Универзитет у Бањој Луци
Природно-математички факултет

Димитрије Д. Чвокић
Драган Корађ

УВОД У СОФТВЕРСКО ИНЖЕЊЕРСТВО

Бања Лука, 2026

Универзитет у Бањој Луци
Природно-математички факултет

др Димитрије Д. Чвокић

др Драган Корађ

УВОД У СОФТВЕРСКО ИНЖЕЊЕРСТВО

Пројектовање, реализација, и развој софтвера
(уз кодове на језику Гоу)

Бања Лука, 2026.

Назив дјела:
Увод у софтверско инжењерство
Прво издање

Аутори:
Доц. др Димитрије Д. Чвокић
Проф. др Драган Корађ

Издавачи:
Универзитет у Бањој Луци
Природно-математички факултет
Универзитетски град, Булевар војводе Петра Бојовића 1А, 78000 Бања Лука

За издавача:
Проф. др Радослав Гајанин, ректор
Проф. др Горан Трбић, декан

Рецензенти:
Проф. др Дејан Симић
Проф. др Љубиша Прерадовић

Аутор прелома и графичке обраде публикације:
Димитрије Д. Чвокић

Лектор:
Никола Париповић

Штампа:
Електронско издање

ISBN: 978-99997-45-43-7

MSC (2020): 68N30 (Software engineering), 68N01 (General theory of computing systems)

Одлуком Наставно-научног вијећа Природно-математичког факултета Универзитета у Бањој Луци под бројем 19-3.1261/26, од 17.06.2026. г., и Одлуком Сената Универзитета у Бањој Луци под бројем 02/04-3.1825-73/26, од 02.07.2026. г., одобрено је издавање ове књиге као *универзитетске наставне литературе*.

Парадокс намјерно остављене празне стране.

ПРЕДГОВОР

Ова књига настала је из потребе да се студентима Информатичког смјера Студијског програма Математика и информатика Природно-математичког факултета Универзитета у Бањој Луци понуди уџбеник који софтверско инжењерство представља не само као збир парадигми, методологија, и техника, већ као начин размишљања. Премда је дисциплина данас обимна и ваљано академски утемељена, и даље постоји јаз између теорије и праксе, између онога што се предаје и онога што се у пракси заиста дешава. Овај уџбеник настоји да тај јаз смањи, не некаквим упрошћавањем, него прецизношћу формулација, критичким осврtом, и умјереном дозом ироније која је изгледа неизоставни сапутник сваког озбиљнијег бављења софтверским инжењерством. Уосталом, да парафразирамо дух Кнутовог односа према програмерској догми: када вам методологија почне уређивати живот више него календар, вријеме је да је преиспитате. Ова књига не иде толико далеко, али барем сугерише опрез.

Прва од специфичности овог уџбеника, која се посебно издваја, јесте настојање да се софтверско инжењерство третира као инжењерска дисциплина у пуном смислу те ријечи. То подразумијева да се кључни појмови и методи освијетле помало и кроз формалне структуре, математичке аналогије, и тамо гдје је смислено помоћу математичких теорема које показују домете и ограничења појединих приступа. Према сазнањима аутора, ријеч је о једном од ријетких покушаја у нашој академској заједници да се теме софтверског инжењерства на појединим мјестима

амтематички дисциплинују, и то не као какав тежак и магловит додатак, већ као инструмент јаснијег разумијевања.

У тексту су поменути и неки од познатијих алгоритама чија улога није да "*украсе*" материју, већ да студенту покажу како се апстрактне инжењерске идеје манифестују у конкретним рачунарским конструкцијама. Софтверско инжењерство, иако фокусирано на процесе, у својој основи увијек почива на алгоритамском размишљању.

Поједине историјске поставке из класичне литературе на нашем језику овдје су кориговане. Посебно се то односи на погрешно дат редослијед појављивања појединих класичних модела животног циклуса развоја софтвера, на нетачности које се тичу Ројсових и Боумових радова, као и на превелико уопштавање "*водопада*" као модела који у изворној форми никада није ни препоручиван за сложене пројекте. Овдје су историјски прегледи дати сажето, али прецизно, како би студенти имали тачну историјску слику развоја дисциплине, без претјеривања у објашњавању онога што се данас рјеђе користи. Посебна пажња посвећена је тестовима вођеном развоју, као моделу који повезује дисциплину са креативношћу.

Специфичност овог уџбеника лежи и у његовом критичком разматрању кад је ријеч о кратким анализама обима и ограничења представљених ставова и техника. Намјера није да се нека парадигма, методологија, или пак модел промовише, а ни оспори, већ управо да се студенту укаже на то да у инжењерском раду нема неупитних догми. Уџбеник стога настоји да формира начин размишљања, а не да само пренесе технички репертоар. Колико је ауторима познато, увођење систематских критичких осврта на

крају сваког поглавља представља изузетак у уџбеничкој литератури ове области.

Трагајући током писања за моделима који би студентима помогли да препознају суптилне облике организационих дисфункција, посебну пажњу аутора привукао је један неочекиван документ из 1944. године — *Simple Sabotage Field Manual,* који је израдила тадашња *OSS* (претеча данашње *CIA-e*), а који је широј јавности објелодањен тек 2008. године. Иако у тренутку настанка није имао никакве везе са савременим појмом софтвера, његова специфичност су препоруке које и данас дјелују као упечатљива аналогија бројним честим антиобрасцима у индустрији. Управо тај контраст, уведен у уџбеник као својеврсно образовно помагало, риједак је у стручној литератури, али се показује изузетно корисним за развијање критичког мишљења.

Такође, овај уџбеник се одликује промишљеним односом према језику. Многи термини су преведени или уобличени како би на природнији начин звучали у српском језику, а гдје год је то било могуће, коришћена је ћирилица. Намјера аутора је да покажу како се савремени информатички садржаји могу излагати јасно, прецизно, и стилски складно на српском језику, без непотребног ослањања на англоцентричну терминологију, осим тамо гдје је она заиста неопходна.

Програмски примјери у уџбенику дати су на језику Гоу (енг. *Go, GoLang*). Разлог за тај избор не лежи у праћењу трендова, већ у инжењерској дисциплини коју Гоу истиче: јасној структури пројеката, изричитом управљању зависностима, брзој повратној информацији (енг. *feedback*), и минимализму који не трпи такозвану “магију“. Према нашим доступним сазнањима ово је

први уџбеник из софтверског инжењерства који се конзистентно ослања на програмски језик Гоу. На тај начин студентима се представља прегледан, чист, и предвидљив језик у којем се непосредно уочавају посљедице добрих, али и лоших архитектонских одлука. Допунске реализације појединих примјера на језицима Пајтон и Јава доступне су преко пратећег репозиторијума.

С друге стране, аутори се у овом тексту не баве детаљно *Unified Modeling Language*-ом (*UML*-ом). Разлог је једноставан: полази се од претпоставке да студенти четврте године већ познају основну прегледну нотацију. *UML* ће бити искључиво коришћен тамо гдје заиста доноси корист, прије свега у раном моделовању, док се неће наметати у ситуацијама у којима његова прецизност нема јасан практични допринос.

Тон излагања је намјерно умјерено ироничан, у духу традиције коју су његовали Дајкстра, Кнут, и Керниген: довољно озбиљан да очува академски карактер градива, али довољно жив да студенти осјете како иза техничких рјешења стоји човјек, а иза модела стварни пројекти, са стварним потешкоћама. Циљ уџбеника није само да информише, него да студента подстакне на самостално и критичко размишљање.

Још једна специфичност би се могла истаћи, а то је да се овим уџбеником уводи, помало експериментално, нови концепт назван *контекстно-интуитивно кодирање.* Ријеч је о новој парадигми софтверског инжењерства заснованој на синергији човјека и генеративних модела, која још увијек није академски ваљано третирана, иако је у потпуности прожела праксу.

На крају, потребно је истаћи да су аутори при изради овог уџбеника посебну пажњу посветили да садржај буде подједнако користан студентима и практичарима. Свака тема настоји да споји теорију и искуство, указујући на границе формалних модела, и простор у којем се доноси професионални суд. Умјесто апсолутних рецепата, уџбеник нуди начела, јер управо способност критичког размишљања и прилагођавања разликује инжењера од простог оператера.

Аутори

САДРЖАЈ

1. УВОД У СОФТВЕРСКО ИНЖЕЊЕРСТВО

Сврха поглавља

У овом поглављу поставља се почетни оквир за разумијевање софтверског инжењерства као инжењерске дисциплине и културе рада засноване на систематичности, одговорности, и критичком преиспитивању претпоставки.

1.1. Шта је софтверско инжењерство?

Софтверско инжењерство је информатичка дисциплина која се бави систематским приступом развоју, одржавању и измјенама софтвера. Њен основни циљ јесте да се сложени софтверски системи развијају предвидљиво, поуздано и одрживо, уз контролу трошкова, времена и квалитета.

На енглеском говорном подручју термин *Software Engineering* појавио се крајем шездесетих година XX века, у контексту првих великих, а неуспјелих софтверских пројеката, који су постали познати широј тадашњој програмерској заједници. Схватило се да развој софтвера, као услужног производа, није тек писање некаквог кода, већ је ријеч о процесу који захтијева и планирање, и сарадњу, и управљање, и изразиту радну

дисциплину, слично свим другим већ тада познатим инжењерским областима.[1]

Слика 1.1. Приказ софтверског инжењерства као дисциплине која повезује парадигме развоја, процесе, методологије, и конкретна инжењерска оруђа. Сврха је да се истакне да ниједна компонента не функционише изоловано.

Међутим, за разлику од грађевинарства или машинства, гдје сама радна материја (супстанца) пројекта је та која ограничава одређену врсту грешака које се могу десити (на основу закона физике), у развоју софтвера ситуација није баш тако пластична. Просто, нема статике, нити физичког трења. А таква природа "материје" омогућава да се грешке лакше шире (али и брже исправе). Даље, управо ова помало "апстрактна природа" чини ову нашу дисциплину посебно подложном кад је ријеч о

[1] Треба имати у виду да се и послије краја 60-их редовно јављају тешки промашаји када је ријеч о развоју великих софтверских система. Један од познатих примјера из 90-их година је развој аутоматског система за управљање пртљагом на аеродрому у Денверу (САД). Кашњења на испоруци софтвера су износила скоро читаве двије године, а поврх тога није се ни успјело остварити задано аутоматско управљање пртљагом у довољној мјери (и даље је значајан дио рада обављан био ручно).

пренаглашавању процедура, а и модела развоја, који често немају упориште у самој веома динамичној пракси (индустрији). Краћи разговор са скоро било којим тренутним вођом тима неке иоле озбиљније *IT* компаније на ову тему ће потврдити да су се и они сусретали са пројектима за велике клијентске системе, и да је инсистирање на ригорозним процедурама знало довести до застоја. Обично су се те ситуације разрјешавале искуством и довитљивошћу старијих програмера, који су знали да преусмјере ресурсе и снађу се "у ходу", како би се испунили захтјеви клијента. Отуда је и видљива потреба да се на софтверско инжењерство гледа трезвеније и приземније, тј. више као на спој парадигми, методологија, модела, али и критичког расуђивања, а можемо слободнији бити и казати дословно речено познавања "тајни заната".

Како је ово уџбеник за студенте *IV* године информатичког смјера, а то значи да су већ поприлично добро информатички писмени, већ на самом почетку ће у табели 1 бити дат преглед кључних појмова, основних концепта, парадигми, методологија, модела, оруђа и техника.

Табела 1.1. Преглед кључних појмова у софтверском инжењерству и њихових улога.

ДИМЕНЗИЈА	КЉУЧНИ КОНЦЕПТИ И ЊИХОВА УЛОГА
ЦИЉ	Систематски развој софтвера који је поуздан, одржив и економски исплатив. ☑ Тражи се баланс између формалности и практичности.
ЖИВОТНИ ЦИКЛУС РАЗВОЈА СОФТВЕРА	Захтјеви → Пројектовање → Реализација → Тестирање → Одржавање

(SDLC)	☑ Итеративни нелинеарни процес.
ОСНОВНИ ПРИНЦИПИ	● Модуларност — подјела на независне цјелине; ● Апстракција — фокус на битно, сакривање детаља; ● Енкапсулација — спајање података и понашања; ● Реупотреба — *DRY* начело[2].
КВАЛИТЕТ	● Провјера: "Да ли радимо производ исправно?" ● Потврда: "Да ли радимо исправан производ?" ☑ Оба су нужна. Ниједно није довољно.
ПРОЦЕСИ	● Традиционални (Инкрементални (и рани итеративни приступ), Водопадни, *V*-модел, Спирални, Итеративни) — карактерише предвидљивост; ● Агилни (*Scrum, Kanban, XP*) — карактерише прилагодљивост; ● *Девопс* — карактерише стална испорука; ☑ У пракси: хибридни приступи.
ОРУЂА	● Контрола верзија (*Git, CVS, SVN, Mercurial*) — праћење промјена ● Моделовање (*UML, SysML, BPMN, ERD, DFD, ArchiMate, IDEF0, IDEF1X*) — визуелизација структуре ● Аутоматизација (*CI/CD, IaC,*

[2] *DRY* је скраћеница од *Don't Repeat Yourself* (срп. Не понављај се), ријеч је о веома познатом канонском начелу у програмирању.

	Configuration Management, Orchestration, Automated Testing Frameworks, Observability) — стално тестирање ☑ Оруђе није замјена за размишљање.

Кад је ријеч о *софтверу,* постоје различити погледи на њега као објекат. Најчешће га посматрамо (истовремено) двојако:[3]

- *Производ.* Софтвер је крајњи резултат развоја, тј. композиција извршног кода, конфигурационих датотека, документације и извршених тестова.
- *Процес.* Софтвер је низ активности којима се (претходно поменути) производ ствара, провјерава, одржава, али и даље развија.

У пракси, квалитет процеса не гарантује квалитет производа. Могуће је формално испоштовати све задате фазе развоја софтвера, а ипак добити нефункционалан производ. На примјер, тим програмера може прецизно слиједити дефинисане кораке одабраног модела развоја свог софтвера/производа, али ако се занемари, рецимо, рано тестирање, резултат може бити апликација која не испуњава кључне захтјеве крајњих корисника Управо због тога савремена индустријска пракса све више и више наглашава повратну спрегу између кода и теста, што је посљедично водило ка концепту *тестовима вођеног развоја* (енг. *Test-Driven Development (TDD)*).

[3] У пракси, многи тимови открију да је процес на крају већи од производа (односно "скупља пита него тепсија"). Ово је познато као "инжењерска верзија термодинамике": ентропија документације расте све док је не обришете.

1.2. Кратак историјат дисциплине

Прије него што упловимо у историјске воде, ово је можда добар тренутак да читалац размисли о симптомима које би он сам очекивао у пројекту који лагано граби ка неуспјеху. Другим ријечима, запитајмо се сад већ на почетку, шта би нас, као будуће софтверске инжењере, навело на помисао да наш пројекат ипак није на добром колосијеку?

Током педесетих и шездесетих година *XX* вијека, развој софтвера био је неформалан и вођен искључиво програмерском праксом. Софтвер се "писао" *ad hoc*, без документације, без стандарда и без дефинисаних фаза развоја.

Када су пројекти постали већи (у војним и научним програмима, као што су *SAGE, Apollo Guidance Computer, IBM System/360*), појавили су се проблеми: кашњења, неконтролисани трошкови, те грешке у критичним системима. Први талас интересовања за систематизацију развоја софтвера појавио се на конференцији НАТО-а 1968. године, у Гармиш-Партенкирхену.[4] Појам „*Software Crisis*" означавао је низ великих неуспјеха — пројеката који су били прескупи, недовршени или потпуно нефункционални. Учесници попут Фридриха Л. Бауера, Брајан Рандела, Едзгара В. Дајкстре, и Питера Наура, нагласили су да је сам развој софтвера потребно третирати као строго инжењерску дисциплину, са фазама, методама/техникама, и, наравно, контролом квалитета.

[4] Погледати [54] и [64].

У деценијама које су слиједиле, утемељио се појам *животног циклуса развоја софтвера* (енг. *Software Development Life Cycle (SDLC)*), за шта су развијани различити модели:

- инкрементални модел;
- водопадни модел;
- *V*-модел;
- спирални модел;
- итеративни модел;
- еволутивни модели (не као један, него као руковјет модела).

У новије вријеме доминира нешто што бисмо могли окарактерисати као *Агилни* (енг. A*gile*)[5] покрет, који наглашава прилагођавање и тимску комуникацију. Ипак, пракса је показала да је многе Агилне тимове брзо захватила супротна крајност, а то је склоност импровизацији без преузимања одговорности. У литератури и заједници која форсира Агилно, та појава се често зове *cargo cult agile*[6] (срп. квазиизведено Агилно). Просто, дешава се да такозване "церемоније" Агилног приступа развоју софтвера (итерације/спринтови (енг. *sprints*), брифинзи (енг. *stand-ups*)) спроводе без икакве суштинске добити.

[5] Агилно је туђица у српском језику која у суштини означава нешто брзо прилагодљиво, окретно, флексибилно, али и предузимљиво и проактивно.

[6] На домаћој *IT* "сцени", овај појам се практично не користи, па га прво наводимо на енглеском језику. Касније ћемо у тексту користити наш дати превод на српски.

1.3. Укратко о самој дисциплини и њеном инжењерском доживљају

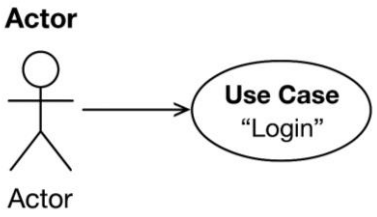
Прави инжењерски приступ, кад је ријеч о развоју софтвера, не лежи у којекаквим ригидним обрасцима, или рецимо, поменутим церемонијама савремених парадигми, већ у јасном моделовању, мјерљивим резултатима и сталним провјерама (тестирањима). Кључне карактеристике таквог приступа су: дисциплина у документовању и верзионирању, минималистички пројекат (али јасан), брза повратна информација, и разуман однос према ризицима (кад се уоче).

Другим ријечима, "инжењерство", као појам, у овом контексту није тек нека формална ознака већ и обиљежје одговорности, тј. способност да се читав пројекат изгура до краја, да се грешке препозна(ва)ју на вријеме, те да се код може разумјети и одржавати дугорочно (енг. *Long Term Support* (*LTS*)). Просто, можемо себи поставити једно практично питање: "Да ли код остављам јаснијим у односу на то што сам затекао?"

Како би се овај начин размишљања који се односи на повезивање логике, структуре и провјере, проширио и на ниво самог пројектовања, користе се модели као што је *Unified Modeling Language (UML)*[7] (срп. *Општи језик за моделовање*). Основни дијаграми који се препоручују почетнику програмеру (енг. *Junior*) да треба да савлада су обично:

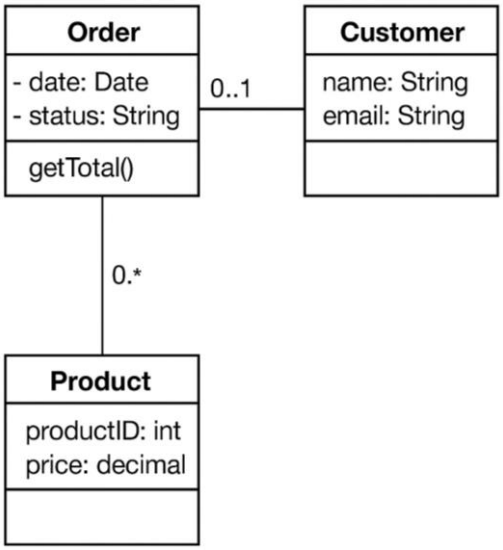
[7] У даљем тексту ћемо користити скраћеницу из енглеског језика *UML*, јер је одавно присута у говору и литератури на српском језику. Штавише, управо због њене поменуте свеприсутности одлучили смо се да је прво наведемо у оригиналу, а онда да дамо превод на српски.

- *дијаграм случај(ев)а употребе* (енг. *Use Case Diagram*), који нам служи за опис функционалности самог система (показни примјер је дат на слици 1.2);



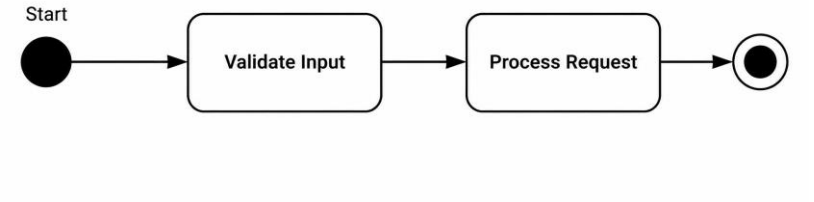
Слика 1.2. Прост показни примјер дијаграма случаја употребе (*Use Case* дијаграм). Учесник (*Actor*) се пријављује (*Login*) за рад (у систему).

- *дијаграм класа* (енг. *Class Diagram*), који користимо за опис односа између ентитета (показни примјер је дат на слици 1.3);



Слика 1.3. Прост показни примјер дијаграма за класу (*Class* дијаграм), у којем су, конкретно дате сљедеће три класе: *Order, Customer, Product*.

- *дијаграм активности* (енг. *Activity Diagram*), који нам обично служи за приказ тока нашег процеса (показни примјер је дат на слици 1.4).



Слика 1.4. Прост показни примјер дијаграма активности (*Activity* дијаграм). По потврди уноса (*Validate Input*) прелази се на њихову обраду (*Process Request*).

Читаоцу треба бити јасно да нећемо у овом уџбенику много говорити о самим дијаграмима. Нешто више о њима биће казано у потпоглављу 4.2 *UML* у служби моделовања. За сад нам је битно да схватимо да моделовање није циљ само по себи, тј. оно треба да послужи тек толико да програмер има добру визуелну представу пројекта, те да може пластичније да разумије његову суштину прије писања одговарајућег програмског кода. Ако сам дијаграм-модел не појашњава ништа програмеру, онда је то у ствари ништа више до украс на папиру (или пак табли).[8] Једно "правило десне руке" може да буде исказано сљедећим питањем: "Да ли нас дијаграм-модел подстиче на постављање нових конкретних питања?" Е, ово питање нас, у ствари, може водити ка смисленијем

[8] Многи пројекти из индустрије су својеврстан искуствени доказ програмерске заједнице да је најкраћи пут до неуспјеха управо прављење савршеног дијаграма-модела, а који се никада неће реализовати. Имајући то у виду, *UML* дијаграми су најбољи када су довољно јасни да буду од неке стварне користи, али у исто вријеме и не баш "раскошни", тј. таман да нам не могу да послуже као постери.

моделовању, умјесто цртању дијаграма који су сами себи (умјетничка) сврха. С друге стране, треба имати и у виду ограничења аутоматизације и да она непосредно проистичу из људске логике. Неки проблеми су доказиво нерјешиви било којим програмом, што је важно да би се схватила граница могућности у аутоматској обради података. На то указује једна од основних тврдњи теоријског рачунарства, Черчове теза[9] из 1936. године, коју наводимо у, за ову нашу материју, прилагођеној формулацији:

> *Све механички израчунљиве функције могу се свести на једноставан модел обраде симбола λ-рачуна.*[10]

На примјер, колико год напредна оруђа за статичку анализу кода користили, не можемо конструисати програм који ће за сваки могући (други) програм одредити да ли ће се он зауставити при извршавању. Ово ограничење није техничко, него фундаментално и формализовано је неодлучивошћу Проблема заустављања. Черчова теза даје шири оквир за разумијевање појма учинковите израчунљивости и граница алгоритамског поступка. Другим ријечима, постоји јасна граница између онога што је израчунљиво (обрадиво као податак) и онога што није, а што је важно за знати приликом процјене домета аутоматске обраде података, сагледавања проблематике која се треба ријешити, и наравно провјеравања добијених резултата.

[9] Черчова теза није математичка теорема, јер појам "механички дјелотворно израчунљиве функције" није формално дефинисан. Она представља идентификациону тезу којом се интуитивни појам алгоритма поистовјећује са класом λ-дефинибилних функција.

[10] Касније је показано да то важи и за Тјурингову машину, за коју се касније, формално, кренуло да сматра да представља еквивалент појму алгоритма.

1.4. Критички осврт: Успостављање баланса између академског формализма и праксе

У бројним академским радовима из области софтверског инжењерства могу се уочити два типична проблема:

1. Хиперформализација — инсистирање на процедурама и шаблонима без стварног разумијевања домена; рецимо, тим од петоро инжењера шест недјеља прави потпуно формалан *UML* пакет документације за модул за који се на крају испостави сувише нестабилан: захтјеви су се промијенили три пута, те документ није помогао у реализацији, и одржавање документације је постало већи трошак од самог кода.

2. Деконтекстуализација — модели се посматрају изоловано од реалног тима, буџета и корисника; Рецимо, тим усваја "микросервисну архитектуру" јер је то "индустријски тренд", али производ има само 300 корисника, два програмера и ограничен буџет, те умјесто скалабилности добија се неоправдана сложеност, кашњење, и више тачака отказа.

У пракси, већина успјешних тимова функционише на пола пута између хаоса и бирократије: довољно структуре да се пројекат води, али и довољно слободе да се решења брзо испробавају. Због тога овај уџбеник не тежи апсолутистичким рецептима, већ разумијевању самих начела како би будући инжењер знао када да примјени модел, а када да га одбаци, или пак како да га модификује.

Конкретна питања која студент већ сад може себи да постави, кад је ријеч о практичном раду, ради бољег разумијевања

казаног су:

- Да ли изабрани модел реално одговара сложености система који градимо?
- Колико су скупи грешка и кашњење у овом домену? Да ли нам је потребна "тежа" методологија или је то претјеривање?
- Колико је тим искусан? Да ли ће додатна документација помоћи или успорити рад?
- Које су три највеће неизвјесности у пројекту и који модел нам стварно помаже да их контролишемо?
- Да ли модел рјешава проблем или га само премијешта у форму дијаграма и шаблона?
- Шта је минимални обим формализације који стабилише рад тима?
- Да ли је модел довољно флексибилан да преживи прве измјене захтјева?

1.5. Закључак поглавља

Софтверско инжењерство није само скуп техника, већ се оно може посматрати као својеврсна култура тимског рада. Оно обухвата дисциплину, одговорност и стално провјеравање претпоставки. У пракси, најуспјешнији системи нису нужно они који су савршено документовани, већ они који су разумљиви, провјерени/провјерљиви, и одрживи током дужег временског периода. На основу историје ове наше дисциплине, можемо

наслутити да ће даљи развој ове наше дисциплине можда захтјевати непрестано унапрјеђење методологија и прилагођавање изазовима новијих и новијих технологија, али и софистицираних захтјева са клијентске стране.

1.6. Питања и задаци за провјеру знања

1. Дебатовати на сљедећу тему: да ли је софтвер више производ или пак процес?

2. Зашто се софтверско инжењерство, као дисциплина, сматра баш "инжењерством", иако је сам софтвер, гледан као производ, на неки начин нематеријалан?

3. Укратко, у чему је суштина *TDD* приступа и како он мијења однос према коду?

4. Навести три основна *UML* дијаграма и њихову сврху. Описати један примјер, осмишљен по жељи, који би говорио када се формални модел развоја не би показао нешто посебно учинковитим.

2. ЖИВОТНИ ЦИКЛУС РАЗВОЈА СОФТВЕРА

Кад је ријеч о софтверском инжењерству, можда се може рећи да је ово поглавље у ствари оно послије којег ће студент/читалац стећи заиста тај неки општи увид у саму дисциплину. Другим ријечима, по завршеном читању овог поглавља, заинтересовани читалац би требао да има у одређеној мјери разумијевање појма животног циклуса развоја софтвера, идеје примјене овог концепта у структуираном процесу развоја софтвера унутар познатих класичних модела, те да ови увиди послуже као основа за касније свеобухватније разумијевање модела, методологија, парадигми, и осталог што је битно кад је ријеч о софтверском инжењерству.

Сврха поглавља

У овом поглављу развој софтвера се сагледава као повезан животни циклус, док се модели његовог развоја разматрају као прилагодљиви оквири за размишљање, а не као непромјенљиви рецепти.

2.1. Шта је то животни циклус развоја софтвера?

Појам *Software Development Life Cycle (SDLC),* у преводу на српски "животни циклус развоја софтвера", није коришћен на НАТО конференцији одржаној чувене 1968. г., када се расправљало о кризи софтвера, али је управо дати скуп сматра иницијалном

капислом која је покренула убрзан развој ове наше дисциплине и њену појмовну карактеризацију. Први документи који систематизују сам појам "животни циклус развоја софтвера" јављају се врло брзо, већ почетком 1970-их:

- 1970. је Винстон В. Ројс[11] у свом семиналном раду[12] под називом "*Managing the Development of Large Software Systems*" (срп. Управљање развојем великих софтверских система; представљен на *IEEE WESCON* конференцији) први графички приказао праволинијски процес управљања развојем софтвера, који је касније постао познат као Водопадни модел. Иако Ројс није користио израз/скраћеницу "*SDLC*",[13] управо је његов рад дефинисао идеју секвентних фаза:

  Анализа → Пројектовање → Реализација →

  → Тестирање → Поставка → Одржавање.

  То је био његов одговор на изазове који се односе на управљање великим и сложеним пројектима, у којима је свака фаза захтијевала да буде до краја завршена како би се у сљедећу фазу могло безбрижније упловити. Такође, сам модел се односио и на систематски приступ тимском раду и на контролу граница самог пројекта.

[11] енг. *Winston W. Royce*

[12] Погледати [64].

[13] У наставку ћемо и ми користити ову скраћеницу из енглеског језика, јер је већ постала широко прихваћена у програмерској заједници и литератури на српском језику. Као што нам циљ није да некритички и помодно прихватамо англицизме, и то чак колоквијализме из енглеског језика, тако нам није циљ да одемо у другу крајност и баш све србизујемо, чиме бисмо учинили садржај овог уџбеника тотално неразумљивим и за ужу академску заједницу.

- Средином 1970-их термин *SDLC* се појављује у интерним техничким документима америчких војних програма (нпр. *DoD-STD-2167*, или *MIL-STD-498*), у којима се прописују фазе развоја за критичне системе.

- Током осамдесетих, *SDLC* постаје општеприхваћен термин у академској и индустријској литератури, нарочито у дјелима Роџера Пресмана [14] (*Software Engineering: A Practitioner's Approach*, 1982) [15] и Јана Сомервила [16] (*Software Engineering*, 1982),[17] који га формализују као основни концепт дисциплине.

SDLC представља структурисан приступ кроз који се систематски осмишљавају, дефинишу, пројектују, реализују, развијају, провјеравају, и одржавају софтверски системи. Основна идеја је да се све фазе пројекта изведу уз добар надзор, ваљано управљање, минималан ризик, и што више предвидљивости. *SDLC* се не односи на један конкретан процес, већ на породицу модела и методологија који описују различите начине развоја.

Иако су се развили разни модели и методологије, кад је ријеч о *SDLC*-у, већина њих (као приступа) садржи сљедеће фазе: прикупљање захтјева, пројектовање, реализација (остваривање

[14] енг. *Roger Pressman*

[15] Погледати [62], као најновије издање књиге Пресмана.

[16] енг. *Ian Sommerville*

[17] Погледати [69], као најновије издање књиге Сомеривла.

замишљеног и испројектованог)[18], тестирање, поставка[19], пуштање у рад (на коришћење), и одржавање.

Међутим, важно је истаћи да ове фазе нису увијек строго праволинијски организоване. У стварним пројектима честа је појава да се оно што се увиди током тестирања искористи као повратна спрега која олакшава побољшање и ревизију пројектних захтјева. Такав начин рада, из циклуса у циклус, доприноси флексибилнијем развоју софтвера, чиме се избјегавају уобичајене заблуде о томе да то треба да буде строго праволинијски процес. Поменути циклус заснован на повратној спрези оствареној кроз тестирање, често би требао да се понавља у савременим моделима развоја софтвера.

Слика 2.1. Приказ основних фаза *SDLC*-а са наглашеном повратном спрегом између реализације, тестирања и одржавања,

[18] Ријеч је о "имплементацији", али се нама више свиђа термин реализација.

[19] Ово је у ствари "инсталација", који је широко прихваћен појам. Ипак, сад смо се одлучили да га мало србизујемо, чисто да на један благ начин натјерамо читаоца да мало мисли на свом матерњем језику док чита.

чиме се указује да савремени развој не мора да буде праволинијски процес.

2.2. Укратко о класичним моделима *SDLC*-а

Током развоја дисциплине, појавило се више класичних модела (неки су већ поменути):

- *Инкрементални модел* (енг. *Incremental Model*). Уведен, у фрагментима, касних 50-их и раних 60-их у *Bell Labs* и *IBM*-у на војним софтверским пројектима.

- *Водопадни модел* (енг. *Waterfall Model*). Формално га је увео Ројс 1970.,[20] али је Херберт Д. Бенингтон[21] представио обрисе оваквог модела још 1956. г. на симпозијуму *Advanced Programming Methods for Digital Computers* (ријеч је било о развоју софтвера за систем *SAGE*). Сам модел карактерише секвентни приступ са јасно раздвојеним фазама;

- *V-модел*. Изведени из радова Берија Боума[22] писаним у периоду 1979-1984, када је развијао концепте провјере и ваљаности, итеративног профињења, ризиком вођеног инжењерства, економијом квалитета софтвера, и размишљања о животном циклусу тестирања.[23] Институционализовао се кроз системско инжењерство,

[20] Погледати [64].

[21] енг. *Herbert D. Benington*

[22] енг. *Barry Boehm*

[23] Погледати [9], [10], [11].

њемачки *V-Modell*, *INCOSE/NCOSE* литературу.[24] Наглашава паралелу између фаза развоја и тестирања (симетрија између развоја и тестирања);

- *Спирални модел* (енг. *Spiral Model*). Увео га је Бери Боум 1986.[25] Нагласак је на процјени ризика и цилусима/итерацијама;
- *Итеративни модел* (енг. *Iterative Model*). Настајао, концептуално гледано, заједно са инкременталним током 60-их, али је као формални назив ушао у литературу тек 90-их (посебно кроз *Rational Unified Processes*[26]). Нагласак је на понављају циклуса развоја (слично Спиралном).

Сви ови модели настоје да уведу дисциплину и предвидљивост у развој, али често по цијени смањене прилагодљивости.[27]

Поменути класични модели најчешће су на мети критике управо због њихове претјеране крутости. *У пракси, тј. на планети Земљи (а не у свијету академске маште), захтјеви клијената и одабир приступа (па и технологија) се мијењају током самог развоја!* Модели који не садрже повратну спрегу често доводе до тога да се развој софтвера заврши много касније него што се предвидјело, а неријетко не задовољава ни стварне потребе крајњих корисника.

[24] Погледати [10], [14], [23], [24], [73].

[25] Погледати [7].

[26] *Rational Unified Process* (*RUP*) је тежак (да се тако изразимо), круто формално дефинисан процесни оквир за развој софтвера, развијен у компанији *Rational Software* (касније постала дио *IBM*-а).

[27] Историчари софтверског инжењерства би рекли да је половина модела настала да би исправила грешке оне друге половине. Ово је једно од ријетких подручја гдје је рефакторисање историјска дисциплина.

Ови класични модели су ипак од користи као својеврстан оквир за разумијевање ове наше дисциплине, али се њиховом примјеном у пракси мора бити врло, врло пажљив. Употреба треба да буде прикладна контексту датог пројекта, и најчешће се користе у комбинацији/хибридизацији са неким другим новијим и флексибилнијим приступима.

2.2.1. Кратак приказ итеративног развоја

У итеративном развоју (не само Итеративном моделу!), свака итерација представља мали *SDLC* циклус у којем се постојећа функционалност систематски побољшава, прецизира, или проширује. Итерације не морају уводити нове модуле, али морају довести до стабилније, квалитетније, или провјерљивије верзије система.

Примјер 2.1. Развој *REST API* сервиса (рецимо унутар гоу-екосистема):

1. итерација: постављање основне *HTTP* структуре са најједноставнијим обрађивачем захтјева [28] и скелетним моделима (основним моделима без реализационе логике);

2. итерација: побољшање постојеће функционалности увођењем потврђивања и јединичних тестова;

3. итерација: оптимизација обрађивача захтјева и додавање управљања и надзором над грешкама, реорганизација кода, те побољшање структуре;

[28] енг. *handler*

4. итерација (посљедња): побољшање учинковитости, додавање кеширања или записивања (*а.k.a.* логовања), те јачање заштитних мјера. ▲

Свака итерација мора резултовати функционалном верзијом апликације која је и провјерљива.

2.3. Кратак приказ савремених приступа

Агилни приступ[29] развоју софтвера уводи итеративност на специфичан начин, а уз то и блиску сарадњу са клијентом и честе испоруке. Основна јединица рада је *итерација*, тј. кратак развојни циклус са јасно дефинисаним циљевима. Најпознатије методологије Агилног приступа су *Скрам*[30] (енг. *Scrum*), *Канбан*[31] (енг. *Kanban*) и *Екстремно програмирање* (енг. *Extreme Programming* (*XP*)).

Девопс представља проширење Агилног приступа који спаја *развој* (енг. *Development*) и *оперативу* (енг. *Operations*). Циљ је

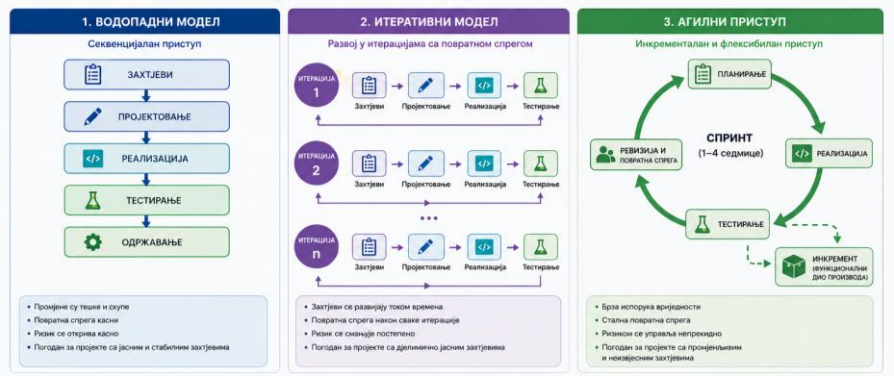
[29] У суштини, ријеч је о парадигми.

[30] Појам долази из енглеског рагбија. У рагбију *scrum* означава ситуацију у којој се игра поново покреће, оба тим су збијена (наслоњена један на други), и циљ је да се заједничким напором избори лопта. Етимолошки, *scrum* долази као скраћеница од *scrummage*, што води поријекло из старофранцуског *escarmoouche* (окршај, чарка). Из овога бисмо могли да закључимо да Скрам није "процес", него контролисан окршај са проблемом уз концентрисан тимски напор.

[31] Канбан је јапанска ријеч која дословно значи "мотрити на плочу". Појам није настао у *IT*-у, него у трговини (знаци испред радњи) и производњи (нпр. у Тојотином производном систему). У аутомобилској индустрији канбан је био просто знак који сигнализира да је нешто потребно произвести (није план, није наредба, него визуелни сигнал). Канбан, у софтверском инжењерству, подразумијева минимално наметање структуре, видљив ток рада, и "повлачење" као тип односа према појединцу и задатку (а не гур(к)ање, карактеристично за Скрам).

стална испорука (енг. *Continuous Delivery*) и *стално интеграција/увезивање* (енг. *Continuous Integration*), у којима се код аутоматски тестира, повезује, а сам софтвер испоручује.

Комбинација Агилног и Девопса данас представља *de facto* стандард у *IT* индустрији, али је важно нагласити да њихова учинковитост зависи од зрелости тима, а не од самог назива методологије. Што се тиче самих модела, више о њима ће бити казано у поглављу 9. Софтверски процеси.



Слика 2.2. Упоредни приказ Водопадног модела, Итеративног модела, и Агилног приступа развоју софтвера. Дијаграм илуструје разлике у флексибилности, повратној спрези, и управљању ризиком.

2.4. Осврт на језик Гоу у контексту развоја софтвера

У контексту *SDLC*-а, избор програмског језика није само техничка одлука. Он утиче на начин како се спецификације преводе у архитектуру, како се реализација прати, те како се сам софтвер тестира,

Гоу (енг. *Go*) (или пак Гоуленг (енг. *GoLang*)) је конструисан у оквиру једног развојног пројекта компаније Гугл (енг. *Google*)

управо да би се боље подржао инжењерски приступ развоју великих, дугорочних система: минимализам (енг. *simplicity-first*), конзистентну организацију пројеката, брзу компилацију која охрабрује честе итерације, олакшано провјеравање/тестирање, високу предвидљивост понашања и начина на који се систем одржава током његовог животног циклуса.

У педагошком контексту *SDLC*-а, Гоу омогућава да се сваки корак циклуса (рецимо, планирање → реализација → тестирање → реализација → одржавање) јасно и видљиво одрази у структури директоријума и самом коду.

2.4.1. Архитектонска структура Гоу пројеката као одраз SDLC дисциплине

Као што је већ речено, Гоу је рачунарски програмски језик осмишљен са циљем да подржи једноставност, јасну организацију и велику брзину компилације. Типичан гоу-пројекат има сљедећу структуру

```
project/              # корјен пројекта
│
├── cmd/              # улазна тачка апликација
│   └── app/          # главна апликација
│       └── main.go   # главна улазна тачка главне апликације
│
├── internal/         # ПРИВАТНИ пакети
│   └── logic/
│       └── service.go  # пословна логика (доступна само унутар модула)
│
├── pkg/              # ЈАВНИ пакети
│   └── models/
│       └── user.go   # структуре података (доступне свима)
│
├── tests/            # тестови
│   └── user_test.go
│
├── go.mod            # дефиниција гоу-модула (име, верзија, зависности)
```

```
└── go.sum          # хеш-checksum за зависности (аутоматски генерисан)
```

cmd/

- Сваки субдиректоријум представља засебну улазну тачку;
- main.go садржи func main() — почетак извршавања;
- На примјер, cmd/api/, cmd/cli/ су субдиректоријуми за различита гоу-прочеља;

internal/

- Гоу спрјечава учитавање ов(акв)их пакета ван тренутног модула;
- Врло добро је за пословну логику и реализационе детаље;
- Карактеристично заштитно својство језика Гоу;

pkg/

- Представља јавни *API* нашег пројекта;
- Ваљани и добро документовани пакети за реупотребу;
- Структуре, помоћне функције, и спољна прочеља

tests/;

- Овдје се налазе тзв. integration/end-to-end тестови;

Напомена. Јединични тестови обично иду у *_test.go поред изворног кода.

2.4.2. Веза са *SDLC* фазама

1. Анализа и спецификација. Структуре домена у суб-директоријуму pkg/models непосредно представљају елементе саме спецификације. Свако поље у овој структури *User, Invoice,* или пак *Event* одговара једном стварном захтјеву. Промјене ових захтјева

се јасно виде у моделима, тј. они су први знак да долази до промјена у самим захтјевима.

2. Архитектура и пројекат. Раздвајање на субдиректоријуме internal/ и pkg/ даје на модуларности управљања архитектуром софтвера:

- internal/ ће да покрива реализацију која се не смије извозити;
- pkg/ је као што је већ казано, јавни *API* модул, тј. стабилно прочеље које подржава развој и трансформацију система без ломљења зависности.

Конструктори језика су сматрали да ова концепција природно води програмера (али и софтверског инжењера) ка такозваној слојевитој/чистој (енг. *layered/clean*) архитектури, без неке посебен потребе за теоретисањем и теоријским упутама шта и како да се ради.

3. Реализација. Постављањем улазних тачака у cmd/ успоставља се организација пројекта која одговара реалним сервисима:

- cmd/api → веб-сервис;
- cmd/cli → оруђе за администрацију;
- cmd/scheduler → периодични процес.

Ово је непосредан пренос *SDLC* планирања на код софтвера, јер свака улазна тачка представља један системски извршни контекст.

4. Тестирање. Гоу има минималистички, али веома строг тестни модел:

- јединични тестови се пишу у *_test.go датотекама;

- интеграциони тестови могу се груписати у tests/;
- тестови се извршавају једном наредбом: go test ./...

2.4.3. Специфичности програмирања на језику Гоу

С друге стране, језик Гоу има неке своје изразите специфичности када се посматра скуп савремених рачунарских програмских језика. Рецимо, Гоу не користи класичну објектно-оријентисану парадигму, већ се ослања на *структуре* (енг. *structs*) и *прочеља* (енг. *interfaces*). Прочеља дефинишу понашање, а не тип података. Тиме он спада у такозване *објектно-засноване језике*, али не и објектно-оријентисане. Сјетимо се са предавања из објектно-оријентисагног програмирања, да можемо на пластиан начин упоредити ова два концепта. Класична објектно-оријентисана парадигма се усредсређује на хијерархију класа и насљеђивање, док философија објектно-заснованог језика Гоу наглашава дефиницију понашања преко прочеља ради веће флексибилности и једноставнијег прилагођавања компоненти (у суштини да се избјегне оно што се у програмерској пракси зове *пакао насљеђивања* (енг. *Inheritance Hell*).

Примјер 2.2. Демонстрираћемо прочеље Printer са методом Print(), структуру Report којом се реализује прочеље, метод Print() за поменути Report, те полиморфизам кроз прочеље, и иницијализацију структуре са пољем Title.

```
package main

import "fmt"
```

```
type Printer interface {
    Print()}

type Report struct {
    Title string}

func (r Report) Print() {
        fmt.Printf("Извјештај '%s' исписан.\n", r.Title)
}

func main() {
    var p Printer = Report{Title: "Дневни извјештај"}
    p.Print()

    // Алтернативни начин
    report := Report{Title: "Мјесечни извјештај"}
        report.Print()
}
```

Прочеља у Гоу посредно дефинишу понашање, што омогућава лаку замјену компоненти и једноставнију провјеру. ▲

Примјер 2.3. Приказ Инјективне зависности (енг. *Dependency Injection*-а) и Начела инверзне зависности (енг. *Dependency Inversion Principle*): прочеље PaymentProvider дефинише уговор, конкретне реализације (StripeProvider, PayPalProvider, MockProvider), лабави спрег (PaymentService, који зависи од прочеља, а не реализације), лако тестирање помоћу *mock*-објеката, и проширивост (додавање нових платних (енг. *payment*) провајдера.

```
package main

import (
    "errors"
        "testing"
)

// Лоше: Конкретне зависности
type PaymentProcessor struct {
```

```
    stripeKey string
    paypalKey string
}

func (p *PaymentProcessor) ProcessStripe(amount float64) error {
    // Директно позивање Stripe API-ја
    // Тешко за тестирање, јако спрегнут
    return nil
}

// Проблем: Тестирање захтијева мрежу, тешко додавати нове плаћања
// Рјешење: Прочеље као уговор
type PaymentProvider interface {
    ProcessPayment(amount float64) (string, error)
    Refund(transactionID string) error
}

type StripeProvider struct {
    apiKey string
}

func (s *StripeProvider) ProcessPayment(amount float64) (string, error) {
    // Реализација за Stripe
    return "stripe_tx_123", nil
}

func (s *StripeProvider) Refund(transactionID string) error {
    // Реализација refund логике за Stripe
    return nil
}

type PayPalProvider struct {
    apiKey string
}

func (p *PayPalProvider) ProcessPayment(amount float64) (string, error) {
    // Реализација за PayPal
    return "paypal_tx_456", nil
}

func (p *PayPalProvider) Refund(transactionID string) error {
    // Реализација refund логике за PayPal
    return nil
```

```
}

type PaymentService struct {
    provider PaymentProvider
}

func (ps *PaymentService) ProcessPayment(amount float64) (string, error) {
    return ps.provider.ProcessPayment(amount)
}

// Тестирање постаје тривијално
type MockProvider struct{}
func (m *MockProvider) ProcessPayment(amount float64) (string, error) {
  if amount <= 0 {
    return "", errors.New("износ мора бити позитиван")
  }
    return "test_tx_123", nil
}

func (m *MockProvider) Refund(transactionID string) error {
  if transactionID == "" {
    return errors.New("неважећи ID трансакције")
  }
    return nil
}

func TestPayment(t *testing.T) {
    service := PaymentService{provider: &MockProvider{}}
  id, err := service.provider.ProcessPayment(100.0)

  if err != nil {
    t.Errorf("Неочекивана грешка: %v", err)
  }

  if id != "test_tx_123" {
    t.Errorf("Очекиван ID 'test_tx_123', добијен '%s'", id)
  }
}

func TestPaymentWithInvalidAmount(t *testing.T) {
  service := PaymentService{provider: &MockProvider{}}
  _, err := service.provider.ProcessPayment(-50.0)
```

```
    if err == nil {
        t.Error("Очекивана грешка за негативан износ")
    }
}

func main() {
    // Производна конфигурација
    stripeService := &PaymentService{
        provider: &StripeProvider{apiKey: "sk_test_123"},
    }

    // Тестна конфигурација
    testService := &PaymentService{
        provider: &MockProvider{},
    }

    // Иста логика, различити провајдери
    _, _ = stripeService.ProcessPayment(100.0)
    _, _ = testService.ProcessPayment(100.0)
}
```

По разматрању претходног кода увиђамо једну практичну инжењерску предност: додавање новог плаћања (нпр. *PayPal*) не мијења постојећи код. ▲

Гоу подржава *лаку конкурентност* преко *гоурутина* (go func() { … }) и *канала* (енг. *channels*), што омогућава једноставно извршавање више токова без сложених механизама синхронизације. Да се подсјетимо, формалне гаранције о међусобној искључивости потичу од Леслија Лампорта [32] и његовог *пекарског алгоритма*,[33] који показује да је могуће постићи уредно међусобно чекање без заједничке меморије. Ово је теоријска позадина конкурентних образаца у Гоу.

[32] енг. *Leslie Lamport*

[33] У литератури на еглеском познат као *Lamport's bakery algorithm*.

Примјер 2.4. Демонстрација секвентне и конкурентне обраде података на језику Гоу.

```
type ValidationResult struct {
    OrderID string
    Valid   bool
      Error   string
}

type TaxResult struct {
    OrderID string
    Tax     float64
      Error   string
}

type InventoryResult struct {
    OrderID  string
    InStock  bool
      Error    string
}

// Секвентна обрада је спора
func ProcessOrders(orders []Order) []OrderResult {
    var results []OrderResult

    for _, order := range orders {
        result := OrderResult{OrderID: order.ID}

        validation := ValidateOrder(order)
        if !validation.Valid {
            result.Valid = false
            result.Error = validation.Error
            results = append(results, result)
            continue
        }

        tax := CalculateTax(order)
        inventory := CheckInventory(order)
        payment := ProcessPayment(order)

        // Агрегација
        result.Valid = true
        result.Tax = tax.Tax
```

```
        result.InStock = inventory.InStock
        result.Paid = payment.Success
        results = append(results, result)
    }

    return results
}

// Конкурентна обрада је бржа (и боље се користе ресурси)
func ProcessOrdersConcurrent(orders []Order) []OrderResult {
    var wg sync.WaitGroup
    results := make(chan OrderResult, len(orders))

    for _, order := range orders {
        wg.Add(1)
        go func(o Order) {
            defer wg.Done()

            validCh := make(chan ValidationResult, 1)
            taxCh := make(chan TaxResult, 1)
            inventoryCh := make(chan InventoryResult, 1)
            paymentCh := make(chan PaymentResult, 1)

            // Покретање свих операција
            go func() { validCh <- ValidateOrder(o) }()
            go func() { taxCh <- CalculateTax(o) }()
            go func() { inventoryCh <- CheckInventory(o) }()
            go func() { paymentCh <- ProcessPayment(o) }()

            // Чекамо на резултате
            valid := <-validCh
            tax := <-taxCh
            inventory := <-inventoryCh
            payment := <-paymentCh

            // Синхронизација и финална обрада
            result := mergeResults(valid, tax, inventory, payment)
            results <- result
        }(order)
    }

    go func() {
        wg.Wait()
```

```
        close(results)
    }()

    return collectResults(results)
}

// Помоћне функције
func ValidateOrder(order Order) ValidationResult {
    // Симулација обраде, рецимо, траје 100ms
    return ValidationResult{
        OrderID: order.ID,
        Valid:   order.Total > 0 && len(order.Items) > 0,
        Error:   "",
        }
}

func CalculateTax(order Order) TaxResult {
    // Симулација обраде, рецимо, траје 50ms
    tax := order.Total * 0.2
    return TaxResult{
        OrderID: order.ID,
        Tax:     tax,
        Error:   "",
        }
}

func CheckInventory(order Order) InventoryResult {
    // Симулација обраде, рецимо траје 200ms
    return InventoryResult{
        OrderID: order.ID,
        InStock: len(order.Items) > 0,
        Error:   "",
        }
}

func ProcessPayment(order Order) PaymentResult {
    // Симулација обраде, рецимо, траје 300ms
    return PaymentResult{
        OrderID: order.ID,
        Success: order.Total > 0,
        Error:   "",
        }
}
```

```
type PaymentResult struct {
    OrderID string
    Success bool
      Error   string
}

func mergeResults(valid ValidationResult, tax TaxResult, inventory InventoryResult, paymen
t PaymentResult) OrderResult {
    return OrderResult{
      OrderID: valid.OrderID,
      Valid:   valid.Valid,
      Tax:     tax.Tax,
      InStock: inventory.InStock,
      Paid:    payment.Success,
      Error:   "",
      }
}

func collectResults(results <-chan OrderResult) []OrderResult {
    var collected []OrderResult
    for result := range results {
      collected = append(collected, result)
    }
      return collected
}

// Побољшана верзија руковања грешкама
func ProcessOrdersConcurrentImproved(orders []Order) []OrderResult {
    var wg sync.WaitGroup
    results := make(chan OrderResult, len(orders))

    for _, order := range orders {
      wg.Add(1)
      go func(o Order) {
        defer wg.Done()

        validCh := make(chan ValidationResult, 1)
        taxCh := make(chan TaxResult, 1)
        inventoryCh := make(chan InventoryResult, 1)

        go func() { validCh <- ValidateOrder(o) }()
        go func() { taxCh <- CalculateTax(o) }()
```

```
        go func() { inventoryCh <- CheckInventory(o) }()

        // Прикупљање резултата
        valid := <-validCh
        if !valid.Valid {
            results <- OrderResult{OrderID: o.ID, Valid: false, Error: valid.Error}
            return
        }

        tax := <-taxCh
        inventory := <-inventoryCh

        // Финална обрада
        payment := ProcessPayment(o)
        result := OrderResult{
            OrderID: o.ID,
            Valid:   true,
            Tax:     tax.Tax,
            InStock: inventory.InStock,
            Paid:    payment.Success,
        }
        results <- result
    }(order)
}

go func() {
    wg.Wait()
    close(results)
}()

    return collectResults(results)
}

func main() {
    orders := []Order{
        {ID: "1", Total: 100.0, Items: []string{"item1", "item2"}},
        {ID: "2", Total: 200.0, Items: []string{"item3"}},
    }

    // Секвентно: ~650ms по наруџбини
    sequentialResults := ProcessOrders(orders)
    _ = sequentialResults
```

```
    // Конкурентно: ~300ms (најдужи корак)
    concurrentResults := ProcessOrdersConcurrent(orders)
    _ = concurrentResults
}
```

Овај код показује размјере изводљивости: 1000 наруџби се обради за око 300 ms, умјесто за 650 секунди! ▲[34]

Компилација кода на Гоу је једноставна и изузетно брза. Једна команда `go build` производи извршну датотеку без додатних корака. Повезивање са другим пакетима врши се преко `go mod` система који управља зависностима:

```
go mod init example.com/project
go get github.com/gin-gonic/gin
```

Овим приступом, према замисли твораца језика Гоу, постиже се поприлично висока преносивост и поузданост пројеката, која је потврђена и у самој индустријској пракси.

2.5. Критички осврт: Култ процеса или пак заиста корисна дисциплина?

Иако *SDLC* и класични модели животног циклуса представљају темеље дисциплине софтверског инжењерства, њихов историјски развој открива неколико важних ограничења и структурних слабости које се често занемарују у уџбеничкој

[34] Када први пут покренете 2026 гоурутина, природно је запитати се: да ли ја управљам овим процесима или они управљају са мном? Ако сте се запитали то, већ сте на добром путу...

литератури.

На почекту одмах да кажемо да ови формални модели, а посебно се ту могу истаћи Водопадни и *V*-модел, настали су у индустријама са стабилним и дугорочним захтјевима (аеронаутика, војни системи, телекомуникације). Стога, очекивано је да њихова примјена у динамичним доменима (веб, смартфон-апликације, софтвер као услуга (енг. *Software as a Service*)) често производи организациону крутост и доводи до значајног јаза између документације и стварног стања производа. Штавише, историографски гледано, већ у чувеном раду Ројса из 1970. године стоји његово лично упозорење да чисто секвентни "водопад" није адекватан за компликоване пројекте, а што је у пракси игнорисано деценијама.

Друго, класични *SDLC* модели претпостављају да су захтјеви колико-толико стабилни, те да се могу у потпуности описати прије почетка икаквог писања кода. У савременим условима, захтјеви су, слободно можемо рећи *по правилу,* непотпуни, нити довољно јасни. Такође, подложни су промјенама под утицајем тржишта, конкуренције или еволуције самих пословних потреба. Модели који немају уграђен механизам брзе повратне спреге доводе до ситуације у којој је већина грешака и пропуста откривена сувише касно, тј. најчешће у фази тестирања или након испоруке производа.

Треће, Итеративни и Инкрементални модели, који на неки начин, историјски претходе Водопадном, дуго су били недовољно формализовани, иако су се показали практичнијим у стварном индустријском програмирању. Њихова каснија формализација кроз *Rational Unified Process* и друге методологије нам је открила

да је учинковитост развоја непосредно везана за способност тима да управља ризиком. Други ријечима, битно је да тим може брзо да потврди архитектурске одлуке и непрекидно унапрјеђује квалитет кода.

Четврто, савремени приступи као што су Агилно и Девопс често се усвајају површно, тј. као скуп оруђа или "церемонија", умјесто као дисциплина заснована на вриједностима, инжењерској зрелости и ригорозном управљању техничким дугом. Без тога, Агилне праксе лако деградирају у оно што смо назвали квазиизведено Агилно, у којем се процедуре формално спроводе, али не производе стварну вриједност, нити доводе до побољшања квалитета самог производа.

Пето (и посљедње у овом разматрању), и класични и савремени модели имају своја ограничења: класични модели су прецизно структурисани, али нефлексибилни, док су савремени модели флексибилни, али њихов успјех зависи од зрелости тима, тј. не од методолошке етикете. Ово нас доводи до дубљег увида, а то је да сваки формални опис има своје границе, што није само инжењерско, већ и математичко начело. На то нам, подсјећања ради, најбоље указује *Геделова теорема о непотпуости*.

Теорема 2.1. (Прва теорема о непотпуности Курта Гедела, 1931). *У сваком конзистентном, ефективно аксиоматизованом формалном систему довољно јаком за аритметику постоје формуле које су неодлучиве у том систему, тј. ни оне ни њихове негације нису доказиве унутар система.*

Другим ријечима, ако ствар посматрамо нешто лабавије, чувена Прва Геделова теорема о непотпунсоти, објављена у

његовом семиналном раду

> Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik, 38, 173–198

(референца [31]), може се узети као упозорење да формална моћ једног система не гарантује његову потпуност. У сваком довољно снажном формалном систему, под одређеним условима, могу постојати искази који су истинити, али недоказиви унутар самог система.

Аналогија у софтверском инжењерству не треба да се схвати као директна примјена Геделове теореме, него као сродна интуиција о границама формалног описивања. Ниједан формалан опис софтвера, ниједна спецификација, UML дијаграм или методолошки процес његовог будућег развоја не могу истовремено бити коначни, потпуно исцрпни и независни од каснијег искуства реализације. Они не могу унапријед обухватити све услове, ограничења, интеракције, изузетке и понашања система који се тек развија. Зато у развоју софтвера увијек остају недоречени, имплицитни или посљедични услови који постају видљиви тек током имплементације, интеграције, употребе или тестирања.

Гедел би, да је данашњи програмер, можда написао сљедеће: “Не постоји методологија која ће вас спасити од лоше комуникације у тиму.” На жалост, ову теорему и даље формално нисмо доказали, али можемо рећи да је њена индуктивна база поприлично јака.

Примјер 2.2. Рецимо, чак и када тим направи детаљан модел

одређеног модула (нпр. обичан *API* за наплату), нови услови се појаве тек у интеракцији са стварним корисницима (било људима, било софтвером). Примјери тога су временска ограничења плаћања, изузеци у пореским правилима, или пак непредвиђена понашања других сервиса. Ништа од тога није формално "погрешно моделовано", већ су то просто природна ограничења било ког крутог формалног описа. Ништа више, и ништа мање. ▲

Овај примјер и претходно разматрање нам говори да је кључно да се *SDLC* и модели схвате као концептуални оквири, а не као какви рецепти. Искусни тимови комбинују елементе више модела, прилагођавајући их домену, тиму (себи самима), процијењеном ризику, технологији коју су одабрали, као и доступним ресурсима.

2.6. Закључак поглавља

У закључку овог поглавља можемо рећи да *SDLC* није ствар догме већ је *оквир за размишљање*. Циљ *SDLC-а* је да омогући ваљан надзор, јасно управљање, добру разумљивост, и дигне ниво одговорности у развоју софтвера за сваког од учесника. Инжењер-програмер треба да познаје многе различите моделе, али да их користи као оруђа, а не као прописане ритуале.

2.7. Питања и задаци за провјеру знања

1. Објаснити разлику између Водопадног и Итеративног модела.
2. Која је главна предност Спиралног модела?

3. Навести основне фазе *SDLC*-а.

4. Према личном мишљењу, која фаза *SDLC*-а, наведена у потпоглављу 2.1., захтјева највише времена у индустријским пројектима?

5. Како Агилни приступ одговара на промјене захтјева током пројекта?

6. Пронаћи у литератури и укратко објаснити шта је *CI/CD,* те како је он од користи.

7. Написати једноставан примјер у коме се једна итерација завршава тестирањем функције.

3. РАД СА ЗАХТЈЕВИМА И ПИСАЊЕ СПЕЦИФИКАЦИЈА

Сврха поглавља

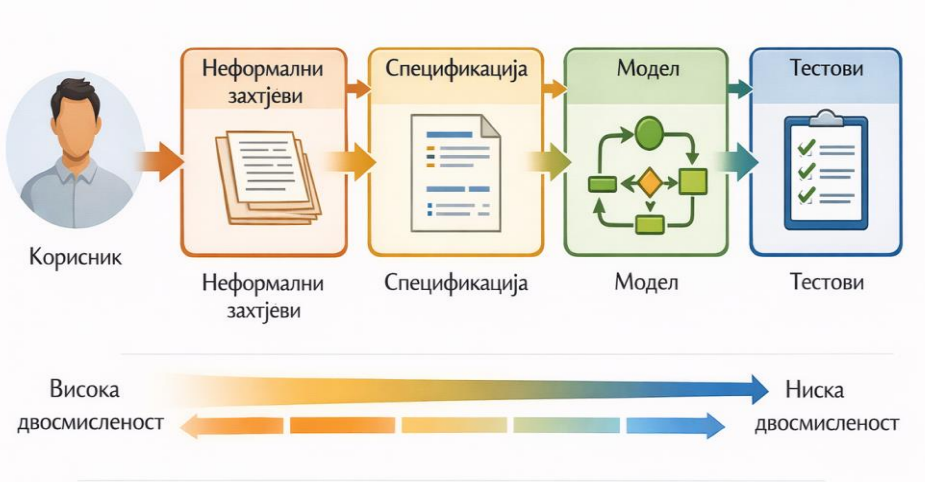
У овом поглављу разматра се како се потребе, очекивања, и пословни циљеви клијента препознају, разјашњавају, и преводе у довољно прецизне и провјерљиве софтверске захтјеве.

3.1. Појам и значај рада са захтјевима

Рад са захтјевима (енг. *Requirements Engineering*) представља процес препознавања, сагледавања, документовања и управљања захтјевима које систем треба да испуни. Циљ није само прикупљање упута од клијента, већ разумијевање пословних циљева и њихово прецизно превођење у техничке спецификације. Квалитет захтјева непосредно утиче на квалитет кода. Нејасни или контрадикторни захтјеви производе нестабилне системе, без обзира на квалитет самог програмирања. Захтјеви се уобичајено дијеле на:

- *функционалне,* који описују шта систем треба да ради;
- *нефункционалне,* који описују ограничења и особине система (брзина, сигурност, употребљивост, преносивост);
- *пословне,* који описују циљеве организације и разлоге зашто систем постоји.

У пракси, нефункционални захтјеви се неријетко занемарују, иако знају бити кључни кад је ријеч о задовољству крајњих корисника нашим софтвером. Неријетко занемаривање нефункционалних захтјева доводи до превиђања значајних скривених трошкова, како финансијских, тако и оних нематеријалних (рецимо, углед компаније).



Слика 3.1. Показни дијаграм који приказује како се неформални захтјеви корисника постепено трансформишу у формализоване спецификације, моделе, и тестове, уз постепено смањивање двосмислености.

Да би се избјегле такве ситуације и омогућило потпуније и свеобухватније разумијевање потреба система (будућег производа), користе се различите технике за препознавање и сагледавање захтјева клијената. Најчешће технике када је ријеч о раду са захтјевима клијената укључују:

- *интервјуе*, тј. директан разговор са клијентом или корисницима;
- *радионе* (енг. *workshops*), тј. групне дискусије ради усаглашавања различитих перспектива;

- *посматрање* (енг. *observation*), тј. праћење стварног начина рада;
- *анализу докумената*, тј. преглед постојећих процедура, уговора, и података;
- *прототипизацију*, тј. креирање ране верзије прочеља ради визуелизације захтјева.

Свака од ових техника има неке своје предности. Обично комбиновање више техника даје задовољавајуће резултате у пракси.

3.1.1. Студија случаја: Нејасни функционални захтјеви

Контекст: Стартап за е-трговину уз хитно “лансирање”[35]

Проблем: Захтјев је формулисан на сљедећи начин: "Систем треба да шаље обавјештења корисницима" (што видно није довољно прецизно и оставља простор за различита тумачења)

Посљедице: програмери реализују обавјештења само кроз сервис ел. поште; након “лансирања”, клијент пријављује: "Зашто нема СМС обавјештења за хитне поруџбе?"; тим мора хитно рефакторисати систем, те се губе двије седмице рада.

Ради илустрације дајемо у овој студији случаја примјер кода који се односи на обавјештавања за хитне поруџбе. Конкретно, сљедећа функција и исказ су типичан примјер нејасног и неизражајног прочеља који код рецензената и програмера подиже бројна питања.

[35] Колоквијализам за пуштање у употребу.

```go
// Шта значи овај позив? Шта се шаље?
func SendNotification(userID int, message string) error

// Позив функције је нејасан!
err := SendNotification(123, "Ваша поруџбаа је спремна") // Како се шаље нотификацији? Email? SMS? Push?
```

Иако се на први поглед чини једноставном, она у ствари нарушава више основних начела ваљаног програмског пројектовања:

1. Недостаје сазнање о начину испоруке обавјештења;
2. Корисник функције нема контролу над понашањем;
3. Крши се начело изричитости;
4. Садржи концептуалну двосмисленост;
5. Цијели систем је теже провјерити (мања провјерљивост) и теже га је проширити (мањи “маневарски простор”).

Боље, мада вјероватно не савршено рјешење, је дато у наставку, и то коришћењем изричитости када је ријеч о типовима.

```go
// Декларација типа за нотификације
type NotificationType string

// Константе за јасне опције
const (
    Email NotificationType = "email"
    SMS   NotificationType = "sms"
        Push  NotificationType = "push"
        )

// Изричита функција
func SendNotification(
        userID int,
        message string,
```

```
    notificationType NotificationType
) error

// Јасни позиви - сада знамо тачно шта се дешава!
err := SendNotification(123, "Ваша поруџба је спремна", Email)
err := SendNotification(456, "Потврда плаћања", SMS)
err := SendNotification(789, "Нова порука", Push)
```

Практичан савјет:[36] Користити конкретне критеријуме прихватања:

- "Систем шаље е-пошту приликом потврде поруџбе".
- "Систем шаље СМС обавјештење када је поруџба испоручена".
- "Обавјештења се шаљу у року од 5 минута од догађаја".

3.1.2. Студија случаја: Занемаривање нефункционалних захтјева

Контекст: Финансијска апликација

Проблем: Фокус је искључиво на просту функционалност, те се игнорише учинковитост саме апликације

Посљедице: 10 000 корисника креће да је користи истовремено → систем пада; трансакције трају 30+ секунди (превише); губи се повјерење корисника/клијената.

Рјешење: Додати нефункционалне захтјеве:

КАО корисник, ЖЕЛИМ да приступим свом профилу

[36] Једна стара Кинеска изрека каже: ако је захтјев написан тако да га сви разумију без додатних питања, онда га нико није разумио. У том случају, питају се само “продактоу(в)нери”, али то је већ други жанр.

ДА БИХ провјерио стање рачуна

КРИТЕРИЈУМИ ПРИХВАТАЊА:

- Страна се учитава за < 2 секунде

- Систем подржава 50 000 истовремених корисника

- Подаци су конзистентни при паду мреже

3.2. *UML* дијаграм случаја употребе у раду са захтјевима

Дијаграми случаја употребе служе за представљање односа између корисника (учесника) и система. Свака функционалност коју наш систем нуди може бити представљена као *случај употребе* (енг. *use case*), при чему сам дијаграм приказује *ко, шта, и због чега* ради (то што ради). Главни елементи овог дијаграма су *учесник*, *системско ограничење* (енг. *system boundary*), и *везе* (енг. *associations*) између њих.

Ови дијаграми, када се користе, обично се користе у најранијој фази сагледавања проблематике, јер пружају *интуитиван преглед очекиваног понашања система без техничких детаља*. За разлику од дијаграма класа или активности, који приказују структуру или ток извршавања, дијаграм случаја употребе описује *намјеру* и *контекст употребе*.

Примјер 3.1. Дат је систем за резервацију термина који има учеснике „Клијент“ и „Администратор“, а случајеве употребе „Преглед доступних термина“, „Резервација“ и „Отказивање термина“. Дијаграм случаја употребе нам помаже да сагледамо очекивања прије писања прве линије кода. ▲

Напомена. Треба указати на то да ови дијаграми имају и своја одређена ограничења. Наиме, они не представљају детаљан ток извршавања унутар сваког случаја, нити описују техничке аспекте реализације. Ништа од тога. Ово обично захтјева додатну анализу и додатно превођење корисничких захтјева (и очекивања) у конкретна техничка рјешења.

Поред основних односа између учесника и система, у *UML* нотацији постоје и неке специјализоване релације које доприносе тачнијем моделовању:

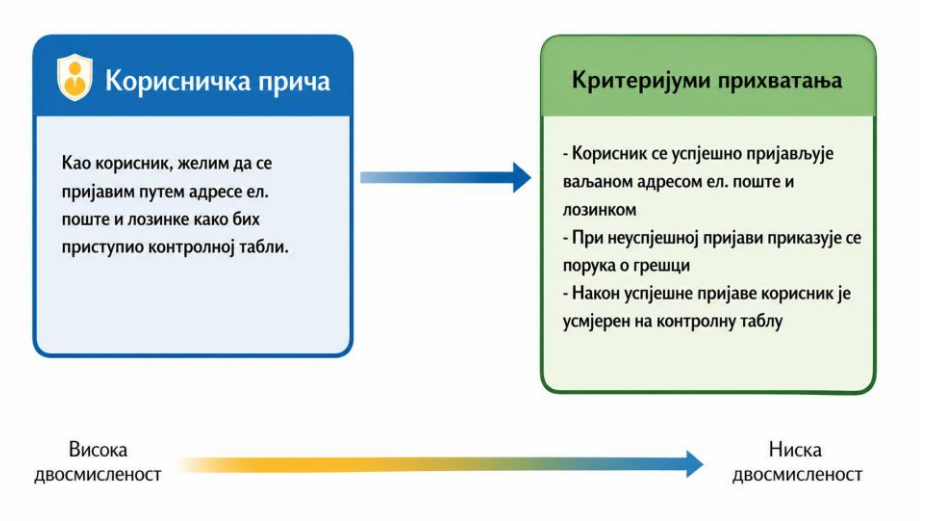
- *Include* (срп. укључи) означава да један случај употребе увијек укључује и неки други (нпр. „Резервација термина“ укључује „Провјеру доступности“).
- *Extend* (срп. прошири) означава опциону или пак условну радњу која се дешава у само неким одређеним ситуацијама (нпр. „Отказивање термина“ може проширити „Преглед резервација“).
- *Generalization* (срп. уопшти) се користи када више актера или случајева дијеле заједничка својства/особине (нпр. „Администратор“ може бити специјализација општег актера „Корисник“).

Дијаграми случаја употребе су корисни не само као оруђе за сагледавање, већ и као средство за вођење разговора између клијената, аналитичара и програмера. Они нам омогућавају да сви учесници разговарају о систему на заједничком и ненападно техничком језику, а што за посљедицу има смањен ризик од погрешних тумачења, те олакшава касније фазе пројектовања.

3.3. Корисничке приче Агилног развоја

У Агилном развоју, класични и неријетко исувише формални захтјеви се замјењују једноставним и разумљивим описима који долазе из перспективе крајњег корисника. *Корисничке приче* (енг. *User Stories*) представљају најмању јединицу функционалности у оквиру *листе ставки* (енг. *product backlog*), и служе као полазна тачка када је ријеч о планирању итерација/спринтова, те процјену посла и дефинисање *критеријума за прихватање* (енг. *acceptance criteria*). Типична форма је:

> *„Као [улога], желим [функционалност] да бих остварио [циљ].“*



Слика 3.2. Приказ односа између корисничке приче и њених критеријума прихватања, који представљају основу за тестирање и потврду функционалности, при чему критеријуми прихватања смањују двосмисленост корисничке приче..

Примјер 3.2. *„Као студент, желим да добијем обавјештење о резултатима испита на адресу е-поште, да бих избјегао свакодневно провјеравање портала.“* ▲

Оваква форма фокусира развојни тим на вриједност за корисника, а не на техничке детаље. Она подстиче разговор између клијента, програмера и тестера, тј. основно начело сада већ чувеног Манифеста Агилног (енг. *Agile Manifest*), који истиче „појединце и садејства испред обраде и оруђа“.

Да би корисничка прича била употребљива, мора задовољити критеријуме познате као *INVEST* (акроним који је предложио Бил Вејк[37] 2003. г.[38]):

- *Independent,* тј. свака прича треба да буде што независнија од других;
- *Negotiable*, тј. није уговор, већ тема за разговор и договор;
- *Valuable,* тј. мора доносити стварну корист кориснику;
- *Estimable,* тј. треба да се може разумно процјенити вријеме и труд;
- *Small,* тј. довољно мала да стане у једну итерацију/спринт;
- *Testable,* тј. мора постојати јасан начин провјере да је реализована.

[37] енг. *Bill Wake*

[38] Погледати [15], [76].

У суштини, један од начина да донекле обезбиједимо да корисничке приче испуњавају *INVEST* критеријуме, је да просто саставимо ДА/НЕ листу према горе датим критеријумима.

Даље, у оквиру сваке приче обично се додају и већ поменути критеријуми прихватања, који описују шта се сматра испуњењем саме корисничке приче. На примјер:

- „Систем аутоматски шаље обавјештење на адресу ел. поште у року од пет минута по објави резултата.“;
- „Обавјештење садржи предмет, датум и линк ка детаљима.“.

Такви критеријуми се касније непосредно употребљавају у тестовима прихватања и у самом *TDD* приступу, чиме се затвара циклус састављен од захтјева, реализације и провјеравања. Другим ријечима речено, критеријуми прихватања формулишу јасне и накнадно провјерљиве услове које софтвер мора да испуњава, како би се корисничка прича сматрала завршеном/окончаном, а то непосредно утиче на тестирање, те провјере како исправности, тако и ваљаности самог софтвера.

Корисничке приче су стога не само средство комуникације већ и својеврсно оруђе за управљање развојем, који обезбјеђује да свака програмска функција настане из стварне потребе, а не из некаквог формалног документа.

3.4. Инжењерство заштите

Када је ријеч о раду са захтјевима, једна од најчешћих грешака до сада је била та што су се сигурност и безбједност

посматрали као нешто што ће се додати касније, на крају. Дуго времена заштита софтвера се сматрала *накнадном активношћу*, која се провјеравала у завршним фазама развоја или тек током тестирања. У савременом софтверском инжењерству овај приступ је окарактерисан као застарио, сматра се да у ствари поскупљује развој софтвера, те је одбачен како у пракси тако и у академским разматрањима.

Умјесто тога, сада се инсистира на принципу „*пројектно заштићеног софтвера*“ (енг. *Secure by Design*), којим се заштита посматра као основни нефункционални захтјев, који повезује и прожима сваку од фаза развоја. Изостанак оваквог приступа у индустријским пројектима неријетко доводи до озбиљних сигурносно-безбједносних пропуста, попут “*Heartbleed*” рањивости у *OpenSSL*-у или неовлашћеном приступу подацима у великом броју *SaaS* платформи. У свим тим сценаријима је неблаговремени рад на сигурности и безбједности резултовао компромитовањем осјетљивих података, губитком повјерења корисника, а и знатним финансијским трошковима. *Рад на заштити* (енг. *Security Engineering*) није само посао стручњака за заштиту информација, већ је то једним дијелом и одговорност сваког софтверског инжењера, па и самих програмера (чак и почетника!).

3.4.1. OWASP Top 10 као полазна тачка

Можемо рећи да се данас, као основни водич за најчешће и најкритичније сигурносно-безбједоносне ризике у веб-апликацијама, *IT* индустрија се добрим дијелом ослања на широко познати *OWASP Top 10*. *Open Web Application Security*

Project (скраћено *OWASP*) је јавна заједница која периодично ажурира поменуту листу. Сматрамо да је важно напоменути (у духу критичког осврта) да *OWASP Top 10* није свеобухватна листа свих могућих врста напада, већ *документ за освјешћење* (енг. *awareness document*) који је од велике помоћи програмерским тимовима да се фокусирају на најкритичније пропусте. Неки од кључних пропуста који се редовно појављују на листи укључују:

- *лош надзор приступа* (енг. *Broken Access Control*), под којим се подразумијева омогућавање корисницима да приступе подацима или изврше акције које нису ауторизоване за њихов ниво привилегија;
- *криптографски пропусти* (енг. *Cryptographic Failures*), под којим се подразумијева неадекватно чување осјетљивих података (лозинке, лични подаци) у чистом тексту или коришћење слабих алгоритама;
- *напади убризгавањем/инјекцијом* (енг. *Injection Attacks*), под којим се подразумејва слање непоузданих података (које уноси сам корисник) непосредно интерпретеру (нпр. бази података, оперативном систему, или *HTML*-страници) без одговарајуће обраде;
- *незаштићен пројекат* (енг. *Insecure Design*), под којим се подразумејвају материјални пропусти у архитектури, тј. они који се не могу "поправити" потоњом реализацијом.

3.4.2. Заштитне праксе језика Гоу

Језик Гоу, због своје философије једноставности и јасних стандардних библиотека, нуди одличне механизме за превенцију најчешћих напада.

Превенција *SQL* инјекције (енг. *SQL Injection*). Напад *SQL* инјекцијом настаје када се кориснички унос директно угради у *SQL*-упит/захтјев, омогућавајући нападачу да измијени логику упита/захтјева.

Примјер 3.3. (Неповољан (рањив) приступ). Дат је фрагмент кода који осликава неповољан, тј. видно рањив приступ.

```
package main

import (
   "database/sql"
      "fmt"
)

// User структура за чување података о кориснику
type User struct {
   ID    int
   Name  string
      Email string
}

// НИКАДА ОВАКО НЕ РАДИТИ!
// 'userID' долази директно из HTTP захтјева (нпр. "105 OR 1=1")
func GetUser(db *sql.DB, userID string) (*User, error) {
   // РАЊИВО: Кориснички унос се лијепи директно у упит.
   query := fmt.Sprintf("SELECT * FROM users WHERE id = %s", userID)

   // Нападач може промијенити 'userID' у "105 OR 1=1" и добити приступ свим корис
ницима
   // Или у "105; DROP TABLE users; --" и обрисати табелу
   // Или у "105; UPDATE users SET password = 'hacked' WHERE 1=1; --" и измијенити лоз
инке
```

```
    rows, err := db.Query(query)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    var user User
    if rows.Next() {
        err := rows.Scan(&user.ID, &user.Name, &user.Email)
        if err != nil {
            return nil, err
        }
        return &user, nil
    }

    return nil, sql.ErrNoRows
}
```

Пожељнији (бољи) приступ у Гоу је database/sql пакет који подржава *припремљене захтјеве* (енг. *prepared statements*). Упит/захтјев и подаци се шаљу бази одвојено, чиме драјвер базе података врши неопходну обраду и спрјечава да се кориснички унос тумачи као *SQL*-исказ.

Примјер 3.4. (Приступ који нуди бољу заштиту). Коришћење припремљених захтјева, како би се побошљала заштита.

```
package main

import (
    "database/sql"
)

type User struct {
    ID       int
    Username string
    Email    string
}
```

```
// ИСПРАВНО: Коришћење параметара (placeholders)
func GetUser(db *sql.DB, userID string) (*User, error) {
    query := "SELECT id, username, email FROM users WHERE id = ?"

    // Подаци (userID) се шаљу као одвојени аргумент.
    // Драјвер базе ће безбједно обрадити унос.
    row := db.QueryRow(query, userID)

    var u User
    if err := row.Scan(&u.ID, &u.Username, &u.Email); err != nil {
        return nil, err
    }
        return &u, nil
}

// Примјер за PostgreSQL
func GetUserPostgreSQL(db *sql.DB, userID string) (*User, error) {
    query := "SELECT id, username, email FROM users WHERE id = $1"

    row := db.QueryRow(query, userID)

    var u User
    if err := row.Scan(&u.ID, &u.Username, &u.Email); err != nil {
        return nil, err
    }
        return &u, nil
}

// Примјер са више параметара
func GetUserByCredentials(db *sql.DB, username, password string) (*User, error) {
    query := "SELECT id, username, email FROM users WHERE username = ? AND password =
 ?"

    row := db.QueryRow(query, username, password)

    var u User
    if err := row.Scan(&u.ID, &u.Username, &u.Email); err != nil {
        return nil, err
    }
        return &u, nil
}
```

```
// Пример са именованим параметрима (за сложеније упите)
func UpdateUser(db *sql.DB, userID int, username, email string) error {
    query := `
        UPDATE users
        SET username = ?, email = ?, updated_at = CURRENT_TIMESTAMP
        WHERE id = ?
    `

    _, err := db.Exec(query, username, email, userID)
        return err
}
```

Превенција *Cross-Site Scripting*-а (*XSS*-а). *XSS* напад настаје када нападач успије да убризга злонамјерна клијентска скрипта (рецимо, написана на *JavaScript*-у) у веб-страницу коју прегледа други корисник.

Гоуова стандардна библиотека html/template је осмишљена са примарним циљем предупрјеђења *XSS*-а. За разлику од text/template, html/template, она омогућује да се врши *контекстно аутоматско "избјегавање"* (енг. *context-aware auto-escaping*). То значи да пакет разумије да ли се податак убацује у *HTML*, унутар *JavaScript* блока, или као *URL*-атрибут, и примјењује одговарајућа одбрамбена/заштитна правила.

Примјер 3.5. Сљедећи фрагмент кода је видно рањив из угла *XSS*-а (text/template):

- ако се користи text/template за формирање *HTML*-а import "text/template"
- 'comment' долази из базе, а уписао га је нападач:

"<script>alert('XSS Напад!');</script>"

```
import "text/template"

tmpl := `Коментар: {{.Comment}}`
```

```
data := map[string]string{"Comment": "<script>alert('XSS')</script>"}
```

Овај код ће дати потенцијално опасан *HTML* (*// Коментар: <script>alert('XSS')</script>*):

- text/template ће слијепо убацити ову знаковну ниску у *HTML*;
- резултат ће бити извршавање *JavaScript*-а у прегледачу жртве.

Слиједи побољшани приступ уз коришћење html/template import "html/template"

```
import "html/template"

tmpl := `Коментар: {{.Comment}}`
data := map[string]string{"Comment": "<script>alert('XSS')</script>"}
// Генерише ескејпован HTML:// Коментар: &lt;script&gt;alert(&#39;XSS&#39;)&lt;/script&gt;
```

- html/template ће аутоматски избјећи тумачење;
- Резултат на *HTML*-у ће бити:

<script>alert('XSS Напад!');</script>

... што прегледач неће извршити, већ само приказати као текст. ▲

Превенција *Cross-Site Request Forgery*-а (*CSRF*). *CSRF* (или *XSRF*) је напад који приморава потврђеног корисника да изврши нежељену акцију на веб-апликацији. Нападач припреми линк или форму на свом сајту која, када је корисник (који је пријављен на нашу апликацију) отвори, шаље захтјев нашем серверу (нпр. "промијени лозинку" или "пребаци новац") са корисниковим *колачићима* (енг. *cookies*).

Гоу нема уграђену *CSRF* заштиту у net/http пакету, јер она захтјева *управљање сесијом* (енг. *session management*). Најчешћа одбрана је коришћење *Anti-CSRF* токена:

- када сервер прикаже осјетљиву форму (нпр. за промјену лозинке), он генерише јединствени, тајни токен и угради га у форму као скривено поље;
- када корисник пошаље форму (*POST* захтјев), сервер провјерава да ли примљени токен одговара оном који је претходно генерисао за тог корисника;
- нападач тешко може да сазна какав је овај токен, тако да захтјев са његовог злонамјерног сајта не би требао проћи провјеру ваљаности.

У пракси се за ово најчешће користе већ готова рјешења (библиотеке), као што је gorilla/csrf пакет, који се интегрише са net/http рутером.

3.5. Критички осврт: Академска формализација насупрот стварном разговору са клијентом

Слободни смо да кажемо да у данашњој академској литератури често се инсистира на прецизној формализацији захтјева кроз дијаграме, шаблоне и табеле. Међутим, у пракси, суштина лежи у *комуникацији* и *повјерењу* између развојног инжењера и клијента. Сами разговори су често неформални, импровизовани, али и продуктивнији од бирократског попуњавања шаблона. Добар инжењер мора знати да препозна

шта клијент стварно жели, па и када то сам клијент не умије јасно изразити. На неки начи, и колоквијално речено, инжењер треба да буде и добар психолог.

Због свега тога овај уџбеник заступа став да *процес није замјена за мишљење*, а *UML* и корисничке приче треба користити као оруђа, тј. не као "догме". Као што смо могли да видимо из студије случаја који се односи на е-трговину, нејасни захтјеви нису тек теоријски проблем, него се они лако претварају у седмице разнообразног састављања кода наново, и наново, и наново, уз, наравно, незадовољне клијенте. Практично рјешење лежи у поменутим *INVEST* начелима и конкретним критеријумима прихватања.

3.6. Закључак поглавља

Рад са захтјевима је темељ успјешног софтверског инжењерства. Да парафразирамо хигијенску крилатицу, *јасан захтјев је пола рјешења*. Како су показала многа истраживања, па и наше студије случаја, нејасни функционални захтјеви и занемаривање нефукционалних захтјева могу довести до поприличних проблема када је ријеч о развоју и коришћењу софтвера. Опет, то води ка неочекиваним дорадама, поправкама, што за посљедицу има и незадовољство клијената и ниже повјерење у производ. Циљ рада са захтјвима клијената није само у документовању, већ у разумијевању клијената, њихових жеља,

али и њихових неразумијевања техничких могућности. Неријетко кардиналне грешке настају у ствари због лоше комуникације.[39]

3.7. Питања и задаци за провјеру знања

1. Објаснити разлику између функционалних и нефункционалних захтјева.
2. Навести најмање три технике прикупљања захтјева и кратко их описати.
3. Размотримо практичну вјежбу. Замислимо да нам је дат задатак да прикупимо захтјеве клијената који се односе на једну здравствену апликацију која мора да ради како са подацима пацијената, тако и одређеним општим административним подацима. Коју од поменутих горе техника бисмо могли да комбинујемо?
4. Шта нам то представља *UML Use Case* дијаграм?
5. Написати примјер једне корисничке приче.
6. У чему лежи разлика између академског и практичног приступа у управљању захтјевима?
7. Поправити захтјев "Направимо добар search", у три корисничке приче са критеријумима прихватања.

[39] Овдје бисмо могли додати и једну чувену реченицу политиколога Јуџина Луџиса Фордсворта (енг. *Eugene Lewis Fordsworthe*): "Претпоставка је мајка свих брљотина." Изворна верзија ове мисли је мало непристојнија.

4. МОДЕЛОВАЊЕ И ОБРАСЦИ

Сврха поглавља

У овом поглављу моделовање и провјерени архитектонски и пројектни обрасци сагледавају се као основа за пројектовање софтвера који се лакше мијења, тестира, и одржава.

4.1. Појам моделовања у софтверском инжењерству

Замислимо ситуацију у којој тим програмера, на самом крају реализације софтвера, увиђа кардиналну грешку у коду. Дата грешка има за резултат велике финансијске трошкове и кашњења у испоруци софтвера као производа. Е, ова, нажалост не баш нешто ријетка ситуација, намеће сама по себи сљедеће питање: како спријечити такве проблеме у раним фазама развоја? Један од приступа да се одговори на ово питање, да се ријеши овакав проблем, јесте моделовање.

Моделовање у софтверском инжењерству је процес поједностављеног приказивања стварног или планираног система ради бољег разумијевања, сагледавања или размјене мишљења међу члановима тима. *Модел* је уопштење стварног, тј. не сам систем, већ његово довољно тачно представљање да може служити као основа за пројектовање и реализацију. Добар модел није декоративан, већ функционалан: он омогућава да се грешке уоче прије него што настану у коду.

Главни *циљеви моделовања* када је ријеч о софтверском инжењерству су:

1. *разумијевање* → помаже програмерским тимовима да сагледају и ширу слику и односе међу ставкама;

2. *комуникација* → модели служе као својеврстан универзални језик носилаца пројекта, програмера, аналитичара и самих клијената;

3. *потврда* → омогућава провјеру задовољења захтјева прије уласка у фазу реализације;

4. *документација* → модели често остају као трајни запис архитектуре софтверског рјешења.

Иако се у пракси појмови моделовање и пројектовање често користе заједно, они представљају двије потпуно различите фазе размишљања и самог рада на софтверу. *Моделовање* нам описује шта систем јесте или шта треба да буде (нпр. појмови у домену, њихови односи, ограничења и сценарији употребе). То је ниво разумијевања проблема, независан од начина реализације. *Пројектовање*, с друге стране, одговара на питање како ће систем бити изграђен, тј. које структуре, који обрасци, који модули, прочеља и зависности требају бити дефинисани да би улазни модел могао да се реализује на ваљан начин. За брзо утврђивање разлике, можемо се послужити сљедећом хеуристиком: *ако је језички независно, онда је моделовање, али ако пак зависи од технологије или кода, онда је пројектовање.*

Другим речима, моделовање се бави доменом,[40] док се пројектовање бави софтверском структуром. Добро пројектовање је могуће само ако је модел јасан, а лош модел скоро увијек доводи до лошег пројекта, а и сложеног и веома "кртог" кода. Важно је нагласити да се *UML* може користити у објема фазама, али са битно различитим улогама: у моделовању као средство за разумијевање, а у пројектовању као средство за структурирање рјешења.

4.2. *UML* дијаграми у служби моделовања

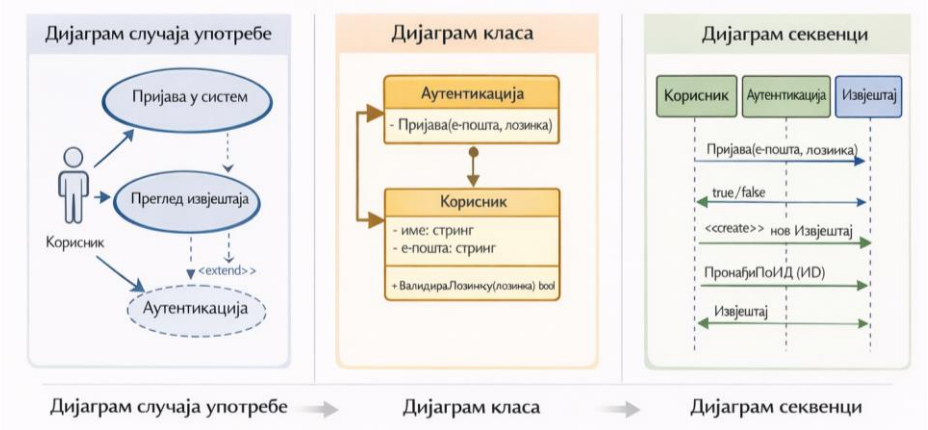
Као што смо то већ рекли, *UML* представља стандардизован визуелни језик за опис, спецификацију и документовање софтверских система. Његова предност лежи у томе што омогућава да се сложени системи представе кроз више комплементарних перспектива: структурну, понашајну и међудејствовну.

С обзиром на искуства саврмене праксе, треба указати на то да *UML* није циљ, већ средство моделовања. Различити дијаграми одговарају различитим питањима која постављамо у раној фази пројектовања. *UML* није листа дијаграма нити сет графичких правила, већ је то је језик за моделовање који омогућава инжењеру да систем прикаже из различитих перспектива. Сваки *UML* дијаграм одговара на неко конкретно питање у самом процесу моделовања:

[40] Прво направите домен. Онда направите модел. Онда се запитате да ли модел заиста одражава домен или само ваше почетно неразумијевање домена. Најчешће је обоје.

- Ко користи систем и са којим циљем?
- Како се компоненте понашају у току извршавања?
- Како је систем структурно организован?
- Које су зависности између дијелова система?

Због тога је важно разумјети врсте *UML* дијаграма не као формалну класификацију, већ као оруђа за подршку различитим аспектима моделовања у оквиру раних фаза *SDLC*-а. У наставку слиједи преглед најчешће коришћених *UML* дијаграма и врста питања за које они помажу да се разјасне током анализе и пројектовања.



Слика 4.1. На слици је један исти софтверски систем приказан кроз дијаграм случаја употребе, дијаграм класа, и дијаграм секвенци, чиме се на пластичан начин илуструје да сваки дијаграм освјетљава различит аспект исте стварности.

Сами дијаграми се уобичајено дијеле на *структурне* (који описују статичку организацију система) и *понашајне* (који приказују динамичко понашање, токове и садејства). Можемо замислити структурне дијаграме као архитектонске планове

зграде: дефинишу трајну организацију и распоред простора. С друге стране, понашајне дијаграме можемо да доживимо као својеврсне дневне операције унутар саме зграде: они показују како се простори и ресурси користе у различитим ситуацијама. Овим, пластичнијим начином предочавања њихових разлика, разумијевање статичких и динамичких погледа на софтверски систем постаје тренутно интуитивно.

Структурни дијаграми:

1. *Дијаграм класа* (енг. *Class Diagram*). Служи да се прикажу класе, њихове атрибуте, методе и односи као што су наслеђивање, агрегација и асоцијација. Овај дијаграм се обилато користи за објектно-оријентисанно пројектовање.

2. *Дијаграм компоненти* (енг. *Component Diagram*). Приказује архитектонску структуру система кроз модуле или компоненте, као и њихова прочеља. Користи се за разумијевање зависности између различитих дијелова кода.

3. *Дијаграм распореда* (енг. *Deployment Diagram*). Показује како су компоненте софтвера распоређене на конкретној физичкој инфраструктури: серверима, клијентима (рачунарима) или у (рачунарском) облаку. Веома је важан у фази пројектовања вишеслојних система (нпр. веб-апликације).

Дијаграми понашања:

1. *Дијаграм случаја употребе* (енг. *Use Case Diagram*). Приказује спољашње учеснике (кориснике, системе) и њихов однос са системом. Овај дијаграм најбоље одговара раним фазама анализе захтјева.

2. *Дијаграм секвенци* (енг. *Sequence Diagram*). Он нам илуструје редослијед порука између објеката током извршавања једног конкретног сценарија. Од користи је да побољша разумијевање динамике система и међусобне сарадње самих ставки.

3. *Дијаграм активности* (енг. *Activity Diagram*). Приказује ток активности, гранања, као и паралелних токова у процесима. Обично га користимо за моделовање пословне логике или представљање алгоритама.

У пракси, три најчешће коришћена *UML* дијаграма су *Use Case, Class* и *Sequence* дијаграми, јер заједно пружају јасан преглед шта систем ради, ко учествује у читавом процесу, те како тече сама комуникација.

Када је ријеч о напреднијим пројектима, додају се природно и други дијаграми (нпр. дијаграм стања или ставки), али за основно разумијевање *UML*-а ове три врсте су неријетко сасвим довољне.

UML дијаграми имају највећу вриједност када се користе као оруђа за размјену мишљења, а не као формални документи. У индустријској пракси, *UML* се најчешће користи у живом разговору, на табли током пројектовања, у пару програмирања или на удаљеним састанцима за разјашњавање сложених идеја. Брзо нацртан дијаграм класа може спријечити недеље погрешне реализације, а једноставан секвентни дијаграм може разјаснити ток података који би у коду био тешко видљив. Кључно је то да дијаграм треба да послужи да нам одговори на конкретно питање или разјасни одређени аспект система, а не да буде потпун и

савршен. Добар инжењер зна када нацртати дијаграм за пет минута да би уштедио читав час дискусије.

Практична вриједност *UML*-а огледа се у томе да ли након његове употребе тим заиста боље разумије систем, да ли има мање грешака при реализацији и да ли је укључивање нових чланова брже (енг. *on-boarding*). Ако дијаграм не појашњава никоме ништа, онда је његово цртање само узалудни трошак времена. С друге стране, ако је од користи тиму да разумије причу на коју се ослања развој софтвера, и да тиме допринесе дјелотворности цијелог тима, онда је, јасно, његова сврха испуњена.[41] *UML* је добро оруђе које може да буде од помоћи да се те границе виде и да се домен правилно организује прије него што настане *архитектонски дуг.* Под овим појмом архитектонског дуга подразумијевамо системски ризик настао када архитектонске одлуке краткорочно оптимизују испоруку, а дугорочно смањују способност система да се мијења без експоненцијалног трошка.

Иако *UML* пружа формалан језик за опис структуре и понашања софтверских система, ипак, сам по себи он не одговара на питање шта тачно моделујемо и зашто су одређене границе у систему постављене управо тако. Управо на том мјесту појављује се *доменом вођено пројектовање* (енг. *Domain-Driven Design* (*DDD*)) као приступ који усмјерава пажњу са техничке организације кода на појмове и правила стварног домена који систем треба да подржи. У контексту *UML*-а, *DDD* не уводи нову нотацију, већ даје смисао постојећим дијаграмима: класе више нису апстрактне техничке конструкције, већ представљају појмове из домена;

[41] Дијаграм је успјешан за програмера онолико колико је и програмер успјешан без дијаграма.

асоцијације и агрегације одражавају стварне односе; а границе између пакета и подсистема постају архитектонски значајне, а не само организационе. На тај начин, *UML* модели престају да буду пука документација и постају средство за очување заједничког разумијевања система кроз вријеме. Доменом вођено пројектовање стога није алтернатива *UML*-у, већ начин да се *UML* користи сврсисходно, као одраз доменске логике, а не као самостална техничка вјежба.

4.2.1. Студија случаја: Лоша апстракција у банкарском систему

Проблеми лошег кода скоро увијек потичу од лошег моделовања самог домена. *UML* дијаграми класа управо служе да идентификују апстракције и односе у систему прије него што се напише прва линија кода. Самим тим, ако је модел погрешан, код ће бити нефлексибилан, крт и тежак за проширивање. У сљедећим редовима приказана је типична грешка у моделовању, а то је да се све врсте банкарских трансакција представљају се једном класом. Гледано из угла *UML*-а, ово у ствари говори да је направљена једна превелика апстракција која покрива више различитих понашања која нису у сродству једно с другим. За посљедицу имамо експлозију условних грана у коду. Исправка се добија примјеном *стратешког обрасца* (енг. *Strategy Pattern*), који у *UML*-у изгледа као прочеље са више конкретних реализација. Овај модел омогућава да се свако понашање изолује у посебну класу, те да систем постане проширив без измјена у постојећем коду.

Проблем. Сви типови трансакција имају исту класу.

```
// ЛОШЕ
```

```
type Transaction struct {
    ID          int
    Amount      float64
    FromAccount string
    ToAccount   string
    Type        string
    Fee         float64
    Currency    string
    Timestamp   time.Time}

// Цјелокупна логика у једној функцији
func ProcessTransaction(t Transaction) error {
    switch t.Type {
    case "transfer":
        // 50 линија кода
    case "withdrawal":
        // 40 линија кода
    case "deposit":
        // 30 линија кода
    }
}
```

Додавање нове врсте трансакције захтијева мијењање свих постојећих функција, што за посљедицу има висок ризик од грешака. Рјешење видимо у бољем пројектовању.

```
// МАЛО БОЉЕ (Strategy pattern)
type Transaction interface {
    Execute() error
    Validate() error
    GetAmount() float64}

type Transfer struct {
    FromAccount *Account
    ToAccount   *Account
    Amount      float64
    Fee         float64}

func (t Transfer) Execute() error {
        // Специфична логика за трансфер
}
```

```
type Withdrawal struct {
    Account *Account
    Amount  float64
      ATMID   string
}

func (w Withdrawal) Execute() error {
      // Специфична логика за исплату
}
```

4.2.2. Антистудија случаја: Игнорисање доменом вођеног пројектовања

Један од најчешћих узрока архитектонских проблема није сам код, него лоше моделовање домена. Ако се ентитети организују искључиво "технички" (нпр. један огроман *models* пакет), тим губи увид у границе домена, нејасне су зависности, и тешко је рећи шта је заправо извор истине (енг. *source of truth*) у читавом систему.

Ова наша мала *антистудија случаја*,[42] као мало боље разрађен примјер, служи да демонстрира грешку која се често јавља у систему за управљање пројектима, а то је да су сви ентитети спаковани у један пакет,[43] без икаквог структурног разграничења домена (корисници, пројекти, задаци, наплата). Другим ријечима, све класе су у истој семантичкој групи, иако припадају различитим граничним контекстима. Посљедица

[42] Класична студија случаја показује шта је рађено и зашто. У нашем случају, указујемо на то шта се може десити (и то на конкретном примјеру) када нешто није урађено.

[43] Ово, иако блиско, ипак није у вези са чувеном искуственом смјерницом из оптимизације портфеља: "Никада не треба стављати сва јаја у једну кошару."

оваквог приступа је изражено међусобно преплитање ентитета, те немогућност ваљаног тестирања према доменима, а што у крајњој инстанци резултује високим трошковима измјена.

DDD исправља овај модел уводећи јасне домене са сопственим ентитетима, услугама и изворима истине. *UML* дијаграм испод показује како исправно моделирање односа између *Project*, *Task* и *Invoice* елиминише погрешне зависности и омогућује флексибилније понашање када је ријеч о домену наплате.

Проблем. Сви ентитети у једном models пакету

```
models/
├── user.go
├── project.go
├── task.go
├── comment.go
├── invoice.go
└── notification.go
```

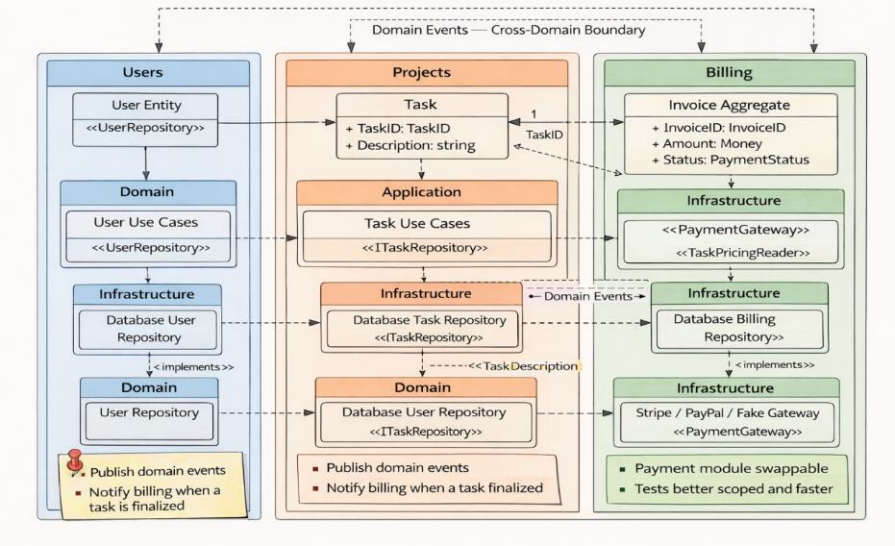
Ово за посљедицу има непрактичну зависност кода за *Invoice,* те је немогуће замијенити модул за плаћање, а тестирање захтијева симулацију читавог система. Другим ријечима, имамо одвише незгодну ситуацију за одржавање.

Једно од рјешења, које се природно намеће, лежи у сљедећој доменима вођеној архитектури (на слици 4.2 се налази одговарајући *UML* дијаграм)[44]

```
project/
├── cmd/
|   └── api/
|       └── main.go
```

[44] Занемарићемо страница ради питања која се односе на то да ли *Projects* одређује наплативост *Task*-а (нпр. *Rate*, *billable flag*) и шаље *Billing*-у, или пак *Billing* има своја правила/политику.

```
├── internal/
│   ├── users/
│   │   ├── domain/
│   │   │   ├── user.go
│   │   │   └── repository.go
│   │   ├── application/
│   │   │   └── service.go
│   │   └── infrastructure/
│   │       ├── http/
│   │       └── postgres/
│   ├── projects/
│   │   ├── domain/
│   │   │   ├── project.go
│   │   │   ├── task.go
│   │   │   ├── ids.go            # TaskID, ProjectID
│   │   │   └── events.go         # TaskBillableFinalized (опционо)
│   │   ├── application/
│   │   │   ├── task_service.go
│   │   │   └── pricing_snapshot.go # DTO експортовано у ACL (опционо)
│   │   └── infrastructure/
│   │       ├── http/
│   │       └── postgres/
│   ├── billing/
│   │   ├── domain/
│   │   │   ├── invoice.go        # Invoice има TaskID (локалног типа)
│   │   │   ├── line_item.go
│   │   │   ├── money.go
│   │   │   └── repository.go
│   │   ├── application/
│   │   │   ├── create_invoice.go
│   │   │   ├── pay_invoice.go
│   │   │   ├── ports.go          # PaymentGateway, TaskPricingReader
│   │   │   └── handlers.go       # управљање догађајима
│   │   └── infrastructure/
│   │       ├── postgres/
│   │       ├── stripe/
│   │       ├── fake/             # тестирање gateway-а
│   │       └── messaging/
│   └── shared/
│       ├── events/               # мале апстракције система догађаја
│       └── auth/                 # попречне функционалности
└── go.mod
```



Слика 4.2. *UML* дијаграм који "спасава"[45] пројекат. Можемо примијетити да је ентитет *Invoice* везан за *Task,* умјесто за цјелокупни *Projects*.

Измјене у разради софтверског рјешења омогућиле су флексибилније наплаћивање.

4.3. Архитектонски и пројектни обрасци

Како софтверски системи крену да расту, повећава се, природно, и сложеност њиховог пројектовања. *Архитектонски и пројектни обрасци* представљају формализоване, поновљиво корисне начине рјешавања уобичајених проблема у софтверским системима. Они не замјењују моделовање, нити пројектовање, него служе као искуствено провјерени радни оквири да убрзавају развој и смање ризик од погрешних архитектонских одлука.

[45] Ипак, да буде казано, не треба претјеривати са оваквим *спасилачким* терминима и тешким ријечима.

4.3.1. Архитектонски обрасци

Архитектонски обрасци (енг. *Architectural Patterns*) дефинишу структуру система на највишем нивоу. Они треба да одреде како су компоненте организоване, како међусобно комуницирају, и како се разлучују одговорности. Уобичајени архитектонски обрасци укључују:

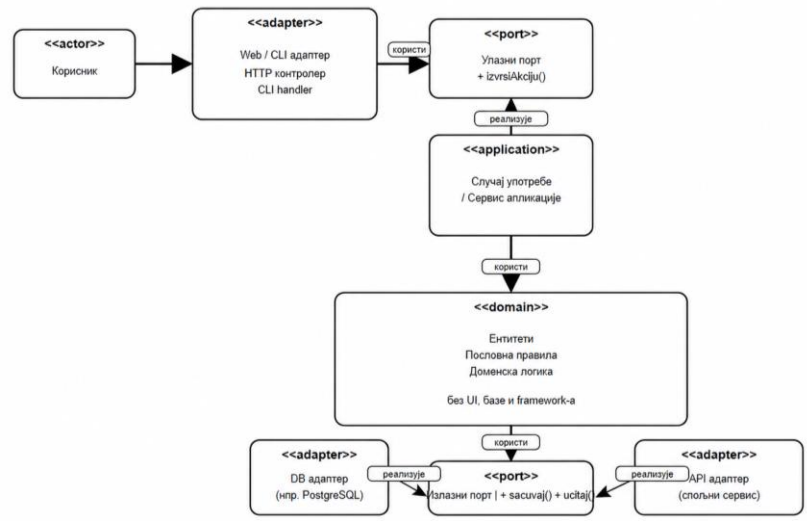
- *Слојевиту архитектуру* (енг. *Layered Architecture*). Раздваја се корисничко прочеље, пословна логика, и сами подаци. Користимо га у класичним корпоративним апликацијама као што је, рецимо, банкарски софтвер, у којима су корисничко прочеље, пословна логика, и приступ бази података стриктно раздвојени.

- *Клијент–сервер* (енг. *Client-Server*). Овај образац (а и концепт) раздваја клијентску логику од централизованог сервера. Познато нам је из курсева који се односе на веб-технологије и интернет-програмирање да је он практично основа за већину веб-апликација данас.

- *Микросервисна архитектура* (енг. *Microservices*). Према овом обрасцу сам систем чине мале, али независне услуге (сервиси). Овај образац се често примјењује у савременим глобалним сервисима попут *Netflix*-а, код којих је свака услуга независна и може се побољшати или скалирати засебно.

- *Архитектура Кокбернових шестоуглова* (енг. *Ports & Adapters*).[46] Код овог обрасца нагласак је на одвајању домена од инфраструктуре (техничких детаља), тако да пословна правила система остају независна од корисничког прочеља, базе података, и околних инфраструктурних механизама који се према језгру односе као замјењиви адаптери. Примјер коришћења је у системима за управљање плаћањима у којима је јасно раздвојен домен саме апликације од њене везе са спољним свијетом (системима).

Архитектонски образац утиче на скалабилност, провјерљивост, модуларност, и одрживост система.



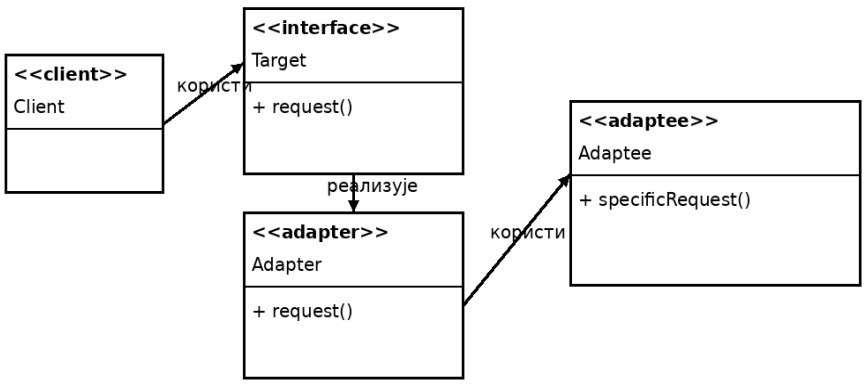
[46] Оригинални (функционални) назив је *Ports & Adapters* који је дао Алистер Кокберн (енг. *Alistair Cockburn*) 2005. г. Назив описује механизам којим апликација комуницира са спољним свијетом. Назив хексагонална архитектура (визуелни назив) се усталио јер је Кокберн користио шестоугао (хексагон) за цртање дијаграма ове архитектуре. Другим ријечима, није ствар у броју шест. Апликација не мора да има тачно шест страна или шест врста веза. Кокберн је просто користио шестоугао да би избјегао традиционални "слојевити" приказ гдје све иде одозго на горе. Термин "чиста" је касније популаризовао Роберт Мартин (енг. *Robert Martin*).

Слика 4.3. Омањи *UML*-ски показни примјер за *Ports & Adapters*. Улазни адаптери (нпр. веб-прочеље или *CLI*) приступају систему преко улазног порта, који реализује апликациони слој. Доменско језгро садржи пословна правила и остаје независно од инфраструктуре. Приступ бази података и спољним сервисима иде преко излазних портова, чије реализације дају конкретни адаптери. На тај начин технички детаљи остају замјењиви, а доменска логика остаје стабилна и прегледна.

4.3.2. Пројектни обрасци

Пројектни обрасци (енг. *Design Patterns*) нам служе да њима ријешимо мање, али често понављајуће проблеме унутар архитектуре. Класична класификација образаца према *Gang of Four*[47] (*GoF*) обухвата сљедеће обрасце:

- *креационе* (*Singleton, Factory, Builder*);
- *структурне* (*Adapter, Composite, Decorator*);
- *понашајне* (*Observer, Strategy, Command*).



[47] Мисли се на четири аутора, а не четири класе образаца. Ријеч је о књизи "*Design Patterns: Elements of Reusable Object-Oriented software* (1994)".

Слика 4.4. Примјер адаптерског обрасца (прост *UML*-ски приказ) за *Client -> Target <- Adapter -> Adaptee*.

У суштини, пројектни обрасци повећавају читљивост, смањују дуплирање логике, и олакшавају измјене у самом систему.

4.3.3. Обрасци у екосистему језика Гоу

Гоу не користи класичан модел насљеђивања, што значи да се пројектни обрасци често реализују кроз композицију (умјесто насљеђивања), кроз прочеља са минималистичким уговорима, или пак помоћу пакета као јединица модуларности.

Због тога Гоу природно подржава обрасце као што су *Strategy* (преко прочеља), *Adapter* (преко композиције) и *Singleton* (преко пакета и иницијализације).

Примјер 4.1. Једноставна архитектура веб-апликације на Гоу се може представити на сљедећи начин:

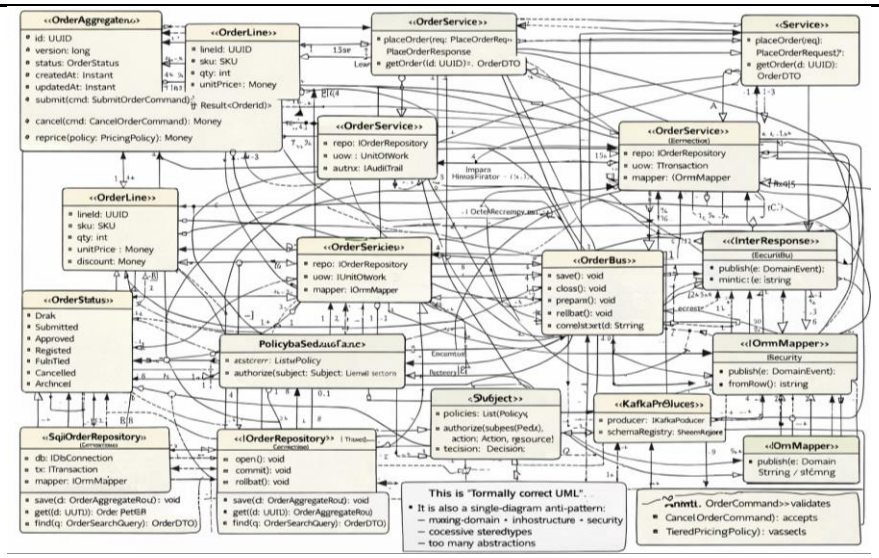
```
project/
│
├── cmd/app/main.go
├── internal/handlers/user.go
├── internal/services/user_service.go
├── internal/repositories/user_repo.go
└── pkg/models/user.go
```

За сваки од ових слојева важи да он има своју, такорећи, одговорност: `handlers` прихвата *HTTP* захтјеве и шаље одговоре, `services` садржи пословну логику, а `repositories` управља приступом бази података. ▲

Овакав приступ подстиче слабију спрегу и једноставније тестирање компоненти.

4.4. Критички осврт: Академска апстракција насупрот индустријској стварности

У пракси развоја софтвера врло брзо се покаже да претјерано збијени и међусобно испреплетени домени у оквиру једног монолитног пакета стварају проблеме управо онда када систем треба мијењати. Оно што је у почетку дјеловало једноставно, рецимо замјена модула за плаћање или прилагођавање новим пословним захтјевима клијента, временом постаје сложено, скупо и ризично. Такав развој догађаја није ни риједак ни неочекиван. У стручној и академској литератури већ се одскора указује на то да сложени системи са нејасно повученим архитектонским границама имају природну склоност ка расту сложености и појави модела који више отежавају него што олакшавају његово разумијевање. Посебно је то уочљиво у областима које се тичу безбједности и сигурности, гдје непрецизно дефинисане границе система често доводе до збрке у тумачењу и примјени различитих модела у пракси. У таквом окружењу постаје тешко не само формално описати рјешење, већ га поуздано потврдити и упоредити са алтернативним приступима.



Слика 4.5. Помало искарикиран примјер *UML* дијаграма који је формално исправан, али практично неупотребљив због прекомјерне сложености. Дијаграм илуструје опасност од хиперформализације.

Иако академски модели пројектовања софтвера наглашавају формалне дијаграме, јасне границе, те систематичну документацију, ипак индустријска пракса је видно више усмјерена на програмски код. Дијаграми се користе само ако ће да заиста помогну разумијевању, тј. као привремено средство, али не и као административна обавеза. Системи ријетко “пуцају” због “мањка *UML*-а“, али често се озбиљни багови јављају због лоше структурираних апстракција. У наставку ћемо издвојити, према нашем мишљењу, најчешће “архитектонске назови-обрасце” (антиобрасце) који настају када се теорија примењује без “прљања руку”, или када се пресликава “индустријска мода” без разумијевања самог домена.

1. “1000 малих датотека“ као антиобразац сложености. На први поглед, многи тимови настоје да постигну “чистоћу“ раздвајањем кода у што више ситних датотека. Без дисциплине у моделовању,

ово води до супротног ефекта. Рецимо, јавља се пораст когнитивне сложености (нико више не зна "гдје се шта налази"), ствара се хаотична мрежа међузависности, или је присутно одсуство јасних домена и граница, а и могућа је немогућност процјене утицаја промјене.

У архитектонском смислу, овај антиобразац је супротност добром *UML* моделовању: много датотека није исто што и много класа, а много класа није исто што и јасна апстракција. Ово је примјер лошег моделовања у коду, гдје се структурални проблеми маскирају такозваном *фрагментацијом пројекта*.

2. Хексагонална архитектура без дисциплине — идеал претворен у хаос. Хексагонална архитектура је снажан концепт, јер је јасна доменска логика окружена контролисаним прочељима. Међутим, без дисциплине тимова и без јаког доменског моделовања, она резултује хаотичним бројем "портова" чија намјена није јасна, гомилом "адаптера" који се дуплирају, доменској логици која може да "цури" у инфраструктуру, инфраструктуром која лако упрља домен, или пак *UML* дијаграмима који више ништа не разјашњавају.

Другим речима, архитектура неће спасити тим који нема модел. Хексагонална архитектура је оруђе, а не магични штапић, тј. без јасних ентитета, агрегата и домена, она само увећава хаос.

3. Када микросервис постане микрофеудализам. Микро-сервиси имају смисла само ако домени постоје и ако су границе (енг. *bounded contexts*) јасно моделоване. Без тога имамо да сваки тим "влада" својим сервисом као каквим феудом, а даље сваки сервис повлачи своју импровизовану верзију домена. Одавде, даље гледано, лако може да настане мрежа међусобних *RPC* позива без

икакве централизоване логике, те да зависности постану неуправљиве, а *UML* дијаграми компонената у суштини непримењљиви (јер систем нема неки разумљив облик). Просто, лоше моделовање домена → лоше архитектонске границе → микросервиси као микрофеудализам (много граница, али ниједна исправно успостављена). Стога, умјесто "савршеног модела", важно је тежити *јасном, досљедном и провјерљивом коду*. "Просто ко пасуљ!", каже наш народ.

4.5. Закључак поглавља

Моделовање и пројектовање су кључни кораци који спајају апстрактно и конкретно. Циљ не треба да буде стварање умјетничког дјела од *UML* дијаграма, већ да се добије разумљива основа за будући стабилан код. Добро пројектовање се мјери лакоћом промјене и тестирањем.

4.6. Питања и задаци за провјеру знања

1. Објаснити сврху моделовања у софтверском инжењерству.

2. Навести три најчешће коришћена *UML* дијаграма и њихову примјену.

3. У чему је разлика између архитектонских и пројектних образаца?

4. Описати једноставну архитектуру гоу-апликације користећи три слоја.

5. Критички анализирати предности и мане *UML* дијаграма у сопствено изабраним практичним пројектима.

6. Поправити лоше написану класу

```
// ДАТА ЛОША КЛАСА
type Order struct {
    ID int
    Items []string
    ShippingAddress string
    BillingAddress string
    PaymentMethod string // "card", "paypal", "bank_transfer"
    CardNumber string
    PayPalEmail string
    BankAccount string
    Status string // "pending", "paid", "shipped", "delivered"
}
// ЗАДАТАК: Рефактериши користећи принципе добре апстракције
```

4.7. Практични домаћи задатак: Развој система за резервацију терена

У овом поглављу дајемо практични домаћи задатак, релативно добро формално представљен. Ипак, овај домаћи задатак не треба узети као примјер добро написане спецификације софтверских захтјева (енг. *Software Requirements*

Specification (*SRS*)). Један, па можда чак идеалистичан, примјер доброг SRS-а је дат као Прилог B, овом уџбенику.

Циљ задатка: Треба осмислити и моделовати систем за резервацију спортских терена. Потом, у наредном кораку, реализовати кључне функције и написати одговарајуће јединичне тестове.

Дио 1. Моделовање система (*UML* дијаграмима)

Формирати *UML* дијаграм класа за систем који омогућава сљедеће редом:

- корисници требају да могу да прегледају доступне терене (фудбал, кошарка, тенис);
- резервацију термина на одређени датум и вријеме;
- приказ сопствених резервација;
- отказивање резервација.

Очекивани *UML* дијаграм класа треба да укључи сљедеће:

- класе *Korisnik, Teren, Rezervacija, Termin*;
- атрибуте и методе за сваку класу;
- потребне односе између класа (асоцијације, агрегације);
- основне типове података.

Дио 2. Реализовати кључне функције на Гоу.

```
// rezervacija.go
package main

import (
   "time"
)
```

```
type Termin struct {
   ID      int
   TerenID  int
   DatumVreme time.Time
   Trajanje time.Duration
   Zauzet   bool
}

type RezervacijaSistema struct {
   termini []Termin
}

// Провјерава да ли је терен слободан у одређеном термину
func (rs *RezervacijaSistema) ProvjeriDostupnost(terenID int, pochetak time.Time, trajanje tim
e.Duration) (bool, error) {
   kraj := pocetak.Add(trajanje)
      for _, termin := range rs.termini {
      if termin.TerenID == terenID && termin.Zauzet {
         terminKraj := termin.DatumVrijeme.Add(termin.Trajanje)

         if pocetak.Before(terminKraj) && kraj.After(termin.DatumVrijeme {
            return false, nil
         }
      }
   }
   return true, nil
}
```

Дио 3. Писање конкретних јединичних тестова. Написати тестове за функцију ProvjeriDostupnost користећи *TDD* приступ.

```
package main

import (
   "testing"
      "time"
)

func TestProvjeriDostupnost(t *testing.T) {
   sistemi := &RezervacijaSistema{
      termini: []Termin{
```

```
        {
            ID:         1,
            TerenID:    1,
            DatumVrijeme:  time.Date(2024, 1, 15, 10, 0, 0, 0, time.UTC),
            Trajanje:   time.Hour,
            Zauzet:     true,
        },
        {
            ID:         2,
            TerenID:    2,
            DatumVreme:  time.Date(2024, 1, 15, 14, 0, 0, 0, time.UTC),
            Trajanje:   time.Hour * 2,
            Zauzet:     true,
        },
    },
}

tests := []struct {
    name     string
    terenID   int
    pocetak   time.Time
    trajanje  time.Duration
    expected  bool
    shouldErr bool
}{
    {
        name:     "Слободан терен у слободном термину",
        terenID:   1,
        pocetak:   time.Date(2024, 1, 15, 16, 0, 0, 0, time.UTC),
        trajanje:  time.Hour,
        expected:  true,
        shouldErr: false,
    },
    {
        name:     "Заузет терен --- преклапање са почетком",
        terenID:   1,
        pocetak:   time.Date(2024, 1, 15, 9, 30, 0, 0, time.UTC),
        trajanje:  time.Hour,
        expected:  false,
        shouldErr: false,
    },
    {
        name:     "Заузет терен --- преклапање са крајем",
        terenID:   1,
        pocetak:   time.Date(2024, 1, 15, 10, 30, 0, 0, time.UTC),
        trajanje:  time.Hour,
        expected:  false,
        shouldErr: false,
    },
    {
        name:     "Различит терен --- слободан",
        terenID:   3,
```

```
        pocetak:   time.Date(2024, 1, 15, 10, 0, 0, 0, time.UTC),
        trajanje:  time.Hour,
        expected:  true,
        shouldErr: false,
      },
    }

    for _, tt := range tests {
      t.Run(tt.name, func(t *testing.T) {
        dostupno, err := sistemi.ProveriDostupnost(
                                              tt.terenID, tt.pocetak, tt.trajanje
                                          )

        if (err != nil) != tt.shouldErr {
          t.Errorf(
                          "ProvjeriDostupnost() грешка = %v,
                          очекивана грешка = %v",
                          err,
                          tt.shouldErr
                      )
          return
        }

        if dostupno != tt.expected {
          t.Errorf(
                          "ProvjeriDostupnost() = %v,
                          очекивано = %v",
                          dostupno,
                          tt.expected
                      )
        }
      }
      )
      }
}

func TestProvjeriDostupnost_EdgeCases(t *testing.T) {
    sistemi := &RezervacijaSistema{
      termini: []Termin{
        {
          ID:        1,
          TerenID:    1,
          DatumVreme:  time.Date(2024, 1, 15, 10, 0, 0, 0, time.UTC),
          Trajanje:    time.Hour,
          Zauzet:     true,
        },
      },
    }

    dostupno, err := sistemi.ProveriDostupnost(1,
      time.Date(2024, 1, 15, 11, 0, 0, 0, time.UTC),
      time.Hour,
```

```
    )

    if err != nil {
        t.Errorf("Неочекивана грешка: %v", err)
    }

    if !dostupno {
        t.Error(
                "Терен би требало да буде слободан када нова резервација почиње
                тачно на крају постојеће")
        }
}
```

5. РЕАЛИЗАЦИЈА

Сврха поглавља

У овом поглављу реализација софтвера се сагледава као дисциплиновано превођење пројектованог рјешења у читљив, провјерљив, и одржив програмски код.

5.1. Мјесто реализације у *SDLC*-у

Реализација је фаза развоја софтвера у којој се логички модел система преводи у извршни рачунарски код. Замислимо програмера током једног дана како у "спринту" (Скрама) сједи за рачунаром и практично претвара сложене дијаграме и архитектуре у линије и линије рачунарског кода.[48] Његови екрани (данас већ множина!) су испуњени прозорима са спецификацијама, и док куца, свака линија кода је један мали корак ка извршној верзији програма. Управо, овај корак је срж софтверског инжењерства. Овдје се идеје претварају у конкретно понашање производа, тј. програма. Но, сматрамо да треба нагласити да иако многи програмери реализацију доживљавају као централни и скоро па једини дио развоја, она је тек једна карика у ланцу који почиње захтјевима и моделовањем.

[48] А слободно се може претпоставити и да се кафа претвара на својеврстан начин у линије рачунарског кода.

Добра реализација није ствар броја линија кода, већ степена разумљивости, стабилности и провјерљивости. Квалитет реализације непосредно зависи од квалитета претходних фаза. На примјер, тимови који остварују висок степен разумљивости често биљеже смањену стопу кардиналних грешака, а што доводи до много мањег времена потребног за отклањање таквих грешака, и много краћег времена за испоруку софтвера. Неке емпиријске анализе показују већу (архитектонску и оперативну) стабилност са смањењем неочекиваног рада поново (енг. *rework*) и нижим трошковима реализације и одржавања, јер стабилније зависности и јасније границе умањују таласна дејства промјена. Ако је пројекат нејасан, реализација ће бити непредвидљива, фрагментирана, и пуна импровизација. У добро организованим тимовима, програмер није "аутор" система, већ реализатор пројектованих рјешења. Основна начела добре реализације су:

- *Јасноћа.* Код треба да буде читљив и логички организован.
- *Досљедност.* Користити јединствен стил писања и именовања.
- *Минимализам.* Избјегавати сувишне апстракције.
- *Изолација одговорности.* Функције и класе требају да имају јасну улогу.
- *Документација.* Коментари морају да објасне зашто, а не шта код "ради", при чему нарација није замјена за јасноћу;
- *Рефакторисање*[49]*.* повремено прочишћавање и побољшавање структуре (састава) кода без промјене његовог понашања (функционалности);

[49] Од енглеске ријечи *refactoring*, што значи "саставити опет".

- *Аутоматизација тестова.* Свака нова функција мора (или би требала?), када је то смислено, да има тест којим се провјерава њено очекивано понашање.

За потребе боље реализације, развијани су различити стандарди писања кода. Њихова сврха је да систем учине предвидљивим, а тиме учинковитијим. Тако, имена треба да носе информацију. На примјер, функција calculate_total() је обично боље име од ct(). Онда, у различитим језицима доминирају различити договори (*snake_case, camelCase, PascalCase*). Добра пракса је да свака датотека садржи једну доминантну апстракцију. Компоненте треба раздвајати по улогама, рецимо на прочеља, реализације, и конфигурације (подешавања). Потребно је избјегавати цикличне зависности и такозвани шпагети-код. Структурно уређење олакшава и промјене и рефакторисање.

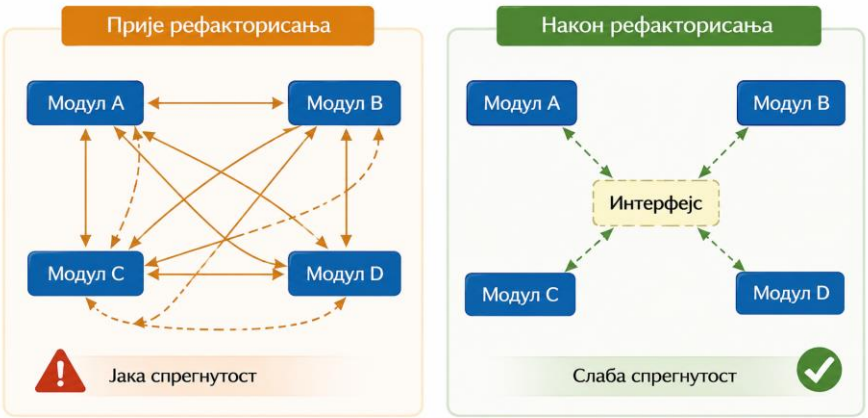
Напомена. Структурне одлуке нису важне само за читљивост, јер у појединим случајевима (нпр. код рекурзивних рјешења) оне директно одређују асимптотско понашање и могу довести до експоненцијалног раста времена извршавања.

5.2. Статичка анализа кода и рефакторисање

Статичка анализа открива сљедеће проблеме без покретања програма: стилистичке грешке, недосљедности у типовима, потенцијално опасне конструкције, као и неке сигурносно-безбједоносне рањивости. Типична оруђа за статичку анализу која се данас користе су: *Flake8, pylint, SonarQube, ESLint,* и

clang-tidy. Наглашавамо да поменута оруђа само допуњују, али не и замјењују људско рецензирање кода.[50]

Рефакторисање је уређење и побољшање састава самог кода без промјене његове функционалности. Потреба за рефакторисањем настаје када функције постану предуге,[51] када апстракције постану непрегледне, исти код постоји на више мјеста, имена више не одговарају улогама, или пак када компонента има "превише одговорности". Рефакторисање је у ствари кључни механизам управљања техничким дугом. *Технички дуг* представља компромис у којем се брзина испоруке плаћа будућом сложеношћу и већим трошковима одржавања.[52]



Слика 5.1. Приказано је поређење структуре зависности модула

[50] Статичка анализа је као зрцало: многи га избјегавају ујутру, али то не мијења чињеницу шта је у њему. Добро оруђе за анализу не уљепшава слику, само вам је приказује прије него што то уради испорука.

[51] Један од аутора је био свједок функције написане на *JavaScript*-у која је имала око 15 000 линија кода. Није познато да ли је особа која је то радила била генијечудак, или је пак таква функција просто производ лошег инжењерства.

[52] Разлика у односу на архитектонски дуг је што је технички локалан (класа, функција, модул).

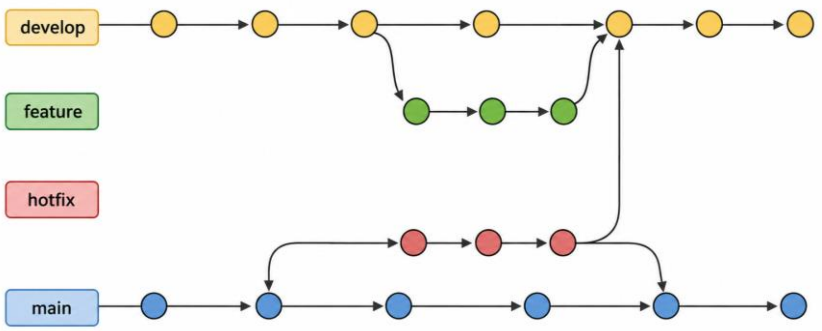
прије и након рефакторисања, са нагласком на смањење спрегнутости и побољшање одрживости кода.

5.3. Управљање верзијама и Гит

Реализација, као фаза, неодвојива је од *управљања верзијама*. Гит (енг. *Git*) као најпопуларније оруђе за то омогућава праћење промјена, сарадњу у тиму и брзо враћање на претходна стања. Због тога ћемо га овдје и мало размотрити. Основни ток рада са Гитом је приказан у наредна три корака:

1. `git add .` — додавање измјена у такозвану *staging* зону;
2. `git commit -m "опис измјене"` — ријеч је о снимању измјене у локалну историју;
3. `git push origin main` — слање измјена на удаљени репозиторијум.

Добра пракса је чест *упис промјена* (енг. *commit*) и писање јасних коментара (једна логичка измјена = један упис (један *commit*)).



Слика 5.2. Типичан модел гранања у систему за контролу верзија који подржава развој, тестирање, и (ваљда) стабилну испоруку софтвера.

С друге стране, само верзионирање мора бити подржано системима за изградњу, праћење зависности, и конфигурисање цјелокупног кода. У већини језика реализација захтјева:

- систем за изградњу (*Make, CMake, Maven, Gradle*);
- управљаче зависностима (*npm, pip, Cargo*);
- скрипте за аутоматизацију послова;
- конфигурационе датотеке (*YAML, JSON*).

Циљ је да се систем може поуздано изградити било гдје и у било којем тренутку.

5.4. Критички осврт: Код је најбоља документација

Иако се академски нагласак често ставља на детаљну документацију, пракса показује да је *читљив и провјерљив код најбоља документација*. Добро написан (оптимизован) код је самодокументујући јер функције имају јасна имена, коментари постоје само гдје је логика сложена, а структура пројекта је предвидљива. Штавише, структурно уређен и досљедно написан код није разумљив само програмерима, већ представља и бољу основу за статичке анализе, оптимизације и аутоматизована оруђа који се на тај код ослањају. Зато се савремено софтверско инжењерство више ослања на добру организацију кода него на вишеструке формалне наративне документе.

5.5. Закључак поглавља

Реализација није изолована активност већ резултат претходног разумијевања захтјева и пројектовања. Добар програмер је онај који пише код који други могу лако разумјети и проширити.[53]

5.6. Питања и задаци за провјеру знања

1. Које су главне карактеристике квалитетног кода?
2. Објаснити улогу прочеља у Гоу.
3. Како се у Гоу реализује конкурентност?
4. Описати основни гит-ток рада.
5. Зашто се каже да је код најбоља документација?
6. Написати кратак гоу-програм који користи прочеље и тестну функцију.
7. Пронаћи проблеме у сљедећем коду и предложити рјешење

```
type GodObject struct {
   db *sql.DB
   cache *redis.Client
   emailSender *smtp.Client
}
```

[53] А лош је онај који има if-исказа више него што *неће*!

```
func (g *GodObject) HandleUserRegistration(user User) error {
    // Потврда
    // Снимање у базу
    // Кеширање
    // Слање email-а
    // Ажурирање статистике
    // Логовање
}
```

6. ТЕСТИРАЊЕ СОФТВЕРА

Сврха поглавља

У овом поглављу тестирање се сагледава као стална повратна спрега у развоју софтвера којом се провјерава понашање система и постепено повећава повјерење у његову исправност.

6.1. Улога тестирања у *SDLC*-у

Тестирање је кључна фаза животног циклуса развоја софтвера. Његов циљ није да докаже да систем ради, већ да открије у каквим ситуацијама и зашто не ради. Откривање у раној фази развоја смањује трошкове који могу настати ако се пропусти открију након пуштања софтвера у употребу. Квалитетна провјера не доказује исправност програма, већ повећава повјерење у његов рад. Ово није само практично ограничење, него и формално.

У теоријском смислу, исправност програма се не може уопштено доказати тестирањем, јер не постоји алгоритам који би могао да одреди да ли произвољан програм има неко нетривијално својство. То ограничење формализује Рисова теорема.

Теорема 6.1. (Хенри Гордон Рис,[54] 1953) *Свако нетривијално*

[54] енг. *Henry Gordon Rice*

семантичко својство функције, односно понашања које програм израчунава, неодлучиво је у општем случају.

У теорији, ослањајући се на Рисову теорему, дошло се до математичких разлога (доказа) зашто статичка анализа може открити неке грешке, али никада све. Из практичног угла гледно, то значи да тестирање не доказује исправност, већ само повећава повјерење у систем!

Тестирање се мора планирати истовремено са пројектовањем и реализацијом, јер накнадно тестирање често постаје скуп и поприлично неучинковит процес. На примјер, истраживања показују да кашњење у тестирању може повећати трошкове и до 50%, будући да се у каснијим фазама пројекта могу открити комплексније грешке које захтјевају већи напор за исправљање. Основна тестирања су:

- *Јединично тестирање* (енг. *Unit Testing*). Тестира појединачне функције или модуле. Циљ му је да смањи ризик од грешака у логици појединачних дијелова кода;

- *Тестирање увезаности* (енг. *Integration Testing*). Провјера комуникације између компоненти. Циљ је смањење ризика везаног за исправност прочеља и садејства различитих дијелова система;

- *Тестирање система* (енг. *System Testing*). Провјера цјелокупног система као функционалне цјелине. Циљ је идентификација ризика повезаних са цјелокупном функционалношћу система, као и са крајњим условима корисничког окружења;

- *Тестирање прихвата* (енг. *Acceptance Testing*). Потврда да систем задовољава потребе корисника. Помаже у минимизовању ризика неприхватања од стране корисника или неиспуњавања корисничких захтјева.

Сваки тип теста има своју улогу и треба бити документован и поновљив.

Слика 6.1. Пирамида тестирања која илуструје однос јединичних, везивних, и системских тестова, као и њихову релативну цијену и учесталост.

У савременом софтверском инжењерству тестирање се све чешће проширује и на *заштитне аспекте* софтвера. Поред претходно поменутих класичних тестова, у пракси се користе и статичка анализа кода (енг. *static analysis*), фази-технике, претрага зависности (енг. *dependency scanning*), као и различити облици тестирања заштићености система. Циљ ових поступака није само откривање функционалних грешака, већ и уочавање потенцијалних рањивости, проблематичних зависности,

непровјерених улазних података и ризичних образаца у реализацији. У оквиру савремених *CI/CD* и *DevSecOps* приступа, овакве провјере све чешће постају аутоматизован дио процеса повезивања и испоруке.

6.2. Јединично тестирање на Гоу

Језик Гоу има уграђен пакет testing који омогућава једноставно писање и извршавање тестова.

Примјер 6.1. Дајемо једноставан јединични тест, као примјер за разумијевање.

```
package mathutils

import "testing"

func Add(a, b int) int {
      return a + b
}

func TestAdd(t *testing.T) {
   result := Add(2, 3)
   if result != 5 {
      t.Errorf("Очекивано 5, добијено %d", result)
      }
}
```

Тест се покреће наредбом go test ./.... Ако тест прође, Гоу исписује ok, док у противном случају имамо приказ грешака. ▲

6.3. Тестовима вођен развој

Тестовима вођен развој (енг. *Test-Driven Development (TDD)*) је приступ у којем се тест пише *прије* кода који треба да га задовољи. Циклус рада је кратак и понавља се у три корака:

1. *Црвена лампица* се односи на неуспјешан тест јер реализација још не постоји;
2. *Зелено свјетло* се односи на најједноставнији код који чини да сам тест прође;
3. *Рефакторисањем* се побољшава структура кода без мијењања понашања.

Слика 6.2. Циклус се извршава у малим корацима и понавља за сваки нови захтјев.

Овај приступ развоју софтвера, посебно погодан када је ријеч о језику Гоу, подстиче дисциплину, баца фокус на стварне захтјеве и минимизује фазу реализације.

6.3.1. Студија случаја: Валидација ЈИБ-а

Познато нам је да ЈИБ мора имати тачно 13 цифара и задовољити познати *MOD 10* алгоритам контроле. Потребно је, имајући у види претходно речено, омогућити потврђивање да је дати број ЈИБ. У наставку је дат почетни радни код.

```
// validacija_jib.go
package main
import (
   "regexp"
      "strconv"
)

func ValidirajJIB(jib string) bool {
   // ЈИБ мора имати тачно 13 цифара
   if len(jib) != 13 {
      return false
   }

   // Провјера да ли су сви знакови цифре
   matched, _ := regexp.MatchString(`^\d+$`, jib)
   if !matched {
      return false
   }

   // MOD 10 алгоритам за провјеру контролне цифре
      return validirajControlnuCifru(jib)
}

func validirajControlnuCifru(jib string) bool {
   factors := []int{6, 5, 4, 3, 2, 7, 6, 5, 4, 3, 2}
   sum := 0
```

```
    for i := 0; i < 11; i++ {
        digit, _ := strconv.Atoi(string(jib[i]))
        sum += digit * factors[i]
    }

    remainder := sum % 11
    controlDigit, _ := strconv.Atoi(string(jib[11]))

    if remainder == 0 {
        return controlDigit == 0
    }

    calculatedControl := 11 - remainder
    if calculatedControl == 10 {
        return controlDigit == 0
    }

        return controlDigit == calculatedControl
}
```

Корак 1. Црвена лампица се односи на писање јединичног теста на којем наш код сигурно “пада”.

```
// validacija_jib.go
package main

// Минимална реализација. Увијек враћа false
func ValidirajJIB(jib string) bool {
        return false
}
```

Тест покрећемо на претходно указани начин са go test -v. Добијамо посљедично извјештај

./validacija_jib_test.go:20:14: undefined: ValidirajJIB.

Корак 2. Зелено свјетло ће нам бити код који треба да резултује успјешном провјером. Пишемо крајње прост код за то.

```
// validacija_jib_test.go
package main

import "testing"

func TestValidirajJIB(t *testing.T) {
    tests := []struct {
        name string
        jib  string
        want bool
    }{
        {
            name: "Неважећи ЈИБ - погрешна дужина",
            jib:  "12345",
            want: false,
        },
        {
            name: "Неважећи ЈИБ - садржи слова",
            jib:  "12345ABC90128",
            want: false,
        },
        {
            name: "Важећи ЈИБ број",
            jib:  "1234567890128",
            want: true,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := ValidirajJIB(tt.jib)
            if got != tt.want {
                t.Errorf("ValidirajJIB(%s) = %v, очекивано %v", tt.jib, got, tt.w
ant)
            }
        })
    }
```

```
}
```

Поново покрећемо тест на уобичајен начин са go test -v. Добијамо сљедећи извјештај

```
=== RUN   TestValidirajJIB
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_погрешна_дужина
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_садржи_слова
=== RUN   TestValidirajJIB/Важећи_ЈИБ_број
    validacija_jib_test.go:22: ValidirajJIB(1234567890128) = false, очекивано true
--- FAIL: TestValidirajJIB (0.00s).
```

Побољшајмо сада реализацију.

```
// validacija_jib.go
package main

import "unicode"

func ValidirajJIB(jib string) bool {
    // Провјера дужине
    if len(jib) != 13 {
        return false
    }

    // Провјера да ли су све цифре
    for _, char := range jib {
        if !unicode.IsDigit(char) {
            return false
        }
    }

    // За сада прихватамо све 13-цифрене бројеве
        return true
}
```

Поново тестирамо са go test -v. Добијамо нови извјештај

```
=== RUN   TestValidirajJIB
=== RUN   TestValidirajJIB/Важећи_ЈИБ_број
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_погрешна_дужина
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_садржи_слова
--- PASS: TestValidirajJIB (0.00s).
```

Корак 3. Рефакторисањем побољшавамо структуру без мијењања понашања.

Додајмо још неколико тестова за контролни број и побољшајмо сам код.

```
// validacija_jib_test.go (проширено)
func TestValidirajJIB(t *testing.T) {
    tests := []struct {
        name     string
        jib      string
        expected bool
    }{
        {
            name:     "Важећи ЈИБ број 1",
            jib:      "1234567890128",
            expected: true,
        },
        {
            name:     "Важећи ЈИБ број 2",
            jib:      "4177475270003",
            expected: true,
        },
        {
            name:     "Неважећи ЈИБ - погрешна дужина",
            jib:      "12345",
            expected: false,
        },
        {
            name:     "Неважећи ЈИБ - садржи слова",
            jib:      "123456789012A",
            expected: false,
        },
        {
            name:     "Неважећи ЈИБ - погрешан контролни број",
```

```
        jib:      "1234567890129",
        expected: false,
    },
    {
        name:     "Неважећи ЈИБ - празан стринг",
        jib:      "",
        expected: false,
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        got := ValidirajJIB(tt.jib)
        if got != tt.expected {
            t.Errorf("ValidirajJIB(%s) = %v, очекивано %v", tt.jib, got,                tt.expecte
d)
        }
    })
    }
}
```

Реализујмо *MOD 10* алгоритам.

```
// validacija_jib.go (финална верзија)
package main
import "unicode"

func ValidirajJIB(jib string) bool {
    // Провјера основних услова
    if len(jib) != 13 {
        return false
    }

    for _, char := range jib {
        if !unicode.IsDigit(char) {
            return false
        }
    }

     return validirajKontrolniBroj(jib)
}
```

```go
func validirajKontrolniBroj(jib string) bool {
    factors := []int{6, 7, 8, 9, 10, 5, 4, 3, 2}
    sum := 0

    for i := 0; i < 12; i++ {
        digit := int(jib[i] - '0')
        sum += digit * factors[i%len(factors)]
    }

    remainder := sum % 11
    if remainder == 1 {
        return false
    }

    controlDigit := int(jib[12] - '0')
    if remainder == 0 {
        return controlDigit == 0
    }

    expectedControl := 11 - remainder
        return controlDigit == expectedControl
}
```

Завршни тест вршимо опет са `go text -v -cover`. Добијамо сљедећи извјештај у којем наш код пролази све написане тестове, и то са високом покривеношћу.

```
=== RUN   TestValidirajJIB
=== RUN   TestValidirajJIB/Важећи_ЈИБ_број_1
=== RUN   TestValidirajJIB/Важећи_ЈИБ_број_2
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_погрешна_дужина
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_садржи_слова
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_погрешан_контролни_број
=== RUN   TestValidirajJIB/Неважећи_ЈИБ_-_празан_стринг
--- PASS: TestValidirajJIB (0.00s)
PASS
coverage: 100.0% of statements
```

6.3.2. Поступно конструисање кода кроз *TDD*

Поступно конструисање кода кроз *TDD* се врши унутар циклуса Црвена лампица-Зелено свјетло-Рефакториши:

1. *ЦРВЕНА ЛАМПИЦА.* Пишемо тест за функционалност која тренутно не постоји. Ово значи да наша провјера мора бити неуспјешна (пали се црвена лампица!). Дефинише се очекивано понашање. У суштини, овај тест ће да нам служи као својеврсна "спецификација кроз код";

2. *ЗЕЛЕНО СВЈЕТЛО*. Пишемо прост код који би требао да резултује успјешном провјером. Битно је да се не оптимизује код прерано и да се фокусирамо искључиво на текући тест. Ово двоје је довољно да се добије такозвано зелено свјетло;

3. *РЕФАКТОРИШИ*. Побољшавамо структуру кода тако што ћемо уклони дупликате, побољшати називе (промјенљивих, структура, функција, класа, итд.), подијелити на мање функције, те држати све тестове зеленим.

Предности *TDD* приступа у развоју софтвера су сљедеће:

- разрада рјешења кроз тестове, те је код од самог свог почетка провјерљив;
- уграђена сигурност рефакторисања, јер тестови детектују регресије;
- живо документовање, јер тестови показују како се користи код;
- фокус на једноставност, јер се реализује само оно што је неопходно.

На крају *TDD*-поступка важно је уочити да се развој не одвија у једном кораку, већ у низу све бољих и бољих реализација програма који задовољава све познате тестове. Свака итерација *TDD*-тока може се разумјети као својеврсна примјена трансформације која проширује или прецизира постојећи код, при чему ниједан корак не смије нарушити наше већ задате тестове. Та поступност дјелотворно дефинише монотан оператор над простором могућих програма. У математичкој теорији фиксних тачака имамо формалну аналогију претходно реченог у виду теореме коју су формулисали пољски математичари Бранислав Кнастер[55] и Алфред Тарски.[56]

Теорема 6.1. (Тарски, 1955)[57] *Ако су испуњени додатни услови континуалности, најмања фиксна тачка може се добити као лимес узлазног низа итерација који почиње од најмањег елемента мреже.*

Ово је један од темељних резултата теорије фиксних тачака и има важну улогу у формалној семантици програма, нарочито при дефинисању значења рекурзивних и итеративних конструкција. У том контексту, најмања фиксна тачка представља најмање стабилно рјешење које задовољава задате једначине семантике.

У контексту TDD-а, ова идеја се не може пренијети дословно, али може послужити као корисна инжењерска аналогија. Развој вођен тестовима може се посматрати као

[55] пољ. *Bronislaw Knaster*

[56] пољ. *Alfred Tarski*

[57] Погледати [72].

поступно приближавање понашању које је задато спецификацијом израженом кроз тестове. Свака итерација додаје или прецизира очекивано понашање система, док тестови служе као контролни механизам који открива када нова измјена нарушава раније прихваћено понашање. У идеалном случају, такав процес води ка стабилној верзији програма у којој су захтијевана понашања задовољена, али ту стабилност не треба схватити као математички гарантовану конвергенцију, него као резултат дисциплинованог инжењерског поступка.[58]

6.4. Аутоматизација тестирања и *CI/CD*

Аутоматизација тестирања је суштински дио савременог развоја софрвера. Комбинује се са *сталном интеграцијом и сталном испоруком* (енг. *Continuous Integration and Continuous Delivery/Deployment (CI/CD)*)[59] у системима као што су *GitHub Actions*, *Jenkins* или *GitLab CI*. На примјер, сваки упис промјене у *git*-репозиторијум може покренути:

1. компилацију кода (`go build`);
2. извршавање тестова (`go test ./...`);
3. анализу покривености тестова (`go test -cover`).

[58] Према једној програмерској досјетки, развој без тестова личи на летење без инструмената: можда ће се завршити добро, али то није метода на коју се озбиљан тим ослања. TDD не гарантује исправност система, али обезбјеђује да тим брзо уочи одступања од понашања која су већ експлицитно описана тестовима.

[59] Неријетко се погрешно мисли да је *CI/CD* само девопс-оруђе. Ипак, ријеч је у ствари о дисцичлини у развоју софтвера, квалитету, и наравно испоруци.

Овим приступом се грешке откривају поприлично рано, а стабилност самог кода се провјерава суштински непрекидно.

Напомена. Циљ *CI/CD*-а није брзина, већ смањење ризика. Аутоматизација тестирања просто смањује број људских грешака када је ријеч о самом тестирању, те тиме доприноси стабилизацији развојних токова.

Покривеност тестовима често се анализира на нивоу контролних токова програма. У таквом моделу програм представљамо као *граф контроле тока* (енг. *Control Flow Graph (CFG)*), у којем чворови означавају програмске тачке (улази у блокове, услови, излази), а гранама су престављени прелази у извршавању.

Узмимо да су u и v два чвора у *CFG*-у која означавају полазну и одредишну критичну тачку система. На примјер, почетак функције и њен излаз, или улаз у условну грану и код који слиједи након њеног извршења. Независни путеви између u и v представљају логички различите сценарије извршавања који морају бити тестирани како би се покриле све кључне гране између дате двије тачке.

Иначе, за ово се формално примјењује чувена Менгерова теорема о повезаности графа, објављена (за информатику) давне 1927. године.

Теорема 6.2. (Карл Менгер, 1927)[60] *Нека је G коначан неусмјерен граф и нека су u и v два различита несусједна чвора графа G. Тада је максималан број паровно интерно чворно дисјунктних u–v путева*

[60] Погледати [53]

једнак минималном броју чворова, различитих од u и v, чијим уклањањем се прекида свака веза између u и v.

У контексту структурног тестирања, овај број може се тумачити као доња граница броја међусобно независних путних сценарија које је потребно размотрити између тачака (u) и (v). Ако граф представља контролни ток програма, онда паровно интерно дисјунктни (u-v) путеви указују на различите начине проласка кроз програмску структуру, при чему сваки од њих пролази кроз бар један дио структуре који није садржан у осталима. Зато структурна својства програма не одређују механички број тестова, али постављају доњу границу сложености коју тестни скуп мора узети у обзир ако жели да покрије све релевантне независне путеве.

6.5. Тестирање учинковитости

Поред функционалног тестирања, важна нам је и *процјена учинковитости система*. Гоу омогућује писање профилационих тестова помоћу testing.B структура.

Примјер 6.2. Писање референтних[61] тестова помоћу testing.B структуре.

```
package mathutils

import "testing"

func Add(a, b int) int {
    return a + b
```

[61] енг. *benchmark*.

```
}

func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Add(10, 20)
        }
}
```

Овакви тестови показују колико пута функција може да се изврши у секунди, што помаже у *оптимизацији рачунарског програма*. ▲

Напомена. Треба имати у виду да оптимизација рачунарског програма има своје границе. Штавише, некад су те границе поприлично високо постављене. Рецимо, *Кук-Левинова теорема* формално доказује да је *SAT*-проблем *NP*-комплетан.[62] Посљедица за софтверско инжењерство је да се вјерује да многи проблеми комбинаторне оптимизације, распоређивања ресурса, тестирања и верификације немају своје полиномне алгоритме[63] (гледане као детерминистичке Тјурингове машине), осим ако се у будућности не докаже да супротно у "спору" *P* "против" *NP*. Због тога се користе хеуристике, апроксимације, и прагматични компромиси.

[62] Ово баш и није тачно, јер су формално теореме Кука (енг. *Stephen Cook*) и Левина (рус. *Леонид Левин*) засебно везане за сродне, али другачије проблеме. Ипак, у математичко-информатичкој пракси Кук-Левинова теорема је постао прихватљив и устаљен колоквијални термин.

[63] Или бар за сад ауторима није познато да је неко конструисао полиномне детерминистичке Тјурингове машине за рјешавање *NP*-тешких проблема.

6.6. Критички осврт: Формалне провјере насупрот здравом разуму

У академским оквирима често се инсистира на формалним метрикама покривености теста, али у пракси је важније тестирати најризичније и најчешће коришћене дијелове система.

Добар тест не мора бити сложен, али би требао бити смислен. Претјерана склоност формализму може довести до тога да се тестови пишу ради статистике, не ради квалитета кода. Здрав инжењерски приступ увијек даје предност стварној поузданости у односу на бирократску мјерљивост.

Питање. Тестови се пишу прије кода, али једно питање остаје отворено: да ли тим *заправо жели* да зна истину коју ће тестови рећи?

6.7. Закључак поглавља

Тестирање није фаза на крају реализације пројекта, већ стална активност. Добар код настаје из доброг тестирања као повратне спреге, а не обрнуто. На крају, тестови су у ствари

најпоузданији облик документације, јер прецизно показују шта систем ради и како се понаша у граничним ситуацијама.

6.8. Питања и задаци за провјеру знања

1. Објаснити разлику између јединичног тестирања у односу на тестирање веза.
2. Шта је суштина *TDD*-а?
3. Како се када је ријеч о језику Гоу пишу и покрећу тестови?
4. Каква је улога *CI/CD*-а у тестирању?
5. Објаснити зашто формална покривеност тестова није довољна гаранција квалитета.
6. Написати функцију за потврду *EMBG* броја користећи *TDD* приступ као у студији случаја 6.3.1.

7. ОДРЖАВАЊЕ И ИЗМЈЕНЕ

Сврха поглавља

У овом поглављу одржавање и измјене сагледавају се као природан наставак развоја софтвера и основни услов његове дугорочне стабилности, прилагодљивости, и употребљивости.

7.1. Појам и значај одржавања софтвера

Одржавање софтвера обухвата све активности које се спроводе након што је производ пуштен у употребу, са циљем да се исправе новоуочене грешке, унаприједи функционалност и по потреби изврши прилагођавање новим условима. Одржавање је најдужа и често најскупља фаза *SDLC*-а. Према *ISO/IEC* 14764 стандарду,[64] постоје четири основне врсте одржавања:

- *Коративно одржавање*: исправљање откривених грешака (“багова”).
- *Адаптивно одржавање*: прилагођавање софтвера промјенама у окружењу.
- *Перфекционо одржавање*: побољшање функционалности софтвера или пак читљивости кода.

[64] Погледати [37].

- *Превентивно одржавање*: спречавање могућих проблема унапријед.

Напомена. У пракси се ове побројане четири врсте одржавања често преплићу, нарочито када је ријеч дугорочним пројектима.

До сада смо научили да софтвер није статичан као производ. Просто, он се мијења заједно са његовим пословним и технолошким окружењем. Уопштено развој софтвера се не односи само на његову изградњу до испоруке клијенту, него и на дугорочно управљање његовим измјенама, уз што је могуће мање ризике и трошкове. Према чувеним Лимановим законитостима еволуције софтвера,[65] сваки (живи?) систем, и ту већ осјећамо нешто што превазилази пуку технику, мора "еволуирати" или ће се неминовно распасти. Ово начело, да га тако назовемо, се јасно огледа у случајевима изразито великих информационих система, као што су банкарски или здравствени софтвери, који су опзнати по томе што захтијевају стална ажурирања због регулаторних промјена, те прилагођавања новим технологијама или усмјеравању на нове тржишне захтјеве. Могли бисмо бити помало цинични и казати да се управо у тој потреби за непрестаном промјеном открива читава трагикомична карактеристика самог (развоја) софтвера: систем који никада није готов, већ стално клизи ка сопственој непотпуности. Но, посматраћемо ту ситуацију као нормалан космички поредак, тј. да без ажурирања, без сталног суочавања са хаосом нових захтјева, систем пуца, баш као што пуца и човјек[66] који избјегава одговорност.

[65] *Lehman's Laws of Software Evolution*

[66] ..."по шавовима", не из ватреног оружја!

Одржавање, као што видимо, није пуко крпљење. То је стална борба против ентропије, односно дисциплина која условљава опстанак. Софтвер који се не развија почиње да "трули": зависности застаревају, окружења се мијењају, корисничка очекивања расту, а систем који остаје исти заправо постаје све гора копија сопствене прошлости. Одржавање је зато врста техничког хабитуса: непрестано увођење реда и стално уклањање хаоса који се природно акумулира.[67] Овако гледано, одржавање није трошак, већ онтолошка нужност. Даље, из праксе је видљиво да учинковито одржавање захтјева дисциплину у раду са програмским кодом и оруђима која се односе на:

- *верзионирање* (*Git, Mercurial, Apache Subversion, Concurrent Versions System,* итд.), јер нам омогућавају праћење промјена и враћања на стабилне верзије;
- *рецензију кода* (енг. *Code Review*), што је у ствари преглед кода који врше други чланови тима ради уочавања грешака и побољшања квалитета;
- *аутоматско тестирање*, јер свако одржавање у суштини треба да буде пропраћено регресионим тестовима;
- *документација промјена*, што се односи на опис разлога и ефеката сваке модификације.

Без ових конкретних пракси, сам систем временом постаје неуправљив и скуп за неко ново даље развијање, ма како да смо га иницијално разрадили. Опет, квалитет одржавања може се пратити кроз различите метрике:

[67] Могли бисмо казати "креативног хаоса".

- *Mean Time To Repair (MTTR).* Односи се на просјечно вријеме потребно за отклањање грешке;
- *Mean Time Between Failures (MTBF).* Односи се на просјечно вријеме између два бага (пуцања система);
- *Change Failure Rate (CFR)*: проценат промјена које доводе до нових грешака.

Поменуте метрике омогућавају рационалну процјену трошкова и учинковитост програмерског тима.

У сложеним системима који имају много конкурентних процеса питање да ли се одређено стање (у овом случају стање развоја једног дијела софтвера) може достићи формално се описује преко *досега* (енг. *reachability*) Петријевих мрежа.

Петријеве мреже су стандардни формални модел за описивање система са више конкурентних токова извршавања, тј. у којима независни догађаји могу настати у различитим редослиједима. Класични примјери су комуникационе мреже, мултипроцесорски системи, производни системи, дистрибуиране базе података, и др. Проблем досега у таквим мрежама показује да нека понашања конкурентних система није могуће алгоритамски предвидјети уопштено.

7.2. Практични примјер регресионог тестирања на Гоу

Регресионо тестирање одговара на сљедеће питање: да нису можда нове измјене нарушиле постојећу функционалност? Другим ријечима, ако јесу, само нам је то још фалило у животу. У

пракси, ова врста тестирања омогућава откривање да ли додате или измијењене компоненте узрокују неочекиване сметње у већ реализованим функцијама, што је од кључне важности за очување поузданости софтвера током његовог развоја и одржавања.

Примјер 7.1. Регресиони тест који провјерава да ли се понашање функције промијенило између два позива.

```
package main

import "testing"

// Претпоставимо да постоји Login функција
func Login(username, password string) bool {
    // Реализација логике пријаве
    return username == "user" && password == "pass"
}

func TestUserLoginRegression(t *testing.T) {
    old := Login("user", "pass")
    new := Login("user", "pass")
    if old != new {
        t.Errorf("Регресија откривена: очекивано исто понашање")
    }
}
```

Сврха регресионих тестова је да открију неочекиване промјене у понашању, бочне ефекте између позива, проблеме са стањем/кеширањем, као и грешке у конкурентном окружењу. У посљедње вријеме су се “наметнули” као методологија за осигуравање стабилности софтвера током његовог одржавања.

7.3. Рефакторисање на Гоу: Од лошег ка одрживом коду

Рефакторисање је процес побољшања унутрашње структуре кода без мијењања спољњег понашања. У Гоу, рефакторисање се ослања на једноставност, јасноћу и аутоматске тестове. Сигурни кораци за рефакторисање су:

1. Покренути тестове и осигурати да сви пролазе;
2. Направити малу промјену, конкретно један концепт по промјени;
3. Поново покренути тестове, тј. провјерити да ли је све у реду;
4. Потврдити промјену ако тестови пролазе;
5. Поновити кораке 1-4 када је ријеч о наредној малој промјени са списка.

```
# Безбједни циклус
go test ./... && git add . && git commit -m "Рефакторинг: подјела OrderProcessor"
```

Критеријуми доброг рефакторисања:

✅ Сви тестови и даље пролазе;

✅ Код је јаснији и читљивији;

✅ Свака функција има једну одговорност;

✅ Мање је дуплирања кода;

✅ Лакше је за тестирање;

✅ Боља именовања типова и функција.

Примјер унапређења према метрикама[68] након рефакторисања:

Cyclomatic complexity: 15 → 4
Lines per function: 80 → 15
Test coverage: 60% → 90%
Code duplication: 25% → 5%

Кад је ријеч о оруђима за рефакторисање на Гоу имамо сљедеће на располагању.

```
# Аутоматско форматирање
gofmt -w .
# Статичка анализа
go vet ./...
# Откривање дупликата кода
gocpd --min-tokens 50 ./...
# Мјерење комплексности
gocyclo -over 10 .
# Безбједносна провјера
gosec ./...
```

7.3.1. Заштитно рефакторисање са тестовима

Рефакторисање у пракси ријетко почиње зато што је сам код "лош", него управо зато што је постао тежак за мијењање и дораду. Функционалност постоји, присутна је, систем ради, али свака нова измјена изазива нову нелагоду, јер није јасно шта ће се све пореметити и гдје се могу појавити нежељена дејства. У таквим ситуацијама заштитно рефакторисање има јасну улогу: структуру кода треба побољшати, а да се његово понашање при

[68] О софтверским метрикама више у сљедећој глави.

томе не промијени. Тестови нам ту не служе као некаква формална потврда исправности, већ управо као практична гаранција да је након промјене систем остао функционално исти.

Примјер који слиједи приказује чест образац у реалним пројектима. Потврда/валидација података, приступ бази, комуникација са спољним сервисима и уписивање (логовање) налазе се на једном мјесту, што код чини поприлично тешко читљивим, а још тежим за само тестирање. Иако такав код није нужно неисправан, свака промјена у једном дијелу повећава ризик да ће нешто друго бити нарушено. Рефакторисање које је приказано у наставку не уводи нове могућности, већ постепено раздваја одговорности у мање, смислене цјелине. На тај начин код постаје прегледнији, једноставнији за тестирање и отпорнији на будуће измјене, што је и основни циљ заштитног рефакторисања.

```
package main
import (
   "database/sql"
      "errors"
)

// Прије рефакторисања један објекат ради све
type OrderProcessor struct {
   db     *sql.DB
   emailer *EmailService
      logger  *Logger
}

func (op *OrderProcessor) Process(order Order) error {
   // Потврда (30 линија)
   if order.Amount <= 0 {
      return errors.New("неважећи износ")
   }
   if order.CustomerEmail == "" {
      return errors.New("email је обавезан")
   }
```

```
    if len(order.Items) == 0 {
        return errors.New("поруџбина мора ити бар један артикал")
    }
    // ... још 25 линија потврђивања

    // Снимање у базу (20 линија)
    _, err := op.db.Exec("INSERT INTO orders (id, amount, customer_email) VALUES (?, ?, ?)",
        order.ID, order.Amount, order.CustomerEmail)
    if err != nil {
        return err
    }
    // ... још 15 линија SQL логике

    // Слање email-а (15 линија)
    err = op.emailer.Send(order.CustomerEmail, "Поруџбина примљена")
    if err != nil {
        return err
    }
    // ... још 10 линија email логике

    // Пријава (10 линија)
    op.logger.Info("Обрађена поруџбина", order.ID)

        return nil
}

// Након рефакторисања
// OrderValidator је једино одговоран за валидацију
type OrderValidator struct{}
func (v OrderValidator) Validate(order Order) error {
    if order.Amount <= 0 {
        return errors.New("неважећи износ")
    }
    if order.CustomerEmail == "" {
        return errors.New("email је обавезан")
    }
    if len(order.Items) == 0 {
        return errors.New("поруџбина мора имати бар један артикл")
    }
    // ... специфична логика
        return nil
}
```

```
// OrderRepository је једино одговоран за перзистенцију
type OrderRepository struct {
    db *sql.DB
}

func (r OrderRepository) Save(order Order) error {
    _, err := r.db.Exec("INSERT INTO orders (id, amount, customer_email) VALUES (?, ?, ?)",
            order.ID, order.Amount, order.CustomerEmail)
    return err
}

// Notifier је једино одговоран за комуникацију
type Notifier interface {
    NotifyOrderCreated(order Order) error
}

type EmailNotifier struct {
    emailer *EmailService
}

func (e EmailNotifier) NotifyOrderCreated(order Order) error {
    return e.emailer.Send(order.CustomerEmail, "поруџбина примљена")
}

// Logger је уговор за пријаву
type Logger interface {
    Info(msg string, args ...interface{})
    Warn(msg string, args ...interface{})
}

// OrderService нам координира различите сервисе
type OrderService struct {
    validator OrderValidator
    repo      OrderRepository
    notifier  Notifier
    logger    Logger
}

func (s *OrderService) Process(order Order) error {
    if err := s.validator.Validate(order); err != nil {
        return err
    }
```

```
    if err := s.repo.Save(order); err != nil {
        return err
    }

    if err := s.notifier.NotifyOrderCreated(order); err != nil {
        s.logger.Warn("Грешка при слању обавјештења", err)
    }

    s.logger.Info("Обрађена поруџбина", order.ID)
    return nil
}

// Помоћне структуре и прочеља
type Order struct {
    ID          string
    Amount      float64
    CustomerEmail  string
    Items       []OrderItem}
type OrderItem struct {
    ProductID string
    Quantity  int
    Price     float64
}

type EmailService struct{}

func (e *EmailService) Send(to, subject string) error {
    // Реализација слања email-а
    return nil
}

type ConsoleLogger struct{}

func (c ConsoleLogger) Info(msg string, args ...interface{}) {
    // Реализација info пријаве
}

func (c ConsoleLogger) Warn(msg string, args ...interface{}) {
    // Реализација warn пријаве
}
```

7.3.2 Практични примјер рефакторисања кроз тестове

У претходном потпоглављу показано је како се код може структурно унаприједити без промјене понашања система. Међутим, такво рефакторисање у реалним условима не ослања се на интуицију или наду да "ништа није покварено", већ на конкретне тестове који изричито описују шта сам систем смије, а шта не смије да ради. Управо зато тестови у овом контексту нису пратећи детаљ, већ централни механизам који омогућава сигурно мијењање кода.

Сљедећи примјер приказује једноставан, али репрезентативан тест који фиксира понашање сервиса за обраду поруџбина. Тест не улази у унутрашњу реализацију, нити га занима како је код организован, већ искључиво провјерава да ли систем коректно реагује на важеће и неважеће улазе. Кроз употребу назови-реализација (енг. *mocks*) за спољне зависности јасно се раздваја логика која се тестира од инфраструктурних детаља, што омогућава да се код касније рефакторише без страха да ће се промијенити његово видљиво понашање. На тај начин тестови постају својеврсни уговор: све док они пролазе, рефакторисање се може наставити са разумним степеном повјерења.

```
// Тест који осигурава понашање током рефакторисања
func TestOrderService_Process(t *testing.T) {
    tests := []struct {
        name    string
        order   Order
        wantErr bool
```

```
}{
    {
        name:    "Важећа поруџбина",
        order:   Order{Amount: 100,
                                    CustomerEmail: "test@example.com"},
        wantErr: false,
    },
    {
        name:    "Неважећа поруџбина --- негативан износ",
        order:   Order{Amount: -50,
                                    CustomerEmail: "test@example.com"},
        wantErr: true,
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        // Setup - мокови за све зависности
        validator := &MockValidator{}
        repo := &MockRepository{}
        notifier := &MockNotifier{}
        logger := &MockLogger{}

        service := OrderService{
            validator: validator,
            repo:     repo,
            notifier:  notifier,
            logger:    logger,
```

```
        }


        // Execute
        err := service.Process(tt.order)


        // Verify
        if (err != nil) != tt.wantErr {
            t.Errorf("Process() error = %v, wantErr %v",
                                        err, tt.wantErr)
        }
    }
        )
    }
}
```

7.4. Софтверска подршка и одржавање у производњи

У претходним дијеловима поглавља софтверско одржавање посматрано је првенствено као скуп активности усмјерених на измјене софтверског артефакта, односно кода и његове структуре. Такав приступ је у складу са класичним инжењерским моделима, у којима се одржавање дијели на коративно, адаптивно и перфективно. Међутим, у савременом окружењу, у којем се софтверски системи извршавају у сложеним продукционим условима и служе великом броју корисника, појам одржавања нужно се проширује.

Потребно је направити јасну разлику између софтверског одржавања и софтверске подршке. Док се одржавање односи на

измјене самог система (нпр. исправљање грешака, унапређење перформанси или прилагођавање новим условима), софтверска подршка обухвата активности које омогућавају стабилан и поуздан рад система у продукцији. Ове активности укључују комуникацију са корисницима, обраду пријављених проблема, праћење рада система, као и реаговање на инциденте.

У индустријској пракси, активности подршке најчешће су организоване кроз више нивоа, који се означавају као *L1–L3*:

- *Први ниво подршке (L1)* представља контакт тачку са корисницима. Његова улога је да прима захтјеве, евидентира проблеме (обично кроз систем тикета) и пружи основну помоћ када је то могуће.
- *Други ниво подршке (L2)* бави се техничком анализом проблема. Овдје се врши дијагностика, анализа логова, репродукција грешака и утврђивање узрока проблема.
- *Трећи ниво подршке (L3)* укључује развојни тим. У случајевима када је потребна измјена кода или дубља интервенција у систему, *L3* преузима одговорност за рјешавање проблема.

Ова подјела омогућава учинковитије управљање инцидентима и рационалну расподјелу ресурса, при чему се сложенији проблеми усмјеравају ка вишим нивоима.

Поред комуникације са корисницима, значајан дио савременог одржавања односи се на праћење рада система пуштеног у употребу. То подразумијева употребу активности као што су надзор (енг. *monitoring*), вођење записника (логовање), и системи за обавјештавање. Циљ је благовремено откривање проблема, често и прије него што их сами корисници уоче. У том

контексту важну улогу имају и појмови као што су *инциденти* и *нивои услуге* (енг. *Service Level Agreement* (*SLA*)), који дефинишу очекивани ниво доступности и времена одзива система.

Савремени приступи развоју софтвера, посебно они који се означавају појмом Девопс (енг. *DevOps*), додатно бришу границу између развоја и одржавања. Умјесто јасне подјеле на фазе, уводи се непрекидни ток који обухвата повезивање, провјеравање, испоруку, и праћење рада софтвера. Аутоматизација ових корака (нпр. кроз *CI/CD* токове) омогућава брже отклањање грешака, смањење ризика, те већу стабилност система током употребе.

Иако класичне категорије одржавања (корективно, адаптивно и перфективно) остају важеће, у пракси се оне реализују у ширем оквиру који укључује и активности софтверске подршке. Због тога се савремено одржавање софтвера не може посматрати искључиво као измјена кода, већ као скуп координисаних активности које обухватају и технички и оперативни поглед на рад производа.

7.5. Критички осврт: Мит о “готовом“ софтверу

Можемо слободно рећи да је један од највећих неспоразума, који карактерише академски и индустријски дискурс, јесте идеја да је софтвер икада “завршен“. У стварности, сваки систем који се користи мора да се одржава. Када се одржавање занемари, настаје

феномен звани *труљење софтвера* (енг. *software rot*). Просто, на специфичан начин пропада и код и архитектура.[69] Због тога је смисленије говорити о *стабилним верзијама,* него о *завршеним производима*.

7.6. Закључак поглавља

Одржавање и измјене задиру у суштину животног циклуса развоја софтвера. Оне захтијевају дисциплину на високом нивоу, предвиђање токова измјене радног окружења, као и сталну комуникацију у тиму. Добро одржаван систем живи годинама, па и деценијама, а лоше одржаван не преживи ни прву праву надоградњу.

7.7. Питања и задаци за провјеру знања

1. Објаснити разлику између одржавања и еволуције софтвера.
2. Навести четири врсте одржавања према *ISO* стандарду.
3. О чему нам говори Лиманова законитост кад је ријеч о еволуцији софтвера?
4. Опишите улогу Гита у одржавању.
5. Шта је то регресионо тестирање и зашто је важно?

[69] Мисли се на архитектуру софтверског рјешења, а не на архитектуру рачунарског система.

6. Објаснити зашто у савременој индустријској пракси не сусрећемо такорећи готов и завршен софтвер.

8. СОФТВЕРСКЕ МЕТРИКЕ

Сврха поглавља

У овом поглављу софтверске метрике разматрају се као оруђа рационалнијег инжењерског одлучивања, чије вриједности добијају смисао тек тумачењем у одговарајућем контексту.

8.1. Метрике и њихово мјесто под сунцем

Софтверске метрике представљају мјерљиве показатеље који служе за процјену квалитета процеса, производа или тима. Циљ ових метрика (које нису класичне математичке метрике) није бирократска контрола, већ доношење рационално заснованих одлука.

Добре метрике су у суштини објективне, поновљиве и лако разумљиве. Ипак, да не буде забуне, наглашавамо да је објективност метрика условна и зависи од начина мјерења. Рецимо, иста метрика може давати потпуно различите резултате у зависности од програмског језика, стила писања кода или пак оруђа који је рачуна. На примјер, цикломатска сложеност функције у Гоу и Пајтону може се значајно разликовати иако је логика програма идентична, управо зато што различита оруђа рачунају гранања на различите начине. Гоуово оруђе gocyclo броји логичке операторе као гранања, док Пајтонов radon не, те се може десити да функција писана на Гоу има сложеност 10, док иста таква писана

на Пајтону да има сложеност 7. Због тога вриједности добијене коришћењем изабраних метрика треба стављати у ваљан контекст, тј. не треба их сматрати апсолутним показатељима квалитета.

Метрике омогућавају поређење различитих верзија система и процјену напретка развоја кода у односу на постављене циљеве. Обично их дијелимо на неколико сљедећих категорија:

- *Процесне метрике,* које мјере ефикасност развојног процеса (нпр. вријеме испоруке, просјечан број дефеката по итерацији);
- *Производне метрике,* које се односе на карактеристике кода (нпр. број линија кода, сложеност функција, или пак покривеност тестовима);
- *Метрике ресурса,* које прате употребу времена, новца, али и људских ресурса;
- *Метрике квалитета,* које процјењују поузданост, употребљивост, и наравно одрживост самог система.

Табела 8.1. Преглед категоризације софтверских метрика.

Категорија метрика	Шта мјере	На шта се односе	Примјери
Процесне метрике	Учинковитост и динамику развојног процеса	Развојни циклус и тимске активности	Вријеме испоруке, број дефеката по итерацији
Производне метрике	Структуру и сложеност софтверског артефакта	Изворни код и тестове	Број линија кода (LOC), цикломатска сложеност,

Категорија метрика	Шта мјере	На шта се односе	Примјери
			покривеност тестовима
Метрике ресурса	Утрошак и расположивост ресурса	Вријеме, новац и људски ресурси	Утрошено вријеме, број инжењера, трошкови
Метрике квалитета	Ниво поузданости и употребљивости система	Понашање система у употреби	Поузданост, употребљивост, одрживост

Код може бити технички исправан, а ипак лошег квалитета ако је превише сложен. Вриједност метрике не произилази из одговарајућег броја, већ из модела унутар којег се тај број тумачи. Значи, метрика је корисна тек када постоји јасна веза између њене вриједности и инжењерске одлуке коју треба донијети. На примјер, висока цикломатска сложеност не значи аутоматски да је код "лош", већ можда само да постоји повећан ризик за теже тестирање или одржавање. Само у контексту архитектуре, домена и циљева тима вриједност метрике постаје информација, а не некакав изоловани сигнал.[70]

[70] Према старој шали: "Када метрика постане циљ, добијете циљ који не мјери ништа." Ово није само шала. Конкретно, ријеч је о Гудхартовом закону.

8.2. Најчешће коришћене метрике и начини сагледавања квалитета

Када је ријеч о метрикама сложености програмског кода, најчешће коришћене су:[71]

- *Цикломатска сложеност* (енг. *Cyclomatic Complexity*). Макејбова[72] метрика којом се мјери број независних путања кроз програм. Формално се дефинише као

$$V(G) = E - N + 2P$$

  гдје су

  - E број грана графа тока управљања,
  - N број чворова,
  - P број повезаних компоненти (обично 1 за једну функцију).

  Тумачење. Цикломатска комплексност представља апроксимацију броја независних путања кроз код. Емпиријске студије и индустријска пракса показују да раст комплексности корелира са већим когнитивним оптерећењем, већом вјероватноћом дефеката и тежим тестирањем. Вриједности изнад 10 често се сматрају сигналом за анализу и могуће рефакторисање, док вриједности изнад 15–20 указују на код

[71] На нашим просторима ова реченица безобразно дрско звучи, јер се метрике у индустрији користе у траговима.

[72] енг. *Thomas J. McCabe*

који постаје значајно тежи за поуздано разумијевање, тестирање и одржавање.[73]

- *Халстедове метрике* (енг. *Halstead Complexity Measure*). Процјењују напор и вријеме потребно за разумијевање кода. Засноване су на броју различитих и укупних оператора (n_1, N_1) и операнда (n_2, N_2). Основне формуле су

  - Рјечник (енг. *Vocabulary*) $n = n_1 + n_2$
  - Дужина (енг. *Length*) $N = N_1 + N_2$
  - Обим (енг. *Volume*) $V = N \times \log_2(n)$
  - Тежина (енг. *Difficulty*) $D = (n_1 / 2) \times (N_2 / n_2)$
  - Напор (енг. *Effort*) $E = D \times V$

  Тумачење. Приближна процјена напора, когнитивног оптерећења, и времена разумијевања.

- *Индекс одрживости* (енг. *Maintainability Index*). Комбинује више фактора у један број који показује лакоћу одржавања. Рачуна се као

$$I = 171.0 - 5.2 \times \ln(V) - 0.23 \times CC - 16.2 \times \ln(LOC),$$

  гдје је

  - V Халстедов обим,
  - CC цикломатска сложеност,

[73] Цикломатска сложеност мјери број независних путева кроз код, али постоји још један пут који није формално моделован: пут којим програмер покушава да схвати шта је његов претходник радио.

- *LOC* број линија кода.[74]

Тумачење. Веће вриједности → лакше одржавање.

Напомена. *LOC* је условно добар показатељ величине софтвера. Различити језици екстремно варирају у експресивности (Гоу / Јава / Пајтон), што слаби ваљаност овог показатеља.

У Гоу, оруђа попут gocyclo и golint помажу у аутоматској анализи ових показатеља. Метрике тестирања служе за праћење квалитета и покривености тестова. Најважније су:

- *Покривеност кода* (енг. *Code Coverage*). Проценат кода који је обухваћен тестовима. Рачуна се као

Code Coverage = (покривени елементи / укупан број елемената) × 100%

- *Густина дефеката* (енг. *Defect Density*). Број грешака по јединици кода. Рачуна се као

Defect Density = број дефеката / *kLOC*,

гдје *kLOC* значи *kilo LOC* (хиљаду линија кода).[75]

- *Стопа пролазности тестова* (енг. *Test Pass Rate*). Однос између успјешно и неуспјешно извршених тестова. Рачуна се као

[74] Бројање линија кода зависи од стила, језика, оруђа, укључивања/искључивања коментара и празних линија. У метрикама се често рачуна *LOC* без коментара и празних линија (тзв. *SLOC/NLOC*).

[75] *kLOC* као и *LOC* нису прецизне јединице мјере, јер не постоји универзална дефиниција "линије кода" (за све програмске језике), различита оруђа различито броје, а различити језици имају различиту изражајност.

Test Pass Rate = passed tests / total tests × 100%.

Иначе, циљ није постићи 100% покривеност, већ тестирати најризичније дијелове система. *ISO/IEC* 25010:2023 стандард[76] дефинише девет димензија/видова квалитета које овдје дословно наводимо:

1. Функционалност (енг. *Functionality*);
2. Учинковитост извршавања (енг. *Performance Efficiency*);
3. Компатибилност (енг. *Compatibility*);
4. Употребљивост (енг. *Usability*);
5. Поузданост (енг. *Reliability*);
6. Безбједност (енг. *Safety*);
7. Одрживост (енг. *Maintainability*);
8. Преносивост (енг. *Portability*);
9. Сигурност (енг. *Security*).

Ови видови сагледавања квалитета, дефинисани стандардом *ISO/IEC* 25010:2023, пружају структуру за планирање одговарајућих тестова и избор оруђа који мјере сваки појединачни аспект квалитета, чиме омогућавају систематску процјену зрелости система у пракси.

Мјерење људског аспекта развоја је деликатно, али корисно ако се правилно тумачи. Примјери корисних показатеља су:

[76] Погледати [38].

- *Вријеме циклуса* (енг. *Cycle Time*). Ријеч је о просјечном трајању једне измјене од захтјева до испоруке;

- *Брзина тима* (енг. *Velocity*). Односи се на број завршених задатака по итерацији;

- *Стабилност спринта*. Везан је за Скрам методологију,[77] и односи се на број планираних задатака који су успјешно реализовани.

Ове метрике имају смисла само ако се користе за побољшање процеса, а не за индивидуалну контролу (поготово не за микроменаџинг!). Када је ријеч о језику Гоу, он нам омогућава лаку интеграцију са оруђима за анализу квалитета:

- `go test -cover`, за мјерење покривености тестова;

- `golangci-lint run`, за откривање стилских и логичких грешака;

- `gosec ./...`, за провјеравање сигурностно-безбедносне рањивости кода.

Наравно, резултати се могу, а и требају, увезати у *CI/CD* ток како би се квалитет аутоматски пратио током сваког уписа.

[77] О њој више мало касније, тј. у сљедећем поглављу.

8.3. Критички осврт: Метрике су подршка, а не сврха

Метрике би требало да буду својеврсно средство разумијевања, а не средство казне "лијеног програмера". Када се бројеви користе изван правог контекста, они губе смисао и мотивишу погрешно понашање (на примјер, "писање кода ради повећања покривености"). Због тога метрике морају бити повезане са стварним циљевима, а то су стабилност, одрживост и задовољство корисника.

Иначе, појава када је метрике циљ је позната као Гудхартов закон: "када мјера постане циљ, престаје да буде добра мјера" (*Charles Goodhart* (1936-), британски економиста). У пракси то значи да тим почиње да се равна према броју, а не уистину квалитету. На примјер, ако се покривеност тестовима постави као строги *кључни индикатор учинковитости* (енг. *Key Performance Indicator*), програмери ће природно писати више-мање тривијалне тестове који повећавају проценат покривености, али не откривају стварне дефекте производа. Слично, инсистирање на повећању брзине (тима) доводи до фрагментације задатака и тиме природно до погоршања архитектуре. Када метрика постане циљ, систем неминовно деградира, тј. стварни квалитет опада, често без могућности да се тај процес обрне, јер су лоше праксе већ пустиле коријење.

8.4. Закључак поглавља

Све софтверске метрике су у ствари инструменти једног професионалног инжењера: омогућују рационално доношење одлука и колико-толико објективно мјерење напретка. Ипак, треба бити свјестан њихових ограничења. Просто, метрике захтевају пажљиво тумачење у одговарајућем контексту и њихово повезивање са оним што називамо инжењерско искуство. Препоручена пракса подразумева коришћење више метрика и њихово тумачење у складу са специфичностима пројекта и потребама тима. Опет, вриједност метрике зависи од зрелости онога ко је користи, тј. број не замјењује здрав разум.

8.5. Питања и задаци за провјеру знања

1. Која је сврха софтверских метрика?
2. Објаснити разлику између процесних и производних метрика.
3. Које су најчешће метрике за сложеност кода?
4. Шта представља *ISO/IEC* 25010:2023 модел квалитета?
5. Зашто није пожељно тежити 100% покривености тестовима?
6. Наведите три гоу-оруђа која се користе за мјерење квалитета кода.

9. СОФТВЕРСКИ ПРОЦЕСИ

Сврха поглавља

У овом поглављу различити софтверски процеси и методолошки оквири упознају се, упоређују се, и критички сагледавају ради њиховог прилагођавања стварном контексту пројекта, тима, и клијента.

9.1. О софтверском процесу

Софтверски процес се може дефинисати као скуп активности, метода и пракси којима се управља развојем (и одржавањем) софтвера. Практично, ријеч је о процесу који формулише како се замисао у глави носиоца пројекта претвара у конкретан производ, кренувши од почетних захтјева (клијента), преко реализације, до одржавања. Уколико немамо јасно дефинисан процес, наш тим ће брзо запасти у хаос, импровизацију и врло вјероватно у непотребна дуплирања посла.

Први процесни модели били су линеарни, али су брзо показали ограничења. Средином 1980-их појављују се итеративни и еволутивни модели, а затим, мало касније, и оно што се назива Агилно (као приступ). Свака нова методологија и процес је настајала да би исправила крутост или пак ограничења претходних. Током историје, у посљедњих пар деценија, то је значило да се тражило да процеси буду прилагодљивији

промјенама, да се побољша сарадња и динамика односа у тиму и са самим клијентом, те да би осигурало да крајња испорука заиста буде корисна, тачна, и од користи за крајњег корисника, тј. на његово задовољство.

9.2. Класични модели

Класични модели развоја софтвера представљају историјски темељ софтверског инжењерства. Настали су у вријеме када су системи били мање динамични, промјене рјеђе, а био је јак нагласак на такозваним формалним поступцима, документацији и просто надзору цјелокупног рада. Иако данас постоје флексибилније методологије, класични модели остају важни јер пружају оно што можемо назвати концептуалним оквиром, а то је како организовати посао, како управљати ризиком, и како структурисати процес уређеног развоја. Да се подсјетимо, класични модели, када је ријеч о софтверском инжењерству, укључују сљедеће познате: Инкрементални модел, Водопадни модел, *V*-модел, Спирални модел, Итеративни модел, Еволутивни модели.

Иако се међусобно разликују, класични модели дијеле четири заједничке претпоставке:

(П1) процес развоја може се подијелити на кристално јасне фазе;

(П2) напредак се може формално пратити;

(П3) измјене захтјева се практично не дешавају;

(П4) квалитет се осигурава надзором, провјером, и документовањем.

Ове претпоставке добро функционишу у окружењима гдје су захтјеви стабилни, промјене малобројне, а ризици морају бити формално управљани. Више-мање пластични примјери таквих окружења су: одбрамбени сектор ((војна) намјенска индустрија), ваздухопловство, развој медицинских уређаја, или пак разноразни регулаторни системи.

9.2.1. Основна својства кључних класичних модела

Инкрементални и итеративни модели. Систем се гради у мањим функционалним цјелинама, уз повратне спреге и постепено побољшавање. Ови модели су се рано јавили, али су у неком облику дуго остали, те су на крају постали својеврсни прелаз ка модерним методологијама, јер омогућавају рану испоруку и учење кроз повратне информације.

Водопадни модел. Линеарни приступ у којем се фаза не напушта док није завршена. Доноси јаснођу и чврсту контролу, али предвиђа да се захтјеви не мијењају значајно након почетне фазе — што ријетко одговара реалности савремених пројеката.

***V*-модел.** Надоградња водопада са наглашеном симетријом између фаза развоја и тестирања. Свака фаза има јасно дефинисану верификацију. Погодан је за системе високог ризика, али и даље ограничено реагује на промјене током развоја.

Спирални модел. Фокус на управљању ризиком. Пројекат се развија кроз циклусе, при чему се у сваком кораку анализирају и

редукују доминантни ризици. Модел је моћан, али захтјева искусан тим и значајне ресурсе.

Еволутивни модели. Производ се развија кроз више верзија уз сталну интеракцију са корисницима. Настао је као одговор на потребу да софтвер "расте" заједно са пословним окружењем.

9.2.2. Гдје класични модели губе своју снагу?

У динамичном окружењу, у којем се захтјеви мијењају из седмице у седмицу, класични модели испољавају неколико битних мањкавости:

- спора реакција на промјене у домену и функционалности;
- касно откривање погрешних претпоставки;
- јавља се видно већи трошак корекција у каснијим фазама развоја;
- значајно се ослањају на формализме, што може поприлично успорити напредак;
- не дозвољавају, у суштини, *минимални одрживи производ* (енг. *Minimal Viable Product*), тј. да се рано види радни софтвер.

Упоредном анализом, видно је да класични модели имају предност у пројектима са високим захтјевима који се односе на стабилност или пак контролу ризика (на примјер, у одбрамбеном или медицинском сектору), али у динамичним и иновативним окружењима постају крути. Стога, избор модела треба заснивати на карактеристикама самог пројекта, стабилности захтјева

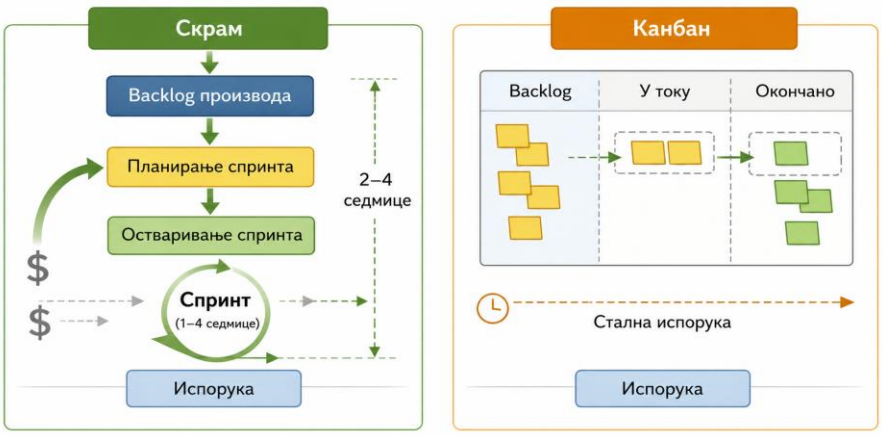
клијената, регулаторној строгости, те потребном нивоу прилагодљивости.

9.2.3. Зашто су ови модели и даље важни?

Иако класични модели нису прилагођени брзим промјенама захтјева и окружења, они ипак представљају темељ разумијевања процеса развоја софтвера. Савремене методологије као што су Агилни приступи или Девопс нису негација традиционалних модела већ њихова еволуција. Итеративност, инкременталност и управљање ризиком су три кључна својства модерног развоја и потичу управо из класичних модела. Отуда имамо да је познавање поменутих класичних приступа важно и за разумијевање контекста, те и разлога зашто су настали нови модели, али и критеријума на основу којих се бира методологија у стварним индустријским пројектима. На примјер, у пракси је често коришћен један, да тако кажемо, хибридни модел у којем развојни тим комбинује итеративни развој из Агилног приступа са формалним фазама провјере и документације старог Водопадног модела. Ово рјешење омогућава прилагођење специфичностима пројекта, попут потребе за прилагодљивошћу, а опет и задовољење одговарајућих регулаторних захтјева.

9.3. Агилне методологије

Агилно (енг. Agile) настаје као одговор на ограничења класичних, суштински линеарних модела, који претпостављају стабилне захтјеве и мало промјена. Агилно као концепт (јер се не ради о једном моделу) не одбацује дисциплину, него напротив, уводи другачију логику развоја: *емпиризам*. Овдје је поменути емпиризам дефинисан као приступ који се заснива на искуству, те сталном прикупљању повратних информација и прилагођавању процеса развоја, а све у складу са стварним сазнањима из праксе (умјесто ослањања на почетна предвиђања). Умјесто предвиђања будућности кроз детаљно планирање, агилност гради софтвер кроз кратке циклусе испоруке, учење и континуирано прилагођавање.[78] Кључна начела (према Манифеста Агилног) истичу сљедеће приоритете: људи и интеракције над процесима, радни софтвер над опсежном документацијом, сарадњу са клијентом над формалним уговорима, као и реаговању на промјене испред строгоће плана.



[78] *Agile* обећава брзу адаптацију, али искуство показује да се најбрже адаптирају управо састанци.

Слика 9.1. Поређење основног тока рада у Скраму и Канбану, са нагласком на разлике у планирању и управљању радом (што укључује и трошкове).

Ови принципи не искључују процесе, документацију или планове. Они наглашавају да њихова вриједност произилази из способности да подрже сарадњу и испоруку. Најпознатији агилни модели (оквири) су *Скрам*, *Канбан* и *Extreme Programming* (*XP*)[79]. У наставку дајемо краће описе Скрама и Канбана.

9.3.1. Скрам оквир

Скрам (енг. *Scrum*) је најраспрострањенији Агилни оквир и почива на три стуба емпиризма: транспарентности, инспекцији и прилагодљивости. Развој се одвија у кратким, временски ограниченим итерацијама под називом "спринтови" (енг. *sprints*), у којима тим испоручује инкремент функционалног, тестирајућег софтвера. Класичне улоге у Скраму су:

- *Product Owner (PO).* Он је носилац производа (софтвера), одговоран за максимизирање вриједности производа. Као обавезу има да управља Product Backlog-ом, поставља приоритете и усмјерава развој ка пословним циљевима. Није ријеч о адмнистративној улози, већ стратешкој;
- *Scrum Master (SM).* Скрам мастер је непосредни носилац процесне дисциплине. Он осигурава да се Скрам правилно примјењује, те уводи тим у добре праксе и подиже организациону зрелост. Не смије да буде "надзорник" (да не

[79] Нећемо обрађивати *Extreme Programming* у овом уџбенику.

дође до микроманаџинга), већ треба да се постави као сервисни вођа;

- *Development Team.* Самоорганизујући и обично мулти-дисциплинарни *развојни тим* стручњака који реализује *инкремент*. Не чине га само програмери! Укључује тестере, аналитичаре, Девопс инжењере и многе друге потребне профиле.

Основни елементи Скрама: *Product Backlog* (функционалности и приоритети), *Sprint Backlog* (план рада за текући спринт), те *Инкремент* (испоручиви дио софтвера).

У Скраму, *церемоније* су дефинисане као формални временски ограничени догађаји, који уводе ритам у рад тима и служе као контролне тачке за планирање, координацију, провјеру, и унапрјеђење начина рада. Оне не производе директно софтвер, већ омогућавају да се рад учини видљивим, да се рано уоче проблеми, и да се тим редовно прилагођава промјенама. Смисао церемонија није у самом извршавању форме, већ у подршци сталној сарадњи и учењу тима. Основне церемоније су: *Sprint Planning*, *Daily Scrum*, *Sprint Review*, и наравно *Sprint Retrospective*. Скрам наглашава ритам, транспарентност и стално побољшавање процеса.

9.3.2. Канбан оквир

Канбан је оквир за управљање протоком посла који потиче из тзв. *Витке*[80] (енг. *Lean*) философије производње. За разлику од

[80] Ријеч је о енглеском термину који нам долази из производног контекста великог јапанског индустријског концерна Тојота. Овај развој је усмјерен на уклањање отпада и оптимизацију протока рада.

Скрама, који ради у временски ограниченим итерацијама, Канбан омогућава континуирану испоруку и постепено побољшавање процеса без увођења фиксних циклуса. Основна идеја је да се визуелизује укупан ток рада, открију уска грла и оптимизује проток тако да тим може испоручивати стабилно и предвидљиво.

Канбан није методологија, нити прецизно прописан процес. То је систем за управљање радом који се може примјењивати самостално или у комбинацији са другим оквирима, укључујући и Скрам. Кључни принципи Канбана су:

- *Визуелизација рада.* Радни ток се приказује на такозваној Канбан табли, подијељеној на колоне које одражавају фазе процеса (нпр. *To Do, In Progress, Testing, Done*). Визуелни приказ омогућава транспарентност, уочавање преоптерећења и боље управљање приоритетима;

- *Ограничење рада у току* (енг. *Work in Progress* (*WIP*)). Ограничење *WIP*-а је суштински елемент Канбана. Ограничење броја ставки које могу бити у одређеној фази спрјечава преоптерећење, смањује вријеме испоруке и открива уска грла у процесу;

- *Управљање протоком.* Циљ Канбана је стабилан, непрекидан и предвидљив проток посла од иницијалног захтјева до испоруке. Тим прати метрике као што су *просјечно вријеме протока* (енг. *lead time*), *трајање рада на задатку* (енг. *cycle time*) и *стопа испоруке* (енг. *throughput*);

- *Изричита правила процеса.* Свако стање на табли има јасно дефинисано значење, критеријуме улаза и излаза (енг.

Definition of Ready / Definition of Done), као и правила преласка из једне фазе у другу;

- *Повратне информације и континуирано побољшање.* Канбан системи редовно користе *рецензије протока* (енг. *flow review*) и ретроспективе усмјерене на уклањање системских узрока проблема. Фокус није на брзој импровизацији, већ на структурном побољшању процеса.

Типична Канбан табла садржи колоне које одражавају фазе у процесу развоја. Примјер минималне структуре:

- *To Do* (спремно за рад);
- *In Progress* (рад у току; ограничен *WIP*);
- *Review/Testing* (преглед/тестирање);
- *Done* (готово).

Свака ставка (*task* (задатак), *user story* (корисничка прича), дефект) представља се картицом која се помјера кроз колоне како напредује кроз процес. Предности Канбан приступа су у томе што омогућава сталну испоруку без потребе за фиксним итерацијама;, побољшава предвидивост кроз мјерење протока, смањује контекстно пребацивање и преоптерећење, открива системске проблеме у процесу (уска грла, недовољни капацитети, неуједначене фазе рада), а и једноставно се уводи и минимално нарушава постојеће токове рада.

Иначе, сам Канбан не дефинише улоге, церемоније ни фиксне циклусе, што може довести до неодговорности или непредвидивости ако тим нема довољан ниво дисциплине. Такође, тимови без зрелости у процесном размишљању могу

погрешно користити Канбан као „слободнији Скрам“, чиме се губи суштина контроле протока. Канбан најбоље функционише у срединама које имају потребу за сталном испоруком, варијабилан прилив задатака, зрелост у самоорганизацији (уз посвећеност мјерењу и побољшању протока.)

9.4. Девопс као продужетак Агилног

Док агилност убрзава развој софтвера кроз кратке итерације, блиску сарадњу и стално прилагођавање, оно не дефинише како обезбиједити да се тај ритам испоруке одржи у производном окружењу. Управо ту настаје Девопс (енг. *DevOps*) као логичан наставак Агилне философије о читавом животном циклусу софтвера.

Девопс није само скуп оруђа, нити методологија у ужем смислу. То је култура и практични оквир који интегрише *развој* (*Dev*) и *оперативу* (*Ops*) тако да тим може стабилно, брзо и поуздано испоручивати софтвер.

Основна идеја је да се елиминише традиционални јаз између тимова који пишу код софтвера и тимова који га одржавају у продукцији. Умјесто уређеног пара (*Dev* → *Ops*), Девопс уводи заједнички проток, заједничке метрике, и заједничку одговорност. Кључни принципи девопс-приступа су:

- *Стална испорука и брзи повратни циклуси.* Умјесто великих, ријетких издања, Девопс оквир омогућава честе, мале и поуздане испоруке. Свака промјена пролази кроз аутоматизовани процес тестирања и провјере ваљаности;

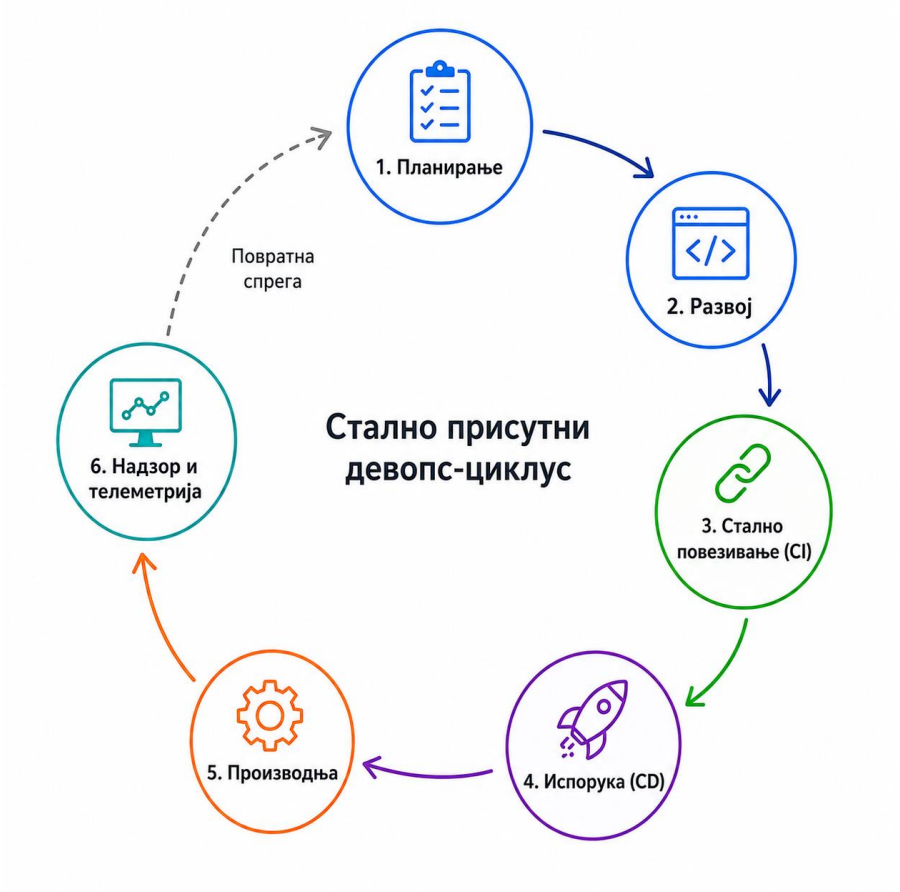
- *Аутоматизација као основно средство.* Аутоматизација елиминише ручне кораке који су спори, непоуздани или подложни грешкама. Она покрива тестирање, грађење апликације, испоруку, те обезбјеђење инфраструктуре;

- *Инфраструктура као код* (енг. *Infrastructure as Code* (*IaC*)). Инфраструктура се описује декларативно као код. То омогућава репродуктивност, надзор и управљање верзијама, брзу изградњу (али и уништавање) окружења и већу поузданост. Популарна рјешења укључују *Terraform*, *Ansible* и *Kubernetes YAML*;

- *Непрестани надзор и телеметрија.* Девопс системи прикупљају метрике у реалном времену: латенцију, грешке, оптерећење, капацитет и корисничке сигнале. Увид у стварно понашање система омогућава брзо реаговање и континуирано побољшање;

- *Заједничка култура, а не само оруђа.* Девопс је учинковит само ако тимови усвоје заједничке циљеве: стабилност, брзину испоруке, транспарентност и заједничку одговорност за квалитет.

Девопс се технички реализује кроз *CI/CD* ланац: *Continuous Integration (CI)* (односи се на непрестано увезивање кода, аутоматско тестирање и аналитику квалитета при свакој промјени) и *Continuous Delivery* или *Continuous Deployment (CD)* (односи се на непрестану испорука или аутоматско пуштање у производњу, уз врло скромну интервенцију људи). *CI/CD* омогућава да се код из развоја у продукцију пренесе у минутима, уз знатно мањи ризик.



Слика 9.2. Поједностављен приказ непрекидног девопс-циклуса који повезује развој, интеграцију, испоруку, пуштање у рад, те надзор софтвера (производа) у јединствен аутоматизован процес.

Гоу је по природи погодан за Девопс окружење јер га карактерише брза компилација и минималне зависности. Најчешће се користе:

- *Docker* за контејнеризацију, јер омогућава исто окружење у развоју, тестирању и продукцији;
- *GitHub Actions*, *GitLab CI* или *Jenkins* за аутоматско покретање тестова, статичку анализу, грађење и испоруку;

- *Kubernetes* за оркестрацију сервиса и такозвано хоризонтално скалирање;
- *Prometheus + Grafana* за надзор, метрике и визуелизацију.

Примјер 9.1. Типичан Девопс процес у Гоу пројекту:

1. Програмер направи промјену и пошаље је у *Git* репозиторијум;
2. *GitHub Actions* аутоматски покреће тестове и статичку анализу;
3. По успјеху, систем гради *Docker* слику и шаље је у регистар;
4. *Kubernetes* аутоматски ажурира сервис на нову верзију;
5. *Prometheus* прати понашање производа и шаље аларме у случају проблема. ▲

Девопс не функционише ако се доживљава само као технологија. Његов успјех зависи од заједничких циљева развоја и оперативе, транспарентности (видљивост метрика и стања система), отворене комуникације, и спремности на аутоматизацију и дугорочну културу побољшања. У том смислу, Девопс није замјена за Агилно, већ његов природни наставак. Док агилни оквири организују начин развоја, Девопс организује начин испоруке и одржавања.

У савременом развоју софтвера, класични девопс-приступ све чешће се проширује концептом Девсекопс (енг. *DevSecOps*), у којем се заштитне провјере непосредно увезују у *CI/CD* токове. Циљ оваквог приступа није да се заштита софтвера посматра као накнадна активност након реализације, већ као саставни дио читавог процеса развоја, тестирања и испоруке.

Посебан значај девсекопс-приступ добија у контексту савремених ВЈМ-агената и аутоматизованог генерисања кода. Иако овакви системи могу убрзати развој, они истовремено могу генерисати небезбједне обрасце реализације, користити рањиве библиотеке, или производити код који пролази функционалне тестове, али не задовољава заштитне захтјеве система. Због тога савремени *CI/CD* токови све чешће укључују аутоматизоване провјере заштићености софтвера.

9.5. Критички осврт: Мит о савршеном приступу

Из искутва је познато да се на стварним пројектима ријетко примјењује једна методологија или један модел у свом чистом облику. Најчешће се комбинују елементи више приступа. На примјер, Водопадни за планирање и Скрам за реализацију. Такав хибридни приступ омогућава стабилност и флексибилност истовремено, односно најбоље од оба свијета (ако је комбиновање урађено како треба). Другим ријечима, ниједна методологија није универзално рјешење. Академски модели често занемарују људске факторе, а то су мотивација, повјерење, и маштовитост. У "земљи стварности" успјех пројекта више зависи од зрелости тима и квалитета међуљудских односа у тиму него од избора процесног модела.

Додатну димензију критике даје и један од аутора Манифеста Агилног, Дејв Томас (енг. *Dave Tomas*), који је у есеју "*Agile is Dead*" истакао да је изворно Агилно током година изгубило суштину. Агилно је, према његовом мишљењу, од

једноставног начина рада заснованог на итерацијама и сарадњи постао производни пакет: скупо сертификован, оптерећен правилима, оруђима и ритуалима који одвраћају пажњу од најважнијег, а то је стварања конкретне вриједности. Томас наглашава да је Агилно првенствено ментални модел, а не академска методологија: "Ако свакодневно испоручујеш мале промјене и учиш из повратних информација, ти си агилан без обзира како се процес (формално) зове."

Овај критички став уклапа се у ширу слику да сваки процесни оквир, ако се примјењује догматски, временом постаје сам себи сврха, нажалост. Зато је у конструкцији процеса важно избјегавати формализам ради формализма (*a.k.a.* "Умјетност ради умјетности" само сад за програмере) и досљедно се враћати суштинским питањима: да ли тим брзо учи, да ли корисник има користи од нашег производа, и да ли се производ заиста побољшава током времена?

Питање. Ако сваки модел развоја обећава да је бољи од претходних, зашто је број модела тако брзо кренуо да расте? Можда зато што су модели добри за људе који их пишу, а не за људе који их користе?

9.6. Закључак поглавља

Софтверски процеси се посматрају као средство, а не као циљ. Добар процес прилагођава се контексту, тимовима и клијентима. Стога, инжењер мора познавати методологије, али и бити спреман да их критички преиспитује и на одговарајући начин комбинује, односно повезује.

9.7. Питања и задаци за провјеру знања

1. Шта је то софтверски процес и која је његова сврха?
2. Које су то главне карактеристике Агилног приступа?
3. Објаснити улогу *Product Owner*-а у Скраму.
4. Шта је то Девопс и како се односи према Агилном приступу?
5. Навести предности и мане неких хибридних модела.
6. Критички анализирати и потом образложити због чега не постоји “савршена“ методологија.

10. ТИМСКИ РАД И УПРАВЉАЊЕ ПРОЈЕКТИМА

Сврха поглавља

У овом поглављу разматра се како квалитет софтверског производа зависи не само од техничког знања већ и од организације рада, расподјеле одговорности, управљања ризицима, и односа међу људима у тиму.

10.1. Значај тимског рада

Развој софтвера се данас сматра колективним подухватом. Уколико нема сарадње између програмера, тестера, аналитичара, и поготово клијената, ниједан софтвер неће моћи бити успјешно испоручен. *Тимски рад* омогућава подјелу одговорности, пренос знања и контролу квалитета. Током времена, у типичном развојном тиму су се искристалисале сљедеће улоге:

- *Управник* (енг. *Project Manager*). Он координише активности, ресурсе и рокове. Сноси највећу одговорност за (не)успјех пројекта;
- *Вођа тима* (енг. *Team Lead(er)*). Технички гледано, води тим и доноси конкретне инжењерске одлуке. Рафинира техничке задатке и осигурава да су спринтови (ако је ријеч о Скраму) завршени на вријеме;

- *Програмери* (енг. *Developers*). Они реализују договорене функционалности производа. Писмено извјештавају о напретку на крају сваког дана (или према договору);

- *Инжењери квалитета софтвера и тестери* (енг. *QA Engineers*). Осигуравају квалитет кроз различите врсте тестирања. Одобравају спринтове (ако је ријеч о Скраму) када су испуњени сви захтјеви за квалитетом;

- *Пројектанти и аналитичари* (енг. *Designers and Analysts*). Дефинишу корисничке захтјеве и прочеља, те осигуравају да су захтјеви јасни и цјеловити прије почетка саме реализације.

Добар тим није скуп појединаца, већ својеврсан организам у којем сваки члан разумије циљ и своју улогу у њему.

Најчешћи узрок неуспјеха пројеката није техничке природе, већ *лоша комуникација*. Откривање и одржавање тог унутрашњег "ритма" тима је битно да би се добило на уједначеној и продуктивној динамици цијелог пројекта. Значи, тимови морају имати јасне канале комуникације: дневни састанци ("откуцаји") који служе за брзи преглед, ретроспективе ("рефлексије") (рецимо, на крају спринтева у Скраму), али и колаборациона оруђа (*Slack, Teams, Jira, Taiga, Confluense, GitHub, BitBucket, итд.*), те транспарентне процесе. Ритмични елементи комуникације дају на добром темпу. Из праксе су проистекла сљедећа кључна правила учинковите комуникације:

1. Јасно дефинисати циљ сваког састанка;

2. Подстицати отвореност и повјерење;

3. Редовно документовати договорене кораке.

10.2. Управљање пројектом

Управљање пројектом обухвата планирање, праћење и контролу извођења активности. Основни циљ је, можемо рећи, да се пројекат заврши у оквиру пројектованог буџета, рока и обима (енг. *scope*). Главни процеси су: иницирање пројекта, планирање активности и ресурса, извршење и праћење напретка, те завршетак и анализа резултата. Већ поменута софтверска оруђа као што су *Trello, Jira* или *Taiga* омогућују визуелно праћење задатака и статуса сваког од појединаца у оквиру пројекта.

У Агилном окружењу пројектно управљање се ослања на итерације и сталне повратне информације. Претходно поменута скрам-оруђа омогућавају визуелно лијеп начин праћење *backlog*-а, спринтова, као и тимских обавеза. Даље, мјерљиви показатељи напретка укључују:

- *Брзину* (енг. *Velocity*) којом тим реализује задатке;
- *Графикон остатка* (енг. *Burndown graph*) којим се прати остатак посла у односу на вријеме;
- *Дијаграм кумулативног тока* (енг. *Cumulative Flow Diagram*) којим се визуализује стабилност читавог процеса.

С друге стране, сваки пројекат садржи неизвјесност. *Управљање ризиком* укључује идентификацију, процјену и планирање одговора на ризике. Типични ризици су: промјене захтјева, недостатак ресурса;, техничка неусклађеност, грешке у интеграцији, или чак одлазак кључних чланова тима. Учинковито

управљање ризиком захтијева редовне ревизије, али и документовање превентивних мјера.

Опет, култура тима има снажан утицај на квалитет рада. Показано је да тимови са високим нивоом повјерења и узајамног поштовања имају већу продуктивност. Просто, мотивација не произилази само из плате, већ и из јасног смисла задатака који се добијају и признања за труд. Зрели тимови развијају такозвани *носећи ментални склоп* (енг. *ownership mindset*), тј. видан осјећај личне одговорности сваког члана тима када је ријеч о резултатима рада.

10.3. Критички осврт: Управљање људима као најтежи дио инжењерства

Иако се често сматра да је најтежи дио пројекта технички, у стварности највећи изазови леже у људском фактору. Академски модели управљања често игноришу психолошке и социјалне аспекте тимског рада. Добар инжењер, поготово вођа тима, мора имати и изражено саосјећање, тј. разумјети људе, а не само код.

У суштини, управљање људима је најтежи дио инжењерства, јер је једино подручје у којем рефакторисање није препоручљиво. Или би пак можда требало да буде...

10.4. Закључак поглавља

На крају овог поглавља можемо казати да успјешан софтверски “бој” не бију добар хардвер и развојна окружења, него “срце” у програмера. Техничко знање је неопходно, и оно је потребан услов, али без доброг тима нема одрживог развоја. Управљање пројектом није бирократија, већ је то начин да се рад учини видљивим, контролисаним, и мотивишућим.

10.5. Питања и задаци за провјеру знања

1. Које кључне улоге у тиму за развој софтвера познајете?
2. Шта подразумијева добро управљање пројектом?
3. Која је сврха познатих скрам-оруђа?
4. Како се то управља ризицима на пројекту?
5. Шта је то носећи ментални склоп?
6. Објаснити зашто је управљање људима најтежи дио инжењерства.

11. ЕТИКА И ПРОФЕСИОНАЛНОСТ

Сврха поглавља

У овом поглављу разматра се професиоинална и етичка одговорност софтверског инжењера према корисницима, колегама, и друштву, као неодвојив дио инжењерског рада.

11.1. Појам професионалне етике

Етика у софтверском инжењерству односи се на одговорност појединца и тима према корисницима, друштву и струци. Инжењер није само техничар, јер он својим радом утиче на људске животе, безбједност и приватност. Из тих разлога, очекује се од инжењера да мора дјеловати у складу са моралним и професионалним начелима, чак и када му нико то формално не прописује. Током развоја, у свакој фази смо на неки начин дужни да себи поставимо питање: “Ко би то могао бити оштећен овом (програмском) функцијом и на какав начин?” Ова врста саморефлексије помаже нам да развијемо културу уграђивања етичких разматрања у свакодневне одлуке нашег рада на софтверу на којем радимо.

Према чувеном *ACM/IEEE Code of Ethics*, кључни принципи етике у развоју софтвера су:

1. Поштовати добробит јавности изнад свега;

2. Избјегавати наношење штете кроз софтверске системе;

3. Бити искрен и транспарентан у раду;

4. Поштовати приватност и интегритет података;

5. Стално унапрјеђивати своје знање и вјештине;

6. Придржавати се закона и прописа, али и етичких норми које их превазилазе.

Треба стално имати у виду да софтвер утиче на друштво (као што и друштво утиче на софтвер): од финансија и здравства, до безбједности и образовања. У том погледу, софтверско инжењерство има карактеристике друштвене научне дисциплине. Ипак, у овом уџбенику сад то нећемо посебно озбиљно разматрати. Инжењер, програмер, мора бити свјестан посљедица својих одлука. У неким случајевима, рецимо у развоју софтвера за медицинске уређаје као што су лабораторијски резултати, тек једна погрешна функција може угрозити животе или тешко пољуљати повјерење јавности. Зато се етичка одговорност не може препустити само управи. Сваки члан тима има моралну дужност да реагује на неправилности. Етичке дилеме у *IT* индустрији су честе: прикривање безбједносних пропуста ради заштите имиџа фирме, коришћење корисничких података без пристанка, развој алгоритама који могу дискриминисати одређене групе, прековремени рад и изгарање као системска појава, и многе друге.

Професионалац мора бити спреман да заузме чврст став, чак и ако то некад није лако.[81] Етички принципи у софтверском

[81] Једном од аутора је познат случај када су наши програмери колективно одбили да раде на пројекту који је везан за НАТО. Или пак случај када је одбијена

инжењерству не односе се само на велике одлуке, већ се примјењују у свакодневном раду кроз конкретне праксе. На примјер, ако ми као програмери примијетимо потенцијалну сигурносно-безбједносну рањивост у коду, онда имамо и етичку дужност да благовремено обавијестимо тим, па и да предложимо неко рјешење, умјесто да игноришемо проблем ради уштеде мало (сопственог) времена. У рецензијама кода то значи давати конструктивне коментаре. Даље, кроз ваљано именовање промјенљивих и функција, етика се огледа у јасноћи која олакшава сарадњу и одржавање. Документовање сложене логике представља одговорност према колегама који ће касније радити на нашем коду. Чак и избор између брзе, али неуредне реализације и чистог, одрживог решења има етичку димензију, јер утиче на дугорочну одрживост пројекта и оптерећење других чланова тима.

11.2. Лиценцирање и етички кодекс компанија

У многим земљама се тежи ка томе да софтверско инжењерство добије статус лиценциране професије, слично адвокатури, архитектури, или медицини. Но, посједовање лиценце нам не гарантује моралност. Сврха тога је да се просто дигне ниво одговорности и повјерења јавности.

У оквиру универзитетских курикулума, ова идеја се обично преноси кроз предмете који детаљније обрађују проблематику

сарадња са програмером који је коментарима у коду деградирао медицински софтвер. Трећи примјер би могао да буде напуштање рада у индустрији игара на срећу.

информатике и друштва, како би се студенти научили да техничке одлуке увијек имају људске посљедице.

IT компаније све више и више усвајају интерне етичке кодексе који дефинишу понашање запослених. Типичне ставке могу да укључују заштиту приватности података, заштиту интелектуалне својине, избјегавање сукоба интересе, одговорно коришћење генеративних модела, и друго. Ипак, етички кодекс има смисла само ако га подржава одговарајућа организациона култура.

Сљедећи примјер илуструје како се етички принципи могу примијенити у пракси.

Примјер 11.1. (Прикривање сигурносно-безбједоносног бага.) Када је откривена рањивост у производњи, управник пројекта је предложио одгађање исправке да би се избјегло кочење система. Тим је ипак инсистирао на етичном приступу: непосредном пријављивању проблема, реализацији хитне закрпе (енг. *patch*-а) и транспарентном обавјештавању корисника о ризику. Иако је овакав развој ситуације привремено утицао на углед компаније, дугорочно је осигурао повјерење корисника. ▲

Овај апстрактни примјер показује да етички ставови не морају бити у сукобу са пословним интересима, већ могу довести до бољих дугорочних резултата за све заинтересоване стране.

11.3. Критички осврт: Етика као отпор формализму

У академским круговима етика се неријетко третира као чисто апстрактна дисциплина. Међутим, у софтверском инжењерству она на неки начи, можда и парадоксално, представља отпор технократији: подсјетник да иза сваког алгоритма стоји човјек. Тамо гдје се све своди на метрике и процесе, етика враћа смисао и човјечност струци.

Опет, кажу да је безбједност највишег степена када је корисник ни не примјећује...

11.4. Закључак поглавља

Етика није додатак струци, већ њен носећи стуб. Програмер је и чувар дигиталног повјерења. Његова одговорност није само да програм ради, већ да се у свом раду води начелима опште правичности и опште добробити.

11.5. Питања и задаци за провјеру знања

1. Који су основни принципи професионалне етике у софтверском инжењерству?
2. Навести три типична етичка изазова у *IT* индустрији.
3. Зашто лиценцирање повећава одговорност инжењера-програмера?

4. Шта садржи етички кодекс компаније?

5. Објаснити зашто је етика отпор технократији.

12. КОНТЕКСТНО-ИНТУИТИВНО КОДИРАЊЕ

Сврха поглавља

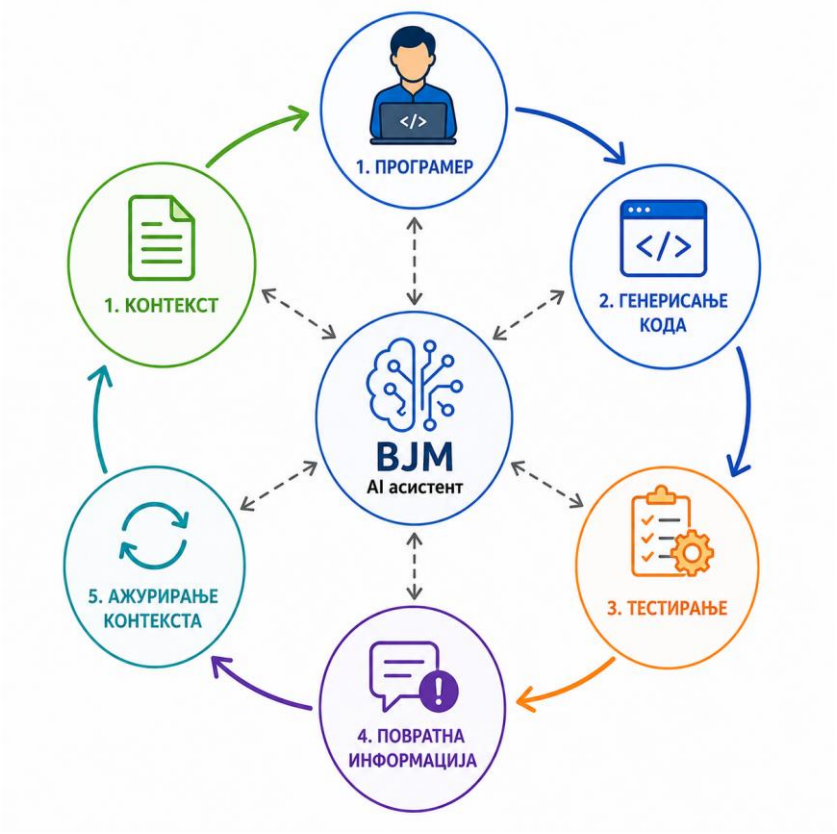
У овом поглављу разматра се како сарадња програмера и генеративних модела мијења процес реализације самог софтвера, те помијера тежиште инжењерског рада са ручног писања кода, ка формулисању намјере, провјери, и управљању контекстом.

12.1. Појам контекстно-интуитивног кодирања

Контекстно-интуитивно кодирање (енг. *Vibe Coding*)[82] се може посматрати као нови приступ/парадигму у софтверском инжењерству у којој програмер и интелигентни агент (најчешће велики језички модел (ВЈМ)) сарађују у оквиру дијалога на природном говорном језику. Програмер је тај који изражава намјеру и дефинише тестове (или услове), а агент на основу контекста и интуитивних сигнала[83] генерише, прилагођава и верификује рачунарски код. Циљ је смањење ручног писања кода и прелазак са синтаксног на семантички и контекстуални ниво развоја софтвера.

[82] Ријеч је о приједлогу термина у академском духу. Вјерујемо да ће се временом искристалисати ваљан назив, како на енглеском, тако и на српском језику.

[83] Који могу бити намјере, корективне упуте, ограничења у значењу, или опет побуде/промптови, и сл.



Слика 12.1. Приказ односа програмера, кода, тестова, и повратне информације BJM-а у циклусу контекстно-интуитивног кодирања.

У класичном *TDD*-у, програмер најприје пише тестове који описују очекивано понашање система, а тек потом пише реализацију која те тестове задовољава. У контекстно-интуитивном кодирању овај однос се проширује: BJM-агент постаје активан учесник у фази реализације, док човјек остаје

ауторитет који формулише тестове, процјењује резултате и усмјерава ток побољшања кода. [84]

Примјер 12.1. Примјена контекстно-интуитивног програмирања у писању кода програма (на језику Гоу) који би требао да провјерава да ли је унијета реченица палиндром.

Фаза 1. (Формулисање теста) У овој фази програмер дефинише тест (задајући тиме контекст) који описује жељено понашање. Агент у овом тренутку не пише никакав код, већ ће касније реализовати функцију која треба да прође дефинисане тестове.

```
package main


import (
   "testing"
)


func TestIsPalindrome(t *testing.T) {
   tests := []struct {
      input string
      want  bool
   }{
      {"Ana voli Milovana", true},
      {"Kajak", true},
```

[84] У нашем случају, *Vibe Coding* не посматрамо као неконтролисано генерисање велики количина кода, као што је то популарно у садашње вријеме, када "програмер" просто одокативним и офрљим побудама изнова и изнова тражи од генеративног модела да му напише гомилу кода, који бива поприлично *баговит*.

```
        {"Golang", false},
    }

    for _, tc := range tests {
        got := IsPalindrome(tc.input)
        if got != tc.want {
            t.Errorf("IsPalindrome(%q) = %v, want %v", tc.input, got, tc.want)
        }
    }
}
```

У овој фази, човјек уноси доменско знање, тј. шта је исправан резултат, а не како га постићи! Иако *TDD* није формални систем доказивања коректности, идеја улазних и излазних услова концептуално је блиска приступу присутном у Хоареовој логици,[85] гдје се исправност програма описује преко тројки *{P} C {Q}*.[86] *TDD* посредно гради улазне и допунске услове, чиме програм постепено добија своју формалну структуру.

Фаза 2. (Агентна реализација) Сада BJM-агент добија побуду (промпт) на коју треба да реагује

[85] Хоареова логика, позната и као аксиоматска семантика, је један формални систем за провјеру коректности програма, Циљ успостављања ове логике је да, прије извршавања програма, имамо могућност да математички докажемо да ће одређени дио кода резултовати очекиваним исходом, наравно, под претпоставком испуњености такозваних улазних услова.

[86] Основни конструкт Хоареове логике: *P* је предуслов, *C* је наредба (програм), *Q* је исход. Тумачи се на сљедећи начин: ако је *P* тачан прије извршавања наредбе *C*, и ако *C* терминира, тада ће *Q* бити тачно након извршавања.

„Реализуј на језику *Go* функцију IsPalindrome тако да сви тестови из TestIsPalindrome прођу успјешно.“

Ова побуда је додатно, имплиицтно, и неформално дефинисање контекста. Конкретније, у овој ситуацији обично имамо да је и човјеку и агенту колико-толико познат шири технички, доменски, и намјенски контекст. Исход прослијеђене побуде заједно са тестовима је сљедећи код.

```
package main


import (
   "strings"
)


func IsPalindrome(s string) bool {
   cleaned := strings.ToLower(strings.ReplaceAll(s, " ", ""))
   n := len(cleaned)
   for i := 0; i < n/2; i++ {
      if cleaned[i] != cleaned[n-1-i] {
         return false
      }
   }
   return true
}
```

Фаза 3. (Потврда и повратна спрега) Програмер покреће тестове на уобичајен начин са go test. На излазу се добија

```
--- FAIL: TestIsPalindrome (0.00s)
    palindrome_test.go:14: IsPalindrome("Ana voli Milovana") = false, want true
FAIL
```

Значи, агент није узео у обзир велика слова и интерпункцију. Људски инжењер сада додатно прецизира свој захтјев:

> „Функција треба игнорисати размаке, велика слова и интерпункцију.“

Фаза 4. (Агентна самокорекција) Агент сада, пратећи допунску инструкцију, мијења претходно произведен код.

```
import (
    "strings"
    "unicode"
)

func IsPalindrome(s string) bool {
    var b strings.Builder
    for _, r := range s {
        if unicode.IsLetter(r) {
            b.WriteRune(unicode.ToLower(r))
        }
    }
    cleaned := b.String()
    n := len(cleaned)
    for i := 0; i < n/2; i++ {
        if cleaned[i] != cleaned[n-1-i] {
```

```
        return false
      }
    }
    return true
}
```

Опет се код тестира, и на излазу се добија

```
ok      palindrome      0.002s
```

Ове четири фазе представљају нешто што бисмо могли назвати типичним током контекстно-интуитивног програмирања (*Vibe-TDD* током). Човјек дефинише тестове и критеријуме, а ВЈМ агент пише и исправља код, при чему тестови служе као својеврсна и објективна арбитража између намјере и резултата. ▲

У контекстно-интуитивном кодирању тежиште се не премјешта са тестирања на импровизацију, већ са ручног писања сваког теста на пажљиво вођење, ревизију и корекцију тестова које иницијално предлаже ВЈМ.

Напомена. У овом случају тестови служе као формална репрезентација намјере. ВЈМ-агент је извршилац и итеративни ученик. Људски програмер остаје креатор логичког оквира и судија квалитета. Интуитивност је присутна у одсуству потпуне формалне спецификације, итеративној корекцији умјесто планирања, и статистичкој селекцији рјешења (ако се предочени ток понавља више пута).

На овај начин, контекстно-интуитивно кодирање надограђује *TDD*, и умијесто ручног преласка са “црвене лампице“

под “зелено свјетло“, агент аутоматски пролази кроз итерације, док човјек осигурава да циљеви остану ваљани.

Примједба. У пракси, писање квалитетних јединичних тестова често захтијева временски напор упоредив са самом реализацијом, а понекад и већи, посебно када је ријеч о искусним програмерима који већ имају јасну слику рјешења.

Имајући у виду претходну примједбу, и сагледавајући предочено контекстно-интуитивно кодирање и употребу ВЈМа, поставља се питање како оптимално расподијелити тај напор. Наиме, умјесто да програмер ручно пише сваки тест од почетка, могуће је иницијално генерисати тестове уз помоћ модела, затим их ручно прецизирати и тек након тога приступити генерисању или доради производног кода. Оваквим приступом задржава се дисциплина TDD-а, али се истовремено користе предности аутоматизованог генерисања, што доводи до бољег односа између уложеног времена и добијеног квалитета.

Напомена. У посљедње вријеме, широка доступност савремених ВЈМ система, као што су ChatGPT, Claude, Gemini, Copilot и слични агенти, додатно је популаризовала овакав начин интеракције човјека и софтверског система. Ипак, у овом тексту нагласак није на конкретној платформи или компанији, већ на самом инжењерском обрасцу сарадње између програмера, тестова и генеративног агента.

12.2. Критички осврт: Заблуда у вези са такозваном вјештачком интелигенцијом

Иако се чини да овај иновативни приступ може убрзати развој и умањити когнитивно оптерећење програмера, он носи и конкретне опасности од којих ми у овом нашем малом уџбенику наводимо сљедеће (а има их сигурно још):

- тестови морају бити *изузетно* прецизни, јер било која двосмисленост може навести агента на погрешну реализацију;
- постоји ризик од повјерења у "зелено свјетло" као доказ исправности, чак и када нам тестови не покривају све случајеве;
- агенти немају стварно разумијевање значења грешке, па ово итеративно учење остаје површинско.

Чињеница да генерисани код пролази дефинисане тестове не представља гаранцију потпуне исправности софтвера. Тестови описују само познате и изричито наведене услове, док велики језички модели могу генерисати реализације које задовољавају тренутни скуп тестова, али не и ширу намјеру или неизричите захтјеве система.

У случајевима непотпуне или двосмислене спецификације, ВЈМ статистички апроксимира намјеру програмера на основу образаца присутних у подацима на којима је обучен. Због тога генерисано рјешење може бити синтаксно исправно и увјерљиво, а ипак концептуално погрешно.

ВЈМ-агенти могу генерисати неисправне *API* позиве,

користити непостојеће библиотеке, водити се погрешним претпоставкама о окружењу, или пак правити логички неисправне алгоритме, при чему излаз често задржава висок степен језичке (софистичке) увјерљивости.

Прекомјерно ослањање на аутоматски генерисан код може довести до постепеног смањења активног аналитичког учешћа програмера, при чему човјек постаје више надзорник извршења него аутор алгоритамске структуре.

Иако тестови представљају централни механизам провјере у контекстно-интуитивном кодирању, они не елиминишу потребу за архитектонским разумијевањем, ревизијом кода и критичком процјеном добијених рјешења.

Контекстно-интуитивно кодирање није замјена за инжењерску дисциплину, већ промјена начина на који се та дисциплина примјењује. Због тога контекстно-интуитивно кодирање не укида потребу за оперативним инжењерским знањем, већ помјера његов фокус са ручне синтаксне реализације ка спецификацији, провјери ваљаности, и контекстуалном управљању развојем.

Заинтересованог читаоца упућујемо да мало “гугла” и прочита по разноразним дискусионим групама нешто о *failures of Vibe Coding*.

12.3. Закључак

Укључивање контекстно-интуитивног кодирања у развој софтвера представља нови приступ у коме се креативност, формална провјера и машинско извођење стапају у један врло специфичан итеративни ток. Човјек постаје инжењер тестова и превасходно намјера, а ВЈМ-агент постаје операционализатор те намјере. Тестови тиме постају један гранични простор у којем се испољава право софтверско знање и умјешност. На овај начин, увезивање контекстно-интуитивног кодирања са класичним техникама програмирања/кодирања обезбјеђује да иновативни агенти раде у складу са утемељеним стандардима.

12.4. Питања и задаци за провјеру знања

1. Објаснити у чему се суштински разликује контекстно-интуитивно кодирање од класичног програмирања.
2. Набројати и укратко објаснити четири главне компоненте које чине инфраструктуру контекстно-интуитивног кодирања (модел, агент, окружење, повратна информација).
3. Зашто се по дискусионим групама на вебу провлачи да контекстно-интуитивно кодирање може довести до губитка дубинског инжењерског знања?
4. Замисли да у тиму уводиш контекстно-интуитивно кодирање за брже добијање прототипова на језику Гоу. Које кораке би поставио да осигураш квалитет кода који генерише агент?

13. БИБЛИОГРАФИЈА

У овом поглављу наведени су извори коришћени током израде уџбеника. Референце су дате у нумерисаном библиографском систему прилагођеном структури овог уџбеника.

[1] Abdiukov, T. (2023). Secure-by-design principles in Agile SDLC: Leveraging formal verification and AI-enhanced code review in CI/CD environments. World Journal of Advanced Engineering Technology and Sciences, 9(1), 494–503. doi: 10.30574/wjaets.2023.9.1.0149

[2] Agile Alliance. (2001). Manifesto for agile software development. URL: https://agilemanifesto.org/ [Accessed: May 17, 2026]

[3] Ahmad, M. O., Dennehy, D., Conboy, K., & Oivo, M. (2018). Kanban in software engineering: A systematic mapping study. Journal of Systems and Software, 137, 96–113. doi: 10.1016/j.jss.2017.11.045

[4] Alenezi, M. (2016). Software Architecture Quality Measurement: Stability and Understandability. International Journal of Advanced Computer Science and Applications, 7(7). doi: 10.14569/IJACSA.2016.070736

[5] Atlassian. (2024). Git tutorials and training. URL: https://www.atlassian.com/git/tutorials [Accessed: May 17, 2026]

[6] Beck, K. (2003). Test-driven development: By example. Addison-Wesley.

[7] Benington, H. D. (1956). Production of large computer programs. In Proceedings of the Symposium on Advanced Programming Methods for

Digital Computers (pp. 15–27). Navy Mathematical Computing Advisory Panel.

[8] Benington, H. D. (1983). Production of large computer programs. IEEE Annals of the History of Computing, 5(4), 350–361. doi: 10.1109/MAHC.1983.10102

[9] Boehm, B. W. (1981). Software engineering economics. Prentice-Hall.

[10] Boehm, B. W. (1984). Verifying and validating software requirements and design specifications. IEEE Software, 1(1), 75–88. doi: 10.1109/MS.1984.233675

[11] Boehm, B. W. (1986). A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes, 11(4), 14–24. doi: 10.1145/12944.12948

[12] Brito, R., & Valente, M. T. (2020). RefDiff4Go: Detecting refactorings in Go. In Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '20) (pp. 1–10). Association for Computing Machinery. doi: 10.1145/3425269.3425274

[13] Brooks, F. P. (1995). The mythical man-month: Essays on software engineering. Addison-Wesley.

[14] Bröhl, A., & Dröschel, W. (Eds.). (1995). Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden (2nd ed.). Oldenbourg.

[15] Cohn, M. (2004). User stories applied: For agile software development. Addison-Wesley.

[16] Čapka, L. (2025, March 4). Go: Security at its core. Cybroslabs. URL: https://www.cybroslabs.com/en/blog/1-go-security-at-its-core/ [Accessed: May 17, 2026]

[17] Defense Acquisition University. (2020). Agile pilots guidebook (Version 1.0, 27 February 2020). U.S. Department of Defense.

[18] DevOps Research and Assessment (DORA). (2021). 2021 Accelerate State of DevOps Report (PDF). URL: https://www.devops-research.com/research.html [Accessed: May 17, 2026]

[19] Dijkstra, E. W. (1972). The humble programmer. Communications of the ACM, 15(10), 859–866. doi: 10.1145/355604.361591

[20] DORA. (2025). DORA's software delivery metrics: the four keys. URL: https://dora.dev/guides/dora-metrics-four-keys/ [Accessed: May 17, 2026]

[21] Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.

[22] Farshidi, S., Bennin, K. E., Babur, Ö., Sallou, J., Kassahun, A., & Tekinerdogan, B. (2026). Advancing research software engineering with AI: A research framework. Automated Software Engineering, 33(79). doi: 10.1007/s10515-026-00621-0

[23] Forsberg, K., & Mooz, H. (1991). The relationship of system engineering to the project cycle. In Proceedings of the First Annual Symposium of National Council on System Engineering (NCOSE) (pp. 57–65). NCOSE.

[24] Forsberg, K., Mooz, H., & Cotterman, H. (2005). Visualizing project management: Models and frameworks for mastering complex systems (3rd ed.). Wiley.

[25] Fowler, M. (2003). UML distilled: A brief guide to the standard object modeling language (3rd ed.). Addison-Wesley.

[26] Freeman, E., & Robson, E. (2020). Head first design patterns: A brain-friendly guide (2nd ed.). O'Reilly Media.

[27] Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2016). A dissection of the test-driven development process: Does it really matter to test-first or to test-last? arXiv [Preprint]. URL: https://arxiv.org/abs/1607.07654 [Accessed: May 17, 2026]

[28] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.

[29] Glass, R. L. (2003). Facts and fallacies of software engineering. Addison-Wesley.

[30] Google. (2023). The Go programming language specification. URL: https://go.dev/ref/spec [Accessed: May 17, 2026]

[31] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik, 38, 173–198. https://doi.org/10.1007/BF01700692

[32] Griesemer, R., Pike, R., & Thompson, K. (2009). Go: Language design in the service of software engineering [Conference presentation]. URL: https://golang.design/history/ [Accessed: May 17, 2026]

[33] Hourigan, A., Hanslo, R. (2025). The integration of agile methodologies in DevOps practices within the information technology industry. African Conference on Information Systems and Technology

(ACIST) 2025. arXiv [Preprint] URL: https://arxiv.org/abs/2508.21811 [Accessed: May 17, 2026]

[34] Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.

[35] Hunt, A., & Thomas, D. (2019). The pragmatic programmer: Your journey to mastery (20th anniversary ed.). Addison-Wesley.

[36] IEEE Computer Society. (2014). Software engineering body of knowledge (SWEBOK V3.0). IEEE Press.

[37] International Organization for Standardization. (2006). ISO/IEC 14764:2006 software engineering - Software life cycle processes - Maintenance. ISO.

[38] International Organization for Standardization. (2023). ISO/IEC 25010:2023 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Product quality model. ISO. https://www.iso.org/standard/78176.html

[39] Kanbanstudy. (2025). Kanban Body of Knowledge (KBOK Guide). Kanbanstudy.

[40] Khan, A. I., Qurashi, R. J., & Khan, U. A. (2011). A comprehensive study of commonly practiced heavy and light weight software methodologies (arXiv:1111.3001) arXiv [Preprint] doi: 10.48550/arXiv.1111.3001 [Accessed: May 17, 2026]

[41] Kitware, Inc. (2024). CMake: The standard build system – Features. CMake. URL: https://cmake.org/features/ [Accessed: May 17, 2026]

[42] Korać, D., Čvokić, D., & Simić, D. (2024). Разматрање концепта безбједности и сигурности у ИТ контексту. ИнфоМ — Часопис за информационе технологије и мултимедијалне системе, 23(78). URL: https://infom.fon.bg.ac.rs/index.php/infom/article/view/2642

[43] Korać, D., Čvokić, D., & Simić, D. (2025). Computational engineering approach-based modeling of safety and security boundaries: A review, novel model, and comparison. Archives of Computational Methods in Engineering. doi: 10.1007/s11831-025-10352-2

[44] Kuhrmann, M., Tell, P., Hebig, R., Klünder, J., Münch, J., Linssen, O., Schmietendorf, A., Schütze, B., Tichy, M., & Weyer, T. (2021). What makes agile software development agile? IEEE Software, 38(3), 72-79. doi: 10.1109/TSE.2021.3099532

[45] Lange, J., Ng, N., Toninho, B., & Yoshida, N. (2016). Fencing off Go: Liveness and safety for channel-based programming. Journal of Logical and algebraic Methods in Programming, 85(6), 1028-1049. doi: 10.1109/MS.2020.3045415

[46] Larman, C. (2004). Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development (3rd ed.). Prentice Hall.

[47] Maass, M., Clement, M.-P., & Hollick, M. (2021). Snail mail beats email any day: On effective operator security notifications in the Internet. In Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES 2021) (pp. 1–13). Association for Computing Machinery. doi: 10.1145/3465481.3465743

[48] Martin, R. C. (2009). Clean code: A handbook of agile software craftsmanship. Prentice Hall.

[49] Martin, R. C. (2011). The clean coder: A code of conduct for professional programmers. Prentice Hall.

[50] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, 2(4), 308–320. doi: 10.1109/TSE.1976.233837

[51] McConnell, S. (2000). Cargo Cult Software Engineering. IEEE Software, 17(2), 11–13. doi: 10.1109/52.841112

[52] McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.

[53] Menger, K. (1927). Zur allgemeinen Kurventheorie. Fundamenta Mathematicae, 10, 96–115.

[54] Muñoz Barón, M., Wyrich, M., & Wagner, S. (2020). An empirical validation of cognitive complexity as a measure of source code understandability. arXiv [Preprint] URL: https://arxiv.org/abs/2007.12520 [Accessed: May 17, 2026]

[55] Naur, P., & Randell, B. (Eds.). (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO.

[56] Nord, R., Ozkaya, I., Sangwan, R., & Koontz, R. (2014). Architectural Dependency Analysis to Understand Rework Costs for Safety-Critical Systems. In Companion Proceedings of the 36th International Conference on Software Engineering (pp. 185–194). ACM. doi: 10.1145/2591062.2591108

[57] Open Worldwide Application Security Project. (2025). OWASP Top 10:2025 (Release). OWASP. URL: https://owasp.org/Top10/2025/ [Accessed: May 17, 2026]

[58] Open Worldwide Application Security Project. (2025, August). OWASP Secure by Design Framework (Draft version 0.5.0). OWASP. URL: https://owasp.org/www-project-secure-by-design-framework/ [Accessed: May 17, 2026]

[59] Parsa, S., Zakeri-Nasrabadi, M., & Turhan, B. (2025). Testability-driven development: An improvement to the TDD efficiency. Computer Standards & Interfaces, 103877. doi: 10.1016/j.csi.2024.103877

[60] Pike, R. (2012). Go at Google: Language design in the service of software engineering [Keynote presentation]. ACM SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2012), Tucson, AZ, United States.

[61] Poppendieck, M., & Poppendieck, T. (2003). Lean software development: An agile toolkit. Addison-Wesley.

[62] Pressman, R. S., & Maxim, B. R. (2020). Software engineering: A practitioner's approach (9th ed.). McGraw-Hill Education.

[63] Reitz, K. (2025). Go - The pragmatic simplifier. URL: https://kennethreitz.org/ [Accessed: May 17, 2026]

[64] Royce, W. W. (1970). Managing the development of large software systems. In Proceedings of IEEE WESCON (Vol. 26, pp. 1–9).

[65] Samira, Z., Wondaferew Weldegeorgise, Y., Osundare, O. S., Ekpobimi, H. O., & Kandekere, R. C. (2024). CI/CD model for optimizing software deployment in SMEs. Magna Scientia Advanced Research and Reviews, 12(01), 056–077. doi: 10.30574/msarr.2024.12.1.0147

[66] Schneider, J., & Smalley, I. (n.d.). What is test-driven development (TDD)? IBM Think. https://www.ibm.com/think/topics/test-driven-development/ [Accessed: May 17, 2026]

[67] Sethi, R. (2023). Software engineering: Basic principles and best practices. Cambridge University Press. ISBN 9781316511944

[68] Shaydulin, R., & Sybrandt, J. (2017). To agile, or not to agile: A comparison of software development methodologies. arXiv [Preprint]. https://arxiv.org/abs/1708.01903 [Accessed: May 17, 2026]

[69] Sommerville, I. (2016). Software engineering (10th ed.). Pearson Education.

[70] Spinellis, D. (2012). Code reading: The open source perspective. Addison-Wesley.

[71] Schwaber, K., & Sutherland, J. (2013). The Scrum Guide: The rules of the game (Scrum Guide). Scrum.org & Scrum Inc. https://scrumguides.org/ [Accessed: May 17, 2026]

[72] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5(2), 285–309.

[73] U.S. Department of Defense. (1988). Defense system software development (DOD-STD-2167A). Department of Defense.

[74] United States Office of Strategic Services. (1944). Simple Sabotage Field Manual. Strategic Services Field Manual No. 3. Washington, DC: Office of Strategic Services.

[75] Vegendla, A., Nguyen Duc, A., Gao, S., & Sindre, G. (2018). A systematic mapping study on requirements engineering in software ecosystems. Journal of Information Technology Research, 11(1), 1–27. doi: 10.4018/JITR.2018010101

[76] Warnars, S. (2010). Object-oriented modelling with Unified Modelling Language 2.0 for simple software application based on agile

methodology. arXiv [Preprint] https://arxiv.org/abs/1006.1683 [Accessed: May 17, 2026]

[77] Wake, B. (2003, November 1). INVEST in good stories, and SMART tasks. XP123. URL: https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/ [Accessed: May 17, 2026]

[78] Wagner, S. (2013). Software product quality control. Springer.

[79] Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C., & Müller, P. (2021). Gobra: Modular specification and verification of Go programs. arXiv [Preprint] https://arxiv.org/abs/2105.13840 [Accessed: May 17, 2026]

[80] Yourdon, E. (1997). Death march: The complete software developer's guide to surviving "mission impossible" projects. Prentice Hall.

[81] Zuberek, W. M. (2003). Petri nets and timed Petri nets in modeling and analysis of concurrent systems - an overview (Technical Report). Memorial University of Newfoundland.

[82] Прерадовић Љ, Дејановић Р. Софтверски инжењеринг, 2010, Архитектонско-грађевински факултет, Универзитет у Бањој Луци, 2010.

ПРИЛОГ А: ИНТЕРВЈУ СА ДИРЕКТОРОМ КОМПАНИЈЕ *TTTechAuto* Д.О.О. БАЊА ЛУКА

У наставку је дат интервју са Тихомиром Винчићем, директором компаније *TTTechAuto* из Бањалуке.

Техничка питања

Питање: Гледано са ваше позције, како *TTTechAuto* Бања Лука балансира између онога што бисмо могли назвати академским принципима, сједне стране, и индустријским потребама, с друге стране, када је ријеч о развоју софтвера за аутомобилску индустрију?

Одговор: "Па, факултет нам је скоро свима полазна тачка, али не и крај приче, јер право учење креће послије факултета. Поменути академски принципи дају структуру и сигурност, али прави тест долази тек када се руке упрљају, тј. када се академска разматрања нађу у рингу са стварним свијетом. У пракси, рецимо, користимо формалне методе верификације, али их комбинујемо са сада Агилним начином рада, јер индустрија тражи и брзину и флексибилност. То је, бар према мом мишљењу, својеврсни ход по жици, у којем се стално на нешто пази, нешто подешава, тј. то није баш нешто што се једном ријеши и онда се на то заборави."

Питање: Који су вам највећи изазови у развоју софтвера, гледано из угла безбједности критичних система, тек како их рјешавате?

Одговор: "Највећи изазов је, можда, према мом мишљењу предвидљивост. Када систем постане довољно сложен, више није довољно да он “углавном ради”, како је то било у мојим млађим програмерским данима. Систем мора да се понаша тачно онако како очекујемо, у сваком могућем сценарију, а то је, поприлично тежак изазов. Зато се ослањамо на врло, врло јасне архитектуре, и ту нам је планирање од изузетне важности, а потом и на врло строге процесе тестирања. Паралелно с тим, много улажемо у едукацију тима, посебно када је ријеч о безбједности, јер је, да се разумијемо, много лакше изградити добру праксу од почетка него је накнадно исправљати."

Питања о развоју каријере

Питање: Шта тражите у младим инжењерима који се пријављују за позицију у *TTTechAuto*?

Одговор: "Техничко знање је важно, али оно се временом надограђује. Оно што је теже научити је радозналост и спремност да се из грешака нешто извуче. Посебно цијенимо када неко не прихвата рјешења здраво за готово, већ покушава да разумије зашто је нешто урађено на одређени начин. Тај ‘зашто’ је често кључ доброг инжењерског размишљања."

Питање: Какав савјет бисте дали студентима који желе да раде у софтверској индустрији?

Одговор: "Тражите прилике гдје софтвер има стварне посљедице. Пројекте у којима код није само апстракција, већ нешто што се сударa са физичким свијетом. Када видите како се ваше одлуке

одражавају на стварни систем, много брже схватите шта је заиста важно. Синтакса се научи, али разумијевање система се гради искуством."

Стратешка питања

Питање: Како се *TTTechAuto* позиционира у односу на глобалне трендове електрификације и аутономне вожње?

Одговор: "Наш фокус је на технологијама које омогућавају те трендове, а не на самим крајњим производима. Развијамо софтверске платформе које омогућавају да електрификација и аутономна вожња буду имплементиране на сигуран и поуздан начин, без обзира на конкретан хардвер. Другим ријечима, ми смо *enabler*, а не произвођач финалног система."

Питање: Како одржавате техничку изузетност у региону са ограниченим ресурсима?

Одговор: "Фокусирамо се на дубину, а не на ширину. Градимо центре изврсности унутар конкретних домена и ту концентришемо знање и искуство. Инжењери тако добијају прилику да раде на сложеним проблемима и да се дугорочно развијају у својој области, што је кључно за задржавање квалитета."

Питања о култури рада

Питање: Како управљате техничким дугом у пројектима са дугим животним циклусима?

Одговор: "На технички дуг, иако не волим тај термин, гледамо слично као и на финансијски. У ствари, они су међусобно поприлично повезани. Ако га игноришемо, камате нас стигну, и тада ситуација буде још гора. Због тога гледамо да имамо јасне планове “амортизације” и да у сваком развојном циклусу остављамо простор за рефакторисање, односно унапрјеђење постојећег кода. У посљедње вријеме смо се посебно фокусирали и на квалитет кода."

Питање: Шта би била најважнија лекција коју сте научили за све ове године, али као инжењер, на челу компаније?

Одговор: "Да, јасно, техника сама по себи није циљ, нити би требала да буде, а надамо се да неће то ни постати. Она је средство и ништа више. У суштини, најбоље техничке одлуке нису нужно оне које најелегантније изгледају на папиру, већ су то управо оне што најбоље рјешавају стварне проблеме и служе људима који ће тај софтвер користити у будућности.”

ПРИЛОГ Б: БРЗИ РЕФЕРЕНТНИ ВОДИЧ

Табела скраћеница и акронима

Сљедећа табела садржи најчешће коришћене скраћенице и акрониме у области софтверског инжењерства, са њиховом пуном формом и кратким објашњењем контекста употребе.

Скраћеница / акроним	Пуни назив	Објашњење / контекст употребе
AES-256	*Advanced Encryption Standard (256-bit)*	Напредни стандардни симетрични алгоритам шифровања са високим степеном заштите.
ACM	*Association for Computing Machinery*	Међународно струковно удружење у области рачунарства
API	*Application Programming Interface*	Прочеље за комуникацију између различитих софтверских компоненти.
BRY	*Build Reuse Yourself*	Принцип поновне употребе кода - градити тако да се компоненте могу лако опет употријебити у будућим пројектима. Ово је ауторски предложено начело.
CD	*Continuous Deployment*	Аутоматизовани процес имплементације нових верзија апликације.

CI	*Continuous Integration*	Практика редовног интегрисања кода ради откривања грешака у раној фази.
CI/CD	*Continuous Integration / Continuous Deployment*	Комбинована Девопс пракса аутоматизације интеграције и испоруке.
CORS	*Cross-Origin Resource Sharing*	Механизам контроле приступа у веб-апликацијама који регулише које домене сервер дозвољава.
CQRS	*Command–Query Responsibility Segregation*	Архитектонски образац који раздваја операције читања и писања.
CSRF	*Cross-Site Request Forgery*	Тип хакерског напада када је ријеч о веб-апликацијама
CSS	*Cascading Style Sheets*	Језик за дефинисање изгледа и стила веб-страница.
DB	*Database*	База података – структурисано складиште за чување информација.
DDD	*Domain Driven Design*	Доменски вођено пројектовање - приступ у којем се структура и понашање софтверског система темеље на стварном домену пословања, а не на техничким слојевима.
DLQ	*Dead Letter Queue*	Ред у који се смјештају неуспјешно обрађене поруке.

DRY	*Don't Repeat Yourself*	Начело којим се настоји избјећи дуплирање истог знања или логике у више дијелова система. Слично је *BRY*-начелу.
HTML	*Hypertext Markup Language*	Језик за означавање структура веб-страница.
HTTP	*Hypertext Transfer Protocol*	Протокол за комуникацију у оквиру Веба.
IDE	*Integrated Development Environment*	Интегрисано окружење за програмирање које обједињује едитор, компајлер и дебагер.
IEC	*International Electrotechnical Commission*	Међународна комисија за електротехнику
IEEE	*Institute of Electrical and Electronics Engineers*	Професионална организација која дефинише бројне стандарде када је ријеч о IT-у.
ISO	*International Organization for Standardization*	Међународна организација за стандардизацију.
JSON	*JavaScript Object Notation*	Формат за размјену података између система.
JWT	*JSON Web Token*	Формат токена који се користи за сигурну размјену података о идентитету и аутентификацију у веб-апликацијама.
OOP	*Object-Oriented Programming*	Парадигма програмирања заснована на објектима и класама.

ORM	*Object-Relational Mapping*	Техника која омогућава пресликавање објектног модела апликације на релациону базу података без ручног писања *SQL*-упита/захтјева.
OWASP	*Open Web Application Security Project*	Међународна организација која објављује стандарде и смјернице у вези са заштитом веб-апликација.
PDF	*Portable Document Format*	Преносиви формат документа
PM	*Project Manager*	Особа задужена за управљање пројектом, роковима и ресурсима.
QA	*Quality Assurance*	Скуп пракси усмјерених на обезбјеђивање квалитета софтвера.
REST	*Representational State Transfer*	Архитектонски стил за пројектовање веб-сервиса.
RPC	*Remote Procedure Call*	Механизам којим програм позива процедуру или функцију на удаљеном систему
SDLC	*Software Development Life Cycle*	Животни циклус развоја софтвера
SLA	*Service Level Agreement*	Уговор о нивоу услуге који дефинише очекивану учинковитост и одговорности
SMS	*Short Message Service*	Услуга слања кратких текстовних порука
SMTP	*Simple Mail Transfer Protocol*	Прости протокол трансфера поште

SQL	*Structured Query Language*	Језик за рад са релационим базама података.
SRS	*Software Requirements Specification*	Формални документ са описом функционалних и нефункционалних софтверских захтјева
TDD	*Test-Driven Development*	Методологија развоја у којој се тестови пишу прије реализационог кода.
TLS	*Transport Layer Security*	Криптографски протокол за безбједну комуникацију преко мреже
UI	*User Interface*	Корисничко прочеље – визуелни дио рачунарске апликације преко које корисник успоставља садејство са апликацијом.
UML	*Unified Modeling Language*	Језик за визуелно моделовање структура и понашања система.
UX	*User Experience*	Корисничко искуство, тј. субјективни утисак корисника током садејства са системом.
VCS	*Version Control System*	Систем за праћење и управљање измјенама у изворном коду.
WIP	*Work in Progress*	Количина рада у току; кључни појам Канбан када је ријеч о управљању и надзору протоком
XSS	*Cross-Site Scripting*	Напад убризгавањем злонамјерног кода у веб-страницу који омогућава извршавање *JavaScript*-а на страни корисника.

YAML	*YAML Ain't Markup Language*	Формат за конфигурационе датотеке, читљив за људе.

Табела гит-наредби

Овај дио прилога садржи најчешће коришћене гит-наредбе, симболе и опис модела гранања. Гит је систем за контролу верзија који омогућава учинковиту сарадњу више програмера и праћење историје измјена кода.

Гит-наредба	Опис функције	Напомена / начин употребе
git init	Иницијализује нови *git*-репозиторај.	Користи се при покретању новог пројекта.
git clone <url>	Умножава постојећи репозиторај са удаљеног сервера.	Почетни корак при раду са постојећим пројектом.
git status	Приказује тренутно стање радног директорија и staging-области.	Користи се прије уписа промјене ради провјере измјена.
git add <file>	Додаје датотеке у staging-област.	Припрема датотеке за упис промјене.
git commit -m "msg"	Снима тренутне промјене у историју верзија са описом.	Основна јединица рада у git-систему.
git log	Приказује историју уписа промјена.	Корисно за праћење измјена кода.
git diff	Приказује разлике	Користи се за ревизију

	између верзија датотека.	измјена прије уписа.
git branch	Приказује постојеће гране или креира нову.	Кључно средство за изоловани развој функција.
git checkout <branch>	Прелази на наведену грану.	Користи се за пребацивање између грана.
git merge <branch>	Спаја наведену грану са тренутном.	Увезује рад из упоредних линија развоја.
git pull	Преузима и увезује измјене са удаљеног репозиторија.	Комбинује `fetch` и `merge` у једну наредбу.
git push	Отпрема локалне измјене на удаљени репозиториј.	Користи се за дијељење рада са тимом.
git stash	Привремено склања тренутне измјене без уписа промјене.	Корисно при брзом преласку на другу грану.
git revert <commit>	Поништава промјене одређеног уписа промјене формирањем новог.	Бољи начин враћања промјена.
git reset --hard	Враћа стање радног директорија на одређени упис промјене.	Опрезно користити — може довести до губитка незабиљежених измјена и промјене тренутног стања гране.
main (или master)	Главна стабилна грана у пројекту.	Садржи званична издања софтвера.
develop	Главна грана за развој.	Садржи најновије, али још нестабилне измјене.
feature/<име>	Привремена грана за развој нове	Спаја се са develop након завршетка рада.

	функционалности.	
hotfix/<име>	Грана за брзе исправке грешака на продукцији.	Спаја се са main и develop након рјешавања проблема.

ПРИЛОГ B: ПРИМЈЕР ДОБРО НАПИСАНЕ СПЕЦИФИКАЦИЈЕ СОФТВЕРСКИХ ЗАХТЈЕВА

Овај прилог садржи спецификацију захтјева за подсистем обавјештавања у оквиру платформе за е-трговину. Документ је намјењен инжењерима, тестерима, девопс-тимовима и свим учесницима у развоју и одржавању система.[87]

1. УВОД

1.1. Сврха документа

Ова спецификација описује функционалне, нефункционалне, везивне, и оперативне захтјеве подсистема за обавјештавање корисника. Документ треба да служи као заједничка референтна тачка за реализацију, тестирање, девопсно увезивање, као и правну усклађеност система.

1.2. Опсег

Подсистем омогућава платформама за е-трговину да поуздано шаљу обавјештења корисницима путем *Email*-а, *SMS*-а и *Push* канала. Систем подржава хитне поруке, механизме поновног слања, *rate limiting* и пуну оперативну видљивост над испоруком.

1.3. Дефиниције и скраћенице

DLQ – Dead Letter Queue

SLA – Service Level Agreement

RTO/RPO – Recovery Time/Point Objective

NFR – Non-Functional Requirement

[87] И наравно, овакве спецификације нећете срести у пракси, не због тога што је пракса заснована на лошим примјерима спецификација, него што је овај примјер ипак поприлично идеалистичан. Другим ријечима, ово је примјер "претјерано добре" спецификације, а можда и "непотребно претјерано добре".

2. ОПШТИ ОПИС

2.1. Перспектива система

Подсистем је реализован као самостални микросервис који излаже *REST API* интерфејс. Унутрашња архитектура обухвата сљедеће компоненте:

- *Delivery Engine*
- *Channel Providers (Email/SMS/Push)*
- *Retry Scheduler*
- *Monitoring & Metrics*
- *DLQ* сервис
- *Admin* интерфејс за преференције

2.2. Актери (Учесници)

- Корисник – прималац обавјештења
- Администратор – управља конфигурацијама/поставкама и надгледа рад система
- Систем наруџби – започиње догађаје за слање порука
- Провајдери канала – спољне услуге за Email, SMS, Push испоруку

2.3. Ограничења

- Систем мора бити усклађен са *GDPR/HIPAA* регулативама
- Сви комуникациони канали морају користити *TLS* 1.3+ (или пак новији)
- Записници (логови) и историја испорука задржавају податке највише 90 дана

2.4. Претпоставке и зависности

- *SMS* провајдер треба да обезбиједи *SLA ≥ 99.9%*
- *Email* сервис подржава *SMTP* преко *TLS* 1.3
- *Push* потпуно зависи од *Firebase* и *APNs* инфраструктуре

3. ФУНКЦИОНАЛНИ ЗАХТЈЕВИ (*FR*-ови[88])

3.1. Слање порука

FR-1: Систем подржава слање порука путем *Email, SMS* и *Push* канала.

FR-2: Иницијатор слања порука мора изричито навести жељени канал.

FR-3: У случају недоступности канала систем мора:

- евидентирати неуспјешну испоруку;
- Извршити поновне покушаје слања уз e*xponential backoff retry* (1s, 2s, 4s, 8s, 16s),
- након пет неуспјеха послати поруку у *DLQ*,
- послати аларм администратору.

FR-4: Свака порука мора имати јединствени *UUID*, временску ознаку и метаподатке.

3.2. Корисничке преференције

FR-5: Корисник може засебно омогућити/онемогућити поједине канале за пријем порука.

FR-6: Ако је канал онемогућен, систем га не користи осим ако корисник није изричито означио да хитне поруке имају *override* (двострука сагласност).

3.3. Хитне поруке

[88] скраћеница за *Functional Requirements.*

FR-7: Хитне поруке се шаљу преко најбржег доступног канала који корисник није изричито забранио.

FR-8: Ако ниједан канал није доступан, систем одмах активира *fallback* на *Email + Push* са *DLQ backup*-ом.

3.4. Историја и *auditing*

FR-9: Систем чува историју свих покушаја испоруке у трајању од 90 дана.

FR-10: Администратор може прегледати историју цјелокупну историју испорука и њихове исходе.

4. НЕФУНКЦИОНАЛНИ ЗАХТЈЕВИ (*NFR*-ови[89])

4.1. Учинак

NFR-1: Времена обраде морају да задовље сљедеће перцентилне услове: P50 < 50ms, P95 < 150ms, P99 < 200ms, P99.9 < 500ms

4.2. Скалабилност

NFR-2: Систем мора скалирати на најмање 100 000 порука у минути.

4.3. Поузданост

NFR-3: Укупна успјешност испоруке мора бити ≥ 99.8%.

4.4. Доступност

NFR-4: Годишњи *Uptime* мора бити ≥ 99.95%

4.5. Пројектно заштићен (*Secure by Design*)

Садржај порука у мировању шифровати *AES*-256.

NFR-5: Сви *API* захтјеви морају бити аутентификовани преко *OAuth2/JWT* механизма.

[89] скраћеница за *Non-Functional Requirements*

4.6. Rate Limiting

NFR-6: Дозвољено је највише 10 порука по кориснику у минуту.

5. OPERATIONS & OBSERVABILITY

5.1. Monitoring

NFR-7: Систем мора излагати метрике:

- *delivered_count*
- *failed_count*
- *retry_count*
- *channel_latency_ms*

NFR-8: *Health-check endpoint* мора одговорити у року < 100ms.

5.2. Distributed Tracing

NFR-9: Свака порука носи *trace-id* који се пропагира кроз све повезане сервисе.

6. *DISASTER RECOVERY*

FR-11: Све неуспјешне поруке након 5 покушаја иду у *DLQ*.

NFR-10: *RTO* < 15 минута.

NFR-11: *RPO* < 5 минута.

7. ЗАХТЈЕВИ УВЕЗИВАЊА

7.1. *REST API* структуре

SendRequest

```
type SendRequest struct {
```

```
    UserID    int             `json:"user_id"`
    Message   string          `json:"message"`
    Type      NotificationType `json:"type"`
    Urgent    bool            `json:"urgent"`
    IdempotencyKey string      `json:"idempotency_key"`
}
```

SendResponse

```
type SendResponse struct {
    MessageID   string    `json:"message_id"`
    Status      string    `json:"status"` // queued, sent, failed
    Timestamp   time.Time `json:"timestamp"`
    RetryCount  int       `json:"retry_count"`
}
```

ErrorResponse

```
type ErrorResponse struct {
    Code    string `json:"code"` // CHANNEL_DISABLED, RATE_LIMITED, INVALID_TYPE
    Message string `json:"message"`
}
```

7.2. *HTTP* статуси

202 *Accepted* (асинхроно слање)

400 *Bad Request*

403 *Forbidden* (канал забрањен)

429 *Too Many Requests*

500 *Provider Failure*

8. *USE CASE* МОДЕЛ

UC-1: Слање нотификације

(већ дефинисани токови)

UC-2: Конфигурација преференција

Корисник мијења дозволе по каналу.

UC-3: Преглед историје

Администратор прегледа уписе и исходе.

UC-4: Поновно слање

Администратор може окидати *resend* (слање наново) из *DLQ*.

UC-5: Масовне поруке

Систем емитује поруке (ограничено *rate limit*-ом).

9. СТРАТЕШКА ТЕСТИРАЊА

9.1. *Load Testing*

- 100 000 порука/мин
- 50 000 истовремених корисника

9.2. *Chaos Engineering*

- пад *SMS* провајдера
- деградација базе
- искусно спори *HTTP* одговори (*latency injection*)

ДОДАТАК В.1: *TRACEABILITY MATRIX*[90]

[90] Прилози су опциони и овај прилог је везан за примјер *SRS*-а, који је такође дат као прилог. Другим ријечима, имамо прилог у прилогу.

Захтјев	Use Case	Код/API	Тест сценарио
FR-1	UC-1	SendRequest.Type	Test-01
FR-3	UC-1	Retry Engine	Test-09
FR-6	UC-2	Preferences API	Test-14
NFR-7	UC-1/5	RateLimiter	Load-03

ДОДАТАК В.2: Проширен рјечник[91]

(додати по потреби)

[91] Исто као и у претходном случају, када је ријеч о прилогу унутар прилога.

ПРИЛОГ Г: ПАРАЛЕЛЕ ИЗМЕЂУ *Simple Sabotage Field Manual* (*OSS,* 1944) И ТЕКУЋЕГ СТАЊА У СОФТВЕРСКОМ ИНЖЕЊЕРСТВУ

Историјски документ *OSS*-а *Simple Sabotage Field Manual* (1944), који је објавила ЦИА у низу докумената са којих је скинута ознака тајности 2008. године, описивао је технике којима обични грађани могу, без ризика и без употребе специјалних средстава, нанијети максималну штету непријатељу. Парадоксално је то што многе препоруке из тог приручника данас готово дословно постоје у јавним и приватним организацијама, само што се јављају не као злонамјерна саботажа, већ неочекивано као својеврстан "стандардни начин рада".

Овај прилог представља једну систематску паралелу између историјских техника саботаже и честих дисфункција у развоју софтвера, с циљем да будући инжењери-програмери препознају управо ове старе ризичне обрасце и спријече да се укоријене у њиховим тимовима и на њиховим пројектима.

1. Саботажа кроз процес: када процедура постаје оружје

OSS препорука: „Инсистирај на строгој примјени свих правила и процедура. Тражи да се све уради формално, без изузетака."

Савремена паралела: Тимови који уводе бирократизоване процесе, са обавезним формуларима, обавезним корацима одобравања, и тиме репродукују суштински исту логику.

У софтверском инжењерству, можемо казати да су најчешћи облици ове саботаже:

- *Pull Request* који стоји данима јер „недостаје један чекбокс" или „још једно туђе мишљење";
- Процеси увођења промјена који трају дуже од саме промјене;
- *Sprint Review* (у Скраму) који се претвара у иритантну административну провјеру умјесто у обичну техничку ревизију.

Дејство ове "саботаже" је да она значајно успорава ток рада, доводи до демотивације тима, а резултат је, наравно, губитак итеративности.

2. Саботажа кроз састанке: производња фрустрације

OSS препорука: "Организуј што више састанака. Држи дискусију што дуже. Понављај већ донесене закључке."

Савремена паралела: Савремени тимови су често подложни дословно истој замци:

- *Daily Standup* од 15 минута постаје 45-минутна дискусија;
- *Sprint Planning* се претвара у философску расправу;
- *Product Owner* инсистира да се „ратосиљамо свих отворених питања", иако нису релевантна за текући инкремент.

Дејство ове саботаже се огледа у томе да непосредно еродира енергију тима, губи се фокус, а то посљедично доводи до пада продуктивности.

3. Саботажа кроз информациони шум

OSS препорука: „Преносите информације нетачно или непотпуно. Заборавите да обавијестите друге."

Савремена паралела: У развоју софтвера то личи на:

- *Slack* поруке без контекста;
- *Task* описе без јасних критеријума коначности (нпр. *Definition of Done*);
- *PR* описе који не наводе мотив промјене, него само списак датотека.

Дејство: повећан број дефеката, погрешне реализације, и дуплирани рад.

4. Саботажа кроз структурно одуговлачење

OSS препорука: „Увијек тражите да се оформи комисија. Понављајте процедуру од почетка ако неко направи грешку."

Савремена паралела:

- Увођење додатних нивоа техничких одобрења за тривијалне промјене;

- Притисак да се организује „архитектонски комитет” за сваки нови микросервис;
- Захтјев да *QA* поново покрене цјелокупно регресионо тестирање због измјене једне линије кода.

Дејство: успоравање развоја, повећање времена циклуса (енг. *cycle time*) и смањење иновација.

5. Саботажа кроз технички дуг као институцију

OSS препорука: „Радите ствари најкомпликованије могуће.”

Савремена паралела:

- Увођење претјерано сложених инжењерских рјешења (микросервиси без потребе, прекомјерна апстракција);[92]
- Зависности које се не одржавају, али се користе јер „тако је већ направљено”;
- Системи без аутоматизованог тестирања, који се све теже мијењају.

Технички дуг није одступање, већ је ту ријеч о акумулацији малих саботажа које временом постају системско оптерећење.

[92] Микросервиси се често препоручују због скалабилности. Но, ствар је у томе да ријетко ко помене да се скалира и број ствари које могу поћи наопако. То је математички симетрично, али маркетиншки не толико пожељно.

6. Саботажа кроз кадровску динамику

OSS препорука: „Унаприједите људе који нису компетентни. Држите способне у ћошку."

Савремена паралела:

- Постављање вође тима према критеријуму „ко је најдуже у фирми" умјесто способности;
- Пребацивање најискуснијег човјека на административне послове;
- Искључивање техничких стручњака из процеса доношења одлука.

Дејство: организациона ентропија, пад квалитета архитектуре и повећање ризика.

7. Саботажа кроз контекстно сљепило

OSS препорука: „Инсистирај на буквалном тумачењу правила, игнориши здрав разум."

Савремена паралела:

- агилне методологије које се спроводе буквално, без разумијевања;
- рецензирање кода (енг. *code review*) који се претвара у стилистичку полицију;

- безбједносне процедуре које ометају инжењере када требају да дијагностикују и отклоне производни дефект.

Дејство: губитак адаптивности, колапс итеративног развоја.

8. Саботажа кроз преувеличавање ризика

OSS препорука: „Увијек нагласите све могуће опасности. Сугеришите да је било каква промјена преурањена или сувише ризична."

Савремена паралела:

- *PM* који "осигурава" управљање ризицима тако што блокира иновације;
- *QA* који инсистира на савршеном регресионом тестирању умјесто ризиком вођеног приступа тестирању;
- Архитекта који тврди да је свака промјена опасност за "системску стабилност".

Дејство: застој, стагнација и блокада еволуције система.

Закључно: приручник за саботаже као негативно зрцало процеса

Паралела између *OSS/CIA* приручника и савременог софтверског инжењерства открива сљедеће обрасце:

- Најопасније саботаже су оне које дјелују као "нормалан рад";

- Дисфункције нису хаотичне, тј. оне имају препознатљиве обрасце;
- Кључни задатак софтверског инжењера је да направи разлику између неопходне процедуре и процедуре која има ефекте саботаже;
- Свако стање у тиму се може посматрати кроз три питања:
  - Да ли ово убрзава или успорава ток вриједности?
  - Да ли је технички оправдано или је ритуал?
  - Да ли велике одлуке доносе способни људи или их одређује инерција?

Ова аналогија треба да постане систематско оруђе за препознавање антиобразаца који утичу на квалитет, темпо и културу инжењерског рада.

CIP - Каталогизација у публикацији
Народна и универзитетска библиотека
Републике Српске, Бања Лука

004.41(075.8)(076)(0.034.2)

ЧВОКИЋ, Димитрије Д., 1984-
Увод у софтверско инжењерство [Elektronski izvor] : пројектовање, реализација, и развој софтвера : уз кодове на језику Гоу / Димитрије Д. Чвокић, Драган Корађ. - 1. онлајн изд. - Ел. књига. - Бања Лука : Универзитет у Бањој Луци : Природно-математички факултет, 2026

Системски захтјеви нису наведени. - Начин приступа (URL): https://blupress.unibl.org/pmf/catalog/book/171. - Насл. са насл. екрана. - Опис извора дана 19.08.2026. - Ел. публикација у ПДФ формату опсега 234 стр. - Напомене и библиографске референце уз текст.

ISBN 978-99997-45-43-7

COBISS.RS-ID 144696833