и оних класа које треба да буду обавештене.

```
public void createNewVideo()
{
    Console.WriteLine("Enter title: ");
    String title = Console.ReadLine();
    Console.WriteLine("Enter description: ");
    String description = Console.ReadLine();
    String link = this.createNewLink();
    Console.WriteLine("Enter length in seconds: ");
    int length = 0;

    try
    {
        length = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException)
    {
        Console.WriteLine("Input error!");
        return;
    }

    Console.WriteLine("Enter local path: ");
    String path = Console.ReadLine();
    YouTubeVideo newVideo = new YouTubeVideo(title, description, link, length, path);
    this.videos.Add(title, newVideo);
    Console.WriteLine("Channel " + this.name + " created video " + newVideo.Title + " ( "
        + newVideo.Length + " ) ");
    Notification notification = new Notification(newVideo.Title, newVideo.Link, this.name);

    foreach (YouTubeChannel channel in this.youtubeSubscribers)
    {
        channel.notifications.Add(notification);
        Console.WriteLine("Notification about recently created video sent from: "
            + this.Name + " to: " + channel.Name);
    }
}
```

Листинг 3.30 *Memoda createNewVideo()***3.3.2.2 Решење**

Решење претходног проблема је дато *Observer* шаблоном. Потребно је разликовати класе у којој се прате промене (*Observable*), у односу на оне које треба да буде обавештене када се унутрашње стање у праћеној класи промени (*Observer*). Промена унутрашњег стања је у овом случају креирање, тј. објављивање новог видео садржаја. Класа која

треба да буде обавештена о промени имплементира интерфејс *IObserver* и пружа имплементацију методе *update()* (листинг 3.31). Метода *update()* се позива сваки пут када се стање надгледане класе промени. Класа која треба да буде надгледана имплементира интерфејс *IObservable* који декларише методе за додавање *observer*-а, уклањање *observer*-а и обавештавање свих *observer*-а (листинг 3.32).

```
interface IObserver
{
    void update(Notification notification);
}
```

Листинг 3.31 Интерфејс *IObserver*

```
interface IObservable
{
    void registerObserver(IObserver observer);
    void removeObserver(IObserver observer);
    void notifyObservers(Notification notification);
}
```

Листинг 3.32 Интерфејс *IObservable*

YouTube налог је и *observer* и *observable*, јер се његове промене (објављивање видео садржаја) прате зарад обавештавања претплаћених *YouTube* налога. Такође, он и прати промене свих налога на које је претплаћен. Из тог разлога, класа *YouTubeChannel* имплементира оба интерфејса, и као поље има листу *observer*-а као претплатнике (листинг 3.33).

```
private List<IObserver> subscribers;
```

Листинг 3.33 Претплатници на *YouTube* налог

На овај начин је избегнуто уско везивање са конкретним класама оних које је потребно обавестити о променама. Свака класа која имплементира *IObserver* интерфејс може бити претплаћена на *YouTube* налог. С обзиром да *YouTubeChannel* имплементира *IObserver* интерфејс, потребно је да пружи логику методе *update()* (листинг 3.34).

```
public void update(Notification notification)
{
    this.notifications.Add(notification);
    Console.WriteLine(this.name + ": " + notification.ToString());
}
```

Листинг 3.34 Метода *update()*

С обзиром да *YouTubeChannel* имплементира и *IObservable* интерфејс, потребно је да пружи имплементацију и метода за додавање, уклањање и обавештавање претплатника када се унутрашње стање промени (листинг 3.35).

```

public void registerObserver(IObserver observer)
{
    this.subscribers.Add(observer);
}

public void removeObserver(IObserver observer)
{
    this.subscribers.Remove(observer);
}

public void notifyObservers(Notification notification)
{
    foreach (IObserver subscriber in subscribers)
    {
        subscriber.update(notification);
    }
}

```

Листинг 3.35 *Имплементација метода интерфејса IObserver*

На листингу 3.36 је дата логика метода за претплаћивање и отказивање претплате, где се сад не додају/избацују конкретни *YouTube* налози као претплатници, већ класе које имплементирају *IObserver* интерфејс.

```

public void subscribe(YouTubeChannel toSubscribe)
{
    if (this == toSubscribe)
    {
        Console.WriteLine("You can not subscribe to your own channel!");
        return;
    }

    try
    {
        this.subscribedTo.Add(toSubscribe.Name, toSubscribe);
        toSubscribe.registerObserver(this);
    }
    catch (ArgumentException)
    {
        Console.WriteLine("You are already subscribed to this channel!");
    }
}

public void unsubscribe(YouTubeChannel toUnsubscribe)
{
    this.subscribedTo.Remove(toUnsubscribe.Name);
    toUnsubscribe.removeObserver(this);
}

```

Листинг 3.36 *Имплементација метода subscribe/unsubscribe*

На листингу 3.37 је дата логика креирања и објављивања видео садржаја. Након креирања и додавања у листу видео садржаја *YouTube* налога, долази до обавештавања свих претплатника преко методе *notifyObservers()*. Као што је претходно приказано на листингу 3.35, метода *notifyObservers()* позива методу *update()* за сваког претплатника, што је могуће јер се сви крију иза интерфејса *IObserver*. Нема уског везивања *YouTube* налога за конкретне класе претплатника, као и мењања програмског кода. Уколико је потребно додати објекат неке нове класе као претплатник, потребно је само да та класа имплементира *IObserver* интерфејс.

```

public void createNewVideo()
{
    Console.Write("Enter title: ");
    String title = Console.ReadLine();
    Console.Write("Enter description: ");
    String description = Console.ReadLine();
    String link = this.createNewLink();
    Console.Write("Enter length in seconds: ");
    int length = 0;

    try
    {
        length = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException)
    {
        Console.WriteLine("Input error!");
        return;
    }

    Console.Write("Enter local path: ");
    String path = Console.ReadLine();

    YouTubeVideo newVideo = new YouTubeVideo(title, description, link, length, path);
    this.videos.Add(title, newVideo);

    Console.WriteLine("Channel " + this.name + " created video " + newVideo.Title
        + " ( " + newVideo.Length + " ) ");

    this.notifyObservers(new Notification(newVideo.Title, newVideo.Link, this.name));
}

```

Листинг 3.37 Метода *createNewVideo()*

3.4 Google налог

Подсистем за *Google* налог представља реалистичну симулацију *Google* налога, где је могуће слање *email*-а са једног *Google* налога на други. Такође је додата функционалност за потребе пројекта, где је могуће да се *Google* налог претплати или откаже претплату на *YouTube* налог.

3.4.1 Observer

Потребно је имплементирати функционалност претплате и отказивања претплате *Google* налога на *YouTube* налог.

3.4.1.1 Опис проблема

На листингу 3.38 је дата класа *GoogleAccount* која представља *Google* налог.

```

class GoogleAccount : ISubscribe
{
    private Dictionary<String, YouTubeChannel> subscribedTo;
}

```

Листинг 3.38 Класа *GoogleAccount*

Неопходно је да *YouTube* налог има знање о *Google* налозима који су претплаћени на њега. Класа *YouTubeChannel*, поред листе *YouTube*

претплатника, има и листу *Google* налога који су претплаћени на њега (листинг 3.39).

```
abstract class YouTubeChannel : ISubscribe
{
    private List<YouTubeChannel> youTubeSubscribers;
    private List<GoogleAccount> googleAccountSubscribers;
```

Листинг 3.39 Класа *YouTubeChannel*

Потребно је имплементирати методу *subscribe()* класе *GoogleAccount*, чија је функционалност претплаћивање *Google* налога на прослеђени *YouTube* налог. Потребно је имплементирати и методу *unsubscribe()* која врши отказивање претплате на *YouTube* налог (листинг 3.40).

```
public void subscribe(YouTubeChannel toSubscribe)
{
    try
    {
        this.subscribedTo.Add(toSubscribe.Name, toSubscribe);
        toSubscribe.GoogleAccountSubscribers.Add(this);
    }
    catch (ArgumentException)
    {
        Console.WriteLine("You are already subscribed to this channel!");
    }
}

public void unsubscribe(YouTubeChannel toUnsubscribe)
{
    this.subscribedTo.Remove(toUnsubscribe.Name);
    toUnsubscribe.GoogleAccountSubscribers.Remove(this);
}
```

Листинг 3.40 Методе *subscribe()* и *unsubscribe()* класе *GoogleAccount*

Неопходно је проширити имплементацију методе *createNewVideo()* класе *YouTubeChannel*. Када *YouTube* налог објави видео садржај, потребно је такође проћи и кроз листу претплаћених *Google* налога како би се извршило њихово обавештавање о објављеном видео садржају (листинг 3.41).

```
foreach (YouTubeChannel channel in this.youTubeSubscribers)
{
    channel.notifications.Add(notification);
    Console.WriteLine("Notification about recently created video sent from: "
        + this.Name + " to: " + channel.Name);
}

String subject = "Channel " + notification.ChannelName + " has uploaded a new video!";
String text = subject + Environment.NewLine;
text += "Go to " + notification.Link + " to check it out!";

foreach (GoogleAccount account in this.googleAccountSubscribers)
{
    MyMail newMail = new MyMail("www.youtube.com",
        account.GMailAddress, Subject, text);
    account.Inbox.Add(newMail);
    Console.WriteLine(this.Name + ": Email about created video sent to "
        + account.GMailAddress);
}
```

Листинг 3.41 Метода *createNewVideo()* класе *YouTubeChannel*

Лоша страна оваквог приступа је да се за сваки нови тип претплатника, мора креирати нова листа као поље класе *YouTubeChannel*. Такође долази до чврстог везивања за конкретне класе претплатника.

3.4.1.2 Решење

Решење претходног проблема је дато *Observer* шаблоном. Неопходно је да класа *GoogleAccount* имплементира *Observer* интерфејс и пружи имплементацију *update()* методе (листинг 3.42).

```
public void update(Notification notification)
{
    String subject = "Channel " + notification.ChannelName + " has uploaded a new video!";
    String text = subject + Environment.NewLine;
    text += "Go to " + notification.Link + " to check it out!";
    MyMail newMail = new MyMail("www.youtube.com", this.gMailAddress, subject, text);
    Console.WriteLine(this.username + ", you have recieved a new mail: " + subject);
    this.inbox.Add(newMail);
}
```

Листинг 3.42 Метода *update()* класе *GoogleAccount*

Уколико дође до потребе да се *Google* налог претплати на *YouTube* налог, неопходно је да само се *Google* налог региструје као *observer* *YouTube* налога (листинг 3.43). Давајући исти интерфејс свакој класи која жели да се претплати на *YouTube* налог (*IObserver*), могуће је регистровати објекат било које класе као претплатник. Довољно је само да класа тог објекта имплементира *IObserver* интерфејс и пружи имплементацију методе *update()* преко које ће бити обавештена о промени. Када *YouTube* налог објави видео садржај, позивом методе *notifyObservers()*, обавештавају се сви регистровани претплатници на тај *YouTube* налог.

```
public void subscribe(YouTubeChannel toSubscribe)
{
    try
    {
        this.subscribedTo.Add(toSubscribe.Name, toSubscribe);
        toSubscribe.registerObserver(this);
    }
    catch (ArgumentException)
    {
        Console.WriteLine("You are already subscribed to this channel!");
    }
}

public void unsubscribe(YouTubeChannel toUnsubscribe)
{
    this.subscribedTo.Remove(toUnsubscribe.Name);
    toUnsubscribe.removeObserver(this);
}
```

Листинг 3.43 Методе *subscribe()* и *unsubscribe()* класе *GoogleAccount*

3.5 Пицерија

Подсистем за пицерију представља симулацију поручивања хране из пицерије. Процес проучивања се разликује у зависности да ли је у питању поруџба на лицу места или *online*.

3.5.1 *Template method*

Потребно је имплементирати разликовање процеса поручивања из пицерије, у зависности да ли је у питању *online* поруџбина или поруџбина са лица места.

3.5.1.1 Опис проблема

Потребно је направити две класе од којих ће свака представљати различити тип поруџбине. *Online* поруџбина је дата класом *OnlinePizzaOrder*, док је поруџбина са лица места дата класом *InPlacePizzaOrder*. Алгоритам поручивања је исти у оба случаја. Корисник прво врши одабир пице, затим се пица пече, након чега се додају прилози. Затим, пица се послужује на сто, ако је у питању поруџбина на лицу места, или се служи кућном доставом, ако је у питању *online* поруџбина. Последњи корак је наплата рачуна, која се врши скидањем новца са картице, ако је *online* поруџбина, или се наплаћује директно, ако је у питању поруџбина на лицу места. Свеукупна логика ових процеса је дата у методи *startOrderingProcess()*, у класама *OnlinePizzaOrder* и *InPlacePizzaOrder*. Потребно је приметити да је имплементација симулације печења пице, као и додавања прилога на пицу, иста за оба типа поруџбине (листинг 3.44).

```

Console.Write("Choose oven: ");
Console.WriteLine("1. Regular");
Console.WriteLine("2. Pizza");
Console.Write("Option: ");
String oven = Console.ReadLine();
Console.WriteLine();
Console.ForegroundColor = ConsoleColor.DarkCyan;
Console.WriteLine("Chosen oven: " + option);
Console.WriteLine();

if (oven == "1")
{
    this.chosenPizza.simulateBakingInRegularOven();
}
else if (oven == "2")
{
    this.chosenPizza.simulateBakingInPizzaOven();
}
else
{
    Console.WriteLine("Bad input! Please try again!");
    return;
}

Console.WriteLine();
Console.ResetColor();
Console.WriteLine("Toppings: ");
Console.WriteLine("1. Ketchup");
Console.WriteLine("2. Mayonaise");
Console.WriteLine("3. Olives");
Console.ResetColor();
Console.Write("Choose toppings (1,3 for example, or press 0 if you dont want any): ");

String toppings = Console.ReadLine();
String[] toppingArray = toppings.Split(',');
Console.ForegroundColor = ConsoleColor.DarkCyan;
Console.WriteLine();

if (toppingArray.Contains("1"))
{
    Console.WriteLine("Adding ketchup...");
    Thread.Sleep(1000);
    this.chosenPizza.addKetchup();
}
if (toppingArray.Contains("2"))
{
    Console.WriteLine("Adding mayonaise...");
    Thread.Sleep(1000);
    this.chosenPizza.addMayonaise();
}
if (toppingArray.Contains("3"))
{
    Console.WriteLine("Adding olives...");
    Thread.Sleep(1000);
    this.chosenPizza.addOlives();
}
if (toppings != "0")
{
    Console.WriteLine("Toppings added!");
}
Console.WriteLine();

```

Листинг 3.44 Део методе *startOrderingProcess()* који је исти за обе класе

Разлика у имплементацији ове методе у класама *OnlinePizzaOrder* и *InPlacePizzaOrder* лежи у процесу одабира пице, послуживања пице, као и наплаћивања рачуна. На листингу 3.45 је дата имплементација процеса послуживања и наплаћивања за класу *InPlacePizzaOrder*, док је на листингу 3.46 дата имплементација ових процеса за класу *OnlinePizzaOrder*.

```

Console.WriteLine("Serving pizza to the table...");
Thread.Sleep(2000);
Console.WriteLine("Pizza is served!");
Console.WriteLine();

this.billHandler.chargeBill(chosenPizza);
Thread.Sleep(1500);
Console.ForegroundColor = ConsoleColor.DarkCyan;
Console.WriteLine();
Console.WriteLine("Pizza is charged!");
Console.ResetColor();

```

Листинг 3.45 Део методе *startOrderingProcess()* класе *InPlacePizzaOrder* који представља послуживање пице и наплату рачуна

```

Console.WriteLine("Getting pizza to pizza guy who will deliver pizza to you...");
Thread.Sleep(500);
Console.WriteLine("Pizza guy is on the way...");
Thread.Sleep(5000);
Console.WriteLine("Pizza guy is here!");
Console.WriteLine();

Console.WriteLine(this.chosenPizza.getName()
    + " is charged online from your credit card with total sum of "
    + this.chosenPizza.cost() + "$");
Console.ResetColor();
Console.WriteLine();

```

Листинг 3.46 Део методе *startOrderingProcess()* класе *OnlinePizzaOrder*, који представља послуживање пице и наплату рачуна

С обзиром да долази до дуплирања дела кода, потребно је приметити да је претходно решење веома нефлексибилно. Долази до нарушавања *DRY* принципа[5]. Сваким наредним додавањем нове врсте поруџбине, дошло би поново до дуплирања кода.

3.5.1.2 Решење

Решење претходног проблема је дато *Template Method* шаблоном. Процес поручивања је потребно издвојити у посебну апстрактну класу, под називом *AbstractPizzaOrder*. Кораци процеса поручивања је неопходно издвојити као посебне методе ове класе. Кораци процеса поручивања су одабир пице - метода *choosePizza()*, печење пице - метода *bakePizza()*, додавање прилога на пицу - метода *addToppings()*, послуживање пице - метода *servePizza()*, као и наплата рачуна - метода *chargeBill()*. У класи *AbstractPizzaOrder* је потребно да се нађе још једна метода, а то је тзв. „*Template metoda*“, у овом случају *startOrderingProcess()*. Метода *startOrderingProcess()* позива специфициране кораке алгоритма у таквом редоследу да дају коначан процес поручивања. С обзиром да су одабир прилога и процес печења пице исти за оба процеса поруџбине, њихову имплементацију могуће је пружити у класи *AbstractPizzaOrder*. Методе оних корака који су различити за сваки тип поруџбина, потребно је прогласити апстрактним, и препустити класама наследницама да пруже њихову имплементацију (листинг 3.47).


```

abstract class AbstractPizzaOrder
{
    protected AbstractPizza chosenPizza;
    private IOven pizzaOven;
    private IOven regularOven;

    public AbstractPizzaOrder() {...}

    public abstract AbstractPizza choosePizza();

    void addToppings(){...}

    bool bakePizza(){...}

    public abstract void servePizza();
    public abstract void chargeBill();

    public void startOrderingProcess()
    {
        this.chosenPizza = choosePizza();

        if (this.chosenPizza == null)
            return;

        bool hasNoError = bakePizza();

        if (!hasNoError)
            return;

        addToppings();
        servePizza();
        chargeBill();
    }
}

```

Листинг 3.47 Класа *AbstractPizzaOrder*

Сваки посебан тип поруџбине наслеђује класу *AbstractPizzaOrder*, и пружа имплементацију метода које су специфичне за конкретан тип поруџбине. На листингу 3.48 је дата класа *InPlacePizzaOrder* са имплементацијом потребних метода, док је на листингу 3.49 приказана класа *OnlinePizzaOrder*.

```

class InPlacePizzaOrder : AbstractPizzaOrder
{
    private AbstractHandler billHandler;

    public InPlacePizzaOrder(AbstractHandler billHandler){...}

    public override AbstractPizza choosePizza(){...}

    public override void servePizza()
    {
        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.WriteLine("Serving pizza to the table...");
        Thread.Sleep(2000);
        Console.WriteLine("Pizza is served!");
        Console.WriteLine();
    }

    public override void chargeBill()
    {
        Console.WriteLine("Waiter is charging "
            + this.chosenPizza.ToString() + " with total sum of " +
            this.chosenPizza.cost() + "RSD...");
        Console.ResetColor();
        Console.Write("Amount of money you give: ");
        int waiterRecieved = 0;

        try
        {
            waiterRecieved = Convert.ToInt32(Console.ReadLine());
        } catch (FormatException)
        {
            Console.WriteLine("Bad input! Please try again!");
            return;
        }

        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.Write("Your change in bills: ");
        this.billHandler.processRequest(waiterRecieved - this.chosenPizza.cost());
        Thread.Sleep(1500);
        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.WriteLine();
        Console.WriteLine("Pizza is charged!");
        Console.ResetColor();
    }
}

```

Листинг 3.48 Класа *InPlacePizzaStore*

```

class OnlinePizzaOrder : AbstractPizzaOrder
{
    public OnlinePizzaOrder(){ }

    public override AbstractPizza choosePizza(){...}

    public override void servePizza()
    {
        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.WriteLine("Getting pizza to pizza guy who will
            deliver pizza to you...");
        Thread.Sleep(500);
        Console.WriteLine("Pizza guy is on the way...");
        Thread.Sleep(5000);
        Console.WriteLine("Pizza guy is here!");
        Console.WriteLine();
    }

    public override void chargeBill()
    {
        Console.WriteLine(this.chosenPizza.ToString() + " is
            charged online from your credit card with total sum
            of " + this.chosenPizza.cost() + "RSD");
        Console.ResetColor();
        Console.WriteLine();
    }
}

```

Листинг 3.49 Класа *OnlinePlacePizzaStore*

Template методу, *startOrderingProcess()*, потребно је прогласити за *final* (не може се мењати у класама наследницама). Користећи *template* методу која позива све кораке, небитно да ли је имплементација корака дата у родитељској или класи наследници, долази до формирања коначног процеса поруџбине, без дуплирања кода.

3.5.2 Chain of responsibility

Потребно је имплементирати функционалност класе *InPlacePizzaStore*, где се омогућује враћање кусура муштерији. Такође, потребно је имплементирати функционалност за додавање новца у касу пицерије.

3.5.2.1 Опис проблема

Класа за враћање кусура - *BillHandler*, мора разликовати новчанице у зависности од њихових апоена, као и имати информацију о количину новчаница за сваки апоен. Метода *chargeBill()*, задужена за враћање кусура, враћа кусур редоследом од највећег до најмањег апоена новчанице (листинг 3.50).

```

class BillHandler
{
    private int numOfThousandBills;
    private int numOffiveHundredBills;
    private int numOfHundredBills;
    private int numOffiftyBills;
    private int numOfTenBills;

    public BillHandler(int num1000, int num500, int num100, int num50, int num10)
    {
        this.numOfThousandBills = num1000;
        this.numOffiveHundredBills = num500;
        this.numOfHundredBills = num100;
        this.numOffiftyBills = num50;
        this.numOfTenBills = num10;
    }

    public void chargeBill(AbstractPizza chosenPizza)
    {
        Console.WriteLine("Waiter is charging " + chosenPizza.getName()
            + " with total sum of " + chosenPizza.cost() + "RSD...");

        Console.ResetColor();
        Console.Write("Amount of money you give: ");

        int waiterRecieved = 0;

        try
        {
            waiterRecieved = Convert.ToInt32(Console.ReadLine());
        }
        catch (FormatException)
        {
            Console.WriteLine("Bad input! Please try again!");
            return;
        }

        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.Write("Your change in bills: ");

        int change = waiterRecieved - chosenPizza.cost();
        while (change / 1000 >= 1 && this.numOfThousandBills > 0)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write("1000 ");
            change -= 1000;
            this.numOfHundredBills--;
        }
        while (change / 500 >= 1 && this.numOffiveHundredBills > 0)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write("500 ");
            change -= 500;
            this.numOffiveHundredBills--;
        }
        while (change / 100 >= 1 && this.numOfHundredBills > 0)
        {
            Console.ForegroundColor = ConsoleColor.Blue;
            Console.Write("100 ");
            change -= 100;
            this.numOfHundredBills--;
        }
        while (change / 50 >= 1 && this.numOffiftyBills > 0)
        {
            Console.ForegroundColor = ConsoleColor.Magenta;
            Console.Write("50 ");
            change -= 50;
            this.numOffiftyBills--;
        }
        while (change / 10 >= 1 && this.numOfTenBills > 0)
        {
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("10 ");
            change -= 10;
            this.numOfTenBills--;
        }
    }
}

```

Листинг 3.50 Класа *BillHandler*

Потребно је приметити да се за новчанице различитих апоена, одговарајуће провере врше засебно за сваку новчаницу, и враћа одговарајући кусур. Проблем оваког приступа настаје онда када је потребно у будућности додати новчанице новог апоена, или избацити новчанице постојећег апоена. Програмски код би се морао мењати у таквом сценарију. Исти проблем се јавља и у имплементацији методе *refill()*, која служи за допуњавање касе (листинг 3.51). Уколико би дошло до додавања новчаница новог апоена, неопходно би било модификовати имплементацију ове методе.

```
public void refill()
{
    Console.WriteLine("Enter bill handler to refill (1000,500,100,50 or 10): ");
    String billHandler = Console.ReadLine();
    Console.WriteLine("Enter amount of bills to insert in handler: ");
    int amount = 0;

    try
    {
        amount = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException)
    {
        Console.WriteLine("Bad input! Please try again!");
        return;
    }

    switch (billHandler)
    {
        case "1000":
        {
            this.numOfThousandBills += amount;
            break;
        }
        case "500":
        {
            this.numOfThousandBills += amount;
            break;
        }
        case "100":
        {
            this.numOfThousandBills += amount;
            break;
        }
        case "50":
        {
            this.numOfThousandBills += amount;
            break;
        }
        case "10":
        {
            this.numOfThousandBills += amount;
            break;
        }
        default:
        {
            Console.WriteLine("Refill failed!");
            break;
        }
    }
}
```

Листинг 3.51 *Метода refill() класе BillHandler*

Исти проблем се јавља и код редефинисања методе *toString()*, због везивања за постојеће апоене новчаница (листинг 3.52). У сва три

случаја долази до чврстог везивања за тренутне апоене новчаница, при чему долази до нарушавања *Open-Closed* дизајн принципа[6].

```
public override string ToString()
{
    string retVal = "Amount of 1000 bills in handler is: " + this.numOfThousandBills;
    retVal += System.Environment.NewLine;
    retVal += "Amount of 500 bills in handler is: " + this.numOfFiveHundredBills;
    retVal += System.Environment.NewLine;
    retVal += "Amount of 100 bills in handler is: " + this.numOfHundredBills;
    retVal += System.Environment.NewLine;
    retVal += "Amount of 50 bills in handler is: " + this.numOfFiftyBills;
    retVal += System.Environment.NewLine;
    retVal += "Amount of 10 bills in handler is: " + this.numOfTenBills;
    retVal += System.Environment.NewLine;

    return retVal;
}
```

Листинг 3.52 Метода *toString()* класе *BillHandler*

3.5.2.2 Решење

Решење претходног проблема је дато *Chain of Responsibility* шаблоном. Шаблон предлаже да се направи посебна класа за сваки апоен новчанице, тзв. *handler* класа. Након тога, *handler* класе је потребно повезати у структуру ланца. Да би то било могуће, потребно је да *handler* класе деле заједнички интерфејс, који је дат *AbstractHandler* апстрактном класом. *AbstractHandler* класа има информацију о томе који је следећи *handler* у ланцу. Метода *processRequest()*, која служи за враћање кусура, проверава да ли је могуће обрадити захтев. Захтев је могуће обрадити, ако је потребно вратити једну или више новчаница којима рукује тренутни *handler*, као и ако је потребна количина новчаница доступна у *handler*-у. Ако захтев није могуће обрадити, он се прослеђује следећем *handler*-у у ланцу. Помоћу оваквог приступа (пропуштањем захтева кроз ланац *handler*-а), имплементирани су и методе *refill()* и *toString()* (листинг 3.53).

```

abstract class AbstractHandler
{
    protected int amountOfBillsInHandler;
    protected AbstractHandler nextHandler;
    protected int billValue;
    protected String handlerName;

    public AbstractHandler(int amountOfBillsInHandler, String handlerName)
    {
        this.amountOfBillsInHandler = amountOfBillsInHandler;
        this.handlerName = handlerName;
    }

    public void setNextHandler(AbstractHandler handler)
    {
        this.nextHandler = handler;
    }

    public void processRequest(int amountToPayOff)
    {
        int amountOfBillsToGive = amountToPayOff / billValue;

        while (amountOfBillsToGive > 0 && this.amountOfBillsInHandler != 0)
        {
            this.payOff();
            amountOfBillsToGive--;
            amountToPayOff -= billValue;
        }
        if (amountOfBillsInHandler == 0)
            Console.WriteLine("[ " + this.handlerName + " IS EMPTY! ]");

        if (amountToPayOff > 0 && this.nextHandler != null)
        {
            this.nextHandler.processRequest(amountToPayOff);
        }
        else if (amountToPayOff > 0 && this.nextHandler == null)
        {
            Console.WriteLine("Waiter owe you " + amountToPayOff);
        }
    }

    public void refill(int amount, String handlerName)
    {
        if (this.handlerName == handlerName)
        {
            this.amountOfBillsInHandler += amount;
            Console.WriteLine("Amount of bills in " + this.handlerName + " is: "
                + this.amountOfBillsInHandler);
        }
        else if (this.nextHandler != null)
        {
            this.nextHandler.refill(amount, handlerName);
        }
        else
        {
            Console.WriteLine("Refill failed");
        }
    }

    public override string ToString()
    {
        string retVal = this.handlerName + ", amount of " + this.billValue
            + " bills in handler is: " + this.amountOfBillsInHandler;
        retVal += System.Environment.NewLine;

        if (this.nextHandler != null)
        {
            retVal += nextHandler.ToString();
        }
        return retVal;
    }

    public abstract void payOff();
}

```

Листинг 3.53 Класа *AbstractBillHandler*

Сваки конкретан *handler*, наслеђује класу *AbstractHandler* и дефинише начин на који исплаћује кусур у новчаницама апоена којим рукује (листинг 3.54).

```
class FiveHundredBillHandler : AbstractHandler
{
    public FiveHundredBillHandler(int amountOfBillsInHandler, String handlerName)
        : base(amountOfBillsInHandler, handlerName)
    {
        this.billValue = 500;
    }

    public override void payOff()
    {
        Console.ForegroundColor = ConsoleColor.Green;
        Console.Write(this.billValue + " ");
        this.amountOfBillsInHandler--;
        Console.ResetColor();
    }
}
```

Листинг 3.54 Пример конкретног *handlera* – класа *FiveHundredBillHandler*

Потребно је приметити да су *handler*-и повезани тако да чине реалан редослед враћања новчаница, од новчаница највећег апоена, до новчаница најмањег апоена (листинг 3.55).

```
AbstractHandler h1 = new FiveHundredBillHandler(10, "FiveHunderBillHandler");
AbstractHandler h2 = new HundredBillHandler(10, "HundredBillHandler");
AbstractHandler h3 = new FiftyBillHandler(10, "FiftyBillHandler");
AbstractHandler h4 = new TenBillHandler(10, "TenBillHandler");

h1.setNextHandler(h2);
h2.setNextHandler(h3);
h3.setNextHandler(h4);

this.inPlacePizzaOrder = new InPlacePizzaOrder(h1);
```

Листинг 3.55 Повезивање *handler-a* у структуру ланца

Handler-е је могуће повезати у било ком редоследу, као и у било ком тренутку извезивати их и увезивати у ланац, тј. могућа је динамичка промена структуре ланца (листинг 3.56).

```
public void addThousandBillHandler()
{
    AbstractHandler h1 = new ThousandBillHandler(10, "ThousandBillHandler");
    h1.setNextHandler(this.inPlacePizzaOrder.BillHandler);
    this.inPlacePizzaOrder.BillHandler = h1;
}
```

Листинг 3.56 Динамичко додавање новог *handler-a* у ланац

3.5.3 Decorator

Потребно је имплементирати функционалност додавања прилога на пицу, као и израчунавања укупне цене пице, у зависности од прилога који су додани.

3.5.3.1 Опис проблема

Потребно је у дефинисану класу *AbstractPizza* додати поља која ће означавати присуство сваког прилога који може да се дода на пицу. Такође, потребно је додати апстрактну методу *cost()*, коју ће свака конкретна пица имплементирати тако да враћа укупну цену те пице (листинг 3.57).

```
abstract class AbstractPizza
{
    protected String name = "No name";
    protected int size = 0;
    protected bool hasKetchup = false;
    protected bool hasMayonaise = false;
    protected bool hasOlives = false;

    public abstract int cost();
    public abstract void simulateBakingInPizzaOven();
    public abstract void simulateBakingInRegularOven();

    public String getName()
    {
        return this.name;
    }

    public void addKetchup()
    {
        this.hasKetchup = true;
    }

    public void addMayonaise()
    {
        this.hasMayonaise = true;
    }

    public void addOlives()
    {
        this.hasOlives = true;
    }
}
```

Листинг 3.57 Класа *AbstractPizza*

На листингу 3.58 је представљена једна конкретна врста пице, која пружа имплементацију методе *cost()*. Уз цену пице, на укупну цену се додају и цене оних прилога који су присутни на пици.

```

class PizzaCapricossa : AbstractPizza
{
    public PizzaCapricossa()
    {
        this.name = "Capricossa";
        this.size = 40;
    }

    public override int cost()
    {
        int cost = 750;
        if (this.hasKetchup)
        {
            cost += 30;
        }
        if (this.hasMayonaise)
        {
            cost += 70;
        }
        if (this.hasOlives)
        {
            cost += 90;
        }
        return cost;
    }
}

```

Листинг 3.58 Класа *PizzaCapricossa*

У клијент класи, од корисника се тражи да унесе жељене прилоге. За сваки жељени прилог, одговарајуће поље у изабраној пици се постављана *true*, што означава да је прилог додат на пицу (листинг 3.59).

```

Console.WriteLine("Toppings: ");
Console.WriteLine("1. Ketchup");
Console.WriteLine("2. Mayonaise");
Console.WriteLine("3. Olives");
Console.ResetColor();
Console.Write("Choose toppings (1,3 for example, or press 0 if you dont want any): ");

String toppings = Console.ReadLine();
String[] toppingArray = toppings.Split(',');
Console.ForegroundColor = ConsoleColor.DarkCyan;
Console.WriteLine();

if (toppingArray.Contains("1"))
{
    Console.WriteLine("Adding ketchup...");
    Thread.Sleep(1000);
    this.chosenPizza.addKetchup();
}
if (toppingArray.Contains("2"))
{
    Console.WriteLine("Adding mayonaise...");
    Thread.Sleep(1000);
    this.chosenPizza.addMayonaise();
}
if (toppingArray.Contains("3"))
{
    Console.WriteLine("Adding olives...");
    Thread.Sleep(1000);
    this.chosenPizza.addOlives();
}

```

Листинг 3.59 Додавање прилога на пицу у класи *InPlacePizzaStore*

Проблем настаје онда када дође до потребе за додавањем новог прилога, или избацивањем постојећег. Тада би се за сваку пицу морао модификовати програмски код, што нарушава *Open-Closed* дизајн принцип[6]. Проблем се такође јавља ако прилог промени цену. Тада би се у свакој класи морала мењати цена прилога.

3.5.3.2 Решење

Решење претходног проблема је дато *Decorator* шаблоном. Класа *AbstractPizza* не треба да има поља за означавање присуства прилога, већ је довољно само оставити апстрактну методу *cost()*, коју ће свака конкретна пица имплементирати. Конкретна пица која пружа имплементацију ове методе приказана је на листингу 3.60. Као повратну вредност *cost()* методе, довољно је само вратити цену конкретне пице.

```
class PizzaCapricossa : AbstractPizza
{
    public PizzaCapricossa()
    {
        this.name = "Capricossa";
        this.Size = 40;
    }

    public override int cost()
    {
        return 750;
    }
}
```

Листинг 3.60 Класа *PizzaCapricossa*

Прилози су додатна својства која пица може а и не мора да има. Потребно је направити класу *AbstractToppingDecorator*, која представља декоратор класу коју сваки конкретни прилог мора наследити. Такође, класа *AbstractToppingDecorator* садржи апстрактну методу *getName()*, чија је намена да врати име пице и свих њених прилога, за даље потребе исписа. Сваки конкретан прилог наслеђује *AbstractToppingDecorator* класу. Осим што је наслеђује, садржи је и као поље класе. Због овакве структуре, могуће је међусобно садржавање прилога и пица, јер су све наследнице класе *AbstractPizza*. Конкретан пример прилога дат је на листингу 3.61, где је такође дата имплементација *cost()* методе. Метода *cost()* је имплементирана тако да узима повратну вредност *cost()* методе објекта који садржи као поље, и на то додаје цену прилога представљеног датом класом.

```
class KetchupTopping : AbstractToppingDecorator
{
    private AbstractPizza pizza;

    public KetchupTopping(AbstractPizza pizza)
    {
        this.pizza = pizza;
        this.name = pizza.getName() + " + Ketchup";
        this.Size = pizza.Size;
    }

    public override int cost()
    {
        return 30 + pizza.cost();
    }

    public override String getName()
    {
        return this.name;
    }
}
```

Листинг 3.61 Класа *KetchupTopping*

На листингу 3.62 је приказана клијент класа која применом декоратор шаблона, додаје прилоге на пицу. Потребно је приметити да није битно знати да ли је неки прилог претходно додат или не. Када је неки прилог потребно додати, само је потребно извршити обавијање претходног објекта новим прилогом. Ово је могуће јер се све конкретне пице и сви конкретни прилози крију иза јединственог интерфејса, датог класом *AbstractPizza*.

```
void addToppings()
{
    Console.WriteLine("Toppings: ");
    Console.WriteLine("1. Ketchup");
    Console.WriteLine("2. Mayonaise");
    Console.WriteLine("3. Olives");
    Console.ResetColor();
    Console.Write("Choose toppings (1,3 for example,
        or press 0 if you dont want any): ");

    String toppings = Console.ReadLine();
    String[] toppingArray = toppings.Split(',');
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine();

    if (toppingArray.Contains("1"))
    {
        this.chosenPizza = new KetchupTopping(this.chosenPizza);
        Console.WriteLine("Adding ketchup...");
        Thread.Sleep(1000);
    }
    if (toppingArray.Contains("2"))
    {
        this.chosenPizza = new MayonaiseTopping(this.chosenPizza);
        Console.WriteLine("Adding mayonaise...");
        Thread.Sleep(1000);
    }
    if (toppingArray.Contains("3"))
    {
        this.chosenPizza = new OlivesTopping(this.chosenPizza);
        Console.WriteLine("Adding olives...");
        Thread.Sleep(1000);
    }
}
```

Листинг 3.62 Додавање прилога на пицу у класи *AbstractPizzaStore*

3.5.4 Bridge

Потребно је имплементирати функционалност симулације печења пице.

3.5.4.1 Опис проблема

Неопходно је извршити симулацију печења пице у зависности од пећи. Потребно је разликовати два типа пећи, уобичајена, и пећ посебно прављена за пице. На листингу 3.63 дате су методе класе *AbstractPizza*, које врше симулацију печења пице за специфичне типове пећи.

```
abstract class AbstractPizza
{
    public abstract void simulateBakingInPizzaOven();
    public abstract void simulateBakingInRegularOven();
}
```

Листинг 3.63 Методе за симулацију печења пице

Сваки конкретан тип пице симулира печење у пећи тако што у методама за симулацију, симулира укључивање пећи, које је специфично за сваки тип пећи. Након тога долази и до симулације промене изгледа пице, услед проласка времена од како је пица у пећи. Симулација промена изгледа пице је такође специфична за сваку пицу. На листингу 3.64 је дата симулација паљења пећи специфично прављене за пицу - метода *simulateBakingInPizzaOven()*, док је на листингу 3.65 дат други део ове методе, који представља симулацију промена изгледа пице.

```
public override void simulateBakingInPizzaOven()
{
    Console.WriteLine("Turning on the pizza oven...");
    Console.WriteLine();

    List<ConsoleColor> colors = new List<ConsoleColor>();
    colors.Add(ConsoleColor.DarkGray);
    colors.Add(ConsoleColor.Gray);
    colors.Add(ConsoleColor.Yellow);
    colors.Add(ConsoleColor.DarkYellow);
    colors.Add(ConsoleColor.Red);
    colors.Add(ConsoleColor.DarkRed);

    for (int i = 0; i < 6; i++)
    {
        Console.ForegroundColor = colors[i];
        Console.WriteLine("#####");
        Console.SetCursorPosition(0, Console.CursorTop + 12);
        Console.WriteLine("#####");
        Console.SetCursorPosition(0, Console.CursorTop - 14);
        Thread.Sleep(1500);
    }

    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.SetCursorPosition(0, Console.CursorTop + 15);
    Console.WriteLine("Oven turned on!");
    Console.SetCursorPosition(0, Console.CursorTop - 15);
}
```

Листинг 3.64 Симулација укључивања пећи

```

double radius = 4;
double thickness = 0.4;
Console.ForegroundColor = ConsoleColor.Yellow;
char symbol = '*';
Console.WriteLine();

double rIn = radius - thickness;
double rOut = radius + thickness;

List<ConsoleColor> colors = new List<ConsoleColor>();
colors.Add(ConsoleColor.White);
colors.Add(ConsoleColor.Yellow);
colors.Add(ConsoleColor.DarkYellow);
Console.SetCursorPosition(0, Console.CursorTop + 10);

for (int i = 0; i < 3; i++)
{
    Console.SetCursorPosition(0, Console.CursorTop - 10);

    if (i != 0)
        Thread.Sleep(1000);

    for (double y = -radius; y <= radius; ++y)
    {
        for (double x = -radius; x < rOut; x += 0.5)
        {
            double value = x * x + y * y;
            if (value >= rIn * rIn && value <= rOut * rOut)
            {
                Console.BackgroundColor = ConsoleColor.DarkRed;
                Console.Write(symbol);
                Console.BackgroundColor = ConsoleColor.Black;
            }
            else if (value < rIn * rIn)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.BackgroundColor = colors[i];
                Console.Write('o');
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.BackgroundColor = ConsoleColor.Black;
            }
            else
            {
                Console.Write(' ');
            }
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}

```

Листинг 3.65 Симулација промена изгледа тиче

Клијент класа која позива ове методе, у зависности која пећ је одабрана од стране корисника, дата је на листингу 3.66.

```

Console.Write("Choose oven: ");
Console.WriteLine("1. Regular");
Console.WriteLine("2. Pizza");
Console.Write("Option: ");
String oven = Console.ReadLine();
Console.WriteLine();
Console.ForegroundColor = ConsoleColor.DarkCyan;
Console.WriteLine("Chosen oven: " + option);
Console.WriteLine();

if (oven == "1")
{
    this.chosenPizza.simulateBakingInRegularOven();
}
else if (oven == "2")
{
    this.chosenPizza.simulateBakingInPizzaOven();
}
else
{
    Console.WriteLine("Bad input! Please try again!");
    return;
}

```

Листинг 3.66 Позив метода за печење пице

Симулација печења конкретне пице је иста у уобичајеној пећи, као и у пећи специфично прављену за пице. Симулација укључивања конкретне пећи се такође понавља у методама сваке конкретне пице. Потребно је приметити да долази до дуплирања кода. Додавањем нових врста пица или пећи, долази до још већег дуплирања кода.

3.5.4.2 Решење

Решење претходног проблема је дато *Bridge* шаблоном. С обзиром да је потребно имати различите врсте пица и пећи, потребно је дефинисати интерфејсе за пицу, као и интерфејс за пећ. *AbstractPizza* представља заједнички интерфејс за пице, који већ постоји, тако да га не треба дефинисати поново. Потребно га је само проширити методом која ће симулирати промену изгледа пице током печења (листинг 3.67).

```

abstract class AbstractPizza
{
    public abstract void simulateBaking();
}

```

Листинг 3.67 Метода за симулацију промене изгледа пице

Сваки посебан тип пице, пружа своју имплементацију ове методе (листинг 3.68).

```

public override void simulateBaking()
{
    double radius = 4;
    double thickness = 0.4;
    Console.ForegroundColor = ConsoleColor.Yellow;
    char symbol = '*';

    Console.WriteLine();
    double rIn = radius - thickness;
    double rOut = radius + thickness;

    List<ConsoleColor> colors = new List<ConsoleColor>();
    colors.Add(ConsoleColor.White);
    colors.Add(ConsoleColor.Yellow);
    colors.Add(ConsoleColor.DarkYellow);
    Console.SetCursorPosition(0, Console.CursorTop + 10);
    for (int i = 0; i < 3; i++)
    {
        Console.SetCursorPosition(0, Console.CursorTop - 10);
        if (i != 0)
            Thread.Sleep(1000);

        for (double y = -radius; y <= radius; ++y)
        {
            for (double x = -radius; x < rOut; x += 0.5)
            {
                double value = x * x + y * y;
                if (value >= rIn * rIn && value <= rOut * rOut)
                {
                    Console.BackgroundColor = ConsoleColor.DarkRed;
                    Console.Write(symbol);
                    Console.BackgroundColor = ConsoleColor.Black;
                }
                else if (value < rIn * rIn)
                {
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.BackgroundColor = colors[i];
                    Console.Write('o');
                    Console.ForegroundColor = ConsoleColor.Yellow;
                    Console.BackgroundColor = ConsoleColor.Black;
                }
                else
                {
                    Console.Write(' ');
                }
            }
            Console.WriteLine();
        }
        Console.WriteLine();
    }
}

```

Листинг 3.68 Једна од имплементација методе *simulateBaking()*

Даље, потребно је направити заједнички интерфејс за пећи, *IOven*, који декларише методу *bake()* која прихвата *AbstractPizza* објекат. Сваки различити тип пећи имплементира овај интерфејс и пружа имплементацију методе за симулацију рада пећи. Симулација рада пећи се састоји из симулације укључивања пећи, након чега се позива метода *simulateBaking()*, ради симулације промена изгледа прослеђене пие. Симулација рада пећи је дата на листингу 3.69, на примеру класе *PizzaOven*, која представља пећ посебно прављену за пие.


```

class PizzaOven : IOven
{
    public PizzaOven() { }

    public void bake(AbstractPizza pizza)
    {
        Console.WriteLine("Turning on the pizza oven...");
        Console.WriteLine();

        List<ConsoleColor> colors = new List<ConsoleColor>();
        colors.Add(ConsoleColor.DarkGray);
        colors.Add(ConsoleColor.Gray);
        colors.Add(ConsoleColor.Yellow);
        colors.Add(ConsoleColor.DarkYellow);
        colors.Add(ConsoleColor.Red);
        colors.Add(ConsoleColor.DarkRed);

        for (int i = 0; i < 6; i++)
        {
            Console.ForegroundColor = colors[i];
            Console.WriteLine("#####");
            Console.SetCursorPosition(0, Console.CursorTop + 12);
            Console.WriteLine("#####");
            Console.SetCursorPosition(0, Console.CursorTop - 14);
            Thread.Sleep(1500);
        }

        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.SetCursorPosition(0, Console.CursorTop + 15);
        Console.WriteLine("Oven turned on!");
        Console.SetCursorPosition(0, Console.CursorTop - 15);

        pizza.simulateBaking();

        Console.SetCursorPosition(0, Console.CursorTop + 4);
        Console.WriteLine();
        Console.ForegroundColor = ConsoleColor.DarkCyan;
        Console.WriteLine("Pizza baked!");
    }
}

```

Листинг 3.69 Класа *PizzaOven*

Оваквом структуром могуће је комбиновати било који тип пећи са било којим типом пице. Додавањем нових типова пећи или пица, не долази до мењања програмског кода ни у постојећим класама које представљају пице, а ни у постојећим класама које представљају пећи. Такође, не долази ни до дуплирања програмског кода. Клијент класа задужена за позивање функционалности симулације печења пице, *AbstractPizzaOrder*, садржи референце ка оба типа пећи (листинг 3.70).

```

abstract class AbstractPizzaOrder
{
    protected AbstractPizza chosenPizza;
    private IOven pizzaOven;
    private IOven regularOven;
}

```

Листинг 3.70 Класа *AbstractPizzaOrder*

Класа *AbstractPizzaOrder* која позива ове методе, прослеђује им пицу коју је потребно испећи (листинг 3.71).

```

bool bakePizza()
{
    Console.WriteLine("Choose oven: ");
    Console.WriteLine("1. Regular");
    Console.WriteLine("2. Pizza");
    Console.Write("Option: ");

    String oven = Console.ReadLine();
    Console.WriteLine();
    Console.ForegroundColor = ConsoleColor.DarkCyan;

    Console.WriteLine("Chosen oven: " + oven);
    Console.WriteLine();

    if (oven == "1")
    {
        this.regularOven.bake(this.chosenPizza);
    }
    else if (oven == "2")
    {
        this.pizzaOven.bake(this.chosenPizza);
    }
    else
    {
        Console.WriteLine();
        Console.ResetColor();
        Console.WriteLine("Bad input! Please try again!");
        return false;
    }

    Console.WriteLine();
    Console.ResetColor();
    return true;
}

```

Листинг 3.71 Позив методе за симулацију печења пице

3.5.5 Factory method

Потребно је имплементирати функционалност корисничког одабира пице.

3.5.5.1 Опис проблема

Процес одабира пице се састоји из два дела. Први корак је да корисник бира коју конкретно пицу жели да наручи. У другом кораку, у зависности од пице коју је корисник одабрао, врши се њено креирање. У зависности од типа поруџбине, процес одабира и креирања пице се разликује. Уколико је у питању поруџбина на лицу места, корисник може наручити било коју пицу (листинг 3.72).

```

Console.WriteLine("Type of pizza to order: ");
Console.WriteLine("1. Capricossa");
Console.WriteLine("2. Vegetariana");
Console.WriteLine("3. Margherita");
Console.WriteLine("4. Domestic Capricossa");
Console.WriteLine("5. Domestic Vegetariana");
Console.WriteLine("6. Domestic Margherita");
Console.ResetColor();
Console.Write("Choose type: ");
int option = Convert.ToInt32(Console.ReadLine());

switch (option)
{
    case 1:
    {
        this.chosenPizza = new PizzaCapricossa();
        break;
    }
    case 2:
    {
        this.chosenPizza = new PizzaVegetariana();
        break;
    }
    case 3:
    {
        this.chosenPizza = new PizzaMargherita();
        break;
    }
    case 4:
    {
        Console.Write("Choose size: ");
        int size = Convert.ToInt32(Console.ReadLine());
        this.chosenPizza = new DomesticPizzaCapricossa(size);
        break;
    }
    case 5:
    {
        Console.Write("Choose size: ");
        int size = Convert.ToInt32(Console.ReadLine());
        this.chosenPizza = new DomesticPizzaVegetariana(size);
        break;
    }
    case 6:
    {
        Console.Write("Choose size: ");
        int size = Convert.ToInt32(Console.ReadLine());
        this.chosenPizza = new DomesticPizzaMargherita(size);
        break;
    }
    default:
    {
        Console.WriteLine("Bad input! Please try again!");
        break;
    }
}
}

```

Листинг 3.72 Део методе *startOrderingProcess()* класе *InPlacePizzaStore* задужен за одабир пице

Уколико је у питању *online* поруџбина, корисник може наручити само пице дате класама *PizzaCapricossa*, *PizzaVegetariana* и *PizzaMargherita* (листинг 3.73).

```

Console.WriteLine("Type of pizza to order: ");
Console.WriteLine("1. Capricossa");
Console.WriteLine("2. Vegetariana");
Console.WriteLine("3. Margherita");
Console.ResetColor();
Console.Write("Choose type: ");
int option = Convert.ToInt32(Console.ReadLine());

switch (option)
{
    case 1:
    {
        this.chosenPizza = new PizzaCapricossa();
        break;
    }
    case 2:
    {
        this.chosenPizza = new PizzaVegetariana();
        break;
    }
    case 3:
    {
        this.chosenPizza = new PizzaMargherita();
        break;
    }
    default:
    {
        Console.WriteLine("Bad input! Please try again!");
        break;
    }
}

```

Листинг 3.73 Део методе *startOrderingProcess()* класе *OnlinePizzaStore* задужен за одабир пице

Потребно је приметити да прогамски код, услед комплексније логице креирања пице, постаје доста дужи. Метода *startOrderingProcess()* сада мора да брине и о логици креирања пице, што нарушава *Single responsibility* принцип. Уколико би дошло до потребе да се овакав програмски код јави на још неколико места, дошло би до великог понављања истог програмског кода. Такође, проблем би настао и уколико би се у будућности јавила потреба за променом у логици креирања пице. Потребно би било претражити сва места на којима се такав програмски код понавља, и извршити модификацију кода по новом захтеву.

3.5.5.2 Решење

Решење претходног проблема је дато *Factory method* шаблоном. У класи *AbstractPizzaOrder* потребно је издвојити посебну методу која ће бити задужена за одабир и креирање пице. Нека то буде метода *choosePizza()*. На листингу 3.74 је дата имплементација ове методе, за класу *InPlacePizzaOrder*.

```

public override AbstractPizza choosePizza()
{
    Console.WriteLine("-----");
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine("Type of pizza to order: ");
    Console.WriteLine("1. Capricossa");
    Console.WriteLine("2. Vegetariana");
    Console.WriteLine("3. Margherita");
    Console.WriteLine("4. Domestic Capricossa");
    Console.WriteLine("5. Domestic Vegetariana");
    Console.WriteLine("6. Domestic Margherita");
    Console.WriteLine();
    Console.ResetColor();

    Console.Write("Choose type: ");
    int option = Convert.ToInt32(Console.ReadLine());

    switch (option)
    {
        case 1:
        {
            this.chosenPizza = new PizzaCapricossa();
            break;
        }
        case 2:
        {
            this.chosenPizza = new PizzaVegetariana();
            break;
        }
        case 3:
        {
            this.chosenPizza = new PizzaMargherita();
            break;
        }
        case 4:
        {
            Console.Write("Choose size: ");
            int size = Convert.ToInt32(Console.ReadLine());
            this.chosenPizza = new DomesticWayPizzaCapricossa(size);
            break;
        }
        case 5:
        {
            Console.Write("Choose size: ");
            int size = Convert.ToInt32(Console.ReadLine());
            this.chosenPizza = new DomesticWayPizzaVegetariana(size);
            break;
        }
        case 6:
        {
            Console.Write("Choose size: ");
            int size = Convert.ToInt32(Console.ReadLine());
            this.chosenPizza = new DomesticWayPizzaMargherita(size);
            break;
        }
        default:
        {
            return null;
        }
    }

    Console.WriteLine();
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine("Chosen pizza: " + this.chosenPizza.ToString());
    Console.ResetColor();
    Console.WriteLine("-----");
    return this.chosenPizza;
}

```

Листинг 3.74 Имплементација методе *choosePizza()* класе *InPlacePizzaOrder*

На листингу 3.75 је дата имплементација методе *choosePizza()*, за класу *OnlinePizzaOrder*.

```

public override AbstractPizza choosePizza()
{
    Console.WriteLine("-----");
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine("Type of pizza to order: ");
    Console.WriteLine("1. Capricossa");
    Console.WriteLine("2. Vegetariana");
    Console.WriteLine("3. Margherita");
    Console.WriteLine();
    Console.ResetColor();

    Console.Write("Choose type: ");
    int option = Convert.ToInt32(Console.ReadLine());

    switch (option)
    {
        case 1:
        {
            this.chosenPizza = new PizzaCapricossa();
            break;
        }
        case 2:
        {
            this.chosenPizza = new PizzaVegetariana();
            break;
        }
        case 3:
        {
            this.chosenPizza = new PizzaMargherita();
            break;
        }
        default:
        {
            return null;
        }
    }

    Console.WriteLine();
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine("Chosen pizza: " + this.chosenPizza.ToString());
    Console.ResetColor();
    Console.WriteLine("-----");
    return this.chosenPizza;
}

```

Листинг 3.75 Имплементација методе *choosePizza()* класе *OnlinePizzaOrder*

Потребно је приметити да се сада програмски код задужен за одабир и креирање пице, налази на једном месту. Родитељска класа *AbstractPizzaOrder*, препушта имплементацију логики креирања својим класама наследницама, односно, конкретним типовима поруџбина. Након што метода *choosePizza()* изврши кориснички одабир и креирање конкретног типа пице, остале методе у којима је неопходно да тако креирана пица већ постоји, могу слободно претпоставити да је она успешно креирана, и даље је користити као такву. Свака промена у логици креирања пице, доводи само до промене у методи задуженој за креирање, односно, методи *choosePizza()*. Логика креирања пице, је одвојена од логики која креирану пицу користи.

4. ЗАКЉУЧАК

У овом раду су кроз реалистичан софтверски пројекат приказани проблеми који се ефикасно решавају применом одговарајућих дизајн шаблона. За сваки проблем дата су два решења. Једно које избегава употребу одговарајућег дизајн шаблона, и друго које га примењује. Поређењем ова два решења, указано је на предности употребе дизајн шаблона.

Приликом објашњења сваког решења, дати су сви потребни делови програмског кода, како би читалац најбоље разумео на који начин су применом одговарајућег шаблона решени проблеми који настају када се они не користе. Дати делови програмског кода су детаљно искоментарисани како би читалац најбоље разумео дату имплементацију.

Дизајн шаблони који су коришћени у софтверском пројекту су претходно детаљно описани тако што је пружен увид у њихову дефиницију и намену, општи опис проблема, решење истог, структуру шаблона као и када их је потребно користити.

Даљи кораци у циљу побољшања софтверског пројекта који је имплементиран, као и разумевања дизајн шаблона уопште, били би проширивање софтверског пројекта додавањем нових функционалности. Такве функционалности биле би одабране тако да демонстрирају употребу осталих дизајн шаблона који су изостављени у овом раду.

ЛИТЕРАТУРА

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, „Design patterns: Elements of reusable object-oriented software,“ 21 Октобар 1994.
- [2] A. Shvets, „Dive into design patterns,“ 2009.
- [3] M. Fowler, „UML distilled: a brief guide to the standard object modeling language,“ 2004. Addison-Wesley Professional.
- [4] R.C. Martin, „Agile software development: principles, patterns, and practices,“ 2002. Prentice Hall.
- [5] R.C. Martin, „Clean code: a handbook of agile software craftsmanship, “ 2009. Pearson Education.
- [6] B. Meyer, „Object Oriented Software Construction,“ 1997. Prentice Hall.
- [7] E. Freeman, K. Sierra, „Head First Design Patterns,“ 2004.
- [8] J. Bloch, „Effective Java,“ 2017. Addison-Wesley Professional.

БИОГРАФИЈА

Душан Рађеновић рођен је 28. јуна 1996. године у Зрењанину. Основну школу „Јован Јовановић Змај“ завршио је у Зрењанину 2011. године, као носилац Вукове дипломе. Након тога уписује, а 2015. године завршава гимназију – природни смер у Средњој школи у Зрењанину. Исте године уписао се на Факултет техничких наука у оквиру Универзитета у Новом Саду, смер Рачунарство и аутоматика. Положио је све испите предвиђене планом и програмом.

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, RBR:	
Идентификациони број, IBR:	
Тип документације, TD:	монографска публикација
Тип записа, TZ:	текстуални штампани документ
Врста рада, VR:	дипломски рад
Аутор, AU:	Душан Рађеновић
Ментор, MN:	
Наслов рада, NR:	Примена одабраних дизајн шаблона у реалистичним софтверским апликацијама
Језик публикације, JP:	српски
Језик извода, JL:	српски / енглески
Земља публикавања, ZP:	Србија
Уже географско подручје, UGP:	Војводина
Година, GO:	2020
Издавач, IZ:	ауторски репринт
Место и адреса, MA:	Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6
Физички опис рада, FO:	6 / 83 / 8 / 10 / 10 / 0 / 0
Научна област, NO:	Рачунарске науке
Научна дисциплина, ND:	Пословна информатика
Предметна одредница/кључне речи, PO:	дизајн шаблони, проширивост софтвера, чист код
UDK	
Чува се, ČU:	Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад
Важна напомена, VN:	
Извод, IZ:	У овом раду дати су спецификација, имплементација и верификација софтверског решења које испољава структуру и својства погодна за примену софтверских дизајн шаблона. Одабрани шаблони су примењени у апликацијама и предности њихове употребе су истакнуте.
Датум прихватања теме, DP:	
Датум одбране, DO:	
Чланови комисије, KO:	
председник	Горан Сладић, ванр. проф. ФТН, Нови Сад
члан	Никола Лубурић, асистент-мастер, ФТН, Нови Сад
ментор	Гордана Милосављевић, ванр. проф, ФТН, Нови Сад
Потпис ментора	

KEYWORDS DOCUMENTATION

Accession number, ANO:	
Identification number, INO:	
Document type, DT:	monographic publication
Type of record, TR:	textual material
Contents code, CC:	Bachelor thesis
Author, AU:	Dušan Rađenović
Mentor, MN:	
Title, TI:	Use of Selected Design Patterns in Realistic Software Applications
Language of text, LT:	serbian
Language of abstract, LA:	serbian / english
Country of publication, CP:	Serbia
Locality of publication, LP:	Vojvodina
Publication year, PY:	2020
Publisher, PB:	author's reprint
Publication place, PP:	Novi Sad, Faculty of Technical Sciences, Square Dositej Obradović 6
Physical description, PD:	6 / 83 / 8 / 10 / 10 / 0 / 0
Scientific field, SF:	Software engineering
Scientific discipline, ND:	Business informatics
Subject / Keywords, S/KW:	design patterns, software extensibility, clean code
UDC	
Holding data, HD:	Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, N:	
Abstract, AB:	This paper presents the specification, implementation, and verification of a software solution that exhibits the structure and properties suitable for the application of design patterns. Selected design patterns are applied to the applications and the advantages they bring are highlighted.
Accepted by sci. board on, ASB:	
Defended on, DE:	
Defense board, DB:	
president	Goran Sladic, associate prof., FTN Novi Sad
member	Nikola Luburic, FTN Novi Sad
mentor	Gordana Milosavljevic, associate prof., FTN Novi Sad
Mentor's signature	