Универзитет у Новом Саду
Технички факултет "Михајло Пупин"
Зрењанин

Проф. др Жељко Стојанов

Животни циклус софтвера

Одабране теме

Едиција уџбеници
244
- 2020/2021 -

Универзитет у Новом Саду
Технички факултет "Михајло Пупин", Зрењанин

Проф. др Жељко Стојанов

Животни циклус софтвера

Одабране теме

Зрењанин
- 2021 -

Проф. др Жељко Стојанов
Животни циклус софтвера: Одабране теме

Рецензенти:
Проф. др Владимир Бртка, ванредни професор, Технички факултет "Михајло Пупин", Универзитет у Новом Саду
Проф. др Далибор Добриловић, ванредни професор, Технички факултет "Михајло Пупин", Универзитет у Новом Саду

Издавач:
Технички факултет "Михајло Пупин", Ђуре Ђаковића бб, 23000 Зрењанин

За издавача:
Проф. др Драгица Радосав, деканка Техничког факултета "Михајло Пупин"

Техничка припрема: Проф. др Жељко Стојанов

CIP - Каталогизација у публикацији
Библиотеке Матице српске, Нови Сад

004.4(075.8)

СТОЈАНОВ, Жељко, 1971-
Животни циклус софтвера [Електронски извор] : одабране теме / Жељко Стојанов. - Зрењанин : Технички факултет "Михајло Пупин", 2021. - 1 електронски оптички диск (CD-ROM) : текст, слика ; 12 cm. - (Библиотека Уџбеници ; 244)

Насл. са насловног екрана. - Библиографија.

ISBN 978-86-7672-347-8

а) Софтвер
COBISS.SR-ID 46581257

Одлуком Наставно-научног већа Техничког факултета "Михајло Пупин" у Зрењанину од 01.09.2021. године, одобрено је издавање и коришћење овог уџбеника као основног наставног средства.

Садржај

Листа слика

Листа табела

Листа коришћених скраћеница

IEEE	Institute of Electrical and Electronics Engineers	1
XP	eXtreme Programming	19
ASD	Adaptive Software Development	20
RAD	Rapid Application Development	20
SWEBOK	Guide to the Software Engineering Body of Knowledge	31
SRS	Software Requirements Specification	34
UML	Unified Modeling Language	81
CI	Configuration Item	92
QA	Quality Assurance	92
SPI	Software Process Improvement	105
ISO	International Organization for Standardization	111
IEC	International Electrotechnical Commission	111
SPICE	Software Process Improvement and Capability Determination	111
KPI	Key Performance Indicator	113
CMMI	Capability Maturity Model Integration	113
SCAMPI	Standard CMMI Appraisal Method for Process Improvement	113
VSE	Very Small Entity	114
PDCA	Plan-Do-Check-Act	119

Предговор

Књига представља основни наставни материјал за савладавање градива и припрему испита студентима који слушају предмет *Животни циклус софтвера* на студијском прогаму *Информационе технологије - Софтверско инжењерство* који се реализује на Техничком факултету "Михајло Пупин" у Зрењанину, Универзитета у Новом Саду. Поред тога, користи се и као наставни материјал за савладавање дела градива и припрему испита студентима који слушају предмет *Софтверска решења за финансије и менаџмент* на студијском прогаму *Информационе технологије*, модул *Менаџмент информационих технологија* који се реализује на Техничком факултету "Михајло Пупин" у Зрењанину, Универзитета у Новом Саду.

С обзирома да животни циклус софтвера обухвата све фазе у развоју, употреби и одржавању софтвера, циљ у избору тема за ову књигу је да се одаберу оне теме којима није посвећено довољно пажње у оквиру предмета који су углавном оријентисани на фазу развоја софтвера. Други разлог за избор тема које су обрађене у књизи је њихов значај и критичност за праксу у софтверској индустрији. Имајући то у виду, поред увода у основне концепте животног циклуса софтвера, одабране су теме инжењеринг софтверских захтева, еволуција и одржавање софтвера и софтверски процеси. Инжењеринг софтверских захтева је област која је најкритичнија за успех софтверских пројеката. Еволуција и одржавање софтвера се односе на управљање континуираним променама софтверских система како би остали употребљиви у домену примене, при чему највећи трошкови у животном циклусу софтвера се односе на одржавање софтвера. Моделовање, имплементација, побољшање и мерење софтверских процеса су фундаментални за ефикасност праксе у софтверској индустрији.

Прво поглавље представља основне концепте управљања животним циклусом софтвера, са фокусом на различите моделе. Друго поглавље представља увод у област инжењеринга софтверских захтева. Треће поглавље представља основне концепте еволуције софтверских система током животног циклуса, са посебним освртом на одржавање софтвера као најзахтевнију и најскупљу фазу у животном цикусу софтвера. Четврто поглавље представља основне концепте софтверских процеса, укључујући моделовање, процењивање и побољшање процеса, и мерење процеса. Детаљан списак коришћене литературе и међународних стандарда који се

односе на одабране теме животног циклуса софтвера су важан извор информација за њихово детаљније проучавање.

Упркос вишегодишњем раду у реализацији наставе, изради пројеката софтверских решења, пројеката процењивања и побољшања софтверских процеса у софтверској индустрији, као и припреми материјала за студенте који је послужио као основа за ову књигу, аутор је свестан да постоје пропусти и техничке грешке. Аутор ће бити захвалан свакоме ко укаже на уочене грешке и недостатке и тиме допринесе квалитету наредних верзија овог текста. Све сугестије које могу унапредити квалитет овог текста су такође добродошле.

Слава Богу, сада, увек, и у векове векова. Амин.

Зрењанин, 2021. лета Господњег.

Жељко Стојанов

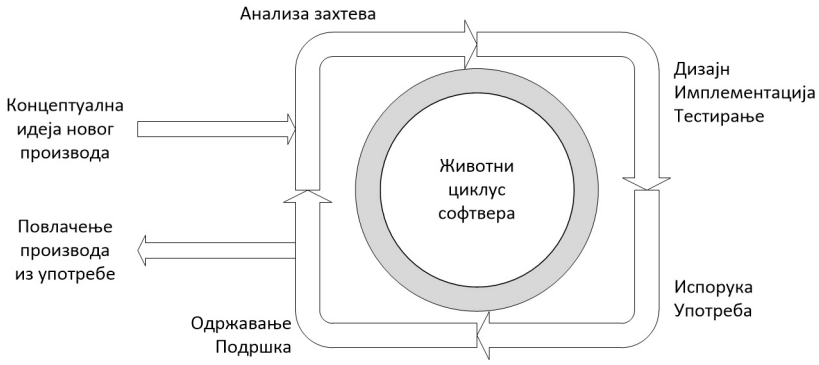
Поглавље 1

Модели и управљање животним циклусом софтвера

Софтвер је рачунарски програм са придруженом документацијом и подацима који се извршава на рачунару. Софтвер се данас користи у разним областима, као што су, на пример, пословни системи, електронско пословање, индустријски системи, образовање, медицина, пољопривреда, транспорт итд. Софтвер је данас најчешће врло сложен и развијају га тимови током дужег временског периода. С обзиром на сложеност софтвера и домена употребе, инжењери који развијају софтвер морају поседовати експертско знање из разних области, укључујући и домен примене. Према стандардном речнику терминологије објављеном од стране Institute of Electrical and Electronics Engineers (IEEE) (*IEEE std. 610.12-1990, IEEE standard glossary of software engineering terminology*), софтверско инжењерство је примена систематичног, дисциплинованог и мерљивог приступа у развоју, одржавању и коришћењу софтвера.

Према стандардном речнику терминологије у софтверском инжењерству, животни циклус софтвера почиње сагледавањем потребе за софтвером и завршава се када софтвер више није потребан и повлачи се из употребе. Типичне фазе у животном циклусу, приказане на слици 1.1 су: дефинисање концепта новог производа, анализа и израда софтверских захтева, дизајн софтвера, имплементација, тестирање, испорука (инсталирање и конфигурисање), употреба и одржавање, и повлачање из употребе. Поједине фазе се могу преклапати и извршавати итеративно, што зависи од конкретног окружења (софтверска организација, корисници, тржиште), али и од карактеристика самог софтверског производа.

Према стандарду *IEEE 12207-2008 - ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes* животни цикус представља еволуцију система, производа, услуга или пројекта од израде концепције па до повлачења из употребе.



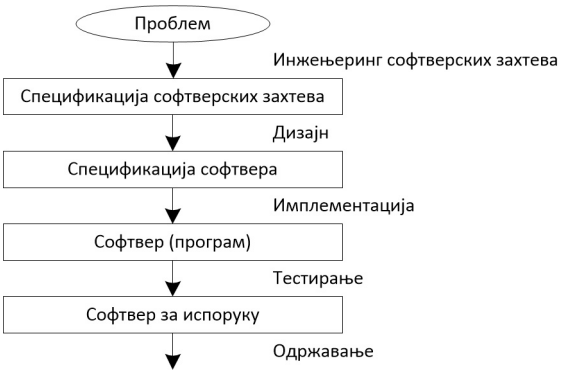
Слика 1.1: *Општи модел животног циклуса софтвера*

У литератури се појављују термини *Животни циклус развоја софтвера* и *Животни циклус софтверског производа*, па је потребно разјаснити ове појмове и релацију између њих.*Животни циклус развоја софтвера* се односи на процесе који се користе за спецификацију и трансформацију софтверских захтева у софтверски производ који се може испоручити. *Животни циклус софтверског производа* укључује *животни циклус развоја софтвера* али и додатне софтверске процесе који омогућују испоруку, одржавање, подршку, повлачење из употребе и еволуцију софтверских производа. Поред тога, *животни циклус софтверског производа* обухвата процесе управљања конфигурацијом софтвера и процесе обезбеђења квалитета који се примењују у свим фазама животног циклуса. Ако је еволуција софтверског производа сложена и софтвер је дуго у употреби, може се десити да *животни циклус софтверског производа* садржи више *животних циклуса развоја софтвера*. *Животни циклус софтверског производа* се обично скраћено назива *животни циклус софтвера*.

У пракси, софтверски процеси могу бити уређени на различите начине, што зависи од контекста где се примењују. То значи да различите софтверске организације могу на различите начине комбиновати процесе и фазе у животни циклус софтвера, при чему настају специфични модели *животног циклуса софтвера*. Ипак, истраживања указују да софтверски инжењери и софтверске фирме избегавају да користе систематичне приступе у индустријској пракси када су у питању модели животног циклуса, што често доводи до повећања трошкова развоја и одржавања софтвера .

У најопштијем случају, животни цикус софтвера обухвата фазе или стања које имају одговарајуће улазе, и након извршења дају излазе који могу бити улази у наредне фазе или самостални производи. На пример, спецификација софтверских захтева која настаје у фази инжењеринга софтверских захтева представља директан улаз за фазу дизајна, али се користи и у фази имплементације (конструкције) и у фази тестирања. Поједностављен модел животног циклуса са типичним фазама и производима који настају у животном циклусу је приказан на слици 1.2.

На слици 1.2 је представљен поједностављен животни циклус софтвера који садржи основне фазе и производе, али се он мора прилагодити свакој



Слика 1.2: *Поједностављени животни циклус софтвера са типичним фазама и производима*

специфичној ситуацији при чему се поједине фазе детаљно разрађују (на пример, детаљна разрада фазе дизајна може укључити различите типове дизајна и моделовања софтверког система). Поред тога, у пракси се ретко среће секвенцијални рапоред фаза у животном циклусу, већ доминира често враћање на претходне фазе због откривања грешака у развоју или употреби софтвера. Основне фазе које садржи поједностављени животни циклус софтвера су:

- **Инжењеринг захтева** (*Requirements engineering*). Основни циљ ове фазе је да се добије комплетан и детаљан опис проблема који треба да се реши развојем софтвера. У овој фази се креира и студија изводљивости која треба да сагледа техничке и економске аспекте развоја софтвера.
- **Дизајн** (*Design*). У овој фази се креира детаљни формални модел софтвера који омогућује имплементацију помоћу одабраних технологија, при чему се често користи декомпозиција на модуле који се потом интегришу у сложени систем.
- **Имплементација или конструкција** (*Implementation*). Током ове фазе се компоненте имплементирају помоћу одговарајућих технологија, а полази се од детаљних модела урађених у фази дизајна. Резултат ове фазе је извршиви програм.
- **Тестирање** (*Testing*). Наком инплементације се извршиви програм тестира да би се откриле потенцијалне грешке. У овој фази се врши верификација (провера да ли је резултат конструкције у складу са претходном спецификацијом или моделом) и валидација софтвера (провера да ли је софтвер конструисан у складу са спецификацијом корисничких захтева).
- **Одржавање** (*Maintenance*). Односи се на одржавање софтвера у оперативној употреби што захтева отклањање грешака, проширење или измену функционалности, или адаптирање на нове услове.

Поједини процеси се у животном циклусу могу извршавати истовремено. На пример, током дизајна софтвера се истовремено одвија и процес управљања конфигурацијом који има за циљ да обезбеди прецизно и стабилно стање свих

елемената у оквиру софтвера (које верзије софтверских елемената се укључују у текућу верзију софтера), као и процес управљања квалитетом. Процеси који се појављују у оквиру животног циклуса софтвера се могу поделити у следеће категорије:

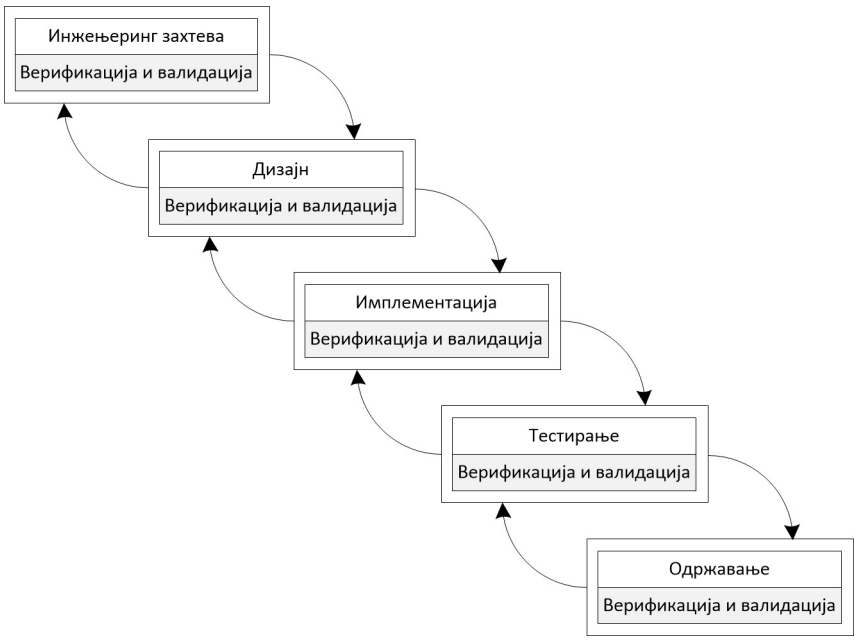
- **Примарни процеси** (*Primary processes*). То су процеси развоја, употребе и одржавања софтвера.
- **Процеси подршке** (*Supporting processes*).То су процеси који се појављују у свим фазама животног циклуса, а представљају подршку основним процесима. У ову категорију спадају процеси управљања конфигурацијом, обезбеђења квалитета, верификације и валидације.
- **Организациони процеси** (*Organizational processes*). То су помоћни процеси, а односе се на тренинг, управљање инфраструктуром, мерења, управљање поновном употребом и управљање моделима животног циклуса.
- **Међу пројектни процеси** (*Cross-project processes*). То су процеси који омогућују сарадњу учесика више пројеката, а обухватају управљање поновном употребом (код, компоненте, библиотеке), управљање производним линијама и управљање доменима.

1.1 Модели животног циклуса софтвера

Због нематеријалне природе и унутрашње сложености софтвера, у пракси се појављују различити приступи развоју софтвера, што је резултирало креирањем различитих модела животног циклуса. Модели животног циклуса варирају од веома једноставних линеарних модела, па до сложених модела који су итеративни и омогућују инкрементални развој софтвера. Према стандарду *IEEE 12207-2008 - ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes*, модел животног цикуса представља радни оквир за процесе и активности које се односе на животни циклус, а које могу бити организоване у стања или фазе, а истовремене служи као референтни модел за комуникацију и разумевање.

Избор модела животног циклуса зависи од природе пројекта или софтверског производа, а подразумева дефинисање сваке појединачне фазе, интеракција између фаза и производа који настају по завршетку фаза. Такође, потребно је одабрати језик или нотацију за моделовање процеса у животном циклусу и алате који подржавају моделовање, а са циљем да се повећа разумевање процеса и омогући њихово побољшање. Избор одговарајућег модела животног циклуса зависи од значајног броја фактора, као што су природа софтверских захтева, величина и састав тима за развој софтвера, величина пројекта, планирани буџет, доступност ресурса, интеракција са корисницима итд.

Два основна типа животног циклуса софтвера су линеарни и инкрементално-итеративни, а суштинска разлика између њих је у начину руковања софтверским захтевима. Код линеарних модела се током развоја



Слика 1.3: *Модел водопада*

креира комплетан скуп захтева према плану из иницијалне фазе пројекта, док се изменама рукује на основу захтева за променама након испоруке софтвера. Код инкрементално-итеретивних модела се софтвер развија као низ сукцесивних производа у којима се имплементирају захтеви.

У пракси постоје различити модели животног циклуса, као што су модел водопада, инкрементални модел, рапидни развој, агилни модели и хибридни модели. Анализом постојећих модел утврђено је да су они комплементарни, пре него компетитивни, па је предложена основна подела модела у три категорије: традиционални модели, агилни модели и хибридни модели (комбинација традиционалних и агилних модела).

1.1.1 Модел водопада

Модел водопада (*Waterfall Model*) је најједноставнији модел животног циклуса и најсличнији основном општем моделу приказаном на слици 1.1. Модел у основној варијанти без повратних веза је веома непрактичан и због тога је он само основа на којој су развијани остали модели. Основна карактеристика овог модела је да фазе следе секвенцијално једна за другом, а у следећу фазу се прелази тек када се комплетирају све активности у претходној фази. На слици 1.3 је приказан основни модел водопада који је проширен тако да омогућује проверу коректности приликом преласка на следећу фазу, као и повратак на претходне фазе у случају потребе. Применом модела водопада са верификацијом и валидацијом резултата сваке фазе обезбеђује се смањење времена и трошкова у софтверским пројектима.

Табела 1.1: *Удео појединих фаза у развоју софтвера код примене модела водопада*

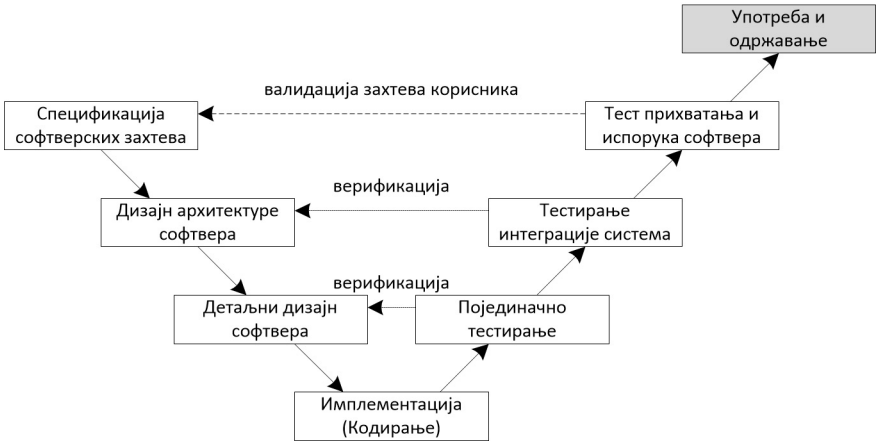
Фаза	**Удео [%]**
Спецификација захтева	41
Дизајн и имплементација	17
Верификација	19
Испорука	23
УКУПНО	100

Провера коректности у свакој фази се врши поступцима верификације и валидације. Верификација (*verification*) је тестирање испуњености услова приликом преласка у следећу фазу, тј. да ли су сви захтеви и ограничења постављени за дату фазу испуњени. Валидација (*validation*) је провера да ли су резултати остварени у некој фази у складу са постављеним захтевима корисника приликом дефинисања пројекта.

Да би се обезбедило што боље испуњење корисничких захтева, велика пажња се посвећује анализи софтверских захтева и дизајна пре него што се софтвер конструише. На основу истраживања великих индустријских софтверских пројеката идентификован је просечни удео појединих фаза у развоју софтвера, што је приказано у табели 1.1. Овакви подаци потврђују ставове из литературе да се највише времена у овом моделу животног циклуса троши у фази анализе и спецификације софтверских захтева. Успешност пројеката базираних на овом моделу у великој мери зависи од стабилности, обима и познавања свих софтверских захтева пре него што се пређе на фазе дизајна и имплементације.

Основни проблем са применом овог модела је велики број дефеката који се појављују након испоруке софтвера, а детаљни списак идентификованих проблема је:

- Велики напор и трошкови за писање и одобравање документације за све фазе развоја софтвера.
- Веома компликовано руковање променама софтвера.
- Понављања одређене фазе подразумева значајан додатни рад и понављање послова.
- Када корисник открије проблеме приликом употребе софтвера који се односе на ране фазе развоја, тада софтвер не одражава стварне захтева корисника.
- Проблеми који се односе на завршену фазу се преносе у следећу фазу да би били решени.
- Управљање великим бројем софтверски захтева који представљају базу за даљи развој софтвера.
- Велика интеграција (Big-bang integration) и тест целог система на крају пројекта развоја може довести до неочекиваних проблема са квалитетом софтвера, високих трошкова и прекорачења рокова.
- Недостатак могућности да корисници дају повратне информације у току развоја софтвера.



Слика 1.4: *V модел животног циклуса софтвера*

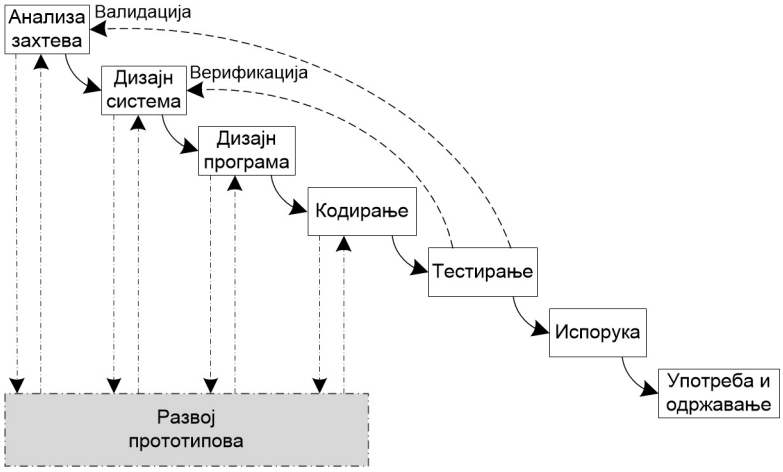
- Значајно повећање времена да се одобре сви артифакти настали у различитим фазама развоја, што омогућује прелазак у наредну фазу.

Основни недостатак овог модела је да не постоје повратне информације (*feedback*) о реализованим активностима, као и одсуство итеративног извршавања појединих фаза. У пракси се програмери врло често враћају на претходне фазе након сагледавања проблема који се појављују током реализације. То је условило развој модела који представљају модификацију основног модела водопада, а укључују повратне информације и итерацију у животни циклус.

V модификација модела водопада

Овај модел је варијација модела водопада у којем се врши декомпозиција развоја софтвера до детаља тако да се свакој фази у развоју придружује одговарајућа фаза у тестирању софтвера пре испоруке. Типичан *V модел* је приказан на слици 1.4. Овај модел приказује везу између активности тестирања софтвера и активности анализе и дизајна. Овај модел указује да проблеми који се уоче током верификације и валидације омогућују да се поједине фазе у животном циклусу понове. Такав приступ омогућује идентификацију и отклањање уочених проблема током развоја софтвера, а пре него што се изврши испорука софтверског система.

Верификација креираног софтвера се врши тестирањем појединачних модула на основу чега се могу вршити одговарајуће измене детаљног дизајна софтвера, или тестирањем интеграције система на основу чега се може вршити модификација архитектуре целог система. Тест прихватања софтвера омогућује валидацију да ли софтвер задовољава корисничке захтеве, и он је предуслов за испоруку софтвера. Ако тест прихватања укаже на проблеме у софтверу, тада се поново врши анализа и спецификација одговарајућих софтверских захтева.



Слика 1.5: *Модел водопада са прототиповима*

Модел водопада базиран на прототиповима

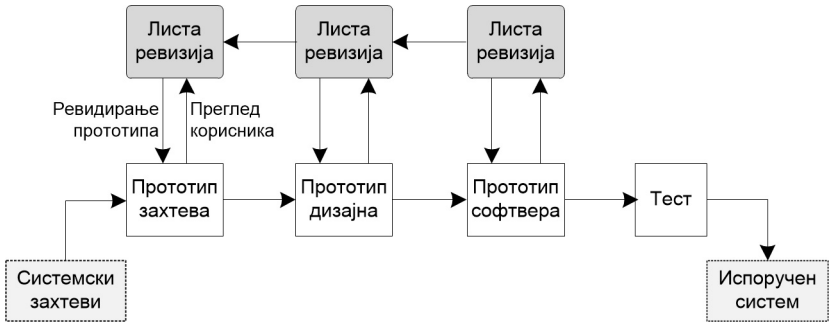
Модел водопада базиран на прототиповима је приказан на слици 1.5, а укључује израду прототипова током свих фаза развоја софтвера. Прототипови омогућују увид у процес развоја софтвера у свим фазама и користе се за добијање повратних информација, чиме се значајно унапређује основни модел водопада.

Овај модел обухвата повратне информације у животном циклусу. Повратне информације обезбеђују верификацију дизајна систем, и валидацију да ли су задовољени постављени кориснички захтеви током фазе анализе захтева. Верификација обезбеђује да све имплементиране функционалности раде коректно. Валидација обезбеђује да су у систему имплементирани сви специфицирани захтеви које су навели корисници система, што омогућује да се може од сваке имплементиране функционалности доћи до скупа захтева које она имплементира.

1.1.2 Модел базиран на прототиповима

Прототипови се могу укључити у модел водопада, као што је приказано на слици 1.5, али могу бити и добра основа за формирање ефикасног модела животног циклуса. Прототипови омогућују брзо конструисање прототипских модела појединих делова система, на основу којих се потом остварује усаглашеност разумевања корисника и пројектаната софтвера. Употреба прототипова уводи вишеструке итерације у току развоја софтвера. Примена овог модела обезбеђује да сви учесници у процесу буду задовољни постигнутим резултатима. Модел базиран на прототиповима је приказан на слици 1.6.

Вишетруке итерације су подржане креирањем *листа за ревизију* које се потом укључују у процес израде нових прототипова све док се не дође до задовољавајућег решења. Најпре се креира прототип софтверских захтева који



Слика 1.6: *Модел животног циклуса софтвера базиран на прототиповима*

се разматра са корисницима, и када се постигне разумевање и усаглашеност по питању захтева, прелази се на израду прототипова дизајна. Прототип дизајна се такође анализира са корисницима па се листа ревизија враћа на дораду прототипа дизајна, или је потребно ревидирати и прототип захтева. Када се постигне разумевање и усаглашеност по питању прототипа дизајна, креира се прототип софтвера који пролази исте поступке анализе као и претходни прототипови.

Овакав модел развоја софтвера омогућује потребан број итерација кроз израду прототипова спецификације захтева, прототипова дизајна и прототипова софтверског система све док се не дође до система који се може испоручити корисницима. Израда прототипова омогућује креирање више алтернатива за софтверско решење и избор оне која задовољава потребе корисника.

У пракси се примењују два типа развоја базирана на прототиповима:

- **Приступ са одбацивањем прототипова** (*throwaway prototyping*). Током развоја софтвера се креира више прототипова, али се при преласку на нови прототип претходни одбацује. У следећој итерације се тада креира нови прототип на основу искустава са претходним.
- **Приступ са еволуцијом прототипова** (*evolutionary prototyping*). Сваки нови прототип је настао на бази измена претходног прототипа, па се тако јавља унапређење прототипова на основу ревизије у коју су укључени корисници софтвера. Претходни прототип се не одбацује већ служи као основа за развој новог прототипа, па се након одређеног броја итерација долази до верзије која задовољава захтеве корисника и она се испоручује за употребу.

У пракси се еволутивни приступ прототиском развоју много чешће користи него приступ са одбацивањем прототипова пошто се остварује много веће искоришћење онога што је већ урађено у току развоја софтвера - почевши од спецификације захтева, преко дизајна па до кодирања.

Предности развоја софтвера на бази прототипова су:

- Испоручени систем је једноставнији за употребу пошто се кроз прототипове долази до решења које највише одговара корисницима.

- Потребе и захтеви корисника су боље сагледани и имплементирани кроз итеративни поступак развоја прототипова. Сваки следећи прототип боље испуњава захтеве корисника.
- Боља и бржа идентификација проблема. Проблеми се врло брзо детектују кроз прототипска решења, па се у наредним итерацијама проблеми елиминишу.
- Високи квалитет дизајна се остварује кроз вишеструко итеративно прочишћавање дизајна у складу са захтевима корисника.
- Креирани систем је једноставан за одржавање, што се постиже итеративним побољшањима прототипова током развоја.
- Развој софтвера захтева мање напора, нарочито приликом еволутивног приступа где се јасно уочавају функционалности и карактеристике које треба сачувати у финалном производу али и оне које треба мењати.

Недостаци развоја софтвера на бази прототипова су:

- Систем који се испоручује обично има више могућности (функционалности) које најчешће нису све потребне корисницима и могу закомпликовати употребу софтвера.
- Овај приступ захтева софтверске инжењере са већим искуством и комплексним скупом знања и вештина које омогућују инкрементални развој кроз еволуцију прототипова.

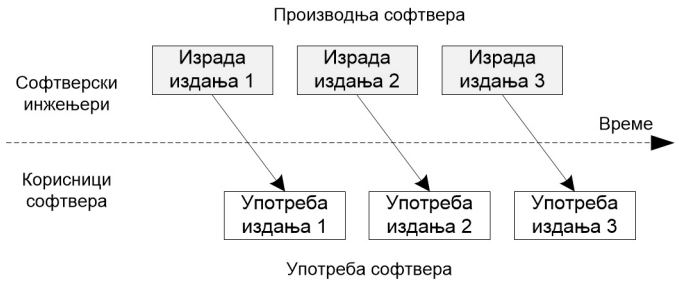
На основу наведених крактеристика, предности и недостатака, могу се дефинисати препоруке за примену модела развоја базираног на прототиповима:

- Ситуације када нису јасни захтеви корисника, па се прототипови користе за њихово прочишћавање. Еволутивним приступом се може доћи до јасно дефинисаног и стабилног скупа захтева.
- За системе са сложеним корисничким интерфејсом и комплексним скупом интеракција корисника са системом, где се прототипови користе за идентификовање одговарајућег начина употребе система од стране корисника.

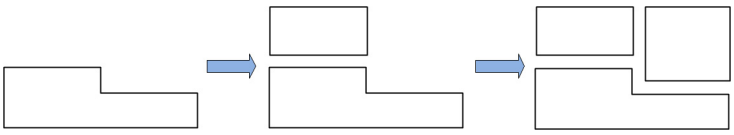
Приликом употребе модела развоја базираног на прототиповима, софтверски инжењери морају бити свесни предности и недостатака овог модела, морају бити спремни на честе измене система да би се систем ускладио измењеним захтевима корисника, и морају контролисати процес развоја кроз јасно дефинисан број могућих итерација и начине комуникације са корисницима.

1.1.3 Фазни модел: Инкременти и итерације

Применом модела циклуса базираних на моделу водопада кашњење са испоруком софтвера је велико, што је у условима савременог пословања које је подложно сталним променама неприхватљиво. Модел фазног развоја је настао са циљем да реши проблем кашњења са испоруком софтверског система корисницима. Фазни модел подразумева да се софтвер развија у



Слика 1.7: *Модел фазног развоја софтвера*



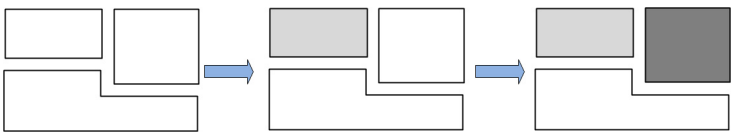
Слика 1.8: *Инкрементални модел развоја софтвера*

деловима који се парцијално испоручују. На тај начин корисници у свом окружењу користе једно издање софтвера, док се у развојном окружењу креира ново издање, као што је приказано на слици 1.7. Издања (верзије) софтвера се означавају нумеричким вредностим, или се уводи посебан начин означавања од стране произвођача софтвера.

Организација развоја софтвера са више издања се може реализовати на више начина, а два најпопуларнија су *инкрементални развој* и *итеративни развој*. Оба приступа подразумевају да се систем у складу са спецификацијом захтева подели у подсистеме према функционалностима које реализују, а коначна верзија система садржи све подсистеме са свим захтеваним функционалностима.

Код инкременталног развоја се сваки подсистем независно развија. Прво се испоручује један подсистем са минималним скупом функционалности које корисник може користити у свом окружењу. Док корисник користи испоручени подсистем, нови подсистеми се развијају и један по један се испоручују кориснику. Инкрементални модел развоја софтвера је приказан на слици 1.8.

Код итеративног развоја се цео систем испоручује као целина са минималним скупом функционалности, а потом се у итерацијама сваки подсистем мења да задовољи потпуни скуп захтева. Итеративни модел развоја софтвера је приказан на слици 1.9.



Слика 1.9: *Итеративни модел развоја софтвера*

У пракси већина софтверских фирми користи комбинацију итеративног и инкременталног развоја. То значи да се у свакој итерацији додају подсистеми или модули са новим скупом функционалности, али се и већ испоручени подсистеми унапређују и проширују додатним функционалностима. Предности употребе фазног развоја и испоруке софтвера су:

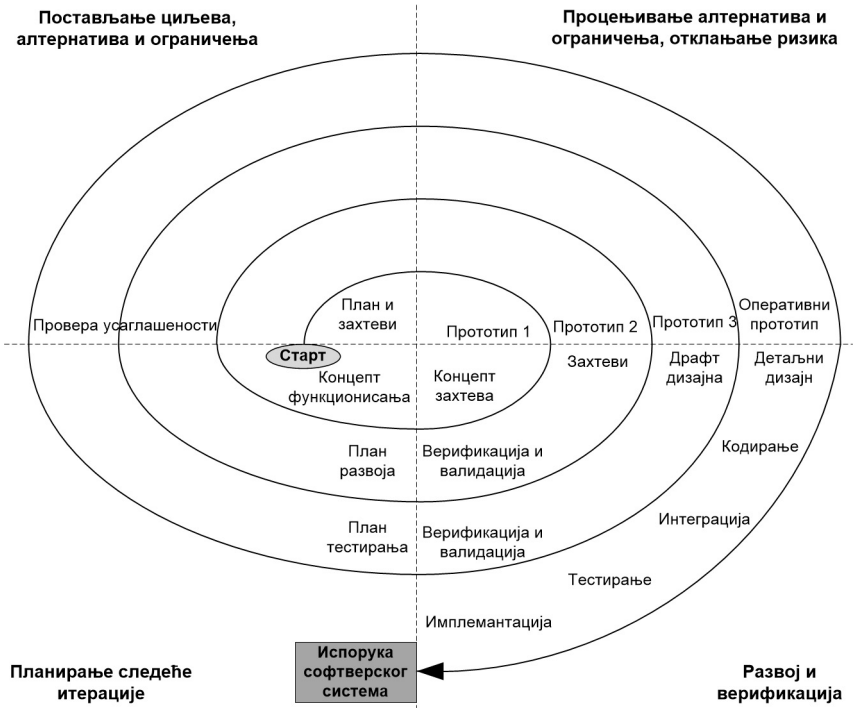
- Раније почиње обука корисника за употребу софтвера, што омогућује да се раније уочи начин функционисања реализованих функционалности. На овај начин се софтвер може брже и ефикасније унапредити, а истовремено се значајно побољшава одзив софтверских инжењера на захтеве и потребе корисника
- Може се раније кренути са маркетингом софтвера, што је нарочито битно за иновативне софтверске производе.
- Често издавање софтвера омогућује да се успешније уоче и отклоне проблеми и грешке (*fixing bugs*) који нису сагледани у претходним фазама развоја.
- Тим за развој софтвера се у свакој фази развоја може посветити одређеном типу подсистема, па се према томе може извршити распоређивање инжењера у складу са њиховом експертизом. На тај начин се обезбеђује ефикасније управљање у тимском раду.

1.1.4 Спирални модел

Спирални модел је уведен са циљем да се смањи и контролише ризик током развоја софтвера. Животни циклус почиње спецификацијом захтева и полазним планом за развој (буџет, ограничења, алтернативе, тим, развојна окружења), и потом укључује активности за процену ризика током развоја. На крају сваког циклуса се креира прототип на одређеном нивоу апстракције и он представља улаз за следећу фазу у циклусу. Након прве итерације се добија документ са анализом шта треба да садржи производ, након друге итерације се добија документ са спецификацијом захтева, након треће итерације се добија дизајн система, а након четврте итерације се добија производ који је могуће тестирати. У свакој итерацији се врши анализа ризика различитих алтернатива у односу на захтеве, ограничења и реализоване прототипове. Спирални модел животног циклуса је приказан на слици 1.10.

Итерације (фазе) које се односе на софтверске захтеве и дизајн система имају много тога заједниочког па се врло често преклапају. То значи да се приликом прикупљања, анализе и спецификације захтева врши креирање првих прототипова дизајна који се приказују корисницима софтвера. У ту сврху се најчешће организују састанци на којима учествују софтверски инжењери и корисници софтвера који су извор захтева.

Спирални модел може садржати више итерација, односно спирала, за сваку фазу развоја. По потреби се за прочишћавање скупа захтева може обићи више спирала док се не дође до стабилног скупа захтева који се могу искористити за дизајн система. Такође је могуће дизајн система



Слика 1.10: *Спирални модел животног циклуса софтвера*

унапређивати у више спирала све док се не дође до прототипа на основу којег се прелази у фазе кодирања, тестирања и испоруке софтвера.

Спирални модел се може применити и у фази одржавања софтвера. Процес одржавања се најчешће иницира захтевима корисника (корекција грешке, проширење функционалности, адаптација) који служе за покретање спиралног процеса који од захтева води ка дизајну промене и испоруци нове верзије софтвера која решава захтев корисника.

Основна предност спиралног модела је његова адаптабилност различитим ситуацијама и окружењима развоја софтвера, што значи да се број спирала појединих фаза може прилагодити конкретном случају. Поред тога модел подржава еволуцију софтвера и кроз развој и касније кроз промене током одржавања софтвера, при чему се сви захтеви за променама своде на циљеве које треба остварити и за које се рачунају алтернативе и ограничења.

1.1.5 Rational Unified Process

Rational Unified Process (RUP) је процес развоја софтвера који је развијен у компанији *Rational Software*. RUP је фокусиран на продуктивност развојног тима чији чланови имају приступ заједничкој бази знања без обзира у којој фази развоја су укључени. Поред тога сви чланови тима користе исти језик и процес у раду, чиме се унифицирају активности и комуникација у тиму.

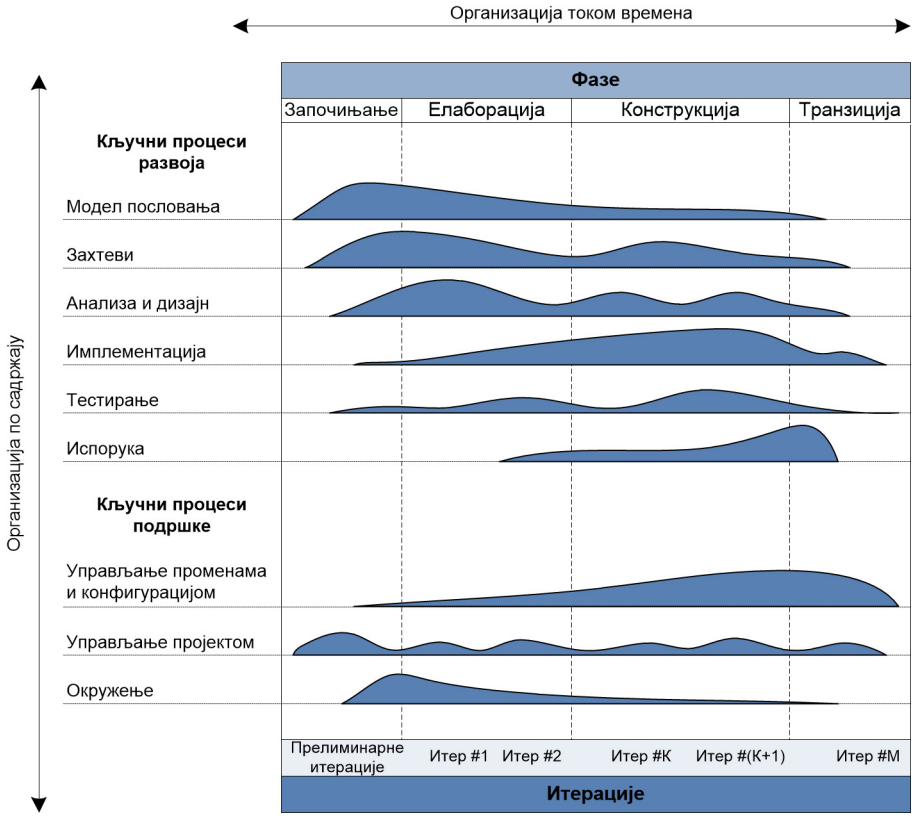
RUP је интерактиван и инкременталан приступ који представља подршку објектно-оријентисаном развоју софтвера базираном на *Unified Modelling Language (UML)* који је индустријски стандард у развоју објектно-оријентисаног софтвера, а омогућује ефикасну комуникацију у фазама спецификације захтева, дизајна архитектуре и детаљног дизајна.

RUP се може прилагодити потребама различитих тимова и софтверских организација тиме што обезбеђује смернице (*guidelines*), обрасце и алате за реализацију активности у процесу развоја. Процес је базиран на следећим принципима добре праксе:

- **Итеративни развој софтвера** (*Develop software iteratively*) омогућује постепено разумевање проблема кроз сукцесивно прочишћавање софтверских захтева и инкрементално развијање софтверског решења кроз више итерација. На крају сваке итерације се испоручује извршива верзија софтвера, чиме се обезбеђује стална провера статуса пројекта и смањење ризика, фокусираност на резултате и поштовање временских оквира.
- **Управљање захтевим** (*Manage requirements*) се започиње активностима прикупљања захтева а простире се на све фазе у развоју. Захтеви су добро документовани помоћу случајева употребе (*use cases*), на основу којих се управља дизајном, конструкцијом и тестирањем софтвера.
- **Употреба архитектура базираних на компонентама** (*Use component-based architectures*) представља основу за пројектовање робустних архитектура које се флексибилне, лако се прилагођавају променама и обезбеђују поновну употребу софтвера (*software reuse*).
- **Визуелно моделовање софтвера** (*Visually model software*) омогућује сагледавање и разумевање структуре и понашања компоненти и софтверског система. Користе се графички језици за визуелно представљање различитих аспеката софтвера. Најчешће се користи језик UML.
- **Верификација квалитета софтвера** (*Verify software quality*) је уграђена у сам процес и укључује све учеснике у објективно мерење перформанси софтвера. Процес подржава планирање, дизајн, имплементацију и извршавање различитих типова тестова.
- **Контрола промена софтвера** (*Control changes to software*) је пресудно важна пошто су промене неизбежне током развоја, а њихово праћење је есенцијално за успешан итеративни развој.

RUP модел наглашава развој софтвера у фазама, при чему се свака фаза може састојати од више итерација. Модел има две димензије које се могу приказати дуж две осе (слика 1.11):

- **Хоризонтална оса** представља временску димензију и приказује динамику извршавања процеса у смислу циклуса, фаза, итерација и кључних прекретница (*milestones*).
- **Вертикална оса** представља статички аспект процеса и опис у смислу активности, артефакта, радника и радних токова.



Слика 1.11: *Организација RUP модела животног циклуса софтвера у две димензије*

Временска димензија RUP модела је представљена са 4 фазе кроз које пролази пројекат: започињање, елаборација, конструкција и транзиција. Свака фаза има јасно дефинисане временске кључне прекретнице (*milestones*) у којима се доносе критичне одлуке и проверава остваривање зацртаних циљева. Основне активности које се обављају у фазама животног циклуса су:

- **Започињање** (*Inception*). У овој фази се започиње пројекат анализом пословног случаја који се моделује. На основу модела случаја се дефинишу захтеви и врши анализа и дизајн система. Започиње са имплементацијом и тестирањем али само да би се верификовали и валидирали захтеви.
- **Елаборација** (*Elaboration*). У овој фази је најзначајнија активност дизајн архитектуре система и детаљни дизајн појединачних компоненти. Имплементација и тестирање се односе на проверу дизајна кроз развој прототипова компоненти.
- **Конструкција** (*Construction*). У овој фази се врши имплементација свих захтеваних функционалности у складу са дизајном архитектуре система и појединачних компоненти. Тестирање компоненти и интеграције система се такође одвијају у овој фази.

- **Транзиција** (*Transition*). У овој фази је основна активност испорука софтвера, што може захтевати додатне активности на имплементацији и тестирању.

Током целог животног циклуса софтвера реализују се и помоћни процеси који су подршка у свим фазама. Ти процеси обезбеђују управљање конфигурацијом и променама софтвера, управљање пројектом и управљање радним окружењем у којем се софтвер развија.

Управљање пројектом (*Software Project Management*) обезбеђује балансирано управљање циљевима, управљање ризицима и управљање ограничењима са циљем да се обезбеди успешна испорука софтвера у складу са захтевима корисника.

Управљање конфигурацијом и променама софтвера (*Configuration & Change Management*) обезбеђује технике, алате и процедуре за управљање разним артефактима који настају у процесу развја софтвера, а које могу произвести различити чланови тима. На овај начин се могу предупредити или решити проблеми постојања више верзија неког артефакта, симултано ажурирање од стране више чланова тима, обавештавање чланова тима о променама артефаката итд. Методе управљања конфигурацијом и променама софтвера су кључне за контролисану еволуцију софтвера.

Управљање радним окружењем (*Environment Management*) обезбеђује софтверској организацији управљање радним окружењем које је подршка развоју софтвера, што подразумева управљање процесима, алатима, развојним окружењима, документацијом, итд.

1.2 Агилне методе

Најзначајнији тренд и најкритичнији захтев у софтверској индустрији крајем 20. и почетком 21. века је рапидно брз развој и испорука софтвера, што је последица динамике тржишта и пословања. Овакав приступ често подразумева компромис по питању испуњења свих захтева клијената и квалитета софтверских производа. Због сталних промена у пословању тешко је одржати стабилан скуп софтверских захтева, па се често јавља потреба за изменама софтвера. У таквим условима пословања, где се захтеви стално мењају примена традиционалних модела развоја софтвера базираних на детаљној и комплетној спецификацији постаје неприхватљива пошто се код њих софтвер испоручује са значајним кашњењем и често не одговара новој ситуацији у окружењу где се примењује. Примена традиционалних метода развоја базираних на детљаној спецификација је и даље значајна код управљачких система који су критични са аспекта сигурности пошто је код таквих система неопходно детаљно специфицирати све аспекте система. Код савремених пословних система је примена традиционалних метода у развоју софтвера неприхватљива и углавном се потенцира примена агилних метода брзог развоја софтвера.

Разумевање корена и историјског развоја, као и чињенице да су основне идеје агилних метода настале у оквиру традиционалног софтверског инжењерства, свакако доприноси бољем разумевању и прихватању принципа

агилних метода у пракси. Узроци појаве агилних метода су: реакција на традиционалне методе и промене у пословању, поновна употреба идеја које су се већ показале као корисне у пракси и ослањање на искуство људи из праксе.

Агилне методе се ослањају на знања, вештине и искуство људи укључених у пројекат развоја софтвера, па се због тога процеси усклађују према специфичностима доступних људи. Агилни пројекти, по природи веома променљиви, наглашавају стручност људи и тимова који су укључени у пројекат развоја софтвера, при чему се агилне организације потпуно ослањају на компетенције и сарадњу људи и тимова.

Појава агилних метода у развоју софтвера у последње две деценије је омогућила лаганији (*lightweight*) али истовремено сигуран и ефикасан приступ управљању софтверским захтевима. Агилне методе омогућују правовремено и интерактивно управљање софтверским захтевима, а истовремено обезбеђују да се раније почне са враћањем уложених средстава у развој софтвера у односу на традиционалне моделе. Практично, код традиционалних модела базираних на моделу водопада повраћај уложених средстава је након завршетка развоја и испоруке целог софтвера, док се код агилних метода повраћај средстава јавља већ након испоруке првих функционалности које имплементира софтвер, док се остале функционалности реализују и испоручују инкрементално. Основне карактеристике агилних метода су:

- Процеси спецификације, дизајна и имеплементације су испреплетени. Документ са спецификацијом захтева садржи само основне информације о систему. Документи са спецификацијом и дизајном су минимални и најчешће аутоматски генерисани у развојном окружењу.
- Систем се развија као низ верзија, при чему сваку верзију оцењују корисници и сви заинтересовани учесници пројекта. Приликом сваког оцењивања се могу изменити постојећи захтеви или додати нови.
- Кориснички интерфејс се обично брзо генерише помоћу већ припремљених компоненти које се брзо интегришу. Значајну улогу у развоју интерфејса имају прототипови.

Основне вредности агилног приступа развоју се огледају у тимском раду где је значајна улога људског фактора, па су према томе и утврђене кључне вредности агилних метода које чине **Агилни Манифест** (*Agile Manifesto*, https://agilemanifesto.org/):

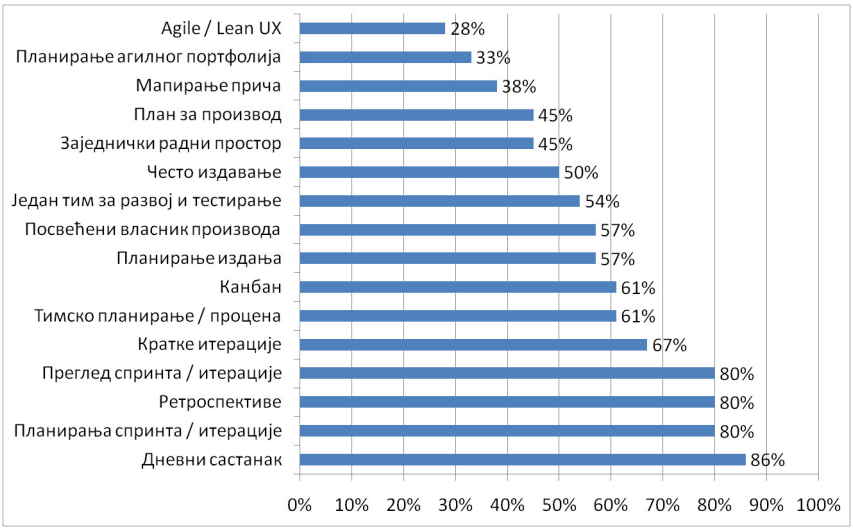
- **Појединци и интеракције испред процеса и алата**. То значи да програмери добијају све потребне ресурсе на почетку циклуса, сами планирају колико ће урадити и у њихову процену се има поверење. Тимови се организују тако да имају међусобну комуникацију уживо, а не помоћу документације.
- **Употребљив софтвер испред обимне документације**. Пажња се посвећује развоју употребљивог софтвера а не креирању документације. Основна мера успеха пројекта је колико добро ради софтвер и колико задовољава захтеве корисника.

- **Сарадња са клијентима испред уговорених аранжмана**. Фокус у развоју је на сарадњи са корисницима у свим корацима, а не на преговарању око детаља уговора.
- **Реакција на промене испред придржавања плана**. Основна претпоставка је да се не могу сви захтеви сагледати на почетку, па су промене неизбежне током развоја. Због тога није могуће направити стриктан план на почетку и придржавати га се, већ се развој прилагођава текућем стању и променама захтева које се јављају.

Значајну улогу у прихватању агилних метода имају принципи на којима се заснива **Агилни Манифест** (*Agile Manifesto*, https://agilemanifesto.org/):

- Задовољан клијент је наш врхунски приоритет, који остварујемо благовременом и континуираном испоруком врхунског софтвера.
- Спремно прихватамо промене захтева, чак и у касној фази развоја. Агилни процеси омогућавају успешно прилагођавање измењеним захтевима што за резултат има предност наших клијената у односу на конкуренцију.
- Редовно испоручујемо применљив софтвер, у периоду од неколико недеља до неколико месеци, дајући предност краћим интервалима.
- Пословни људи и девелопери свакодневно сарађују у току целокупног трајања пројекта.
- Пројекте остварујемо уз помоћ мотивисаних појединаца. Обезбеђујемо им амбијент и подршку која им је потребна и препуштамо им посао с поверењем.
- За најпродуктивнији и најефикаснији метод преноса информације до и унутар развојног тима сматрамо контакт лицем у лице.
- Применљив софтвер је основно мерило напретка.
- Агилни процеси промовишу одрживи развој. Покровитељи, девелопери и корисници морају бити у стању да континуирано раде усклађеним темпом, независно од периода трајања пројекта.
- Стална посвећеност врхунском техничком квалитету и добар дизајн поспешују агилност.
- Једноставност – вештина довођења до највишег степена количине рада који није потребно урадити – је од суштинске важности.
- Најбоље архитектуре, захтеви и дизајн, резултат су рада само–организованих тимова.
- Тимови у редовним интервалима разматрају начине како да постану ефикаснији, затим се усклађују и на основу тих закључака прилагођавају даље поступке.

Агилне методе укључују кориснике софтвера у сваки корак развоја. Циклуси развоја су мали и инкрементални, што значи да се софтвер развија у малим инкрементима који додају део по део функционалности (погледати фазни модел развоја). Серија циклуса развоја се не планира у потпуности,



Слика 1.12: *Најчешће коришћене агилне технике у пракси*

већ се након сваког циклуса сагледава ситуација и врши планирање. На крају сваког циклуса се добија систем који може да се користи и има одређену вредност за кориснике. Преглед најчешће коришћених и најкориснијих техника у агилним методологијама, према истраживању компаније *CollabNet VersionOne* у 2018. години, је приказан на слици 1.12.

У пракси се користе разне агилне методе, а све су базиране на Агилном манифесту који представља оквир за агилни развој софтвера. Најчешће коришћене методе су:

- **eXtreme Programming (XP)**. Метод екстремног прогамирања наглашава основне принципе агилног развоја: комуникација, једноставност, смелост и повратне информације. Посебно су важне повратне информације које се размењују између програмера који раде заједно да би остварили најбољи дизајн софтвера, али и повратне информације од корисника. Кључни елемент је програмирање у пару, које даје добре резултате са аспекта квалитета кода, и ефикасног отклањања грешака.
- **Scrum**. Базира се на итеративном развоју у којем се свака итерација назива *sprint*. Сваки *sprint* има исто трајање (нпр. 7 дана или 30 дана). У оквиру сваког *sprint* се решавају захтеви који су евидентирани и приоритизовани у бази захтева за софтверски пројекат (*backlog*). Може радити више тимова паралелно на решавању захтева, а кооридинација и сагледавање стања се врши на кратким дневним састанцима (*daily scrum*).
- **Crystal** је скуп метода које полазе од претпоставке да сваки софтверски пројекат захтева различити скуп метода и правила за успешну реализацију. Базира се на учесталој комуникацији и честој испоруци производа. Метода је настал вишегодишњим истраживањем индустријске праксе које је спровео *Alistair Cockburn*. Метода је

Табела 1.2: *Удео појединих агилних методологија у индустрији*

Методологија	**Удео [%]**
Scrum	54
Other hybrid / multiple	14
Scrum / XP hybrid	10
Scrumban	8
Kanban	5
Iterative development	3
Do not know	3
Lean startup	2
Extreme programming (XP)	1

фокусирана на људе, њихове вештине и таленте, као и на међусобну интеракцију и комуникацију. Постоји више метода у *Crystal* фамилији метода, а означене су различитим бојама у зависности од сложености пројекта и броја људи који раде у тиму. На пример, *Crystal Clear* се користи за тимове до 6 људи, а *Crystal Yellow* за тимове до 20 људи.

- **Adaptive Software Development (ASD)**. Метод је настао током примене принципа рапидног развоја софтвера (*Rapid Application Development (RAD)*), а основна идеја је да је сасвим природно да се процес развоја стално мења и адаптира околностима. За пројекат постоји циљ, али не постоји правило како остварити циљ. Пројекат се организје око функционалних елемената који носе вредност за корисника софтвера, а који се развијају и испоручују инкрементално. Фиксни интервали испоруке подржавају да се увек изаберу есенцијалне функционалности за верзију која се испоручује.

Истраживање агилне праксе у индустрији из 2018. године указује да је *Scrum* и даље најчешће коришћена методологија, где најмање 72% испитаника користи *Scrum* или неки хибрид који укључује *Scrum*. Удео у индустријској пракси појединих методологија је приказан у табели 1.2.

1.2.1 Предности и недостаци агилних методологија

Као и све друге методологије које се користе у производњи софтвера, и агилне методе имају своје предности и недостатке које утичу на њихово прихватање и успешност примене у пракси. У табели 1.3 су приказане предности агилних методологија, а у табели 1.4 су приказани недостаци агилних методологија.

Истраживање спроведено на глобалном нивоу, а спонзорисано од стране компаније *CollabNet VersionOne* у 2018. години, указује да су најзначајнији разлози за прихватање агилних метода разлози који се односе на ефикасност развоја софтвера, као што су убрзавање испоруке, ефикасније управљање приоритетима и променама, повећање продуктивности и усклађеност са пословањем. На слици 1.13 су приказани сви разлози и њихов значај према оцени учесника истраживања.

У истом истраживању из 2018. године компаније су навеле параметре агилних пројеката које су мериле да би утврдиле успех у примени агилних метода у својој пракси, што је приказано на слици 1.14.

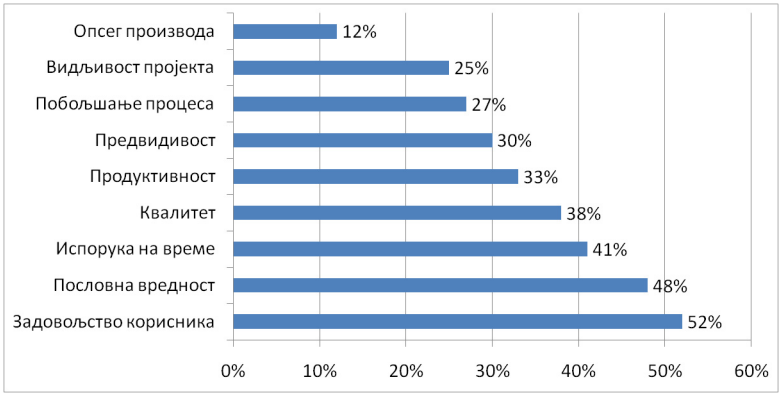
Табела 1.3: *Предности агилних методологија*

Предности	**Опис**
Прихватљивост промена	Промене се могу појавити у било ком тренитку реализације пројекта, и ако се процени да су промене корисне и потребне, оне ће бити и прихваћене. На тај начин се обезбеђује да се коначни производ усаглашава са реалним потребама. Прихватањем промена, мења се листа задатака у пројекту (*backlog*) и то прихватају сви тимови који раде.
Крајњи циљ може бити непознат (није јасно дефинисан)	У реализацији појединих пројеката се крајњи циљ не може јасно дефинисати на самом почетку реализације пројеката. Током реализације пројекта циљеви се мењају, усаглашавају и у неком тренутку се долази до јасно дефинисаног циља на којем се ради до окончања пројекта.
Брзе и квалитетне вишеструке испоруке производа	Реализација пројекта у малим итеративним циклусима омогућује да се брзо развију најбитније функционалности, да се изврши тестирање и открију грешке. Такође се брзо долази и до повратне инфорррмације од клијента, па све то доприноси повећању квалитета испорученог производа.
Динамична и снажна интеракција у тиму	Основна карактеристика је честа комуникација чланова тима, што је подржано одржавањем редовних дневних и недељних састанака. Подржава се комуникација лицем-у-лице чиме се чланови тима подстичу да међусобно помажу једни другима у раду и решавању проблема.
Континуирана побољшања на основу повратних информација	Честе испоруке софтвера након сваког циклуса омогућују клијенту да стекне увид у реализоване функционалности и да достави информације о томе шта треба мењати и радити даље. Такође, на основу повратних информација се у самом тиму обезбеђује учење кроз лекције које је тим заједно савладао у претходним етапама рада на пројекту или у претходним пројектима.

Табела 1.4: *Недостаци агилних методологија*

Недостаци	**Опис**
Недовољно прецизно планирање	Због начина рада агилних тимова и комплексности пројеката, некада је немогуће прецизно одредити све рокове у реализацији пројекта, као и рок испоруке крајњег производа. Због тога је често потребно извршити реприоритизацију задатака и наставити пројекат са новим планом.
Мали и веома стручни тимови	Агилни тимови су обично мали и због тога чланови тима морају бити вешти и имати знања и вештине из разних области које су обухваћене пројектом који се реализује.
Потпуна посвећеност чланова тима	Потпуна посвећеност и сарадња чланова тима је неопходна за реализацију пројекта, што често подразумева више утрошеног рада од првобитно планираног.
Занемаривање документације	Агилне методе и тимови често занемарују документацију на уштрб рада на развоју производа. Међутим, одређени ниво документације је потребан да би се јасно образложила реализација пројекта, па агилни тимови треба ту да пронађу своју меру.
Могући су различити финални производи	Због флексибилности у реализацији пројекта и могућности промене захтева, може се десити да финални производ буде другачији од првобитно договореног. То треба добро документовати током реализације пројекта.

Слика 1.13: *Разлози за прихватање агилних метода*



Слика 1.14: *Начин мерења успешне реализације агилних пројеката*

Слика 1.15: *Елементи скрам методологије*

1.2.2 Скрам (Scrum)

Агилна методологија која се најчешће користи у индустријској пракси је *Scrum*. Превасходно је дизајниран за мале тимове до 10 чланова који свој рад деле на циљеве који се могу комплетирати у оквиру временски дефинисаних интервала који се називају спринтови (*sprints*). Назив методологије је позајмљен из рагби игре, где представља формацију играча која заједно ради на остварењу циља, при чему сви чланови тима морају дати свој одговарајући допринос на позицији где се налазе у тиму (https://en.wikipedia.org/wiki/Scrum_(rugby)).

Скрам је базиран на тимовима који решавају сложене проблеме, и уз повећану креативност и продуктивност чланова тима, обезбеђује испоруку производа високог квалитета. Радни оквир скрама се састоји од улога у тиму, догађаја, предмета и правила, што је приказано на слици 1.15. Правила повезују улоге у тиму, догађаје и предмете.

Скрам се данас користи у разним областима производње (софтвер, хардвер, индустрија), али и у другим областима људске делатности (образовање, маркетинг) и живота (уређење куће). Скрам су креирали *Ken Schwaber* и *Jeff Sutherland* током деведесетих година прошлог века на бази емпиријских искустава из привреде. Основна идеја је да се све одлуке доносе на основу доступних чињеница и да се цео процес базира на инкременталном и итеративном приступу који смањује ризик. Основне претпоставке контроле процеса у скраму су:

- **Транспарентност**. Сви аспекти процеса морају бити видљиви свим релевантним учесницима који су одговорни за резултате процеса, чиме се обезбеђује боље разумевање целог процеса.
- **Интроспекција**. Сви предмети који настају током процеса се често прегледају чиме се обезбеђује напредовање у процесу и избегавају нежељене ситуације.
- **Прилагођавање**. Процес се стално прати и анализира да би се прилагодио датим околностима и избегла одступања од постављених циљева.

Улоге у тиму

Скрам тим се састоји од власника производа (*Product Owner*), *Scrum Master*-а и тима за развој (*Development team*). Основна одлика тима је да се сам организује и поправља током рада, што значи да тим сам доноси одлуке како ће нешто урадити. Такође, тим се формира тако да има сва потребна знања за реализацију пројекта, а основне одлике су креативност, флексибилност и продуктивност.

Власник производа (*Product Owner*) је члан тима који је одговоран за добијање максималне вредности производа кроз рад развојног тима. Власник производа управља производним задацима у листи ставки производа (*product backlog*), што подразумева јасно дефинисање ставки тако да их развојни тим потпуно разуме, као и одређивање редоследа и приоритета ставки.

Scrum Master је одговоран за имплементацију скрам методологије у оквиру пројекта, што значи да треба да обезбеди да сви у тиму разумеју цео процес, све елементе који чине скрам, као и да подстиче интеракције тима. *Scrum Master* сарађује са власником производа око ставки у листи производа, а са тимом око реализације свих ставки које чине листу ставки спринта (*sprint backlog*).

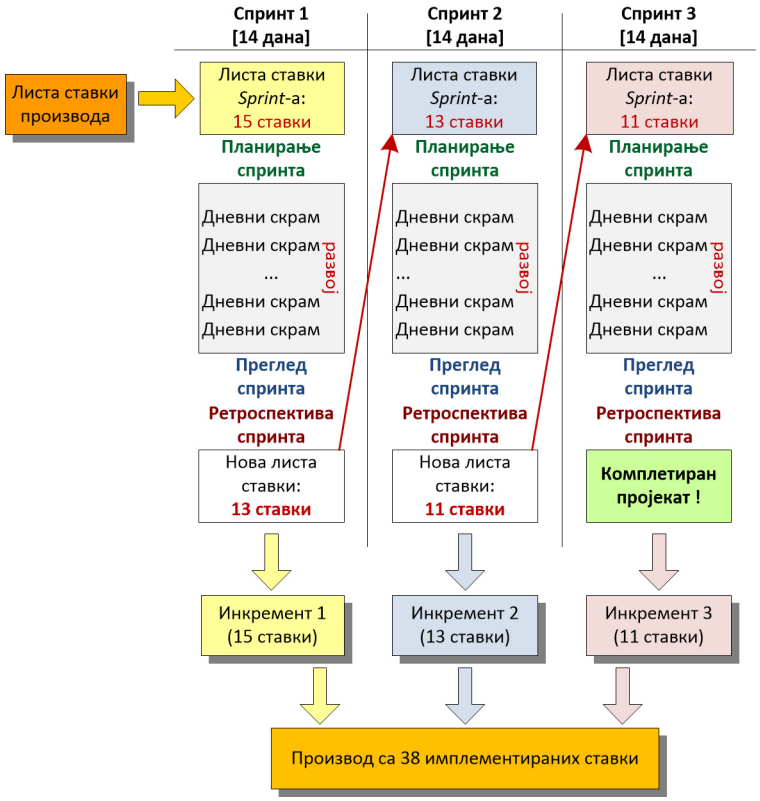
Развојни тим (*Development team*) је скуп професионалаца који раде на решавању ставки из листе ставки производа са циљем да се испоручи употребљив производ на крају сваког спринта. Тимови су обучени да раде самостално и да самостално доносе одлуке о реализацији пројекта. Тим садржи експерте који имају сва потребна знања за реализацију свих ставки пројекта.

Предмети (Артифакти)

Предмети представљају вредности или рад који је предмет интроспекције и прилагођавања. Све информације о предметима морају бити транспарентне да их разуме цео тим и да може потом радити на побољшањима. Основни предмети су: листа ставки производа, листа ставки спринта и инкременти.

Листа ставки производа (*Product Backlog*) је уређена листа свега што је потребно за развој производа. За листу ставки производа је одговоран власник производа, што укључује садржај, доступност и редослед свих ставки. Листа ставки се може мењати током реализације пројекта, што зависи од тренутне реализације која има за циљ да се увек ради актуелан и користан производ. Листа садржи функционалности, функције, захтеве, побољшања, корекције за будућа издања. Листа је динамична пошто се захтеви могу стално мењати. Из листе ставки производа се бирају ставке које ће се реализовати у спринтовима. Власник производа прати стање релизованости сваке ставке из листе ставки производа након сваког реализованог спринта.

Листа ставки спринта (*Sprint backlog*) је скуп ставки из листа ставки производа које су одабране за реализацију у текућем спринту. Ставке у листи ставки спринта представљају процену тима које функционалности ће бити реализоване и колики посао је потребан да се те функционалности реализују,



Слика 1.16: *Пример реализације пројекта применом скрам методологије*

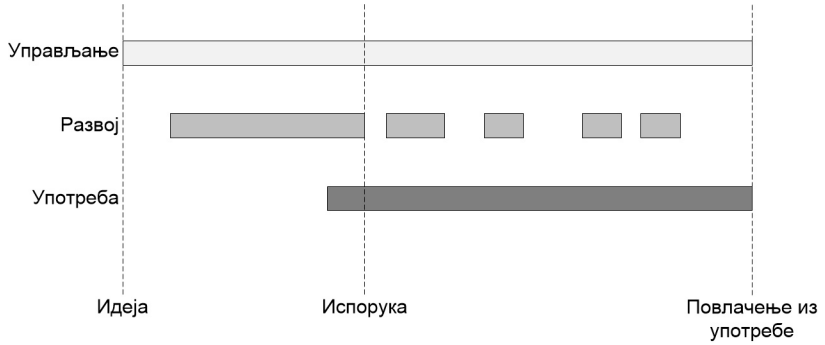
тј. стигну до стања "урађено" (*done*). Листа ставки спринта мора садржати довољно објашњења да се могу реализовати свакодневни скрамови.

Инкремент (*Increment*) је збир свих ставки из листе ставки производа које су реализоване током спринта. Свако побољшање мора бити означено са "урађено" и мора бити употребљиво, а власник производа одлучује да ли ће га пустити у наредно издање производа.

Догађаји

Догађаји који се реализују у скраму обезбеђују да се пројекат што ефикасније реализује. Сваки догађај има тачно предвиђено време када треба да се деси, јасно дефинисано и ограничено трајање, и скуп предмета на које утиче. Основни догађаји су: спринт, планирање спринта, дневни скрам, преглед спринта и ретроспектива спринта. Реализација пројекта применом скрам методологије на којој се јасно могу уочити сви битни догађаји и предмети је приказана на слици 1.16.

Спринт (*Sprint*) је догађај који траје одређени временски период, типично од 1 недеље до месец дана и на крају којег се испоручује готов производ (инкремент). Трајање свих спринтова је непроменљиво, а реализују се један за другим. Сваки спринт се састоји из планирања, свакодневног скрама, рада



Слика 1.17: *Аспекти животног циклуса софтвера*

на развоју, прегледа спринта, и ретроспективе спринта. Пример реализације 3 спринта са инкременталном интеграцијом коначног производа који се састоји од елемената развијаних у сваком од спринтова је приказан на слици 1.16. Када се стартује спринт, не смеју се вршити никакве измене које могу угрозити његову реализацију.

Планирање спринта (*Sprint planning*) се ради пре почетка развоја, а у планирању учествује цео тим. За планирање је одговоран *Scrum Master* који руководи тимом током планирања. Током планирања се дефинише шта може бити урађено током спринта, како то може бити урађено и шта је циљ спринта. Циљ **прегледа спринта** (*Sprint review*) је да тим сагледа шта је остварено током спринта (имплементиране ставке) и да сагледа шта треба радити у наредном спринту. Током **ретроспективе спринта** (*Sprint retrospective*) тим сагледава шта је добро рађено током спринта, шта може бити унапређено и како то треба унапредити.

Дневни скрам (*Daily Scrum*) је 15-минутни састанак развојног тима који се одржава сваког дана. Током састанка тим анализира шта је урадио током претходног дана, планира шта трба да уради у наредном, и на које проблеме је наишао током рада. На тај начин се оптимизује рад тима, унапређује комуникација и прати се реализација ставки из листе ставки спринта. Тим је одговоран за реализацију састанка.

1.3 Управљање животним циклусом софтвера

Управљање животним циклусом софтвера обухвата три аспекта животног циклуса: управљање, развој софтвера и употреба софтвера. Поједине фазе животног циклуса су разграничене значајним догађајима, као што је приказано на слици 1.17.

Животни циклус почиње са *идејом* да је потребно креирати софтвер који је неопходан за решавање одређене групе проблема у некој области. Након тога следи фаза интензивног развоја софтвера, што је предмет детаљног истраживања многих књига и курсева које се баве овом тематиком у склопу софтверског инжењерства. Након фазе развоја следи *испорука* софтверског производа (*delivery*) који је спреман за употребу, и потом његово

инсталирање и подешавање. Након тога је софтвер у фази употребе где се као значајне активности појављују активности одржавања софтвера (*software maintenance*), које по потреби укључују циклусе новог развоја. Нови развој у фази одржавања се јавља када је потребно у софтверски производ уградити нове функционалности, или када је потребно софтвер прилагодити новом окружењу. Када софтвер више не доноси корист у пословању, престаје његово сервисирање, он се *повлачи из употребе* (*retirement*), и тиме се завршава животни циклус софтвера.

Фазе развоја и употребе се односе само на поједине етапе у животном циклусу, док се управљање животним циклусом јавља током целог животног циклуса. Управљање је неопходно током целог животног циклуса да би се пратило стање софтвера током развоја, његова употреба и да би се донеле одговарајуће одлуке у сваком тренутку. Управљање животним циклусом је најчешће током развоја софтвера планирано, а након испоруке је базирано на реактивним активностима које се јављају као одговори на одређене догађаје (нпр. захтеви за модификацијом или за корекцијом грешака).

Употреба софтвера почиње непосредно пре официјалне (званичне) испоруке, а односи се на фазу тестирања софтвера од стране корисника да би се сагледали потенцијални недостаци и утврдило да ли софтвер задовољава све захтеве корисника.

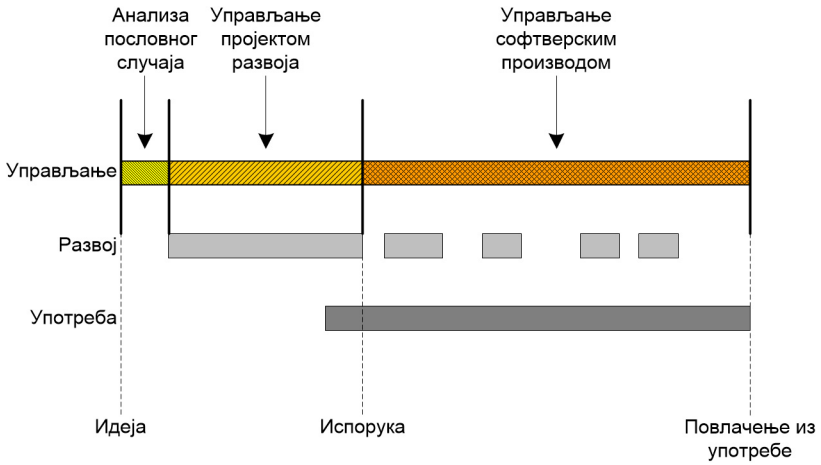
1.3.1 Аспект управљања у животном циклусу софтвера

Управљање животним циклусом треба да обезбеди да софтвер увек задовољава пословне потребе корисника, па се управљање мора реализовати током целог животног циклуса. Управљање почиње од идентификовања идеје за развој софтвера, а затим се наставља анализом случаја употребе софтвера (најчешће пословно окружење). Овај аспект се јавља у фази развоја, али пре него што започну конкретне активности које резултују елементима новог софтверског производа (прикупљање и спецификација захтева, дизај, кодирање, тестирање). Када почне стварни развој софтвера, управљање животним циклусом превасходно обухвата активности управљања пројектом развоја. Након испоруке софтвера, управљање циклусом се своди на скуп активности и метода за управљање апликацијама које се користе и за које се пружају услуге одржавања. Процес управљања животним циклусом софтвера је приказан на слици 1.18.

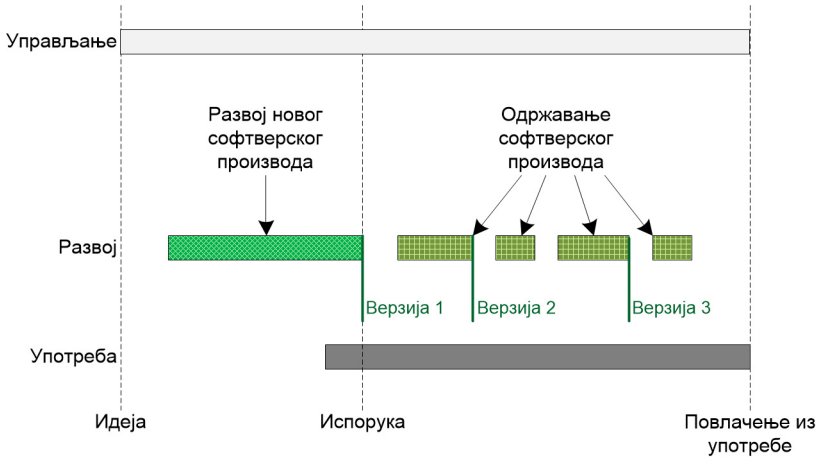
Управљање током употребе софтвера је базирано на праћењу употребе софтвера, пружање подршке корисницима и спровођење активности одржавања на основу захтева корисника. Управљање у животном циклусу је од примарне важности за остваривање максималне пословне вредности софтвера.

1.3.2 Аспект развоја у животном циклусу софтвера

Развој, или пројектовање софтвера је основна фаза у животном циклусу софтвера, када се на основу пословне идеје врши реализација производа. Развој софтвера је итеративан процес, иако је овде приказан правом линијом.



Слика 1.18: *Аспект управљања у животном циклусу софтвера*



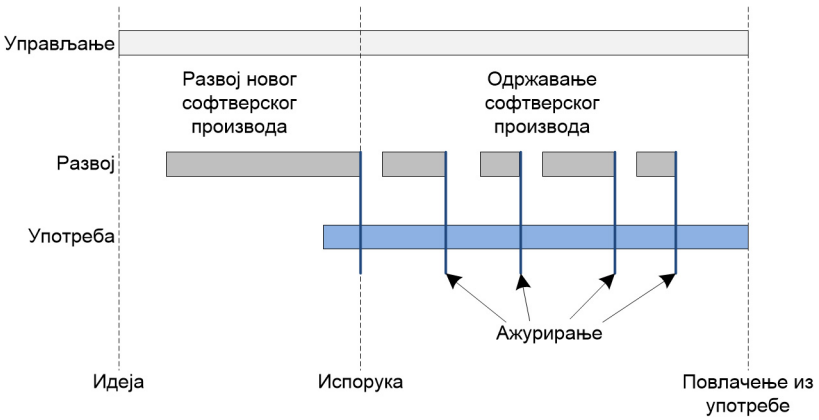
Слика 1.19: *Аспект развоја софтвера у животном циклусу софтвера*

Пре испоруке софтвера се јавља класична фаза развоја софтвера, док се након испоруке софтвера, у склопу одржавања постојећег производа, могу јавити активности развоја. Аспект развоја софтвера током животног циклуса је приказан на слици 1.19.

Врло је важно напоменути да развој није сасатавни део свих активности одржавања (на пример, приликом корекције грешке се не реализује развој софтвера) већ само када се врши унапређење или проширење функционалности софтвера, а приликом значајнијих радова развоја се издаје нова верзија софтвера.

1.3.3 Аспект употребе у животном циклусу софтвера

Употреба софтвера од стране корисника почиње након испоруке, мада се иницијална употреба јавља и пре испоруке да би се сагледале могућности и



Слика 1.20: *Аспект употребе и одржавања у животном циклусу софтвера*

карактеристике новог производа и по потреби извршиле корекције на основу постављених захтева. Након тога започиње фаза праћења употребе софтверске апликације и њено одржавање. Током одржавања постоје активности које укључују развој и измене постојећег софтвера, након чега се јавља промена верзије и ажурирање нове верзије за употребу, као што је приказано на слици 1.20.

Фаза употребе и одржавања може бити веома дуга, чак значајно дужа од фазе развоја. Истраживања указују да су трошкови ове фазе између 40 и 90% од укупних трошкова животног цикуса, а поједини аутори указују да су трошкови ове фазе преко 80% укупних трошкова животног циклуса. За софтверске производе који су веома дуго у употреби трошкови одржавања вишеструко премашују трошкове развоја.

Поглавље 2

Инжењеринг софтверских захтева

Сви модели животног циклуса софтвера укључују активности које се односе на сагледавање и спецификацију захтева које софтверски производ треба да испуни. Софтверски захтеви треба да обезбеде разумевање намене и функционалности софтверског система у посматраном домену употребе. У својој књизи *The Mythical Man-Month: Essays on Software Engineering*, *Brooks* указује да је најтежи део развоја софтвера одлучити шта тачно треба да се креира, тј. дефинисање захтева које он треба да испуни. Због тога се посебна пажња у развоју софтвера посвећује итеративном откривању, прочишћавању и јасном дефинисању захтева. Област софтверског инжењерства која се бави систематским руковањем софтверских захтева се назива *Инжењеринг софтверских захтева*.

Потреба корисника за новим софтвером се изражава кроз спецификацију софтверских захтева. Софтверски захтеви одражавају потребе корисника које се односе на функционалне или друге карактеристике софтверског производа, али не садрже техничку спецификацију имплементације као што је избор архитектуре софтвера или система за руковање подацима. Циљ спецификације софтверских захтева је да се разумеју потребе и проблеми корисника софтвера. Софтверски захтеви треба да укажу које функционалности софтвер треба да садржи, али не и *како* се те функционалности имплементирају. Најједноставнија дефиниција софтверског захтева према књизи *Guide to the Software Engineering Body of Knowledge (SWEBOK)* је:

> *Софтверски захтев је својство које се исказује на одговарајући начин да би се омогућило решавање преоблема из реалног света помоћу софтверског система.*

Софтверски захтеви за сваки софтверски систем су сложена комбинација захтева више људи који на различит начин користе софтвер у посматраном домену употребе. Без обзир ко је извор софтверског захтева, сваки захтев мора бити верификован или као специфичан функционални

захтев или као нефункционални захтев на нивоу софтверског система. Софтверски захтеви се изводе из доменских захтева, који треба да одсликавају специфичности домена употребе софтвера али и специфичне потребе корисника. Основни проблем са доменским захтевима настаје због тога што софтверски инжењери могу имати проблема са разумевањем карактеристика домена у којем се софтвер користи, због чега некада нису у могућности да сагледају да ли су сви захтеви обухваћени, као и да ли постоје конфликти у спецификацијама различитих захтева. С обзиром да различити учесници у процесу спецификације софтверских захтева имају различите погледе на захтеве и софтверски систем, добра пракса је да се у документу са спецификацијом захтева дефинишу сви потребни појмови како би се смањиле могућности за неразумевање и грешке.

Главни извор сложености области инжењеринга софтверских захтева је њена хетерогеност са аспекта примене техничко-технолошких метода и алата у процесу захтева, али и са аспекта укључења у процес различитих учесника и организација које су заинтересоване за развој софтвера. Области које утичу на инжењеринг софтверских захтева су домен проблема где се софтвер користи, карактеристике организације и људи које користе софтвер и организације која производи софтвер, примена различитих технолошких решења у имплементацији софтвера, као и примена различитих методологија, метода и алата у развоју софтвера. Према томе, квалитетни и добро специфицирани софтверски захтеви подразумевају одговарајућу комбинацију следеће три димензије у инжењерингу софтверских захтева: људи, организација и технологија. Ове три димензије се допуњују и омогућују да се проблеми у инжењерингу софтверских захтева сагледају из различитих перспектива.

Основна подела софтверских захтева је на функционалне захтеве и нефункционалне захтева.

Функционални захтеви (*Functional requirements*) описују понашање софтверског система, скуп активности које се реализују и стање система пре и након извршења активности. Функционални захтеви такође описују и могуће изузетке у понашању, улазе (ресурсе) који су неопходни за реализацију активности и излазе (резултате) активности. Функционални захтеви могу бити општи у погледу шта систем треба да ради, али и врло специфични и одређени локалним правилима и начином рада у неком подсистему. Функционални захтеви морају бити комплетни, што значи да све функционалности морају бити идентификоване и прецизно дефинисане. Такође, веома је важно да су функционални захтеви конзистентни, што значи да различити захтеви не садрже противречне дефиниције функционалности софтверског система.

Нефункционални захтеви (*Nonfunctional requirements*) су захтеви који се најчешће односе на карактеристике квалитета и ограничења функционалности које пружа софтверски систем. Ови захтеви често одсликавају очекивања корисника о томе како софтвер треба да ради, као на пример: колико се лако софтвер користи, колико брзо се извршава, колико често се јављају откази или како рукује непредвиђеним ситуацијама. Ова ограничења су најчешће наметнута стандардима, правилима и процедурама који важе у домену проблема, али могу бити и временска ограничења или

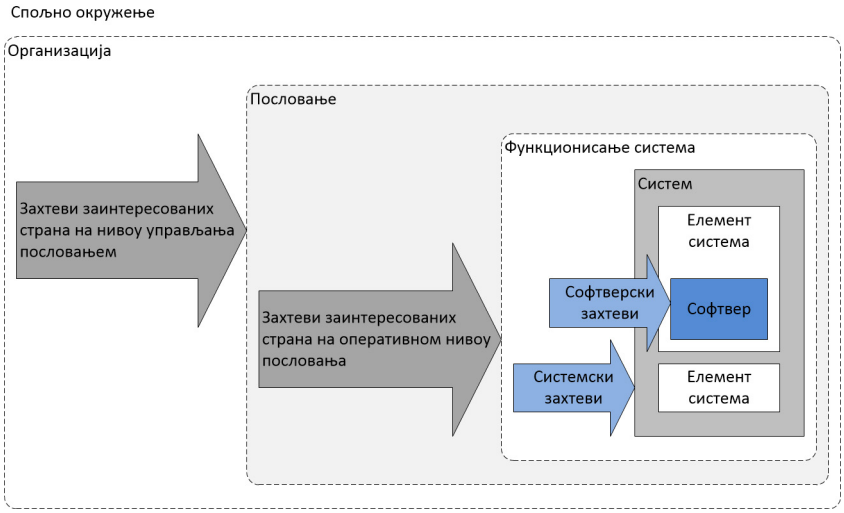
ограничења појединих процеса. Нефункционални захтеви се обично односе на систем као целину, али се могу односити и на поједине делове система. Нефункционални захтеви се могу односити на начин презентовања података, карактеристике корисничког интерфејса, карактеристике улазно/излазних уређаја, сигурност система итд. Нефункционални захтеви се на основу извора могу поделити на следеће групе:

- **Захтеви производа** су захтеви који потичу од карактеристика и ограничења у вези софтверског производа и његових перформанси, као што су брзина извршавања, употреба меморије, прихватљиви степен отказивања, употребљивост, итд.
- **Организациони захтеви** потичу од ограничења која постоје у правилима и процедурама организације која је наручилац софтвера и организације која производи софтвер. У ову групу спадају захтеви који се односе на избор оперативног система за извршавање софтвера, начин употребе софтвера, избор радних оквира и програмског језика за реализацију софтвера, као и поштовање процеса и стандарда у домену примене и развоја софтвера.
- **Спољашњи захтеви** се односе на све захтеве који се не односе на софтверски производ и организације које су укључене у његов развој (произвођач) и употребу (корисници). У ову групу спадају ограничења у вези разних регулатива пословања, законодавна ограничења и етички аспекти употребљивости софтвера.

Спецификација нефункционалних захтева треба да буде изражена квантитативно да би се могло извршити објективно тестирање тих захтева. Уобичајене метрике које се користе за специфицирање нефункционалних захтева су: брзина (трансакције, брзина одзива система), величина (употреба меморије, хардверски уређаји), лакоћа употребе (број потребних тренинга), поузданост (средње време отказа), робустност (време рестарта након отказа) и портабилност (број система на којима је могућа имплементација).

Јасна граница између функционалних и нефункционалних захтева врло често не постоји. На пример, захтев по питању сигурноси и идентификације корисника је нефункционалан захтев, али се приликом детаљне анализе и потом конструкције софтвера преводи у низ функционалних захтева који обезбеђују механизме заштите сигурности система и корисника.

У складу са *ISO/IEC/IEEE 29148:2011(E)* међународним стандардом који се односи на софтверске захтеве у конетксту инжењеринга сложених система, софтверски захтеви се морају посматрати у ширем контексту који обхвата пословање и организациони контекст организације која је наручилац софтвера, али и спољно окружење, као што је приказано на слици 2.1. Фактори спољног окружења који могу утицати на захтеве корисника и свих заинтересованих учесника у процесу захтева су: трендови на тржишту, закони и регулативе, радна снага која је доступна за постављене технолошке захтеве, стандарди, друштвено и културно окружење и животна средина. Спољни фактори утичу на организацију која поставља захтеве за софтверски производ (или сложени систем), али и на софтверску организацију која развија и испоручује систем и софтвер.



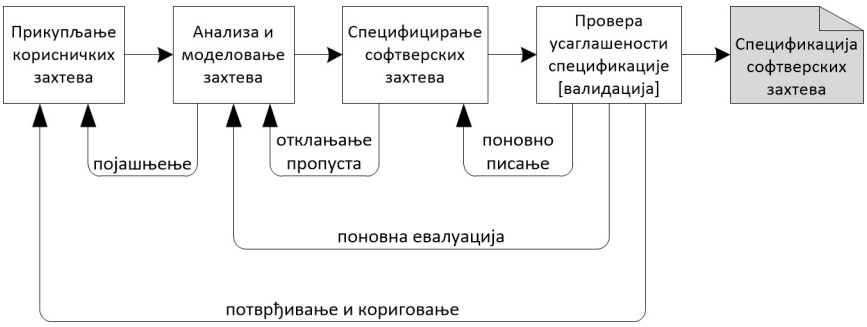
Слика 2.1: *Софтверски захтеви у контексту сложених пословних система*

Захтеви заинтересованих страна у процесу захтева се могу посматрати на нивоу организације и тада се односе на управљање пословањем, или на оперативном нивоу пословања када се могу диференцирати на системске захтеве (односе се на систем као целину или на поједине делове система) и софтверске захтеве (односе се на софтверске компоненте у систему). Захтеви на организационом нивоу морају у обзир узети правила и процедуре у организацији, стандарде и примењене технологије, али и организациону кутуру. На оперативном нивоу пословања захтеви морају узети у обзир правила и ограничења пословних процеса, постојећи систем квалитета у пословању и пословну структуру организације.

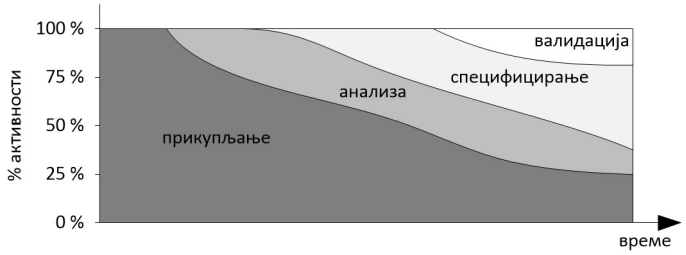
2.1 Процес софтверских захтева

Процес софтверских захтева омогућује сагледавање процесирања софтверских захтева на високом нивоу, али и идентификовање ресурса и ограничења који утичу на реализацију процеса. Типичан процес утврђивања и израде спецификације софтверских захтева за софтверски систем је приказан на слици 2.2. Резултат процес спецификације софтверских захтева је документ са спецификацијом захтева (*Software Requirements Specification (SRS)*), чија је спецификација приказана стандардом *IEEE Std. 830-1998 (Revision 2009) Recommended Practice for Software Requirements Specifications*.

Особе које су од стране софтверске организације ангажоване у процесу софтверских захтева се обично називају *аналитичар захтева* или *систем аналитичар*. Први посао је да се од клијената и корисника прикупе захтеви кроз постављање питања, испитивање уобичајених понашања или приказивање сличних софтверских решења. Након тога се анализом прикупљених захтева креира модел или прототип који омогућује софтверским



Слика 2.2: *Процес израде спецификације софтверских захтева*



Слика 2.3: *Временска димензија и паралелизам активности у процесу израде спецификације софтверских захтева*

инжењерима да боље разумеју софтвер који треба изградити. У поступак анализе се укључују и корисници и клијенти да би се кроз додатна питања прочистили захтеви. Тек када се кроз овај итеративан процес прочисте захтеви прелази се на спецификацију софтверских захтева који ће бити имплементирани у софтверу. Спецификација се валидира од стране корисника софтвера и као коначни резултат се добија документ са спецификацијом захтева (Software Requirements Specification (SRS)).

Процес софтверских захтева је итеративан процес у којем се поједине активности преклапају, што је приказано на слици 2.3. У почетку пројекта се сво време троши на прикупљање захтева, а касније се тај део смањује на рачун активности анализе и специфицирања захтева које се постепено уводе и троше све више времена. Валидација захтева се реализује тек пред крај процеса захтева, тако да се тада све четири активности могу реализовати паралелно. Итеративност процеса софтверских захтева указује да је могуће појављивање нових захтева корисника, или измена постојећих захтева након валидације захтева од стране корисника, што је основна карактеристика агилних приступа у развоју софтвера који доминирају у индустријској пракси.

Софтверска организација која пројектује софтвер према захтевима корисника треба да има одређене карактеристике да би ефикасно креирала софтвер и задовољила захтеве корисника. У ту сврху је уведен **Индекс способности у спецификацији захтева** (*Requirements Specification Capability Index*) који се састоји од следећих шеста аспеката праксе спецификације софтверских захтева: изградња визије захтева, комуницирање

у процесу захтева, дизајн захтева као производа, дизајн процеса захтева, оптимизација инвестиције у захтеве, и развој људских ресурса који се баве захтевима. Свака софтверска организација прилагођава и мери вредности ових индексе у складу са специфичностима процеса које реализује.

2.1.1 Прикупљање софтверских захтева

Прикупљање софтверских захтева је процес тражења, откривања и елаборирања захтева које треба да испуни софтвер. У литератури се често користе и називи аквизиција или откривање софтверских захтева. У овој фази процеса софтверских захтева софтверски инжењери раде заједно са корисницима, клијентима и осталим заинтересованим странама на идентификацији софтверских захтева који обезбеђују ефикасну употребу софтвера у одабраном домену. Поред тога кроз интеракцију са корисницима се идентификују и ограничења која утичу на развој и употребу софтверског производа, као и захтеване перформансе које треба да обезбеде очекивани квалитет софтвера. Прикупљање захтева је вишеслојни и итеративни процес који подразумева комуникационе вештине инжењера који прикупља захтеве, али и потпуну посвећеност и сарадњу свих учесника и заинтересованих страна.

Прикупљање софтверских захтева је прва и најкритичнија фаза у процесу софтверских захтева која значајно утиче на успешност целог пројекта развоја софтвера. На критичност фазе прикупљања захтева највише утичу следеће околности и фактори:

- Корисници и клијенти најчешће нису сигурни шта тачно желе да добију софтвером који треба пројектовати, већ то најчешће описују уопштеним терминима. Такође, врло често су њихови захтеви нереални пошто не могу да сагледају шта је изводљиво приликом пројектовања софтвера.
- Корисници и клијенти најчешће захтеве описују употребом специфичне терминологије коју само они разумеју, при чему подразумевају да и софтверски инжењери имају довољно знања да све то разумеју, тј. да имају довољно доменског знања.
- Различити учесници имају различите захтеве које описују на специфичан начин, а софтверски инжењер треба да открије све могуће изворе захтева и да сагледа њихове сличности и могуће конфликте.
- Различити учесници који представљају изворе софтверских захтева имају различит утицај на важност одређених захтева. Због тога се може десити да се прави захтеви не прикупе пошто утицајне особе, као што су на пример менаџери, могу својим захтевим дати већи значај од захтева осталих запослених у клијентској организацији.
- Организација која наручује израду софтвера је динамична и врло често се захтеви могу мењати током пројекта. Разлози за то могу бити појава нових корисника софтвера са својим специфичним захтевима, измена постојећих или појава нових процеси који раније нису разматрани, промена пословне политике организације, и измена стандарда или

регулатива које утична на домен делатности организације која наручује софтвер.

Активности које се реализују приликом прикупљања софтверских захтева морају обезбедити комуникацију, преговарање и сарадњу свих релевентних актера, при чему је важно превазићи све разлике и проблеме и остварити ефикасан и конструктиван дијалог. Основни типови активности приликом прикупљања софтверских захтева су:

- **Разумевање домена примене софтвера**. Домен проблема је потребно истражити са аспекта политичких, организационих и социјалних аспеката. Такође треба сагледати и могућа ограничења која могу утицати на развој софтвера. Познавање домена проблема и идентификоване знања о домену су кључни за идентификовање и квалитет софтверских захтева.
- **Идентификовање извора за прикупљање захтева**. Захтеви се могу прикупити из различитих извора и могу постојати у различитим облицима (форматима). Најчешће су извор захтева корисници софтвера, али се често користе и увид у постојећи систем, процесе и документацију.
- **Анализа актера**. Актери су особе које имају интерес у развоју и имплементацији софтвера па је њих потребно идентификовати и укључити у активности прикупљања захтева. То могу бити људи у организацији која наручује софтвер, људи у организацији која производи софтвер али и спољни актери који утичу на домен пословања (реулатори, друштвена заједница, итд.) Анализа и идентификовање одговарајућих актера приликом прикупљања захтева значајно утиче на квалитет прикупљених захтева.
- **Избор приступа, технике и алата за прикупљање захтева**. Сваки пројекат, са специфичним захтевима у домену проблема и специфичним актерима захтева пажљив избор техника и алата за прикупљање захтева. Најчешће се приликом прикупљања захтева користи више различитих техника и алата. Врло често избор техника зависи од знања и искуства софтверских инжењера који су ангажовани у прикупљању захтева. За прикупљање захтева се најчешће користе технике као што су колаборативне сесије, интервјуи, анализа задатака и послова, тимски састанци, брејнсторминг сесије, приче корисника, етнографија, модели, упитници, сценарији, прикупљање података из постојећих система и израда прототипова.
- **Прикупљање захтева од актера и из осталих извора**. Прикупљање захтева почиње када су идентификовани сви актери, извори захтева и одабране технике и алати. Најважније је да се приликом прикупљања захтева јасно утврди опсег софтверског система, као и начин интеграције софтвера у постојеће пословно окружење. С обзиром на критичност фазе прикупљања захтева потребно је идентификовати све изворе захтева, што у принципу не морају бити само актери већ могу бити и разни документи, стандарди или пословно окружење.

2.1.2 Анализирање софтверских захтева

Анализа корисничких захтева треба да омогући разумевање шта корисници очекују од софтвера, што треба да резултира имплементацијом само неопходних функционалности уз одговарајуће перформансе. Анализа захтева треба да омогући решавање конфликата међу захтевима, идентификовање границе софтвера и начин интеракције са окружењем, као и елаборацију системских захтева на основу којих ће се извести софтверски захтеви. Анализа захтева почиње још током прикупљања захтева од актера, када софтверски инжењер води процес прикупљања недвосмислених захтева на основу којих се могу проценити трошкови развоја софтвера и валидација од стране корисника у свим фазама развоја. Уобичајене активности у анализи захтева су:

- **Класификација и организација**. Класификација захтева полази од неструктурисане колекције захтева који се групишу у смислене повезане групе. Груписање захтева се најчешће врши на основу архитектуре целог система, идентификације подсистема, а потом се врши придруживање група захтева сваком од подсистема. У пракси се најчешће током груписања захтева врши профилисање дизајна архитектуре система, што утиче на успешност наредних фаза у развоју софтвера. Даље се класификација захтева врши на функционалне и нефункционалне или на захтеве придружене производу и процесима.
- **Приоритизација и усаглашавање**. Због учешћа великог броја актера у прикупљању захтева долази до конфликата у захтевима. Циљ ових активности је да се сви конфликти реше кроз усаглашавање и компромис свих учесника и да се на основу тога изврши приоритизација захтева. Истраживање индустријске праксе указује да постоји значајан број техника за приоритизацију захетав, али да се у сваком посебном пројекту мора пажљиво одабрати метода која највише одговара. Приоритизација захтева се врши у јасно дефинисане класе приоритета на основу балансирања трошкова и ризика у развоју и имплементацији софтвера. Најчешће класе приоритета захтева су: обавезни, веома пожељни, пожељни и опциони.
- **Концептуално моделовање**. Успешна анализа захтева подразумева концептуално моделовања домена проблема где ће се софтвер користити. Циљ концептуалних модела је да се постигне боље разумевање проблема и да се скицирају могућа решења. Појмови који постоје у домену проблема и њихове међусобне релације се приказују концептуалним моделима, који касније еволуирају у друге формалне моделе који се користе у развоју софтвера. У ту сврху се могу креирати различити модели, као што су дијаграми случајева коришћења, дијаграми токова података, модели стања, модели базирани на циљевима, модели интеракције или објектни модели. За разумевање захтева се најчешће код традиционалних метода базираних моделу водопада користе дијаграми случајева коришћења, док се код агилних метода користе приче корисника.

У каснијим фазама анализе захтева, када је скуп захтева стабилан и елаборирaн у конкретне функционалности и карактеристике софтвера, приступа се формалним техникама моделовања које производе моделе који се директно користе у конструкцији софтвера. Формално моделовање захтева је корисно због јасно дефинисаног начина описа захтева који користе сви софтверски инжењери ангажовани у конструкцији софтвера. Формална спецификација се користи за ефикаснију примену софтверских алата у пројектовању софтвера и за прочишћавање неформалне спецификације која се формира у почетним фазама анализе захтева.

2.1.3 Специфицирање софтверских захтева

Специфицирање софтверских захтева се односи на израду документа са спецификацијом захтева за софтверски систем. Креирани документ омогућује систематичан преглед, евалуацију и одобравање захтева. Приликом креирања документа са спецификацијом захтева треба користити стандардне нотације и терминологију која је јасно дефинисана да би се обезбедила што ефикаснија сарадња свих ученика у процесу софтверских захтева и касније у осталим фазама развоја софтвера. Спецификација захтева не сме бити превише техничка да би могли да је разумеју корисници софтвера и сви остали актери (менаџери организације која наручује софтвер, аналитичари, итд.) који нису директно укључени у развој софтвера.

Измена спецификације захтева се врши ако се у наредним фазама, као што су валидација захтева или детаљни технички дизајн софтвера, открију проблеми или недостаци у захтевима. Постоје следећи типови недостатака у спецификацији захтева:

- **Изостављање**. Информације потребне за разумевање захтева су изостављене у документу.
- **Двосмисленост**. Спецификација захтева је записана тако да се може интерпретирати на више начина.
- **Противречност**. Поједини делови у документу са спецификацијом захтева су међусобно противречни.
- **Излишност**. Поједини делови спецификације су непотребни или нису релевантни за опис проблема и спецификацију решења.
- **Некоректност**. Поједине информације у документу су међусобно контрадикторне или нису усаглашене са претходно креираним документима.
- **Неусаглашеност са стандардима**. Захтеви нису специфицирани у складу са стандардима који прописују квалитет спецификације захтева, процеса или пословања.
- **Неизводљивост**. Због ограничења која се могу односити на систем, људске ресурсе, буџет или технологију, захтеве није могуће имплементирати према наведеној спецификацији.
- **Ризичност**. Непостојаност спецификације или превелика повезаност захтева утиче на ризик у њиховој имплементацији.

За сложеније системе који садрже и компоненте које нису софтвер креирају се три документа: документ са дефиницијом система, документ са системским захтевима и документ са софтверским захтевима. Код једноставнијих система са само софтверским компонентама креира се само документ са софтверским захтевима који се назива **Спецификација софтверских захтева** (*Software Requirements Specification (SRS)*). Спецификација софтверских захтева се најчешће пише природним језиком који се користи у окружењу где се софтвер интегрише и користи, а може бити допуњен формалним или полуформалним описима у виду текста или дијаграма. Квалитет спецификације софтверских захтева у SRS документу се описује следећим карактеристикама:

- **Коректност** (*correctness*). SRS документ је коректан ако се сви наведени захтеви могу испунити. Не постоји јединствен "рецепт" за остваривање коректности документа, али се коректност може проверити разним методама верификације и валидације захтева.
- **Недвосмисленост** (*unambiguousness*). SRS документ је недвосмислен ако се сви наведени захтеви могу интерпретирати на само један начин при чему се користи јединствена терминологија. Двосмисленост се најчешће јавља због употребе природног језика приликом спецификације захтева, али се може смањити употребом формалних језика за специфицирање и одговарајућих софтверских алата.
- **Комплетност** (*completeness*). SRS документ је комплетан ако садржи све битне захтеве без обзира да ли су функционални или нефункционални, опис функционисања софтвера за све могуће врсте улазних података (коректне и погрешне), дефиниције свих коришћених термина и референце на све табеле слике и дијаграме који се појављују у тексту документа.
- **Конзистентност** (*consistency*). Конзистентност се односи на интерну конзистентност документа и огледа се у усаглашености са осталим документима на вишем нивоу у пројекту и организацији која производи софтвер. Конзистентан документ не садржи конфликте, који могу бити: конфликти карактеристика објеката у домену проблема, логички конфликти између појединих функционалности које реализује софтвер (нпр. редослед акција у реализацији функционалности) и случај када више захтева описује исти објекат из домена али користе различиту терминологију.
- **Приоритизација** (*prioritization*). Приоритизација захтева се врши на основу њиховог значаја и стабилности. Немају сви захтеви исти значај за функционисање софтвера, па су тако неки захтеви есенцијални док су други само пожељни, па се приоритизација врши на принципу значаја за функционисање софтвера. Приоритизација обезбеђује ефикаснију организацију софтверских инжењера на пословима развоја софтвера.
- **Могућност верификације** (*verifiability*). SRS документ је могуће верификовати само ако се могу верификовати сви наведени захтеви. Захтев се може верификовати ако постоји коначан процес процене да

ли је захтев коректно специфициран, што се може радити ручно или аутоматски помоћу наменских софтвера.

- **Могућност измене** (*modifiability*). SRS документ се може лако мењати (одржавати) ако се измене на свим захтевима могу извршити лако, потпуно и конзистентно а да се очува структура и стил документа. Услови које треба да испуни документ су: кохерентна структура са садржајем, нема понављања садржаја и сваки захтев се специфицира независно од других.
- **Следљивост** (*traceability*). SRS документ подржава следљивост ако се сваки захтев може пратити од његовог извора па кроз све фазе развоја софтвера и израде документације. Постоје два типа следљивости: (1) **следљивост унапред** (*Forward traceability*) - за сваки захтев се могу идентификовати објекти и документи у наредним фазама развоја и (2) **следљивост уназад** (*Backward traceability*) - сваки захтев има референце на документе и објекте из претходних фаза развоја, па све до извора захтева.

Креирани документ са спецификацијом софтверских захтева омогућује ригорозну проверу свих захтева пре него што се крене са детаљним техничким дизајном софтвера, а исто тако може значајно смањити потребу за изменом дизајна или кода. Поред тога спецификација захтева представља основу за управљање процесом захтева и касније целим пројектом развоја софтвера пошто обезбеђује реалну основу за процењивање трошкова, ризика и распореда послова.

Најзначајнији фактор који доприносе квалитету спецификације захтева јесу добро написани захтеви без двосмислености и понављања. Ако је опис захтева издељен у више докумената, то негативно утиче на квалитет. Фактори који индиректно утичу на квалитет спецификације софтверских захтева су неодговарајуће искуство тима, доступност клијената, организациони фактори и закаснела валидација. Спољни фактори, као што су већ постојећи уговори између заинтересованих страна у пројекту развоја софтвера, постојећа законска регулатива, као и стандарди и прописи у домену пословања, такође утичу на квалитет спецификације софтверских захтева

2.1.4 Валидација софтверских захтева

Валидација и верификација се често мешају као технике које се примењују у различитим фазама животног циклуса софтвера иако су то суштински две различите активности. Верификација се спроводи са циљем да се утврди да ли неки производ настао током развоја софтвера испуњава претходно постављене захтеве (нпр. да ли је функционалност у софтверу реализована у складу са захтевом описаним у SRS документу). Валидација се спроводи да би се утврдило да ли производ настао у току развоја софтвера (нпр. SRS документ, софтверски модул, веб сервис) задовољава потребе корисника исказане у корисничким захтевима.

Валидација софтверских захтева, тј. документа са спецификацијом софтверских захтева, треба да обезбеди да су софтверски инжењери

разумели захтеве корисника и да ће пројектовани софтвер бити у складу са њиховим потребама. Валидација се може радити неколико пута у процесу софтверских захтева, а у складу са потребама пројекта подразумева активно учешће свих релевантних актера. Врло често се валидација преклапа са активностима анализе и специфицирања захтева пошто треба да обезбеди да се идентификују сви проблеми у спецификацији. Валидација обезбеђује да су захтеви специфицирани коректно и да имају жељене карактеристике које су у складу са потребама свих заинтересованих страна. У пракси се може десити да документ са спецификацијом захтева испуњава све перформансе приликом прегледања, али приликом конструкције софтвера се могу открити проблеми који нарушавају жељене перформансе софтвера. У таквим случајевима се врши поновна анализа и измена спецификације захтева у SRS документу.

Валидација захтева је веома важна пошто проблеми који нису откривени могу проузроковати значајне трошкове због накнадних послова у наредним фазама развоја софтвера. У пракси су трошкови измене захтева када је већ започето кодирање софтвера много већи него када се измене врше током фазе израде спецификације захтева. Провере које се врше на захтевим у SRS документу су:

- **Провера валидности**. Различити актери имају свој интерес од софтвера који се пројектује, па је неопходно проверити да ли су њихови захтеви и очекивања испуњени у смислу функционалности и перформанси које пружа софтвер.
- **Провера конзистентности**. Проверава се да ли постоје конфликти између различитих захтева, као и да ли постоје контрадикторна ограничења и описи за сваки од захтева.
- **Провера комплетности**. Проверава се да ли документ са захтевима садржи описе свих захтева и ограничења за софтвер који се пројектује.
- **Провера реалности**. Проверава се да ли се захтеви могу имплементирати у оквиру софтвера у задатим роковима ако се сагледају ограничења и могућности технологија које се користе, буџет пројекта и распоред послова приликом развоја.
- **Верификација**. Захтеви се пишу тако да се могу верификовати, тј. да се може проверити њихова усклађеност са претходним документима и да се могу искористити у дизајну, конструкцији и тестирању софтвера.

Валидација софтверских захтева уноси додатне трошкове у пројекат развоја софтвера и захтева одређено време да се спроведе, али ти трошкови и утрошено време на крају пројекта имају оправдање кроз смањене трошкове у осталим фазама развоја софтвера, повећан квалитет производа и задовољење потреба клијената. Иако инжењери нерадо одвајају време за послове верификације и валидације пошто сматрају да то уноси кашњење у реализацији пројекта без повраћаја уложених средстава, у реалности се ове активности вишеструко исплате и значајно смањују додатни рад на корекцијама у наредним фазама развоја софтвера. Валидација софтверских захтева се може вршити применом различитих техника које се могу користити појединачно или заједно:

- **Прегледање захтева**. Прегледање (рецензирање) документа са захтевима је најчешћи начин валидације захтева. Преглед најчешће врши неколико особа које траже грешке и неконзистентности у спецификацији. Преглед може бити неформалан, што укључује тражење крупних грешака и неконзистентности, али се озбиљнији проблеми (нпр. контрадикције између различитих захтева) не идентификују. Формални прегледи користе јасно дефинисан и систематичан протокол прегледа и на крају се креира документ о извршеном прегледу. У пракси се најчешће користи техника инспекције захтева која подразумева одржавање састанака на којима заједно раде људи који су израдили документ, људи који су били извор информација за захтеве, људи који ће конструисати софтвер и људи који ће користити софтвер. Инспекција се базира на ад-хок читању захтева или на читању према контролној листи, при чему су за сам процес инспекције битне техничка, економска и организациона димензија, као и димензија софтверских алата који се користе у процесу.

- **Израда прототипова**. Израда прототипова на основу спецификације захтева је уобичајени начин да се визуализује функционисање софтвера, што је немогуће постићи само прегледом спецификације захтева. Прототипови омогућују да корисници веома брзо добију реално искуство са софтвером који се пројектује на основу њихових захтева чиме се обезбеђује потпуно укључење корисника у процес развоја софтвера. Прототипови такође позитивно утичу на смањење неразумевања између софтверских инжењера и корисника софтвера у току развоја, а посебно у фази израде спецификације захтева. Примена прототипова такође омогућује да се прикупе додатни захтеви или да се изврши корекција постојећих. Израда прототипова поскупљује процес пројектовања софтвера, али прототипови могу еволуирати у софтвер који ће бити испоручен клијентима.

- **Валидација модела**. Анализа и специфицирање захтева као једну од метода користе израду модела софтверских захтева. Креиране моделе је потребно прегледати и валидирати. Код објектних модела се врши статичка анализа да би се верификовале везе између објеката у моделу, а код формалних модела се може користити формално резоновање да би се утврдиле карактеристике спецификације захтева. Најчешће се за представљање информација у спецификацији захтева користе визуелни модели који омогућују да се идентификују информације које недостају и да се боље разуме контекст проблема. Модел мора користити једноставну нотацију да би могли да га разумеју корисници који се укључују у специфицирање и валидацију захтева.

- **Тестови прихватања**. Креирање тестова прихватања је уобичајени начин како крајњи корисници валидирају имплементацију њихових захтева у спецификацији захтева и у крајњем софтверском производу. Валидација помоћу тестова прихватања се најчешће спроводи за функционалне захтеве, док је много сложеније тестирати нефункционалне захтеве за које је најпре потребно дефинисати одговарајуће метрике. За валидацију функционалних захтева се

најчешће генеришу тестови што значајно повећава ефикасност и смањује трошкове развоја софтвера.

2.2 Спецификација софтверских захтева

Документ са спецификацијом софтверских захтева (*Software Requirements Specification (SRS)*), који се често назива и **Спецификација софтверских захтева**, је документ у којем је званично и формално наведено шта све треба имплементирати у софтверу. Овај документ садржи и корисничке захтеве и детаљну спецификацију системских захтева. Структура документа са спецификацијом захтева је предложена стандардом *IEEE Std. 830-1998 (Revision 2009) Recommended Practice for Software Requirements Specifications*. Употреба документа са спецификацијом софтверских захтева обезбеђује:

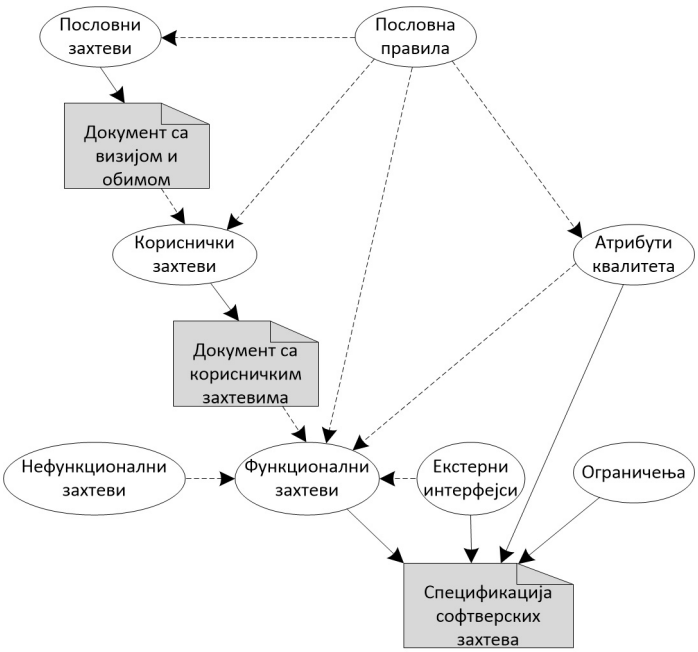
- **Успостављање основе за постизање сагласности између клијената и испоручилаца софтвера**. Клијенти и корисници могу на основу документа да утврде да ли спецификација софтвера испуњава све њихове захтеве или је треба модификовати.
- **Смањење напора и трошкова развоја софтвера**. Клијенти и корисници могу детаљно и ригорозно да прегледају све захтеве пре него што се почне са дизајном софтвера, чиме се смањује потреба за изменама дизајна, изменама кода и додатним тестирањем током развоја софтвера. Детаљни преглед SRS документа омогућује откривање недостатака, неконзистентности и неразумевања између свих заинтересованих страна у фази развоја када је то најједноставније и најјефтиније.
- **Дефинисање прецизне основе за процену трошкова и распореда послова у развоју софтвера**. Документ је основа за реалну процену трошкова развоја софтвера и као такав се може користити за одобравање трошкова развоја од стране клијената. Такође, може послужити за детаљну разраду потребних послова у пројектовању и конструкцији софтвера према договореној спецификацији.
- **Креирање полазне основе за валидацију и верификацију**. Планови за верификацију и валидацију се могу прецизније и ефикасније дефинисати на основу SRS документа, где се јасно дефинише шта се мери и проверава и у односу на које полазне основе.
- **Подршка за једноставнију испоруку софтвера корисницима**. Јасно дифинисана спецификација захтева указује којим корисницима и на који начин треба испоручити софтвер (на који начин софтвер треба да буде доступан). Спецификација захтева се може искористити и за реализацију обуке корисника након испоруке, тако што се за сваког корисника припрема обука за функционалности које су њему доступне на основу улоге која му је додељена.

- **Формирање основе за побољшања софтвера**. Пошто SRS документ дефинише спецификацију производа, он представља основу за будућа побољшања производа.

Предложена структура SRS документа према стандарду *IEEE Std. 830-1998 (Revision 2009) Recommended Practice for Software Requirements Specifications* треба да садржи следеће елементе:

- **Предговор**. Дефинише се намена документа и циљне групе којима је документ намењен.
- **Увод**. Дефинише се назив софтвера, његова намена и употреба у оквиру специфичног домена употребе, као и интеграција у сложеније системе.
- **Дефиниције, акроними и скраћенице**. Дефинишу се сви појмови, акроними и скраћенице неопходни за разумевање садржине документа. Овде се могу навести и референце на друге документе који се могу користити у ту сврху.
- **Дефиниција корисничких захтева**. Садржи опис корисничких захтева употребом терминологије која је карактеристична за домен употребе софтвера. Ако се приликом спецификације корисничких захтева користе други документи или стандарди њих треба навести.
- **Архитектура система**. Садржи приказ система на високом нивоу, са прегледом распореда функционалности у оквиру подсистема, модула и компоненти.
- **Интерфејси система**. Детаљан опис корисничког интерфејса софтвера, интерфејса према другим софтверским системима који се користе у окружењу и интерфејса софтвера према хардверским компонентама.
- **Спецификација софтверских захтева**. Садржи детаљан опис функционалних и нефункционалних захтева. Функционални захтеви морају бити специфицирани и организовани на логичан начин.
- **Оганичења система**. Систематизација и детаљан опис свих ограничења која се односе на софтверски производ, окружење у којем се користи (пословно окружење, законска регулатива и стандарди у домену примене) и процес развоја софтвера (методе, технологије, платформе).
- **Еволуција система**. Опис антиципираних промена система због промене корисничких захтева, хардверске платформе или промена у законској регулативи која уређује домен у којем се софтвер користи.
- **Додаци**. Детаљни описи специфичних елемената система, као што су хардверске компоненте, сервери, базе података.

Информације које се укључују у SRS документ зависе од типа софтвера који се пројектује (уграђени системи, веб апликације, пословни софтвер базиран на структури података итд.) и приступа који се користи у развоју софтвера (традиционалне или агилне методе). Због тога се структура и организација SRS документа прилагођава сваком специфичном софтверском пројекту.



Слика 2.4: *Информације које се користе у спецификацији софтверских захтева*

Софтверски захтеви укључују пословне захтеве, корисничке захтеве, функционалне захтеве и нефункционалне захтеве, што је приказано на слици 2.4, где пуне линије са стрелицама означавају припадање, док испрекидане линије са стрелицама означавају утицај између елемената на слици. Документи који садрже информације о софтверским захтевима могу бити електронски или традиционални штампани документи у којима се налазе подаци у виду текста, табела, дијаграма и слика. **Пословни захтеви** се односе на опис циљева организације за коју се производи софтвер, при чему су ти циљеви описани на високом нивоу, тј. у домену пословања организације. **Пословна правила** су правила, водичи, стандарди или регулативе које дефинишу ограничења у пословању, а могу бити извор за више функционалних и нефункционалних софтверских захтева. **Атрибути квалитета** представљају нефункционалне захтеве који описују перформансе софтверског производа. **Кориснички захтеви** се односе на опис циљева или задатака које одређене групе корисника треба да обављају помоћу софтверског производа, а могу се специфицирати у посебном документу (**Документ са корисничким захтевима**) или могу бити део документа са спецификацијом софтвера. **Ограничења** се односе ограничења која утичу на изборе у дизајну и конструкцији софтверског производа. **Екстерни интерфејси** описују везу између софтверског система који се пројектује са корисницима, другим софтверским системима или хардверским компонентама.

Скуп софтверских захтева (*set of requirements*) представља решење проблема помоћу групе софтверских захтева, при чему сваки укључени

захтев има своју специфичну улогу у групи. Сваки скуп захтева треба да има следеће карактеристике:

- **Комплетност**. Скуп захтева садржи све што је потребно за имплементацију система или дела система.
- **Конзистентност**. Не постоји контрадикција између индивидуалних затева у посматраном скуп захтева. Такође нема дуплирања захтева и подразумева се употреба истих термина у дефинисању и опису свих захтева у скупу.
- **Доступност**. Софтверско решење обухвата све захтеве из посматраног скупа захтева, без обзира на ограничења у вези софтверског производа или процеса развоја софтвера.
- **Ограниченост**. Скуп захтева јасно одређује опсег предложеног софтверског решења и при томе садржи све потребно да се задовоље потребе корисника.

2.3 Корисничке приче

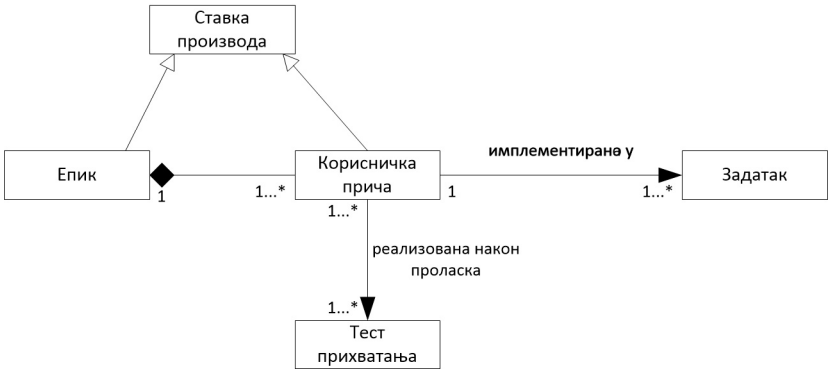
SRS документи се користи у пројектима код којих се користе традиционални модели развоја софтвера, док се код агилних метода захтеви често мењају па је креирање SRS документа непрактично пошто он не одсликава право стање захтева. Због тога се код агилних метода развоја софтвера кориснички захтеви често евидентирају на картицама као **корисничке приче** (*user stories*) које се приоритизирају за следећу фазу у инкременталном развоју софтвера. Приче корисника се често користе за презентовање корисничких захтева и односе се на мале делове система које треба имплементирати, а користе се за организовање рада софтверских инжењера на дневној бази. Приче корисника треба да буду интуитивне и разумљиве, што побољшава процес прикупљања информација и структурирања послова. Најједноставнија интуитивна дефиниција корисничке приче би била:

> *Корисничка прича је кратак и јасан захтев или навод шта систем треба да омогући кориснику.*

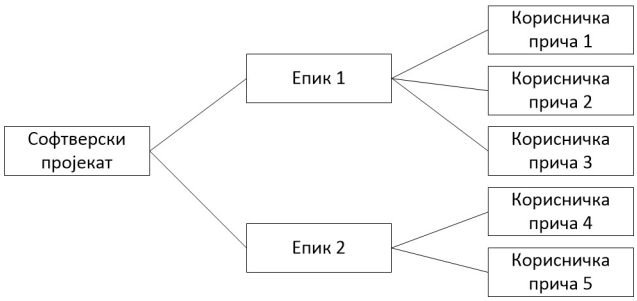
Инкрементални приступ развоју софтвера је базиран на сталној испоруци вредности корисницима кроз додавање нових функционалности у софтвер. Корисничка прича је део агилног приступа која служи за дефинисање софтверских захтева који се реализују као ставке производа и улазе у листу ставки производа. Улога корисничке приче у агилном развоју софтвера је приказана на слици 2.5.

Свака корисничка прича се током реализације може имплементирати кроз један или више задатака, а мора се завршити у оквиру једне итерације у развоју. Корисничка прича се сматра комплетираном када прође један или више тестова прихватања.

Епик (*Epic*) је сложенији део посла који се не може реализовати у оквиру једне итерације, а обухвата више међусобно повезаних корисничких прича.



Слика 2.5: *Корисничка прича као део агилног развоја софтвера*

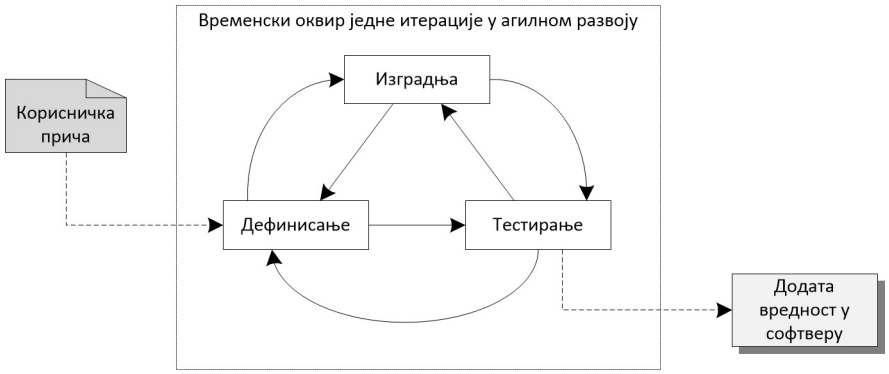


Слика 2.6: *Организација агилног развоја са више епика и корисничких прича*

Величина епика зависи од природе проблема и од начина организације рада у софтверској фирми. Примери епика у развоју софтвера су: потврда идентитета корисника (user authentication), евиденција робе у магацину (goods recording in a warehouse) или наручивање робе преко интернета (ordering goods online). Организација софтверског пројекта са више епика и корисничких прича је приказана на слици 2.6.

2.3.1 Животни циклус корисничке приче

Основна јединица рада у агилном развоју је корисничка прича, која се имплементира кроз реализацију неколико придружених послова. У оквиру једне итерације у агилном развоју софтвера циљ је да се неколико одабраних корисничких прича реализују тако да то представља додатну вредност у следећем издању софтвера. Свака корисничка прича се реализује кроз итеративни циклус који укључује дефинисање приче (спецификација и дизајн), имплементацију (изградња софтверског елемента) и тестирање, као што је приказано на слици 2.7. Приказани циклус се за сваку причу понавља више пута током једне итерације (на пример, спринта у скрам методологији), али се може понављати више пута и у току једног дана. По завршетку циклуса корисничке приче, мора бити креирана и испоручена додата вредност за софтверски производ.



Слика 2.7: *Животни циклус корисничке приче*

Активности дефинисања, имплементације и тестирања се могу понављати и извршавати више пута паралелно током једне итерације у развоју софтвера (на пример, током једног спринта у скрам методологији). Свака комплетирана корисничка прича остаје интегрисана у софтверу који се испоручује корисницима. Циклус корисничке приче је:

- **Конкурентан** (*concurrent*). Све активности се могу извршавати истовремено, што значи да се током тестирања приче може радити и на редефинисању приче, а истовремено и на кодирању да се редефинисана прича прилагоди новим захтевима.
- **Колаборативан** (*collaborative*). На имплементацији приче ради тим који се састоји од чланова који раде на различитим задацима. С обзиром на конкурентност процеса, тим мора сарађивати током свих активности.
- **Атомичан** (*atomic*). Након реализације циклуса, прича ће бити или реализована у потпуности или ће бити означена као нереализована па ће се разматрати за следећу итерацију. Није прихватљиво да се прича делимично реализује, тј. да се само неки од послова који су асоцирани причи реализују.

2.3.2 Формат корисничке приче

Корисничке приче су кратки записи о својству које је потребно кориснику, а њихове основне карактеристике су:

- Кратке су и написане су тако да их лако разумеју и програмери и корисници и сви остали учесници у процесу.
- Представљају мале инкременте функционалности која носи вредност кориснику, а њихова имплементација је обично у периоду од једне до неколико радних недеља.
- Лако се процењују време и напор потребни за реализацију.
- Лако се организују у листе које се могу преуредити ако се појаве нове информације.

- Не садрже детаљну спецификацију, која се ради тек када се крене са имплементацијом приче.

Основни елементи корисничке приче су:

- **Картица** (*Card*) - садржи две до три реченице које укратко износе суштину или намеру причу.
- **Конверзација** (*Conversation*) - садржи детаље о дискусији о корисничкој причи између развојног тима, клијента, власника производа и осталих заинересованих особа. На основу садржаја конверзације се могу сагледати детаљи функционалности које треба имплементаирати. Додатни детаљи се често прикупљају као документи који се придружују картици корисничке приче, а то могу бити алгоритми, табеле (*spreadsheet*), модели и макете (*mock-up*), слике, итд.
- **Потврда** (*Confirmation*) - представља тест прихватања (*acceptance test*) који описује како клијент или власник производа утврђује да је корисничка прича имплементирана тако да задовољи корисничке захтеве. Критеријуми за прихватање корисничке приче нису функционални (*functional*) или јединични (*unit*) тестови, већ представљају услове и ограничења која се постављају за функционисање система.

Опште прихваћени формат за писање корисничких прича укључује улогу корисника, активност коју жели да изврши и пословну вредност коју жели да оствари реализацијом активности. Модел корисничке приче је

> Као <улога>, ја могу <активност> тако да <пословна вредност>.

Овако написана корисничка прича одражава "глас корисника" (*user voice*), дефинисана форма корисничке приче јасно обухвата простор проблема кроз пословну вредност која се испоручује (<пословна вредност>), али и простор решења проблема (<активност>) који се односи на акцију коју корисник реализује на систему. Примери корисничких прича су:

> Као магационер (<улога>), ја могу да видим све дневне промене стања одабране робе (<активност>), тако да фирма увек зна на шта може рачунати у производњи (<пословна вредност>).
>
> Као руководилац производње (<улога>), ја могу да пратим тренутно стање опреме у погону (<активност>), тако да увек могу да откријем да ли постоји проблем који би довео до застоја производње (<пословна вредност>).
>
> Као клијент банке (<улога>), ја могу да вршим плаћање рачуна онлајн (<активност>), тако да то могу урадити било када и са било ког места без губитка времена због одласка у банку (<пословна вредност>).

Агилни тимови троше значајно време у откривању, елаборацији и разумевању корисничких прича, као и на писању тестова прихватања. Искуства из индустрије указују да је значајно тежи посао разумети потребе корисника и формулисати јасне захтеве, него писати код који представља

софтверско решење за проблем. Због тога је веома важно уложити време и напор у креирање квалитетних корисничких прича, што је одсликано у принципу *INVEST*, који описује добре атрибуте корисничких прича:

- **Независна** (***I**ndependent*). Свака корисничка прича може се имплементирати, тестирати и испоручити самостално. То подразумева да се може и вредновати независно од других прича. Ако постоји зависност између корисничких прича, треба сагледати да ли се приче могу преформулисати или груписати на одговарајући начин да се та зависност елиминише.
- **Предмет је договора** (***N**egotiable*). Корисничка приче не представља стриктну спецификацију функционалности већ флексибилне захтеве о којима се може дискутовати пре и у току имплементације и тестирања. Флексибилност корисничких прича омогућује развојном тиму да оствари циљеве испоруке софтвера, што позитивно утиче на међусобно поверење укључених страна.
- **Има вредност** (***V**aluable*). Циљ агилног развоја је да се испоручи што више вредности у задатим временским оквирима и са постојећим ограничењима. Због тога је ово најважнији атрибут корисничке приче. Свака корисничка прича мора обезбедити исту вредност за клијента, кориснике и све заинтересоване стране. У листи ставки производа (*product backlog*) све корисничке приче се приоритизује према процењеној вредности, а приче које имају највећу вредност се прослеђују на имплементацију у следећој итерацији.
- **Процењива** (***E**stimable*). Корисничка прича треба да буде тако дефинисана да се лако може проценити обим посла и временски оквир за имплементацију. Приликом процењивања, прича се сматра добро дефинисаном ако се може имплементирати у оквиру једне итерације. Ако тим не може да процени причу то је индикатор да је прича превише велика или неодређена. Велике приче треба поделити на више мањих прича, а неодређене приче треба поново преиспитати.
- **Мала** (***S**mall*). Корисничке приче треба да буду довољно мале да се могу комплетирати у оквиру једне итерације, што обезбеђује вредност за корисника. Мале корисничке приче подржавају агилност и продуктивност развојних тимова. Комплексније приче треба разделити у више мањих да би се обезбедило комплетирање сваке мање приче у оквиру једне итерације.
- **Лако се тестира** (***T**estable*). Основна карактеристика агилног кода је да је тестиран, што подразумева да се свака корисничка прича може лако тестирати. Ако се прича не може тестирати, тада она није добро формирана, превише је комплексна, или зависи од других прича. Основно правило приликом писања корисничких прича је да се избегавају неодређене речи (брзо, лепо, јасно, управљиво, итд.), које свако може протумачити на други начин.

2.4 Учесници у процесу софтверских захтева

У процес софтверских захтева су укључени различити учесници с обзиром да област софтверских захтева обухвата домен примене софтвера (специфична област пословања или људске делатности), домен који уређује пословање или област људске делатности (правна регулатива, тржиште, стандарди) и домен софтверског инжењерства који обезбеђује методе и алате за изградњу софтвера. Процес софтверских захтева, а посебно фаза прикупљања захтева, се базира на сарадњи свих учесника (заинтересованих страна) при чему се креирају сложени односи међу учесницима, што као последицу има крајње сложен и интерактиван процес софтверских захтева Учесници, или заинтересоване стране, у процесу софтверских захтева су:

- **Клијенти**. То су особе или организације који плаћају развој софтвера, због чега имају пресудну улогу у валидацији и прихватању софтверских захтева. Одређене групе клијената могу купити софтверски производ и након што је потпуно креиран, а да претходно нису учествовале у спецификацији софтверских захтева.
- **Корисници**. То су особе које користе софтверски производ и које најбоље сагледавају све потребне функционалности и карактеристике које треба да садржи софтвер.
- **Доменски експерти**. То су особе које познају домен проблема за који се креира софтвер, као и окружење у којем ће се софтвер користити.
- **Истраживачи тржишта**. То су особе које истражује трендове тржишта и потребе потенцијалних клијената и корисника.
- **Регулатива**. То је скуп законских регулатива, стандарда и других јавних аката који уређују одређену област људске делатности у којој ће се користити софтвер.
- **Софтверски инжењери**. То су особе које креирају софтвер у складу са спецификацијом софтверских захтева. Поред техничких аспеката креирња софтвера они су задужени и за процену трошкова, ризика и осталих ресурса неопходних за креирање софтвера.

Сваки учесник у процесу захтева има свој специфичан поглед на процес захтева као и на саме софтверске захтеве. Такође, сваки учесник очекује одређени ниво детаља у спецификацији захтева у складу са својом улогом, што значи да се софтверски захтеви морају представити на различите начине за различите учеснике у процесу. У пракси је врло често тешко дефинисати јасну границу између начина прдстављања захтева различитим учесницима, али се грубо могу издиференцирати два нивоа представљања захтева:

- **Кориснички захтеви** представљају апстракцију захтева на високом нивоу и обично су изражени језиком домена проблема. Кориснички захтеви се дефинишу тако да су разумљиви крајњим корисницима, менаџерима из клијентских организација и архитектама софтверског система.
- **Системски захтеви** представљају детаљну спецификацију шта систем треба да ради и које карактеристике треба да има. Системски захтеви

се дефинишу тако да су разумљиви крајњим корисницима и свим софтверским експертима који су укључени у креирање софтвера (архитекте, програмери, тестери).

С обзиром на сложеност домена проблема за који се развија софтвер и сложеност процеса развоја софтвера, област инжењеринга софтверских захтева је интердисциплинарна и захтева ангажовање стручњака који имају широк скуп не само техничких већ и нетехничких вештина (*soft skills*) и знања. С обзирома да се техничке вештине као што су програмирање, управљање базом података или конфигурисање софтвера и хардвера могу развити током рада или кроз одговарајуће тренинге, постаје очигледно на основу истраживања у индустрији да је поседовање и унапређење нетехничких вештина изузетно важно за успешног софтверског инжењера. Врло је важно да су софтверски инжењери који су ангажовани у процесу софтверских захтева свесни постојања нетехничких вештина, да их разумеју и да су спремни да их усавршавају кроз различите курсеве или радионице.

Поред разумевања људског фактора који утиче на процес софтверских захтева, неопходно је разумети и контекст у којем ће софтвер бити коришћен, што подразумева сагледавање искустава из претходних софтверских пројеката и укључивање постојећег знања или експерата из домена проблема или из друштвених наука (социолози, психолози, лингвистичари итд.). Ако се у процес софтверских захтева укључе експерти са различитим доменима експертизе потребно је обезбедити механизме и алате који ће умањити њихово неразумевање и позитивно утицати на реализацију пројекта.

У циљу постизања свеобухватног разумевања стручности и компетенција софтверских инжењера ангажованих на пословима прикупљања, анализирања и специфицирања софтверских захтева креиран је модел компетенција које треба да поседују софтверски инжењери. Модел је резултат квалитативног истраживања у софтверској индустрији и садржи следећих 16 компетенција: консултовање са другима, тестирање претпоставки и истраживање, објашњавање концепата и мишљења, систематичан рад, вођење пројекта ка постављеним циљевима, прикупљање информација, фокусираност на потребе и задовољство клијената, примена техничке стручности, креирање решења, самопоуздано деловање, вођење конверзације, деловање, усмеравање комуникације, постизање договора, анализирање и оцењивање информација и координирање и усмеравање активности

Поглавље 3

Еволуција и одржавање софтвера

Промена се може појавити у било ком тренутку животног циклуса софтверског система. У пракси, систем ће се мењати без обзира на тренутак у животном циклусу, и потреба за његовом променом ће стално постојати. Један од основних разлога за то је што се софтвер може *лакше* променити него неки други објекти који чине техничке системе (нпр. хардверске компоненте или грађевински елементи)[*The Mythical Man-Month: Essays on Software Engineering*].

Разумевање односа између организације, техничких система и правила пословања доприноси ефикасном управљању променама у организацијама. Због значаја софтверских производа у функционисању организационих система и пословних процеса, еволуција и одржавање софтвера могу обезбедити "поуздане путање" за реализацију промена у оквиру организација. Софтверски производи одсликавају организационо и индивидуално понашање, што се одражава на рапидни раст функционалности, понашања, величине и структуралне комплексности софтверских производа. То резултира неизбежним потребама за променама. Поуздано, економски исплативо и довољно брзо управљање променама захтева свеобухватан увид у карактеристике софтверског производа и његовог радног окружења од стране и корисника и софтверских инжењера. Због динамике пословања, данас је практично немогуће специфицирати све софтверске захтеве унапред, а као резултат таквог стања софтвер се често мења и прилагођава, чиме се заправо појављује *еволуција софтверског система*.

Перформансе, функционалне могућности и квалитет се не могу у потпуности обезбедити и уградити у софтверски производ током пројектовања, већ се оне постепено остварују кроз еволутивне промене. Основни покретачи еволуције, тј. континуираних промена софтвера, се могу поделити у две групе: (1) фактори тржишта или разни друштвени феномени, и (2) техничко-технолочки фактори. Разматрањем ових фактора се сагледава

пресудан утицај људског фактора на еволуцију софтвера, што се може исказати следећим тезама:

> ***Теза 1:*** *Људски ставови и разматрања су основни спољашњи покретачки фактори еволуције софтвера*
>
> ***Теза 2:*** *Људска (и друштвена) ограничења у процесирању информација (прикупљање и структурирање) постављају ограничења у еволуцији софтвера.*

Еволуција и одржавање софтвера се могу посматрати као низ сукцесивних индивидуалних промена које се реализују на софтверском производу током његовог животног циклуса. Свака индивидуална промена подразумева постојање захтева за променом (*software change request*), рализацију посла промене, и скуп модификација на компонентама система.

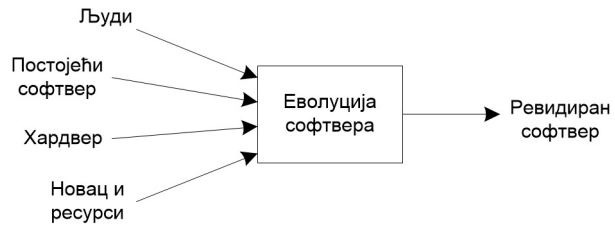
3.1 Основни принципи еволуције софтвера

Еволуција је феномен који се појављује у многим областима људског живота (друштво, теорије, идеје, ...). Еволуција се односи на прогресивне, а пожељно и корисне, промене атрибута ентитета који се мења, тј. еволуира. Оно што се сматра прогресивним зависи од домена и контекста где се еволуција посматра. Еволуција неког ентитета обично као последицу има више промена током животног века, при чему својства ентитета која више не одговарају нестају (ишчезавају), а нова својства настају (дефинисана новим потребама).

Концепт еволуције софтвера је препознао и дефинисао *Meir "Manny" Lehman* 60-тих година прошлог века. Еволуција софтвера је у почетку посматрана као континуирани раст програма који се односио на унапређење функционалних карактеристика, а огледао се у повећању броја програмских модула, линија кода или захтева за складиштењем. Данас је еволуција софтвера нешто што је незаобилазно у животном веку софтвера и не односи се само на недостатке у развоју софтвера. Појам еволуција софтвера се односи на континуиране промене својстава или карактеристика које воде до стварања нових својстава или унапређења постојећих, а одсликава динамику одржавања софтвера током његовог животног века.

Међутим, у софтверском инжењерству еволуција није карактеристика само софтверских система већ и одговарајућих ентитета као што су спецификација, дизајн, компоненте или документација. Појам еволуције се може применити и на парадигме, алгоритме, језике, процесе и под-процесе, циљеве, и практичну употребу. Два основна приступа у проучавању еволуције софтвера су:

- **Објашњавање**. Односи се на разумевање узрока, процеса и ефеката. Овај приступ је свеобухватан и разматра утицај еволуције софтвера на ефикасност организације и планирање организационих промена.
- **Побољшавање**. Односи се на развој бољих метода и алата као подршке еволуцији, а узима у обзир активности софтверског инжењерства као што су дизајн, одржавање, реинжењеринг итд.



Слика 3.1: *Фактори који утичу на еволуцију софтверских система*

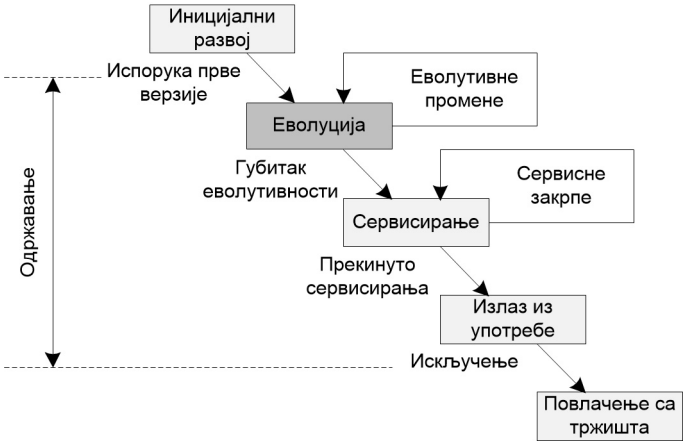
Сложеност софтверских система и социо-техничких окружења у којима се користе утичу на еволуцију софтвера, што је приказано на слици 3.1. Људски фактор има пресудну улогу у креирању и употреби софтверских система, па према томе значајно утиче и на еволутивне промене софтвера. Корисници софтвера пријављују захтеве за модификацијом софтвера, а софтверски инжењери реализују модификације на софтверу при чему настаје нова верзија софтвера (ревидиран софтвер). Када се разматра утицај људског фактора треба сагледати број људи који је укључен у развој софтвера, али и број људи који су корисници софтвера. Такође је потребно узети у обзир разноликост улога које имају људи у еволуцији софтвера. Софтверски инжењери могу радити послове на спецификацији модификације, имплементирању модификације, тестирању и испоруци ревидираног софтвера. Такође корисници могу имати увид само у ограничен скуп функционалности софтвера, па треба сагледати на које групе корисника може утицати имплементирана модификација у ревидираном софтверу.

Такође значајну улогу имају и ресурси које је потребно обезбедити да се реализују све модификације софтвера, а најважнији фактор јесте новац. Еволуције подразумева постојање софтверског система који се мења али зависи и од хардвера који је неопходан за извршавање софтвера (рачунари, комуникациона опрема, остала опрема која комуницира са софтвером).

Еволуција софтвера је специфична за сваки софтверски систем, али зависи и од контекста организације која производи софтвер, од клијената (организације које користе софтвер) и корисника софтвера, али и од динамике пословања организације која користи софтвер и стања и потреба тржишта.

3.1.1 Модели процеса еволуције софтвера

Процес еволуције софтвера је скуп међусобно повезаних процеса који омогућују континуирану промену софтверских система. Процес еволуције подразумева употребу одговарјућих методологија, технологија, алата, техника управљања и документације. Процес еволуције софтвера се најчешће односи на процес модификације софтвера након његове иницијалне испоруке кориснику, тј. у фази одржавања. То подразумева да је еволуцију потребно посматрати као део животног циклуса софтвера. Процес еволуције софтвера у фази одржавања има много сличности са процесом развоја (нпр. израда модела, кодирање, тестирање). Међутим еволуција софтвера у фази одржавања подразумева додатне активности које нису део процеса развоја



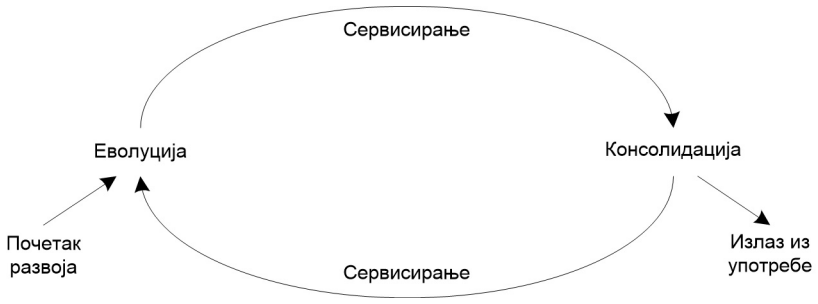
Слика 3.2: *Модел стања животног циклуса софтвера у фази одржавања*

софтвера. Те активности се односе на пријем захтева за променама и њихову анализу и процену (добар део захтева за промена је последица неразумевања документације и функционалности софтвера), техничка анализа трошкова реализације промене (ресурси, време, новац), анализа утицаја промене (*change impact analysis*), регресивно тестирање (*regression test*) тј. тестирање система са имплементираном променом. У литератури се могу пронаћи различити модели процеса еволуције, а сви модели се базирају на пријему захтева за модификацијом, након чега се реализују све активности неопходне за модификацију софтвера.

Животни циклус софтвера представљен помоћу модела стања (*staged model*) фазу одржавања софтвера посматра као низ стања међу којима је једно од стања еволуција, што је представљено на слици 3.2. Овај модел обухвата цео животни циклус, а еволуција је у њему дефинисана као стање у којем се јављају итеративне промене софтвера са циљем да се одговори на захтеве корисника. Фаза одржавања у овом моделу се практично састоји из следећих стања:

- **Еволуција** (*Evolution*). Итеративне промене софтверског система које резултују креирањем нових верзија софтвера.
- **Сервисирање** (*Servicing*). Занемарљиве интервенције на софтверском систему које не резултују креирањем нових верзија софтвера.
- **Излаз из употребе** (*Phaseout*). Софтверска фирма пружа подршку за софтверски производ који се још увек користи, али више не реализује промене на основу захтева корисника.
- **Повлачење** (*Closedown*). Софтверска компаније повлачи производ са тржишта и кориснике упућује на одговарајућу замену.

Основна карактеристика стања еволуције у моделу приказаном на слици 3.2 је итеративно додавање, модификација или уклањање функционалности у софтверу које резултира издавањем нове верзије софтвера. Када систем више није *еволутиван*, тј. када се више не реализују комплексне промене које доводе



Слика 3.3: *Модел стања животног циклуса са наизменичним изменама стања еволуције и консолидације*

до креирања нових верзија софтвера, он прелази у стање сервисирања где се сервисирају једноставнији захтеви, а врло често потом се софтвер повлачи из употребе. Узроци за повлачења софтвера из употребе су драстичне промене у радном окружењу (хардвер, оперативни систем), значајне промене у захтевима корисника или промене на тржишту (потребне су новије и напредније верзије, прелазак у ново окружење као што је на пример *cloud*).

ПСПМ модел процеса софтвера (*Pizka-Seifert Process Model (PSPM)*) омогућује конструктивно планирање животног циклуса софтвера. Овај модел је модификација модела стања, а омогућује да се из стања *сервисирања* врати у стање *еволуције* што је уочено код неких софтвера отвореног кода (*open source software*) или код комерцијалних система као што је *SAP*. Модел, такође, уводи стање *консолидације*, па софтверски систем наизменично прелази из стања еволуције у стање консолидације и обратно кроз стање сервисирања, што је приказано на слици 3.3). Када је софтвер у стању консолидације, тада се на њему не врше никакве промене. Овај модел омогућује да софтвер остане у стању еволуције колико год је то потребно или оправдано са аспекта произвођача софтвера, али и са стране потреба тржишта и корисника софтвера.

Следећа својства процеса еволуције софтвера се могу уочити анализом постојећих модела процеса еволуције:

- **Итеративност**. Многе активности, или скупови активности се понављају и то са већом фреквенцијом него у развоју софтвера. Еволутивне промене се чешће догађају након иницијалне испоруке софтвера.
- **Конкурентност**. Конкурентност има више аспеката, а може се односити на верзије, процесе, активности и послове. Ниво конкурентности је већи него у процесу развоја. На пример, више конкурентних верзија софтвера могу бити коришћене од различитих група клијената.
- **Разноврсност континуираних и неповезаних промена**. У току еволуције промене представљају основни облик понашања, а те промене могу бити различите по природи, као на пример, унапређење постојећих функционалних карактеристика, проширење додавањем нових функционалности или отклањање грешака.

- **Процес управљан повратним информацијама**. Еволуција је управљана повратним информацијама које потичу од корисника или из окружења. Најчешће се активности еволуције покрећу када се за то појави разлог и добију одговарајуће информације од корисника или са тржишта.
- **Радни оквир са више нивоа**. Процес еволуције је интересантан за различите учеснике у животном циклусу софтвера (корисници, софтверски нинжењери, менаџери, анлитичари). Свака од група учесника захтева различити ниво детаља и обима информација, али и укључења у активности животног циклуса.

Анализа карактеристика представљених модела процеса еволуције софтвера указује на следеће:

- Модели дају само генералне описе корака у процесу, док су детаљи или упутства који се односе на саму реализацију корака у датим активностим или контексту изостављени. Сваки модел процеса се прилагођава (кроји) специфичним аспектима софтверске организације која производи софтвер и типу софтверског производа.
- Модели су ограничени у могућностима да интегришу постојеће знање (нпр. експертиза корисника, информације о изворном коду, алате) и ново стечено знање (нпр. искуство стечено у приликом реализације претходних послова одржавања) у оквиру процеса. Због тога је у дизајнирање ових модела неопходно увести принципе управљања знањем и организационог учења.
- Не постоје алати који активно дају подршку корисницима у датом контексту. Подршка корисницима се обезбеђује за сваки софтвер на специфичан начин.

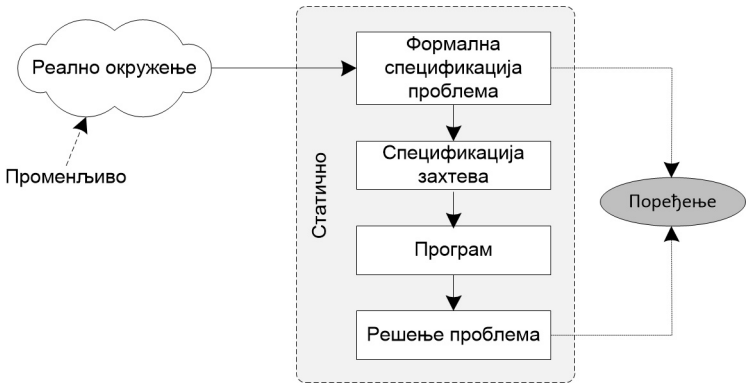
3.1.2 Таксономија еволутивних софтверских система

Концепт еволуције програма (софтвера) је базиран на *Lehman*-овој **СПЕ таксономији** (*SPE Taxonomy*) еволутивних софтверских система. Ова таксономија се односи на *ниво еволутивности* софтвера, тј. на ниво могућности реализације промена у току животног века софтвера. У основи, са аспекта еволутивности, тј. могућности промене софтвера кроз побољшање карактеристика, софтвери се деле на:

- Софтвере који су пројектовани да стриктно решавају одређени проблем, и они се не мењају тј. не еволуирају.
- Софтвере који су пројектовани да решавају проблем у одређеном контексту па се са променом контекста и софтвери морају мењати.

С-тип програма

С-тип програма (*Specified program, S-program*) је програм чије је функционисање изведено из *комплетно дефинисане спецификације*. Дефиниција имплицитно подразумева да је спецификација урађена пре него



Слика 3.4: *Еволутивност С-типа програма*

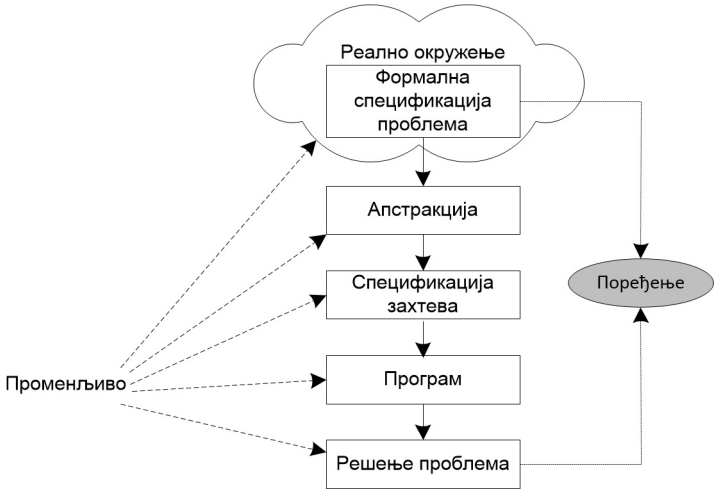
што почне развој (писање кода), тј. да су све функционалности формално специфициране и непроменљиве. То значи да је процес учења и праћења софтвера у току животног циклуса ограничен на предефинисане методе и решења. Спецификација, као формална дефиниција проблема из неког реалног домена, усмерава креирање програма који дефинише решење проблема. Коректност *С-програма* се процењује на основу спецификације која рефлектује проблем који се решава. Промене у радном окружењу током употребе *С-програма* проузрокују да ови програми временом постају неодговарајући за новонастало стање у окружењу, што је приказано на слици 3.4. Промена софтверског решења, која би се извела из постојећег програма, у овом случају подразумева да се добија ново решење са новим скупом функционалности, које је применљиво на други проблем који је различит од првобитног. У пракси то значи да се морају изменити и спецификација и дефиниција проблема да би се имплементирало ново решење. Еволуција *С-програма* се појављује, практично, током иницијалног развоја програма, тачније у фази израде спецификације. Након тога, нема еволуције овог типа програма.

Најчешћи примери *С-програма* су програми који имају стриктно дефинисано рачунање, математичке функције или формално дефинисане трансформације. Примери таквих програма су компајлери или доказивачи.

П-тип програма

П-тип програма (*Problem program, P-program*) је програм који представља практичну апстракцију *проблема* из реалног света, што је приказано на слици 3.5. Проблем који се решава мора бити прецизно дефинисан, док се прихватљивост решења одређује у контексту окружења где се програм користи. Коректност функционисања и валидација *П-програма* се процењују на основу увида у примену решења у реалном контексту.

С обзиром да промене у реалном контексту утичу на промене *П-програма*, овај тип програма временом губи на ефикасности. Истраживања показују да у пракси *П-програми* задовољавају услове или *С-програма* или *Е-програма*. Пример овог типа програма је програм који игра шах.



Слика 3.5: *Еволутивност П-типа програма*

E-тип програма

У већини домена где се софтвер користи *С-програми* и *П-програми* се не могу применити због тога што је практично *врло тешко* креирати формалну спецификацију која је комплетна и коначна, што је последица динамике окружења где се програм користи.

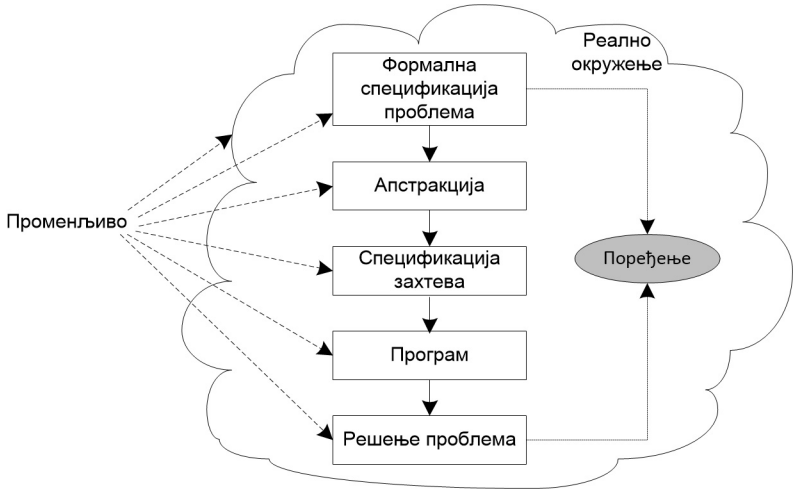
Са аспекта управљања еволутивним променама софтвера које настају након иницијалне испоруке, најинтересантнији је **E-тип програма** (*Evolving program, E-program*), који је најчешће интегрални део неког радног окружења у оквиру којега *еволуира* у складу са промена самог окружења, што је приказано на слици 3.6. У том окружењу су ови програми подложни променама које настају због људског фактора и промена у окружењу. Основна намена *Е-програма* је решавање неког реалног проблема или активности у реалном домену (окружењу).

Е-тип програма је интегрисан у реално окружење и има јасно дефинисане интерфејсе ка корисницима и осталим елементима окружења, помоћу којих пружа сервисе окружењу. Прихватање овог типа софтвера потпуно зависи од мишљења и процене корисника софтвера.

Употреба програма *Е-типа* у домену је у принципу врло комплексна и подразумева континуирано модификовање и адаптирање које прати промене у домену употребе. Практично због уграђености програма у домен, немогуће је предвидети еволуцију програма, али и раздвојити промене домена и промене програма.

Важна карактеристика овог типа софтвера је да он утиче на окружење у којем се користи и да може чак и мењати окружење, па се процес еволуције може посматрати као систем са повратном спрегом (*feedback system*).

У пракси доминирају програми *Е-типа*, док су програми *С-типа* и *П-типа* посебни случајеви који захтевају и посебан третман.



Слика 3.6: *Еволутивност Е-типа програма*

3.1.3 Закони еволуције софтвера

Законе еволуције софтвера је формулисао *Lehman* са циљем да идентификује њене узроке и процесе. Закони еволуције настали су директним посматрањима и мерењима еволуције софтверских система у пракси, а не односе само на аспекте софтверског инжењерства, већ обухватају и аспекте као што су управљање, организација, социолошки фактори и активности корисника. *Lehman*-ови закони у централни фокус еволуције софтвера постављају промене. Основни закони еволуције софтвера су:

I **Континуирана промена**. Системи Е-типа се морају континуирано адаптирати или они постају прогресивно све мање задовољавајући. Еволуција се испољава кроз процес одржавања софтвера, који је базиран на повратним информацијама из окружења у којем се софтвер користи. У пракси се софтвер мора континуирано мењати да би се адаптирао променама окружења.

II **Повећање сложености**. Током еволуције система Е-типа, његова сложеност се повећава упркос раду на њеном одржавању или смањењу. Континуиране измене софтвера и окружења у којем се користи усложњавају сам софтвер и окружење али и њихове интеракције.

III **Саморегулација**. Глобални процес еволуције система Е-типа је саморегулишући, пошто се током дуготрајног одржавања софтвера динамика карактеристика софтвера стабилише и то утиче позитивно на процес одржавања и еволуцију.

IV **Конзервација организационе стабилности**. Просечан ниво глобалних активности за еволутивни систем Е-типа тежи да остане константан током целог животног века система. Активности одржавања које утичу на еволуцију су управљане менаџерским одлукама организације која производи софтвер и повратним информацијама од корисника, које временом долазе до констатног нивоа.

V **Конзервација фамилијарности**. Инкрементални раст и честе промене система E-типа су ограничени потребом да се одржи ниво фамилијарности са системом, тј. нивоом познавања система. Честе промене током еволуције могу довести да људи који су ангажовани у животном циклусу софтвера (софтверски инжењери и корисници пре свега) више не могу да испрате промене функционалних карактеристика и перформанси ситема, што се свакако жели избећи.

VI **Континуирани раст**. Функционални садржај система E-типа мора континуирано да расте да би се одржало задовољство корисника током животног века система. Извори промена током еволуције могу бити разни, али је повратна информација од корисника пресудна за сагледавање и развој кључних карактеристика софтвера.

VII **Опадање квалитета**. Квалитет система E-типа ће опадати све док систем не буде био ригорозно одржаван или адаптиран променама у радном окружењу. На еволуцију софтвера поред карактеристика самог софтвера утичу разни фактори окружења у којем се софтвер користи (организациони фактори, људски фактор, техничко окружење, регулатива, итд.) који су често међусобно конфликтни и могу довести до непредвиђених промена и понашања окружења па и самог софтвера, што се дугорочно може негативно одразити на карактеристике софтвера.

VIII **Систем повратних информација**. Процеси еволуције система E-типа чине сложени систем повратних информација, са произвољним бројем петљи у којима се информације појављују, што обезбеђује унапређење процеса. Повратне информације могу потицати из различитих извора у окружењу у којем се софтвер користи, а њихов утицај на стабилност процеса еволуције и квалитет софтвера је сложена функција која се мора засебно анализирати за сваки софтверски систем.

Основна карактеристика ових закона је да не користе формалне математичке дефиниције и релације као што је то случај у неким другим научним дисциплинама (на пример у природним наукама). Ови закони обухватају уобичајена својства или понашања софтверских система који се мењају током њиховог животног века. Ови закони исказују генералне принципе којима подлеже еволуција софтвера.

Као последица еволуције јавља се старење софтвера (*software aging*) током животног циклуса. Старење софтвера се односи на процес извршавања (употребе) што се манифестује кроз деградацију перформанси или повећану појаву отказа током извршавања, али и на деградацију квалитета кода и документације што је последица честих промена током животног циклуса. Симптоми старења софтвера су:

- **Загађење** (*pollution*). Односи се на додавање кода или модула који не испоручују битне функционалности корисницима, а почињу да доминирају у софтверским системима током послова одржавања.
- **Уграђено знање** (*embedded knowledge*). Односи се на застаревање знања о домену проблема које је уграђено у софтвер, па је питање да ли софтвер заиста има актуелно знање о домену у којем се користи.

- **Лоше именовање** (*poor lexicon*). Називи и идентификатори у софтверу су измењени и нису у складу са значењем у домену проблема
- **Повезивање** (*coupling*). Начин повезивања компоненти и елемената у софтверу више не одговара стању у домену проблема

3.1.4 Повратне информације у процесу еволуције софтвера

Повратне информације имају веома важну улогу за процесе у животном циклусу софтвера. Најзначајнији извор повратних информација је реално окружење у које је софтвер интегрисан са циљем решавања проблема. Ове информације могу бити планиране или непланиране и оне представљају главни извор промена софтверских система. Планиране повратне информације се односе на искуства у употреби софтвера и на прилагођење софтвера промењеним околностима у окружењу где се користи, док се непланиране информације углавном односе на решавање проблема који настају употребом софтвера.

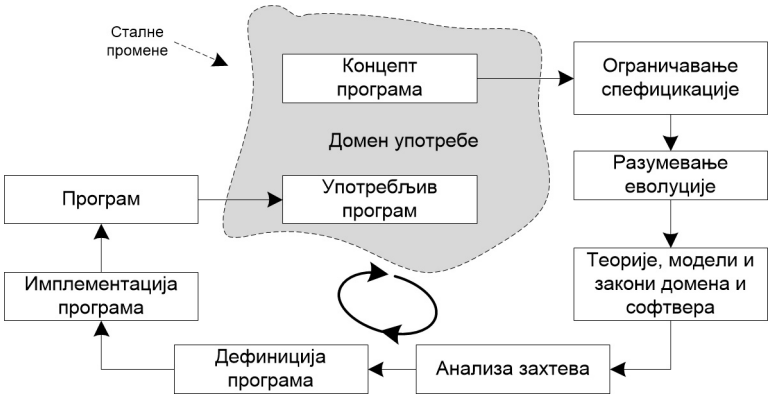
Lehman је формулисао *Feedback, Evolution And Software Technology (FEAST)* хипотезу у којој наглашава улогу повратне информације за стабилност карактеристика и оптимизацију процеса еволуције софтверских система Е-типа. Ову хипотезу је *Lehman* формулисао на следећи начин:

> *Процес софтвера обухвата повратне информације које су организоване у више нивоа и у више итерација, а са основним циљем да се оствари напредак у планирању, контроли и унапређењу процеса еволуције.*

Овде се процес софтвера посматра у најширем смислу и он поред техничких аспеката обухвата и аспекте управљања, маркетинга, продаје, подршке корисницима, као и аспект употребе у реалном окружењу од стране корисника.

Извор повратних информација могу бити извештаји о дефектима, промене у домену где се софтвер употребљава, промене потреба корисника, унапређење технологије, као и разни економски фактори. Употреба софтвера од стране корисника утиче на њихову перцепцију и разумевање детаља софтвера, системских концепата, апстракција и претпоставки, што доводи до стварања нових захтева за променама. Ови захтеви треба да стигну до испоручиоца или произвођача софтвера и да проузрокују одговарајућу реакцију. Одговор на захтеве корисника најчешће није тренутан пошто подразумева техничку, пословну и економску анализу захтева. Тек након ове анализе, и уз сагласност корисника са техничким и финснсијским аспектима промене, софтвер може бити модификован у складу са захтевима и приоритетима.

Промене које настају као последица овакве повратне спреге утичу на начин употребе система. Повратна спрега у контексту еволуције софтвера и домена употребе софтвера је приказана на слици 3.7. Повратна спрега између произвођача и корисника софтвера може имати и позитиван и негативан



Слика 3.7: *Еволуција софтвера и домена употребе као итеративни систем са повратном спрегом*

утицај на даљу употребу софтвера, што може довести до стабилизације и раста система или до његове нестабилности и деградације.

Повратне информације о софтверу у домену употребе подразумевају разумевање домена и ограничења која утичу на еволуцију софтвера у домену, на основу чега се креирају модели и закони еволуције софтвера у домену. Ови модели представљају основу за идентификовање захтева који се потом јасно специфицирају, анализирају и имплементирају у модификованом софтверу. Модификовани софтвер се потом испоручује у домен употребе, при чему се завршава једна итерација или циклус еволуције. Модификовани софтвер утиче на домен употребе, па се према потребама у домену могу појавити нови циклуси еволуције софтвера да би он остао употребљив.

Пре испоруке софтвера потребно је испланирати и спрегу за повратне информације које достављају корисници софтвера. То пре свега значи идентификацију и оптимизацију корисних повратних информација (или захтева) које ће резултирати унапређењима софтвера, али исто тако и минимизацију бескорисних повратних информација које практично представљају *сметње* у процесу еволуције.

3.2 Одржавање софтвера

Одржавање система има значајан утицај на перформансе пословања као што су продуктивност и профитабилност. Проблеми у функционисању система се могу значајно смањити употребом ефикасних процедура и стратегија у одржавању. Одржавање у контексту инжењерских система има широк спектар значења и обухвата акције које се односе на очување конзистентности и ефикасности комплексних система. Дефиниција одржавања према стандарду *EN 13306:2001 Maintenance - Maintenance terminology* је:

> *Одржавање је комбинација техничких, административних и управљачких акција током животног циклуса објекта (система) са*

циљем да га задржи или врати у стање у којем може да обезбеди захтевану функционалност.

Према већини модела животног циклуса, одржавање софтвера се посматра као последња фаза у животном циклусу софтвера. У поређењу са фазом развоја софтвера, фази одржавања се не посвећује довољно пажње ни у истраживањима а ни у индустријској пракси. Као последица тога, број грешака је знатно већи у одржаваним софтверским системима него у софтверским системима након иницијалне испоруке. Послови одржавања се у индустријској пракси посматрају као краткорочни послови које треба одрадити што је брже могуће. У складу са тим, истраживања указују да више од 70% инжењера ангажованих на пословима одржавања софтвера сматра да је ефикасност процеса одржавања веома мала.

Одржавање софтвера се генерално дефинише као било какав рад на софтверском систему након што он пређе у фазу употребе, чиме се јасно указује на границу између фаза развоја и одржавања софтвера. Најопштија дефиниција одржавања софтвера је:

> *Одржавање софтвера је посао који се реализује на софтверу након што он пређе у употребу. Одржавање софтвера обухвата: разумевање и документовање система, проширење функционалности система, проналажење и корекција грешака, одговарање на питања корисника и оператера, тренинг, конверзију и "прочишћавање софтвера", као и друге активности које се односе на софтвер у употреби.*

Дефиниција одржавања софтвера која је усвојена према стандарду *IEEE 1219-1998 - IEEE Standard for Software Maintenance* је:

> *Одржавање софтвера је модификација након испоруке са циљем да се коригују грешке, унапреде перформансе или други атрибути, или да се производ адаптира промењеном окружењу.*

Као резултат добре праксе индустријске праксе идентификована су 23 различита посла који се појављују у пракси одржавања софтвера. Сви ови послови се разликују по многим аспектима, али им је заједничка карактеристика да се односе на модификацију постојећих софтверских производа, а не на развој нових. Послови одржавања софтвера су:

1. **Главна побољшања** (*Major enhancements*) се односе значајно повећање функционалности софтвера.
2. **Мања побољшања** (*Minor enhancements*) се односе на минимална повећања функционалности софтвера.
3. **Одржавање** (*Maintenance*) се односи на отклањање дефеката од стране софтверских инжењера без обавезе према корисницима и без њихових захтева.
4. **Поправке у складу са гаранцијом према уговору** (*Warranty repairs*) се односи на отклањање дефеката у складу са формалним уговором.
5. **Подршка корисницима** (*Customer support*) се односи на одговарање на позиве корисника и на њихове извештаје о проблемима.

6. **Уклањање модула склоних грешкама** (*Error-prone module removal*) се односи на уклањање кода који показује проблематично понашање током извршавања.
7. **Обавазне промене** (*Mandatory changes*) се односи на обавезне или законске промене софтвера (нпр. усклађивање софтвера са регулативом у домену пословања).
8. **Анализа комплексности или структурална анализа** (*Complexity or structural analysis*) се односи на анализу структуре и комплексности софтвера који се одржава у складу са захтевом корисника.
9. **Реструктурирање кода** (*Code restructuring*) се односи на измене кода са циљем смањења сложености.
10. **Оптимизација** (*Optimization*) се односи на побољшање перформанси софтвера
11. **Миграција** (*Migration*) се односи на пребацивање софтвера на другу платформу.
12. **Конверзија** (*Conversion*) се односи на измене интерфејса или структуре софтвера.
13. **Реверзни инжењеринг** (*Reverse engineering*) се односи на издвајање информација о дизајну софтвера из кода који се извршава.
14. **Реинжењеринг/реновирање** (*Reengineering/renovation*) се односи на трансформисање постојећих софтвера (*legacy application*) у нову модернију форму.
15. **Уклањање "мртвог" кода** (*Dead code removal*) се односи на уклањање делова кода који се више не користе током извршавања софтвера.
16. **Елиминисање неактивних софтвера** (*Dormant application elimination*) се односи на уклањање и архивирање софтвера који се не користи.
17. **Национализација** (*Nationalization*) се односи на модификовање софтвера за интернационалну употребу (вишејезична употреба).
18. **Масовно ажурирање због великих глобалних промена** (*Mass updates due to global changes*) се односи на значајне промене због промена на глобалном нивоу као што је промена валуте.
19. **Рефакторинг** (*Refactoring*) се односи на додатно програмирање са циљем повећања јасноће кода.
20. **Повлачење из употребе** (*Retirement*) се односи на повлачење софтвера из употребе или укидање сервисирања.
21. **Сервис на терену** (*Field service*) се односи на слање инжењера из одржавања на локацију клијента да би одрадио посао одржавања.
22. **Пријављивање грешака или недостатака добављачу софтвера** (*Reporting bugs or defects to software vendors*) се односи на пријављивање грешака или недостатака произвођачу софтвера чије компоненете софтверска организација уграђује у софтвер који производи.

23. **Инсталирање ажурирања добијених од продаваца софтвера** (*Installing updates received from software vendors*) се односи на инсталирање нових верзија компоненти које произвођач софтвера добија од друге софтверске организације чије компоненте уграђује у свој софтвер.

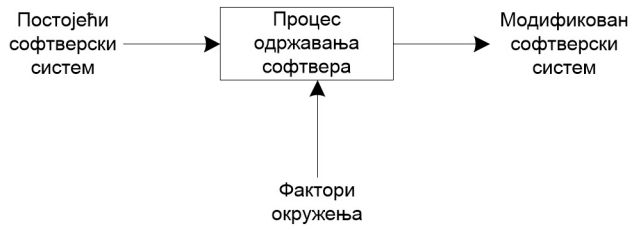
Истраживања указују да се највећи део трошкова за софтверске системе јавља након првог издавања, тј. у фази одржавања. У литератури се могу пронаћи подаци да су трошкови одржавања између 40 и 90 процената од укупних трошкова у животном циклусу софтвера. Шта више, трошкови одржавања за системе који су веома дуго у употреби вишеструко превазилазе трошкове развоја, што указује на значај унапређења процесе одржавања и еволуције софтвера, чиме се може постићи смањење трошкова, унапређење квалитета и брзине одговора на потребе корисника.

Одржавање софтвера се може појавити из много разлога, али су по правилу послови у склопу одржавања много *захтевнији* или *компликованији* у поређењу са пословима у току развоја. Разлози за сложеност послова одржавања софтвера су:

- Документација најчешће не постоји, а и ако постоји није довољно јасна и комплетна. Због тога инжењери у одржавању генерално немају поверења према документацији.
- Тешко је разумети код који је неко други писао, што је последица начина структурирања кода или специфичног начина решавања проблема.
- Програмери који су развијали код најчешће нису доступни у фази одржавања.
- Генерално, софтвер се не пројектује са циљем да се лако мења и одржава.
- Одржавање је много мање атрактиван посао од развоја нових производа, па у пракси инжењери често избегавају послови одржавања.

Одржавање софтвера се реализује у софтверским организацијама које су сложени социо-технички системи, у којима се могу идентификовати следећи фактори који утичу на одржавање софтвера:

- **Захтеви корисника** (*User requirements*). То су захтеви за модификацијом софтвера који могу укључити додавање функционалности, корекцију грешака, побољшање перформанси софтвера, али могу бити и захтеви за корисничком подршком која не укључују модификацију софтвера.
- **Организационо окружење** (*Organizational environment*). Односи се на организациону структуру и интерна правила пословања у организацији која реализује послове одржавања, а такође обухвата и спољашње регулативе (закони, стандарди, препоруке, итд.), као и услове тржишта.
- **Оперативно окружење** (*Operational environment*). Односи се на техничу инфраструктуру у коју је софтвер интегрисан и која омогућује његово извршавање, а укључује друге софтверске системе, оперативне системе, хардвер и комуникациону инфраструктуру, као и доменски



Слика 3.8: *Концептуални модел процеса одржавања софтвера*

специфичне компоненте (на пример, опрема за мониторинг у индустријским системима).

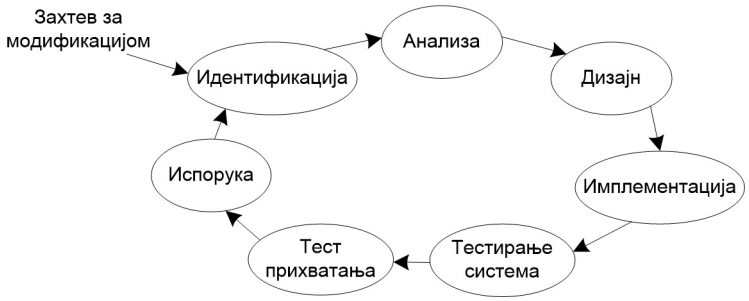
- **Процес одржавања** (*Maintenance process*). Односи се на активности које обезбеђују разумевање корисничких захтева кроз креативан рад базиран на претпоставкама без употребе докумената, употребу различитих приступа и метода у развоју и одржавању софтвера, као и брзо детектовање и отклањање грешака у софтверу.
- **Софтверски производ** (*Software product*). Односи се на зрелост и проблеме домена примене софтвера, квалитет или недостатак документације, комплексност и флексибилност структуре софтвера, и промене у квалитету софтвера.
- **Особље задужено за одржавање софтвера** (*Maintenance personnel*). Односи се на велику флуктуацију особља у софтверским организацијама, па је основно питање да ли софтвер одржавају особе коју су учествовале у развоју. Такође значајан је утицај доменског знања за специфичне софтверске производе, нарочито ако се особље пребацује на друге пројекте.

3.2.1 Процес одржавања софтвера

Основна разлика између развоја и одржавања софтвера је да је развој управљан захтевима корисника (*requirement-driven*), а одржавање је управљано догађајима (*event-driven*) који иницирају појединачне активности одржавања. Догађаји који иницирају процес одржавања најчешће потичу од корисника (клијената, или у ширем смислу тржишта), али могу потицати и од људи који су ангажовани у развоју и одржавању софтвера.

Процес одржавања је према *ISO/IEC 12207 Software Life Cycle Processes* стандарду дефинисан као примарни процес у животном циклусу софтвера. Концептуални модел процеса одржавања, приказан на слици 3.8, као улаз има постојећу верзију софтверског система, а као излаз даје модификовану верзију софтверског система. На процес одржавања софтвера утичу разни фактори из окружења (нпр. способности и вештине особља које је задужено за одржавање, пословна политика организације која производи софтвер, трендови тржишта, итд.).

Многе активности у току одржавања су присутне и у току развоја софтвера (моделовање, кодирање, тестирање, документовање), међутим

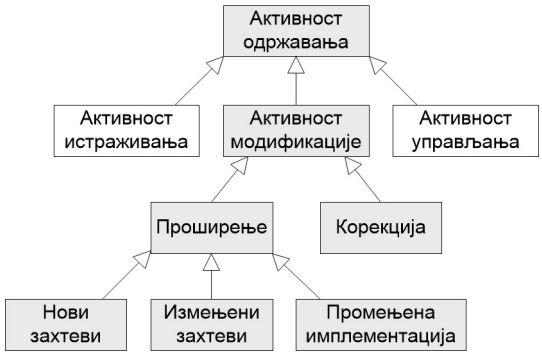


Слика 3.9: *Активности у процесу одржавања софтвера према стандарду IEEE 1219-98*

постоје активности које су карактеристичне само за фазу одржавања, као што су миграција, анализа утицаја промене, повлачење софтвера из употребе, итд. Преглед активности у оквиру типичног процеса одржавања софтвера, према стандарду *IEEE 1219-98 Standard for Software Maintenance*, је приказан на слици 3.9. Активности у оквиру процеса одржавања су специфичне за софтверске организације које их примењују, што значи да свака организација примењује и адаптира ове активности специфичном контексту одржавања који имплементира. Разноликост процеса и активности одржавања зависе од разних фактора, као што су: домен примене софтвера, величина софтвера, учесталост промена, ограничења у распореду послова, ресурса и буџета. На комплексност процеса одржавања утичу следећи фактори:

- Одржавање се реализује на комплексним софтверским системима, који временом постају све сложенији (према законима еволуције).
- Процес одржавања укључује људе који имају различите улоге, као што су корисник који пријављује захтев за одржавањем, инжењер који анализира и имплементира промену, или тест инжењер који тестира модификовани софтвер.
- Процес је итеративан са више петљи у којима учесници процеса размењују информације неопходне за реализацију послова одржавања.

Слика 3.9 указује да је есенцијални процес у фази одржавања софтвера процес који обезбеђује руковање захтевом за модификацијом софтвера. Процес управљања захтевом за модификацијом је неопходно планирати још у фази развоја софтвера чиме се обезбеђује ефикасна подршка за одржавање и управљање конфигурацијом софтвера током целог животног циклуса софтвера. Типичан процес одржавања започиње пријемом захтева за модификацијом, а завршава се испоруком нове верзије софтвера или модула за који је захтев постављен (осим у случајевима када захтев није прихваћен). Модификација софтвера се састоји од следећа три посла: разумевање постојећег система (посао који захтева највише времена и најкритичнији је за успех модификације), имплементација модификације у софтверу и валидација модификације. Два типична процеса који су део одржавања софтвера су:



Слика 3.10: *Сегмент онтологије активности одржавања софтвера који се односи на активности модификације софтвера*

- Процес појединачног захтева за модификацијом који реализује инжењер у одржавању са циљем да имплементира конкретну модификацију софтвера.
- Процес управљања свим захтевима за модификацијама на нивоу целе софтверске организације.

Процес одржавања даје опис како организовати специфичне активности одржавања. Део онтологије активности у оквиру одржавања софтвера који се односи на активности модификације софтвера је означен осенченим блоковима на слици 3.10. Дефиниције које се односе на активности модификације софтвера у онтологији одржавања су приказане у табели 3.1.

Врло често је тешко захтев за одржавањем, а самим тим и тип одржавања јасно класификовати према постојећим типологијама одржавања софтвера. У пракси не постоји једнозначна релација између извешаја о проблему и корективног типа одржавања, што је случај када корисник као проблем пријављује понашање софтвера које није специфицирано у оригиналним захтевима. У таквим случајевима, извештај о проблему за последицу има активност одржавања која се класификује као проширење а не као корекција. Веома је важно јасно разликовати да ли је активност одржавања корективног типа или је проширење софтвера, пошто оне у пракси имају различит финансијски ефекат за софтверску организацију и клијента, а могу захтевати различите начине организовања послова одржавања. Наиме, корекције се у великом броју случајева не наплаћују корисницима, док се проширење наплаћује пошто подразумева нови развој и захтева шире разумевање целог софтверског производа и контекста његове употребе.

3.2.2 Типови одржавања софтвера

Прву типологију одржавања софтвера је предложио Swanson 1976 године. Према тој типологији основни типови одржавања софтвера су:

1. **Коректив**но (*Corrective*). Одржавање које се предузима да би се отклониле уочене грешке у функционисању софтвера.

Табела 3.1: *Дефиниције у онтологији активности одржавања софтвера које се односе на активности модификације софтвера*

Назив	**Дефиниција**
Активност	Акција која може бити истраживање, модификација, управљање или обезбеђење квалитета, а може се састојати од скупа под-активности. Улаз може бити један или више објеката, а излаз може бити један или више промењених објеката што зависи од типа активности.
Активност истраживања	Активност која процењује утицај реализације модификације која настаје на основу захтева за модификацијом или извештаја о проблему.
Активност модификације	Активност која као улаз има један или више објеката и на излазу даје један или више модификованих објеката, који мењају понашање или имплементацију система.
Активност управљања	Активност која се односи на управљање процесом одржавања или на управљање конфигурацијом софтверских производа.
Корекција	Активност која се реализује са циљем да коригује грешку идентификовану у софтверском производу.
Проширење	Активност која реализује промену у софтверском систему тако што пробољшава или проширује понашање или имплементацију система.

2. **Адаптивно** (*Adaptive*). Одржавање које се предузима да би се софтвер прилагодио и остао употребљив у промењеном окружењу (промена техничког или организационог контекста где се софтвер користи).

3. **Перфективно** (*Perfective*). Одржавање које се предузима са циљем да се побољшају перформансе, способност одржавања (*maintainability*), или остале карактеристике софтверског производа.

Ова типологија је касније допуњена у стандарду *ISO/IEC 14764 - Standard for Software Engineering-Software Maintenance*, тако да укључује и **превентивно одржавање** (*preventive maintenance*) које се дефинише на следећи начин:

> *Превентивно одржавање је модификација софтвера након испоруке са циљем да се детектују и коригују скривене грешке у софтверу пре него што се испоље приликом употребе.*

Као подтип корективног одржавања се јавља и **ургентно** или одржавање у хитним случајевима (*emergency maintenance*), када је важније да се брзо интервенише него да се формално испрати стандардни процес одржавања. Ургентно одржавање се јавља у следећим случајевима:

- Грешка која се мора отклонити да би се обезбедило нормално функционисање софтвера и система у који је софтвер интегрисан
- Промене у окружењу које имају неочекиване ефекте на софтверски систем.

- Непредвиђене промене у пословању које настају због промена законске регулативе или услова пословања на тржишту.

Као посебан облик корективног одржавања јавља се такође и **управљање проблемима** (*problem management*) који се уочавају током употребе софтвера. Приликом дефинисања процеса управљања проблемима треба размотрити следеће:

- Идентификација и опис проблема.
- Груписање сличних проблема што може помоћи у препознавању и предикцији потенцијалних проблема.
- Време појављивања проблема.
- Критичност и ургентност проблема.
- Класификација узрока проблема.

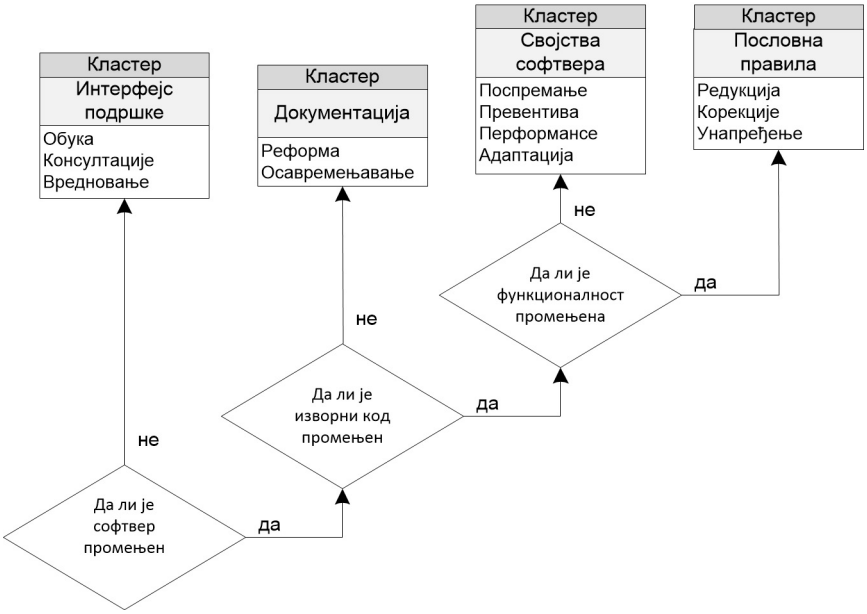
Ова терминологија је делимично прихваћена, али је у многим случајевима у пракси редефинисана и прилагођена конкретним околностима одржавања софтвера. Тако се у пракси може пронаћи и следећа класификација одржавања софтвера која садржи активности проширења софтвера (*enhancements*), поправки софтвера (*repairs*) и превентивног одржавање (*preventive maintenance*).

Терминологија одржавања софтвера најчешће зависи од гране индустрије или домена примене софтвера, као и од специфичности софтверске организације која је примењује. Типологија одржавања је у пракси много детаљнија и може се препознати више типова одржавања од основна три прописана у Swanson-овој типологији или у стандарду *ISO/IEC 14764 - Standard for Software Engineering-Software Maintenance*. Класификације типова одржавања се може посматрати и на основу евиденције прикупљене из праксе, па је тако уведена нова типологија која је базирана на променама које су резултат практичних активности у оквиру одржавања софтвера.

У пракси, процеси и активности одржавања не морају довести до промена на софтверу, али исто тако могу проузроковати више промена различитог типа на софтверу. За овакву класификацију типова одржавања користе се следећа три питања изведена из евиденције активности које проузрокују промене:

1. Да ли је реализовани посао одржавања променио софтвер?
2. Да ли је мењан изворни код софтвера?
3. Да ли је промењена функционалност за корисника софтвера?

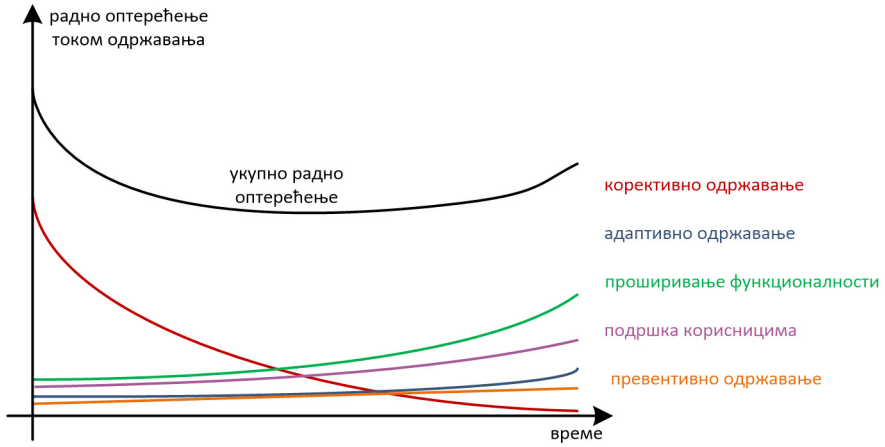
Промена софтвера не мора укључивати промену кода, већ на пример промену конфигурационих параметара који утичу на употребу. Исто тако промена кода не мора укључити промену функционалности већ може бити отклањање грешке или козметичка промена интерфејса. Ова нова типологија одржавања има дванаест различитих типова груписаних у четири кластера, што је приказано на слици 3.11. Кластери у класификацији типова одржавања су:



Слика 3.11: *Кластери и типови одржавања софтвера базирани на евиденцији из праксе*

- **Кластер интерфејса подршке** (*Support interface cluster*). Ово су типови одржавања који на сва три претходно постављена питања имају негативан одговор, тј. послови одржавања не мењају софтвер у било ком облику. Овде пре свега спадају активности подршке корисницима како да најефикасније користе софтвер.
- **Кластер документације** (*Documentation cluster*). Ово су типови одржавања који имају потврдан одговор на прво пиптање, а негативан на следећа два. Практично то значи да се мења документација, док код и функционалности које се пружају корисницима остају не промењени.
- **Кластер својстава софтвера** (*Software properties cluster*). Ово су типови одржавања код којих је позитиван одговор на прва два питања, док је негативан одговор на треће питање. То значи да се промене односе на измене софтвера, укључујући и измене кода, али да са аспекта функционалности које су доступне корисницима софтвер није промењен. Промене овог типа се односе на перформансе софтвера.
- **Кластер пословних правила** (*Business rules cluster*). Ови типови одржавања имају позитиван одговор на сва три постављена питања, што практично значи да се мењају сви аспекти софтвера укључујући и код и функционалности доступне корисницима.

У пракси се најчешће јављају типови одржавања из *кластера пословних правила* и она имају највећи утицај и на сам софтвер и на пословне процесе у које је софтвер интегрисан или их подржава. Типови одржавања који утичу на корисничке функције софтвера обезбеђују реализацију редукције (*reductive maintenance*), корекције (*corrective maintenance*) или проширења (*enhancive*



Слика 3.12: *Типичан удео појединих типова радног оптерећења током одржавања софтвера*

maintenance) функционалности. Следећи кластер по значају и утицају на софтверски производ и функције доступне кориснику је *кластер својстава софтвера* који такође утиче на промене самог софтвера. Ова два кластера се најчешће јављају у пракси и подрзумевају постојање процеса промене софтвера у току одржавања.

Типична дистрибуција радног оптерећења за различите типове одржавања након испоруке софтвера је приказана на слици 3.12. Грешке у раду софтвера се најчешће идентификују убрзо након испоруке софтвера, и највећи број се врло брзо реши. Откази због грешака у софтверу се касније много ређе јављају што резултује опадајућом кривом за послове корективног одржавања (црвена крива на слици 3.12). Обим послова који се односе на остале типове одржавања временом се благо повећава што указује на унапређење или проширење функционалности и перформанси софтвера, или на адаптацију софтвера новим условима у окружењу где се користи (остале обојене криве на слици 3.12). У каснијим фазама употребе софтвера, након што је већина грешака отклоњена, доминирају послови проширивања функционалности софтвера (зелена крива на слици 3.12), као и активности подршке корисницима софтвера (љубичаста крива на слици 3.12).

У индустријској пракси, свака софтверска организација (компанија или фирма) идентификује и имплементира типологију одржавања која највише одговара њеном техничком и организационом контексту.

Поред уобичајених шема класификације активности одржавања према типовима одржавања, активности одржавања се могу класификовати и према грануларности у односу на елементе софтверског система који су предмет одржавања. Могу се идентификовати следећи нивои грануларности активности одржавања:

- Ниво линија кода. Измене на нивоу линија кода се могу испратити употребом услужног алата *diff* који је интегрисан у савремена развојан окружења.

- Ниво линија кода у више датотека или модула.
- Ниво модула, при чему се све измене у елементима модула посматрају као јединствена измена реализована на модулу.
- Ниво програма, при чему се промене посматрају на нивоу целог софтверског производа, али се мора водити рачуна о опсегу промене на нижим нивоима.

Резултати истраживања о факторима који одређују активности одржавања указују да постоје *предвидљиве* шеме у одржавању софтвера, што указује да се одговарајућим методама управљања и планирања разне активности у процесу одржавања могу унапредити.

3.3 Реинжењеринг софтвера

Еволуција софтвера је базирана на променама које треба имплементирати да би софтвер био употребљив. Међутим, већина софтверских система, а нарочито оних који су дуго у употреби, су тешки за разумевање и измене. Разлози за проблеме у разумевању и измени софтвера су најчешће промене које су се често реализовале и након којих је комплексност система повећана, а промене при томе нису документоване.

Значајан број софтверских система у употреби је од велике користи у пословању и организације нису спремне да се одрекну тих система. Овакви системи се називају "наслеђени софтверски системи" (*legacy software*), а њихово одржавање је веома скупо, па се често прибегава реинжењерингу ових система да би се смањила њихова комплексност и унапредила структура, а они и даље остали употребљиви. Реинжењеринг може укључити поновну израду документације система, рефакторисање архитектуре, превођење програма на савремене програмсе језике и радна окружења, модификовање и унапређење структуре података. Међутим, основна идеја реинжењеринга је да се систем побољша, али да задржи старе функционалности које су се показале корисне. Због тога се за овакве системе чешће ради реинжењеринг него што се системи замењују новим. Основне добити које доноси реинжењеринг су:

- **Смањење ризика**. Софтвери су често критични за успешно пословање, па је развој новог софтвера уместо већ постојећег који функционише ризично са аспекта настанка грешака у функционисању или кашњења испоруке новог софтвера због чега се могу појавити губици у пословању.
- **Смањење трошкова**. Трошкови реинжењеринга постојећих система су мањи од трошкова развоја нових система.

Реинжењеринг софтверских система се може јавити због више разлога, а истраживање индустријске праксе указује на следеће:

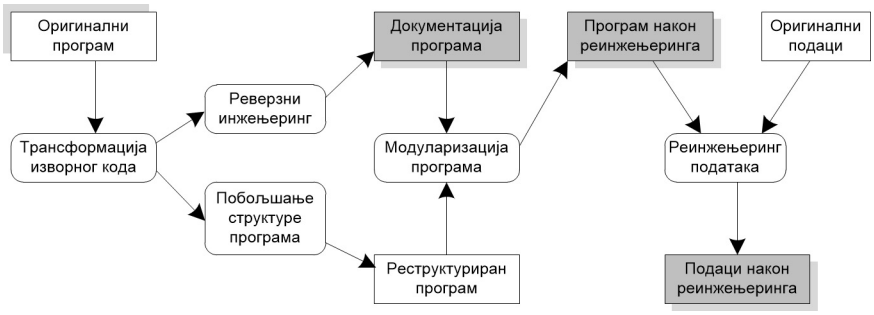
- **Реструктурирање монолитног система** на независне модуле који се потом могу комбиновати у систем са одговарајућом архитектуром који је једноставнији за одржавање.

- **Побољшавање одређених перформанси постојећег система**, као на пример доступност појединих модула или брзина приступа.
- **Пребацивање система на нову платформу**, што подразумева јасно раздвајање платформски зависног и платформски независног кода у систему.
- **Екстракција дизајна система** да би се олакшало одржавање или проширење система новим модулима.
- **Примена нових технологија, стандарда или библиотека** са циљем да се смање трошкови одржавања система.
- **Документовање знања о систему** као помоћ софтверским инжењерима и корисницима софтвера.

Уопштени процес реинжењеринга софтвера је приказан на слици 3.13. Улаз процеса је постојећи софтверски систем који је у употреби, а излаз је реструктурирана верзија софтвера, која укључује реструктурирани програм, реструктуриране податке и креирање одговарајуће документације. Активности у процесу реинжењеринга, према слици 3.13 су:

- **Трансформација кода** (*Source code translation*). То је активност превођења изворног кода у нови програмски језик или у нову верзију на постојећем програмском језику.
- **Реверзни инжењеринг** (*reverse engineering*). Активност који се базира на анализи постојећег програма са циљем да се добију информације о организацији и функционалностима програма. Овај поступак је обично аутоматизован и као резултат даје модел организације програма.
- **Побољшање структуре програма** (*program structure improvement*). Активност се односи на побољшање структуре да би се боље разумела, што касније олакшава одржавање програма. Поступак је полуаутоматизован и укључује ручне интервенције.
- **Модуларизација програма** (*program modularization*). Активност регруписања елемената софтвера у модуле тако да се уклоне редундансе и неправилности из претходне структуре. Такође је у овој активности могуће и рефакторисање архитектуре целог система, а процес је најчешће ручни.
- **Реинжењеринг података** (*data reengineering*). Активност се односи на измену структуре података тако да се усклади са променама у целом софтверу. То подразумева измену шеме базе података и креирање нове базе. Овде се врши и прочишћавање података, отклањање грешака, уклањање дуплираних записа. Ови послови могу бити подржани софтверским алатима који аутоматизују посао.

Поступак реинжењеринга не мора обухватити све активности приказане на слици 3.13, што зависи од тога шта је све потребно одрадити у конкретном случају реинжењеринга. На пример, реинжењеринг података је потребно радити само ако се приликом реинжењеринга структуре софтвера мења и структура података. Чест случај реинжењеринга је да се стари систем и структура података задрже, а да се креира слој са сервисима који



Слика 3.13: *Уопштени модел процеса реинжењеринга софтвера*

Слика 3.14: *Сложеност и трошкови реинжењеринга софтвера*

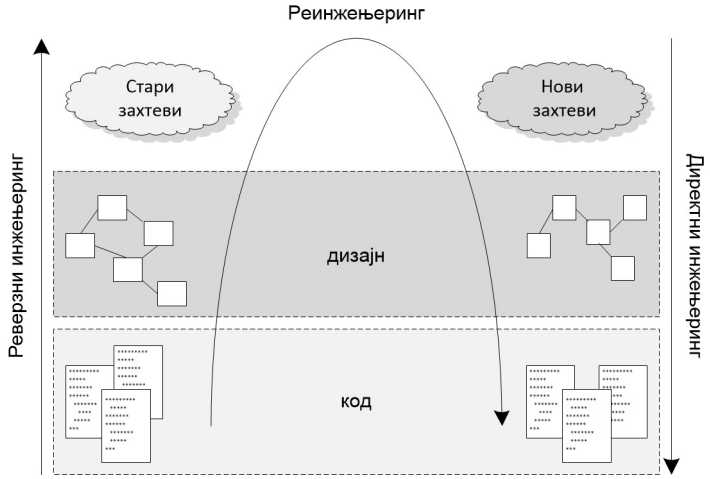
обезбеђују приступ старом систему (*leagacy system wrapping*). Сложеност посла који се обавља приликом реинжењеринга утиче на трошкове, што је приказано на слици 3.14. Такође је врло важно препознати и ситуације када се не исплати радити процес реинжењеринга система, што су обично случајеви са радикалним променама архитектуре софтвера и приступа развоју софтвера (нпр. прелаз са структурираног и функционалног развоја на објектно-оријентисани развој софтвера).

Проблем који се јавља у домену реинжењеринга софтверских система је конфузија са терминологијом, а пре свега се ту мисли на појмове реверзног инжењеринга (*reverse engineering*) и реинжењеринга (reengineering). Са циљем разумевања ова два термина уведене су дефиниције:

> **Реверзни инжењеринг** (*reverse engineering*) је поступак анализе постојећег система са циљем да се идентификују његове компоненте и релације између компоненти и да се креира представа система на вишем нивоу апстракције, чиме се олакшава разумевање система.
>
> **Реинжењеринг** (*reengineering*) је поступак анализе и измене постојећег система тако да се нови систем креира у новој форми, а обезбеђује функционалности које су претходно биле доступне.

Реинжењеринг обично решава неке проблеме идентификоване у постојећем систему и ради реструктурирање и измене система тако да су у ревидираном систему уклоњени проблеми.

Циклус реинжењеринга система укључује процес реверзног инжењеринга са циљем да се идентификује структура постојећег система и његови захтеви,



Слика 3.15: *Реинжењеринг, реверзни и директни инжењеринг софтвера*

и да се потом процесом директног инжењеринг (*forward engineering*) креира нови систем почевши од измењене спецификације захтева, што је приказано на слици 3.15.

Да би се потпуно разумео циклус реинжењеринга, уводи се следећа дефиниција директног инжењеринга који као резултат има ревидирани систем:

> **Директни инжењеринг** (*forward engineering*) је поступак креирања система на основу апстракције система на вишем нивоу (нпр. модел система).

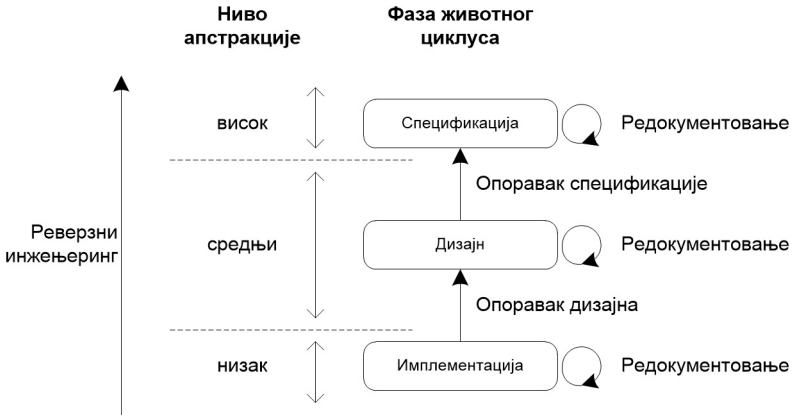
3.3.1 Реверзни инжењеринг

Концепт реинжењеринга је настао у области производње која је базирана на већ утемељеним инжењерским дисциплинама (машинско инжењерство, електротехничко инжењерство, технолошко инжењерство, итд.), и као такав је прихваћен и примењен у области софтверског инжењерства. Реверзни инжењеринг је кључна фаза у реинжењерингу пошто обезбеђује разумевање структуре и функционисања постојећег система. У области софтверског инежерства, реверзни инжењеринг се обично реализује на нивоу кода софтвера пошто дугорочно одржавање и еволуција софтвера доводе до неусаглашености кода и дизајна, као и до постојања неадекватне документације (ако је уопште има). Имајући то у виду, основни циљ реверзног инжењеринга је да олакша промене софтвера тако што омогућује његово боље разумевање са аспекта шта и како ради (функционише), као и са аспекта бољег разумевања његове архитектуре. Реверзни инжењеринг се спроводи да би се обезбедило:

- **Повратак или опоравак изгубљених информација**. Због честих модификација система током одржавања, а које се спроводе под временским и организационим ограничењима, информације о

структури и компонентама система настале током спецификације и дизајна врло често нису актуелне. Због такве ситуације је код једини извор информација о актуелном стању система, па се из кода могу добити информације о дизајну и спецификацији захтева методама реверзног инжењеринга.

- **Подршка за миграцију између платформи**. Применом реверзног инжењеринга се могу добити документи дизајна и спецификације који се потом могу директном инжењерингом прилагодити новим софтверским или хардверским платформама на које се систем жели мигрирати.
- **Алтернативни погледи на софтверски систем**. Током анализе кода и израде нове верзије документације могу се креирати и алтернативни типови документације поред већ постојећих типова. На пример, поред објектно-оријентисаних модела креираних применом Unified Modeling Language (UML), могу се креирати дијаграми токова података или контроле који пружају другачије погледе на систем.
- **Идентификација компоненти за поновну употребу**. Алати и методе за анализу кода и структуре програма током процеса реверзног инжењеринга омогућују идентификацију и екстракцију софтверских компоненти које се могу поново употребити, што омогућује унапређење ефикасности процеса развоја и повећање квалитета софтверских производа.
- **Побољшање или креирање документације**. Један од кључних проблема наслеђених система у дугој употреби је непостојање документације, или документација која не одсликава актуелно стање система. Креирање нове документације или корекција постојеће се може спровести применом алата за реверзни инжењеринг.
- **Ефикасно решавање проблема због комплексности софтвера**. Један од основних проблема са променама постојећих софтверских система током одржавања је да еволуција система обично резултује повећањем комплексности (што је идентификовано и као један од закона еволуције). Да би се управљало комплексношћу система, потребно је приликом сваке модификације извршити апстракцију релевантних информација, а игнорисати све што није релевантно.
- **Идентификовање нежељених ефеката**. У случајевима одржавања софтвера када недостаје глобални поглед на систем, промене на појединим компонентама могу проузроковати нежељене промена на другим компонентама (*ripple effect*). Методе реверзног инжењеринга могу обезбедити да архитектура система постане доступна, што омогућује боље сагледавање и предикцију последица модификација софтвера.
- **Смањење напора и трошкова одржавања**. Значајан удео укупног времена потребног за модификацију се троши на разумевање софтверског система. Два главна разлога за то су недостатак одговарајуће документације и недовољно доменско знање. Методе реверзног инжењеринга могу обезбедити информације које недостају о



Слика 3.16: *Нивои апстракције софтвера у реверзном инжењерингу*

софтверском систему и домену примене, па на тај начин и утицати на смањење напора и трошкова одржавања.

Реверзни инжењеринг подразумева спровођење једне или више апстракција на више различитих нивоа, тако што се креће од ниског нивоа, тј. имплементације (изворног кода програма). Циљ је да се идентификују елементи (конструкти) на ниском нивоу и да се замене одговарајућим елементима (конструктима) на вишим нивоима, средњем нивоу где се софтвер представља дизајном, и високом нивоу где се софтвер представља у виду спецификације. Нивои апстракције и њихов однос са фазама животног циклуса у оквиру реверзног инжењеринга је приказан на слици 3.16. Резултат реверзног инжењеринга може бити на било ком нивоу апстракције, од кода на ниском нивоу, па до спецификације на високом нивоу, што зависи од циљева пројекта. Поступак реверзног инжењеринга који резултира новим дизајном се назива **опоравак дизајна** (*design recovery*), а поступак који резултира новом спецификацијом софтвера се назива **опоравак спецификације** (*specification recovery*).

Редокументовање (*redocumentation*) је поступак поновног креирања документа са семантички еквивалентном репрезентацијом софтверског система на истом нивоу апстракције (слика 3.16). Циљеви редокументовања су:

- Креирање алтернативних погледа на софтверски систем са циљем бољег разумевања његове структуре и функционисања.
- Унапређење и поправка постојеће документације, што је неопходно за системе у којима документација није ажурирана током еволуције система.
- Креирање документације за модификовани софтверски производ, што треба да помогне у наредним активностима одржавања.

Опоравак дизајна подразумева екстракцију значајних информација на вишем нивоу апстракције на основу анализе изворног кода. За реализацију опоравка дизајна се користе постојећи код, постојећа документација о дизајну

која је обично застарела (не прати еволутивне промене софтвера), лично искуство особља ангажованог на одржавању и постојеће знање о домену примене софтвера. Дизајн који се добије након процеса опоравка дизајна обично не одговара почетном дизајну софтвера, а може се користити за поновни развој система. Опоравак дизајна се спроводи у следећим фазама:

- Разумевање изворног кода и идентификација постојећих модула и структуре података. Анализа обично почиње на вишем нивоу архитектуре софтвера и спроводи се ка елементима на нижем нивоу, све до појединачних фајлова са изворним кодом.
- Идентификовање софтверских компоненти које се могу користити за поновну употребу и њихово додавање у библиотеку компоненти за поновну употребу. Такође се врши идентификовање и систематизација знања о компонентама које се може користити за изградњу сличних компоненти.
- Креирање апстрактне репрезентације идентификованих софтверских компоненти за поновну употребу, креирање новог знања о домену примене, и сагледавање ширег обима примене идентификованих компоненти..

Опоравак спецификације се користи у случајевима када опоравак дизајна није довољан за развој новог система, што се дешава када информације о дизајну нису корисне или довољне за нови развој. То су ситуације када се мења парадигма развоја софтвера, као на пример када се врши реинжењеринг са структурног програмирања на објектно-оријентисано програмирање. У том случају је одговарајући поступак опоравак спецификације са циљем да се добије оригинална спецификација система ако је то могуће након неконтролисане и недокументоване еволуције. Издвајање информација о спецификацији на високом нивоу се може вршити на основу изворног кода система или постојећег дизајна, а од користи могу бити и постојећа документација, искуство особља које врши реверзни инжењеринг и постојеће знање о домену проблема. Спецификација која се добије треба да буде у форми која омогућује лаку поновну имплементацију применом другог програмског језика или програмске парадигме. Користи од опорављене спецификације током одржавања софтвера су: подршка активностима одржавања без потребе да се захтева приступ изворном коду, помоћ у разумевању ефеката захтеване модификације на софтверски систем, и помоћ у развоју и одржавању сличних софтверских система.

Поглавље 4

Софтверски процеси

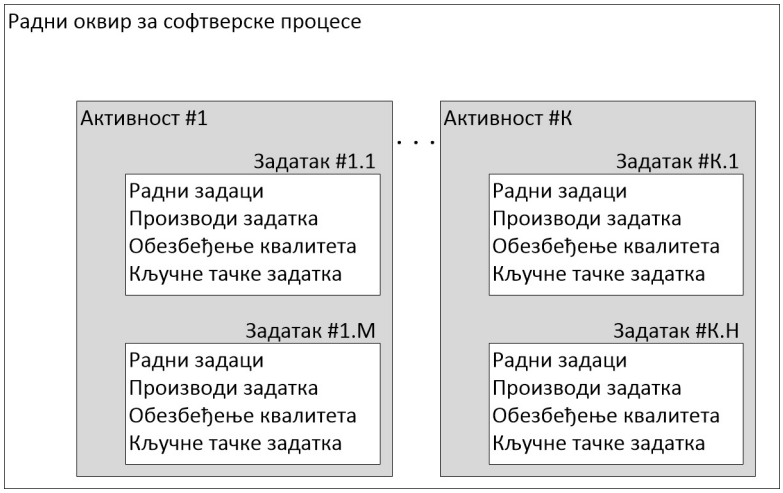
Процес је у најширем смислу скуп повезаних активности које трансформишу скуп улазних величина у скуп излазних величина, при чему активности троше ресурсе за реализацију трансформације. У софтверском инжењерству се процеси односе на рад који се реализује са циљем развоја, одржавања или употребе софтвера. Процеси у софтверском инжењерству се уобичајено називају софтверски процеси (*software processes*). Процеси се састоје од активности, а свака активност од задатака за чију реализацију је потребно пратити процедуру извршавања, као и од скуп алата и техника који омогућују извршавање. Производња и одржавање квалитетних софтверских система захтева усклађеност и дисциплиновану примену јасно структурираних софтверских процеса. Основни принципи инжењеринга процеса су:

- Бољи процеси обезбеђују боље производе рада, што подразумева побољшане функционалности и квалитет, мање додатног рада и прераде производа, као и једноставније модификовање.
- Процес мора бити тако дизајниран да задовољава постављене процесне захтеве и ограничења, да одговара потребама индивидуалних пројеката и да обезбеди ефикасан рад.
- Процеси у оквиру свих пројеката у организацији се морају извести из радног оквира за процесе (*process framework*) који представља генерички модел процеса који се може прилагодити различитим ситуацијама на пројектима. Прилагођење процеса укључује додавање, уклањање или модификовање елемената процесног модела у складу са специфичним захтевима.
- Дизајн и унапређење процеса за резултат треба да имају једноставнији рапоред активности и задатака, повишен квалитет, смањене трошкове, задовољније раднике и кориснике.
- Унапређење процеса се ретко дешава спонтано, већ је последица потребе да се смањи утрошак времена, напора и трошкова рада, што захтева додатно улагање времена, напора и ресурса.

Ефикасно управљање процесима подразумева да људи одговорни за процесе имају тачне и ажурне податке о свим активностима, ограничењима које се односе на те активности, као и упутства за имплементацију активности у оквиру процеса. У ту сврху неопходно је прикупити следеће информације о процесима:

- **Процесна средства** (*Process Assets*) се могу прикупити из више извора у организацији (људи, документација, електронски репозиторијуми), а обухватају, шаблоне, дефиниције, контролне листе, документе пословне политике, стандарде и документе о систематизованом искуству.
- **Дефиниција процеса** (*Process Definition*) укључује специфичне информације о активностима у оквиру процеса, улоге у оквиру процеса, објекте који се користе, као и услове који се јављају на почетку процеса, током реализације и на завршетку.
- **Статус процеса** (*Process Status*) карактерише стање перформанси процеса у појединим фазама реализације, за шта се користе мерљиви атрибути процеса. Информације о статусу се користе за праћење и контролу процеса.
- **Подаци о мерењу процеса** (*Process Measurement Data*) се прикупљају као информације које карактеришу специфичне перформансе процеса у смислу уложеног труда, перформанси особља, употребе ресурса итд. Подаци се прикупљају током реализације процеса и карактеришу аспекте процеса који се односе на производе и ресурсе, а посебно аспекте везане за квалитет.
- **Подаци специфични за пројекат** (*Project-specific Data*) настају и прикупљају се током имплементације процеса. Ови подаци представљају конкретан запис о перформансама процеса у специфичном контексту, а допуњују податке о статусу и мерењу процеса. Ови подаци укључују: посебне документе настале током имплементације пројекта, записе са радних састанака, белешке, систематизоване лекције о искуству итд.

Средње и велике софтверске организације најчешће имају развијене процесне моделе за различите типове пројеката, па се за сваки специфичан пројекат из репозиторијума процесних модела бира процесни модел који се може прилагодити специфичним потребама пројекта. Радни оквир са дефинисаним процесним моделима се назива и архитектура софтверских процеса (*software process architecture*), а треба да обезбеди конзистентну употребу процеса у различитим пројектима. Дефинисањем радног оквира са процесима се обезбеђује одређени ниво стандардизације софтверских процеса, али уз очување флексибилности због специфичних потреба различитиих пројеката. Микро и мале софтверске организације немају довољно ресурса (запослени, експертиза, време или новац) за успостављање процеса и процесних модела у животном циклусу софтвера, већ процесе ад-хок креирају и имплементирају за сваку специфичну ситуацију. Процесни модел за софтверске процесе садржи: опис радних активности које треба реализовати, редослед извршавања задатака и радних активности, опис преклапања и повезаности радних активности и задатака, и опис производа који су резултат реализације активности у процесу. Спецификација



Слика 4.1: *Радни оквир за софтверске процесе*

софтверских процеса омогућује боље разумевање процеса, побољшану комуникацију и координацију, боље управљање софтверским пројектима, мерење и унапређење процеса и производа, као и аутоматизацију процеса или појединих задатака у оквиру процеса. Изостанак спецификације процеса и ефикасног управљања процесима резултира следећим проблемима: прекорачење буџета, неодговарајући квалитет производа, касно откривање проблема и дефеката, конфузија са улогама и одговорностима чланова тима, као и непотребан вишак рада због неразумевања ситуације.

4.1 Дефиниција софтверског процеса

Потреба за дефинисањем софтверског процеса се појавила пре више од 30 година са значајним повећањем интересовања и све динамичнијим развојем софтверског инжењерства као научне и инжењерске дисциплине. Првобитна дефиниција софтверског процеса се односила само на процес развоја софтвера, пошто термин одржавање софтвера није био јасно дефинисан. Ова дефиниција софтверски процес посматра као скуп активности које је потребно реализовати да би се кориснички захтеви трансформисали у софтверски производ. На сличан начин, Pressman дефинише софтверски процес као радни оквир за активности и задатке који се реализују у пројектовању софтвера. Најопштија дефиниција софтверског процеса према књизи *Guide to the Software Engineering Body of Knowledge (SWEBOK)* из 2014. године дефиницију софтверског процеса проширује и на одржавање и употребу софтвера.

Радни оквир за софтверске процесе са дефинисаним скупом оквирних активности и скупом задатака за сваку активност је приказан на слици 4.1. Према овом радном оквиру сваки софтверски задатака садржи јасно дефинисане радне задатке, производе рада, дефинисан начин за обезбеђење квалитета и кључне тачке за дефинисање прогреса задатка (*milestones*).

Радни оквир за софтверске процесе садржи опис кључних активност за софтверске процесе, као што су комуникација, планирање, моделовање, конструкција, тестирање, испорука, али и додатне активности као што су управљање ризицима, контрола пројекта, обезбеђење квалитета, управљање конфигурацијом, итд. Ток процеса се реализује комбиновањем процесних активности у различите временске оквире извршавања, при чему се могу разликовати:

- **Линеарни ток** са секвенцијалним током извршавања активности.
- **Итеративни ток** са понављањем појединих активности или целог скупа активности.
- **Еволутивни ток** са цикличним понављањем целог скупа активности при чему се на крају сваког циклуса добија комплетнија верзија производа.
- **Паралелни ток** код којег се поједине активности извршавају истовремено.

Избор оквирних активности у радном оквиру зависи од начина организовања софтверске организације, природе проблема који се решава, тима који реализује активности и корисника производа. Поред описа активности и задатака у оквиру процеса, опис процеса такође мора садржати:

- **Опис производа** који настају на крају појединих активности или целог процеса као целине. Примери производа могу бити дизајн архитектуре софтвера, модел података, софтверска компонента, модификован софтвер на основу захтева за одржавањем итд.
- **Опис улога** одсликава позиције и одговорности људи ангажованих на реализацији процеса. Улоге зависе од типа процеса, а могу бити програмер, софтверски архитекта, аналитичар, инжењер за тестирање, инжењер за обезбеђење квалитета, менаџер пројекта, менаџер конфигурације итд.
- **Опис предуслова за извршење процеса** се односи на услове који треба да буду испуњени да би се процес могао реализовати. На пример, реализација дизајна архитектуре софтвера подразумева да су софтверски захтеви јасно специфицирани и валидирани од стране корисника.
- **Опис стања након извршења процеса** се односи на испуњене услове након реализације процеса. На пример, након извршења процеса дизајна архитектуре софтвера сви модели који описују архитектуру морају бити прегледани и валидирани.
- **Опис индикатора и приступа за мерење перформанси процеса** да би се могао утврдити ниво ефикасности процеса.
- **Опис метода и алата** који се користе за реализацију задатака у оквиру активности.

Стварни софтверски процеси су секвенце техничких, управљачких и комуникационих активности са циљем да се реализује софтверски пројекат развоја или одржавања софтвера. Свака софтверска организација креира скуп процеса у складу са својим потребама, могућностима и пословном

стратегијом. У животном циклусу софтвера се јављају процеси са различитим наменама, а основна категоризација свих процеса је на:

- **Примарни процеси** су процеси развоја, одржавања и употребе софтвера. Процеси развоја су процес софтверских захтева, процес дизајна софтвера, процес конструкције и процес тестирања.
- **Процеси подршке** се користе привремено или континуирано током животног циклуса софтвера као подршка примарним процесима. Овде спадају процеси који се односе на управљање конфигурацијом, обезбеђење квалитета, верификацију и валидацију.
- **Организациони процеси** обезбеђују подршку софтверским процесима, а обухватају обуке и тренинг, мониторинг и мерење процеса, управљање инфраструктуром (нпр. развојна окружења, софтверске библиотеке итд.), управљање портфолиом софтверских производа и поновном употребом, унапређење организационих процеса, управљање моделима животног циклуса.
- **Међупројектни процеси** су процеси који обухватају више пројеката, а обухватају процес поновне употребе, управљање производним линијама, инжењеринг домена итд.

Извршилац процеса (агент) је особа или машина која извршава процес са циљем остваривања постављених захтева. Ако је извршилац људска особа, онда она интерпретира документ или скрипт са описом процеса. Ако је извршилац машина (програм или софтвер) онда она интерпретира процесни програм. **Власник процеса** је особа или организација која поставља захтеве и циљеве процеса и одговорна је за њихово остварење.

4.2 Категорије софтверских процеса

Према стандарду *ISO/IEC/IEEE 12207:2008* који дефинише софтверске процесе у животном циклусу софтвера, процеси се могу поделити на примарне процесе, процесе подршке и организационе процесе (слика 4.2). Између ових процеса постоји повезаност, а њихова имплементација зависи од специфичности организације која их реализује. Организација може имплементирати и друге организационе процес који обезбеђују успостављање, контролу и побољшавање процеса у животном циклусу софтвера.

Дефинисање, имплементација и управљање добро дефинисаним процесима омогућује софтверској организацији да изврши транзицију од хаотичног окружења у којем се процеси реализују у слободној форми (ад-хок реализација) до окружења у којем се помоћу контроле и комуникације обезбеђује успешна имплементација процеса и побољшање перформанси организације, њених производа и услуга. У дефинисању и имплементацији процеса софтверске организације се могу ослонити на препоруке међународних стандарда, али могу процесе саме дефинисати и

Слика 4.2: *Категорије софтверских процеса према стандарду ISO/IEC/IEEE 12207:2008*

имплементирати у складу са својим специфичним потребама (што и јесте случај код малих и микро организација).

4.2.1 Примарни процеси у животном циклусу софтвера

Примарни процеси током животног циклуса софтвера треба да обезбеде покретање идеје о софтверском производу,развој, употребу и одржавање. Ове процесе реализују све заинтересоване стране за креирање софтвера, његов развој и употребу, а то су чланови организације која користи софтвер и чланови организације која развија софтвер. Примарни процеси су: аквизиција, снабдевање, развој, употреба и одржавање.

Аквизиција (*Acquisition*). Процес аквизиције дефинише активности организације која жели да набави софтверски производ или услугу. Процес започиње дефиницијом потребе за набавком система, софтверског производа или софтверске услуге. Процес се наставља припремом и издавањем захтева за предлог, избором добављача и управљањем поступком набавке до прихватања система, софтверског производа или софтверске услуге. Набављени производи или услуге треба да задовоље иницијално постављене потребе организације. Препоруке за аквизицију софтвера независно од типа, сложености и критичности софтвера се могу пронаћи у стандарду *IEEE 1062-2015 - IEEE Recommended Practice for Software Acquisition*. Аквизиција треба да обезбеди јасно дефинисање софтверских захтева који су у складу са потребама организације.

Снабдевање (*Supply*). Процес снабдевања дефинише активности добављача софтвера (софтверске организације), а директно се ослања на документе и уговоре креиране у процесу аквизиције софтвера који покреће

клијентска организација. Процес обухвата утврђивање процедура и ресурса потребних за управљање пројектом испоруке софтвера или софтверске услуге, што укључује и израду пројектног плана. Циљеви процеса снабдевања су јасно дефинисана комуникација са клијентом, дефинисање уговора са прихваћеним захтевима клијента, успостављање механизма за праћење потреба клијента и статуса реализације испоруке софтвера или софтверске услуге. Препоруке за проверу, преглед и прихватање софтвера приликом испоруке се могу пронаћи у стандарду *IEEE 1028-2008 - IEEE Standard for Software Reviews and Audits*.

Развој (*Development*). Процес развоја софтвера дефинише активности и послове софтверских инжењера који учествују у развоју. Процес подразумева управљање на нивоу пројекта, што треба да буде усклађено са процесом управљања и процесом инфраструктуре у софтверској организацији. Процес се завршава испоруком софтверског производа или услуге клијенту. Циљеви процеса развоја су идентификација софтверских захтева који су у складу са потребама клијента, развој стратегије развоја и испоруке софтвера, развој архитектуре система, развој специфичних елемената система, интеграција елемената у систем, тестирање система и валидација од стране корисника. Према стандарду *ISO/IEC/IEEE 12207:2008*, основне активности у процесу развоја софтвера су: имплементација процеса развоја, анализа системских захтева, дизајн архитектуре система, анализа софтверских захтева, дизајн архитектуре софтвера, детаљни дизајн софтвера, кодирање и тестирање софтвера, интеграција софтвера, интеграција система, тестирање система, инсталирање софтвера и прихватање софтвера од стране клијента. Активности у процесу развоја софтвера су детаљно описане у међународним стандардима, а стандард *IEEE 1074-2006 - IEEE Standard for Developing a Software Project Life Cycle Process* даје препоруку за дефинисање пројекта који управља процесом развоја и одржавања софтвера.

Употреба (*Operation*). Процес употребе софтвера обухвата активности и задатке људи који користе софтвер, али и оперативну подршку корисницима софтвера. *Оператор*, као особа која је задужена за управљање процесом употребе софтвера код клијента, је задужен за управљање свим активностима које се односе на употребу софтвера, укључујући тренинге корисника и управљање неопходном инфраструктуром. Циљеви процеса употребе су да се сагледају сви потенцијални ризици у увођењу и употреби софтвера, да се обезбеди употреба софтвера у складу са организационим процедурама, да се обезбеди оперативна подршка свим корисницима и испорука свих сервиса које софтвер обезбеђује у складу са потребама корисника. Припрема корисничког упутства и упутства за оператора су значајан део подршке у складу са стандардом *IEEE 1063-2001 - IEEE Standard for Software User Documentation*.

Одржавање (*Maintenance*). Процес одржавања софтвера садржи активности и задатке које особље задужемо за одржавање (*maintainers*) реализује са циљем модификације софтвера како би остао употребљив за кориснике, а да се уједно очува и интегритет софтверског производа. Управљање процесом одржавања се реализује на нивоу пројекта, а у једноставнијим случајевима на нивоу појединачних задатака. Циљеви процеса одржавања су модификација софтвера у складу са оперативним процедурама

и потребама корисника, одржавање валидне и ажурне документације о производу, управљање верзијама софтвера код различитих клијената, и адаптација софтвера за измењене услове у радном окружењу код клијента.

4.2.2 Процеси подршке у животном циклусу софтвера

Процеси подршке обезбеђују специфичан и фокусиран скуп активности које помажу у имплементацији примарних процеса у животном циклусу софтвера. Ови процеси имају значајан утицај на успешност реализације пројеката и квалитет софтверских производа и услуга. Процеси подршке су: документовање, управљање конфигурацијом, обезбеђење квалитета, верификација, валидација, заједнички преглед, провера и решавање проблема.

Документовање (*Documentation*). Документовање је процес који обезбеђује бележење информација које настају током животног циклуса софтвера. Активности које се спроводе су планирање, дизајн, креирање, дистрибуција и одржавање докумената који су неопходни различитим учесницима у животном циклусу софтвера (менаџери, програмер, особље за одржавање, корисници, итд.). Процес документовања треба да обезбеди да се идентификују сви неопходни документи у неком процесу или пројекту, да се специфицира њихов садржај, да се документи креирају и публикују у складу са одговарајућим стандардима и процедурама софтверске организације.

Управљање конфигурацијом (*Configuration management*). Процес управљања конфигурацијом се спроводи током целог животног циклуса софтвера са циљем да се идентификују, дефинишу и укључе у базну конфигурцију система сви неопходни елементи конфигурације (Configuration Item (CI)). Управљање конфигурацијом такође обезбеђује контролу модификација и издавања елемената конфигурације, управљање захтевима за модификацијом и праћење статуса елемената конфигурације, као и комплетност, конзистентност и коректност свих елемената конфигурације софтверског система. Управљање конфигурацијом обезбеђује интегритет сваког софтверског елемента и софтверског производа током целог животног циклуса.

Обезбеђење квалитета (*Quality assurance*). Процес обезбеђења квалитета (Quality Assurance (QA)) треба да обезбеди да сви процеси и производи током животног циклуса задовољавају специфичне захтева и да су усклађени са постављеним плановима у организацији или у оквиру пројекта. Такође, обезбеђење квалитета треба да осигура евалуацију и тестирање софтверског производа, при чему користи резултате других процеса подршке, као што су верификација, валидација, заједнички преглед, провере и решавање проблема. Све активности у оквиру обезбеђења квалитета морају бити идентификоване и јасно дефинисане у плану спровођења мера на обезбеђењу квалитета процеса и производа, што је прописано стандардом *IEEE 730-2014 - IEEE Standard for Software Quality Assurance Processes*.

Верификација (*Verification*). Верификација је процеес којим се утврђује да ли производ неке активности испуњава захтеве или услове који су постављени током спровођења претходних активности. Овај процес укључује

активности као што су анализа, прегледање и тестирање. Верификација треба да буде што раније интегрисана у пројекте који се спроводе током животног циклуса софтвера, и да обезбеди потврду испуњености захтева на основу објективне евиденције (на пример мерењем одређених параметара производа). Идентификација критеријума за верификацију је кључна за добијање валидних резултата и елиминисање свих идентификованих аномалија производа.

Валидација (*Validation*). Валидација је процес који има за циљ да утврди да ли финални софтверски производ (систем) задовољава постављене захтеве и испуњава своју намену на основу испитивања и пружања објективних доказа. Валидације се обично реализује тестирањем на различитим нивоима и применом различитих приступа.

Заједнички преглед (*Joint review*). Процес заједничког прегледа се користи за евалуацију статуса производа током трајања уговора између софтверске организације и клијентске организације. Преглед се може реализовати на нивоу управљања и на техничком нивоу, а базира се на заједничком раду добављача софтвера и клијента. Преглед се може односити на анализу софтверских захтева, дизајн архитектуре, детаљни дизајн, интеграцију система и прихватање производа од стране клијента.

Провера (*Audit*). Процес провере утврђује степен усаглашености организације и пројекта са плановима, захтевима и уговором. Проверу може радити независна организација, или одговарајући експерти у софтверској организацији ако постоје. Најчешће мале софтверске организације не спровеоде екстерне провере, а интерне провере реализују самостално или у договору са клијентом. Резултат процеса провере је листа идентификованих проблема која се прослеђује процесу у оквиру којег се врши решавање проблема. Провере се могу поновити да се оцене резултати корективних акција пропиних током претходне провере. Провера треба да буде подржана јасно дефинисаним планом мерења параметара процеса и производа у складу са стандардом *ISO/IEC/IEEE 15939:2017 Systems and software engineering - Measurement process*.

Решавање проблема (*Problem resolution*). Процес решавања проблема има за циљ да обезбеди анализу и решавање проблема током животног циклуса софтвера без обзира на њихову природу и извор. Међутим, значајан део проблема у пракси решавају инжењери који су открили проблем у производу, без ослањања на дефинисане процесе и процедуре у организацији. Ад-хок решавање проблема, без усклађивања са било каквим процесом у организацији, је нарочито карактеристично за мале софтверске организације и тимове где појединци дневно решавају настале проблеме. Процес треба да омогући да се проблеми идентификују и реше на време, као и да се креира потребна документација која се може употребити за потребе учења у организацији, али и за спровођење организационих и финансијских процеса.

4.2.3 Организациони процеси у животном циклусу софтвера

Организациони процеси се морају успоставити пре имплементације примарних процеса и процеса подршке у животном циклусу софтвера. Да би то остварила, софтверска организација треба да омогући управљање на нивоу пројеката, успостави потребну инфраструктуру за подршку имплементацији процеса, планира могућности прилагођења и побољшавања процеса, и обезбеди квалитет процеса кроз заједничке прегледе, верификацију, валидацију и провере. Организациони процеси треба да пруже јасне индикације да ли нови, модификовани или процес који је делегиран другој организацији на имплементацију (*outsourced processes*) треба подржати.

Организациони процеси у животном циклусу софтвера су: управљање, процес инфраструктуре, побољшавање и тренинг.

Управљање (*Management Process*). Процес управљања садржи генеричке активности и задатке који омогућују управљање процесима. Активности које садржи овај процес су иницијализација процеса и дефинисање обима (домена), планирање, извршење и контрола, преглед и евалуација, и затварање процеса или пројекта. Основни циљ управљања је да се јасно дефинише обим рада, а потом да се обезбеди планирање, праћење и мерење неопходних задатака и ресурса за комплетирање пројекта. Такође је неопходно дефинисати жељене перформансе производа и начине мерења квалитета, као и потенцијалне ризике и начин да се они предупреде или њихов утицај минимизира. Ефикасно планирање и управљање пројектима у животном циклусу софтвера је подржано стандардом *IEEE/ISO/IEC 16326-2009 - Systems and Software Engineering - Life Cycle Processes - Project Management*.

Процес инфраструктуре (*Infrastructure Process*). Процес инфраструктуре обезбеђује и одржава инфраструктуру потребну за реализацију осталих процеса. Инфраструктура може укључити софтвер, хардвер, алате и методе, стандарде и регулативу који подржавају остале процесе. Циљеви овог процеса су да се формира и одржава радно окружење софтверског инжењерства неопходно за реализацију пројеката. Овакво радно окружење треба да буде прилагодљиво за реализацију пројекта и да обезбеди подршку за пројектни тим независно од начина имплементације активности.

Побољшање процеса (*Improvement Process*). Овај процес обезбеђује успостављање, процењивање, мерење, контролу и побољшање осталих процеса у животном циклусу софтвера. За побољшање се користе историјски подаци о процесу који указују на начин извршавања процеса и потенцијалне проблеме, недостатке и ризике. Циљеви овог процеса су: дефинисање и успостављање добро дефинисаног скупа стандардних процеса у софтверској организацији, детаљна идентификација и спецификација активности и задатака за све процесе, подршка за прилагођавање стандардних процеса специфичним ситуацијама и потребама, управљање подацима о скупу стандардних процеса, разумевање предности и недостатака стандардних

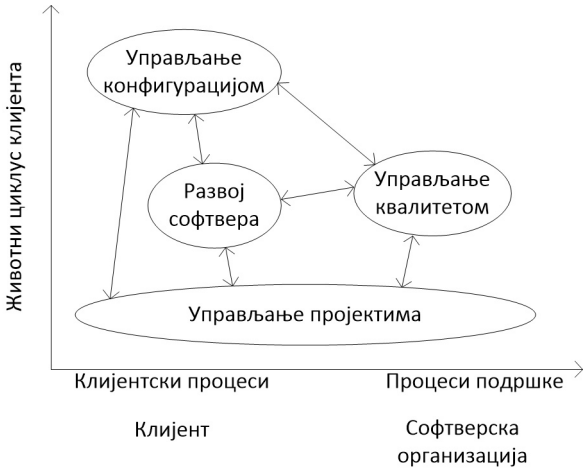
процеса, и имплементација планског праћења активности побољшања процеса у организацији.

Тренинг (*Training Process*). Процес тренинга обезбеђује да у организацији увек постоји обучено особље за све послове у животном циклусу софтвера. С обзиром да послови у софтверском инжењерству захтевају особље са одговарајућим знањем и вештинама, неопходно је тренинг планирати и спроводити на време и у складу са потребама организације или специфичних пројеката. Такође је потребно успоставити формалне процедуре за селекцију и транзицију нових инжењера на одговарајућа задужења у организацији. Мерење индивидуалних перформанси особља након тренинга треба редовно спроводити да би се указало на користи и проблеме који постоје са особљем, што може значајно побољшати перформансе целе организације

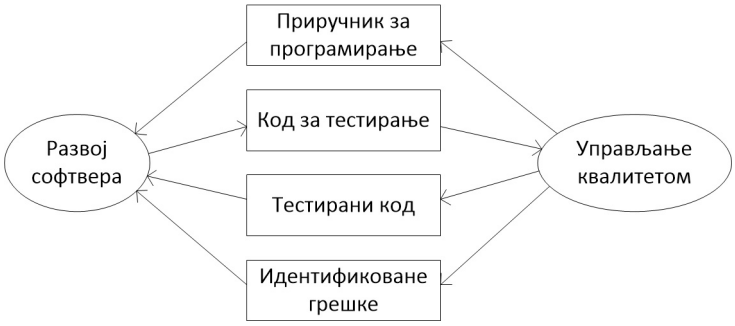
4.3 Кластеризација софтверских процеса

Основни проблем при моделовању софтвера је идентификовати кључне процесе (*core processes*) које треба најпре моделовати, као и идентификовати интерфејсе између процеса. Први корак у правцу идентификовања основних процеса је да се јасно разграниче процеси који су интерни за софтверску организацију и процеси који зависе од окружења (корисници, клијенти, тржиште, законска регулатива, итд.). На тај начин се креира листа или шема свих процеса на високом нивоу апстракције (без детаља о реализацији процеса), што омогућује и хијерархијску организацију процеса (на пример према значају, ризицима или трошковима). Процеси се организују у кластере који садрже процесе који имају сличну логичку намену (нпр. процеси развоја, управљања квалитетом, управљања конфигурацијом итд.), а интерфејси представљају везе између кластера, што је представљено на слици 4.3. Кластери процеса су уређени према томе да ли су оријентисани ка клијентима или су процеси подршке који се реализују у софтверској организацији (X оса на слици 4.3), и према животном циклусу клијента (Y оса на слици 4.3).

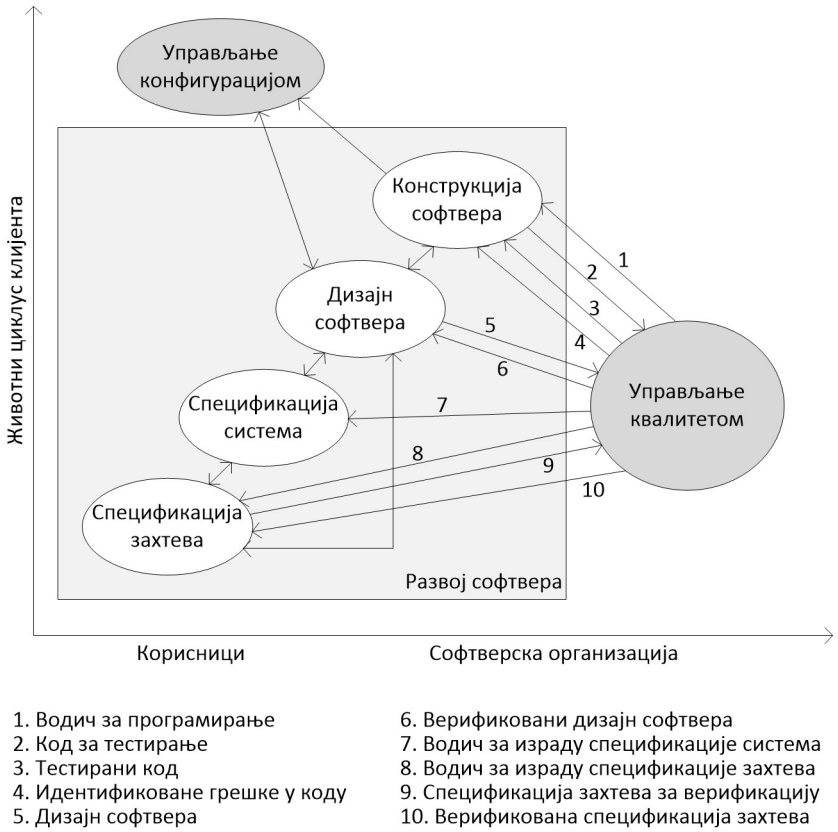
На најнижем нивоу је кластер са основним процесима управљања пројектима који се увек јавља на почетку животног циклуса софтвера. Кластер процеса који се односе на развој софтвера и управљање конфигурацијом софтвера се реализују тако да на њих значајан утицај имају клијенти, док се интерни процеси, као што је управљање квалитетом реализују искључиво у софтверској организацији. Кластери софтверских процеса су повезани ако процеси из тих кластера размењују информације, што је на слици 4.3 представљено линијама са стрелицама у оба смера. Пример повезаности кластера развоја софтвера и управљања квалитетом је приказан на слици 4.4, где се јасно уочавају правци размене информација или објеката. Процеси обезбеђења квалитета дефинишу упутства за програмирање, и врше тестирање кода, а потребне документе са информацијама или производе рада достављају процесима развоја софтвера. Процес развоја софтвера као резултат има код који се доставља на тестирање, што је у надлежности процеса за управљање квалитетом.



Слика 4.3: *Кластерска организација софтверских процеса*



Слика 4.4: *Интеракција између кластера развој софтвера и управљање квалитетом*



Слика 4.5: *Детаљни приказ кластера развоја софтвера и интерфејса ка кластерима управљања квалитетом и управљања конфигурацијом*

Сваки од основних кластера се може декомпоновати на подкластере који обједињују процесе нижег нивоа који су логички повезани. На тај начин се обезбеђује "фина" хијерархијска организација процеса. Оваква декомпозиција процеса подразумева да се сваки подкластер може повезати са другим основним кластерима процеса у оквиру софтверске организације. Детаљни приказ кластера развоја софтвера, са подкластерима и везама ка кластерима управљања квалитетом и управљања конфигурацијом је приказан на слици 4.5

Кластер управљања квалитетом садржи процесе који обезбеђују израду документације и водича за спровођење активности у оквиру организације (у другим кластерима), као што су водичи за израду спецификације захтева или водич за програмирање. Такође процеси у овом кластеру обезбеђују проверу, тестирање и верификацију објеката који настају током развоја софтвера (верификација спецификације захтева, тестирање кода). Процеси у овом кластеру обезбеђују коректно спровођење активности у оквиру других кластера.

Кластер процеса који се односе на управљање софтверским пројектима садржи опис модела процеса као што су припрема пројектног плана, преглед

пројектног плана, контрола трошкова, процена ризика, анализа успеха и неуспеха током реализације пројеекта (*post mortem analysis*) итд.

Кластер процеса за управљања конфигурацијом софтвера садржи моделе процеса који се односе на издавање верзија софтвера, интеграцију софтверских елемената, управљање променама, управљање елементима и базним линијама у софтверу, инспекцију конфигурације, руковање догађајима који се односе на конфигурацију софтвера и извештавање о статусу конфигурације. Ови процеси су доступни свим софтверским пројектима који се реализују у оквиру софтверске организације. Управљање конфигурацијом софтвера обезбеђује јединствену идентификацију, контролисано складиштење, контролу промена и извештавање о статусу софтверских компоненти и производа током животног века система.

4.4 Моделовање софтверских процеса

Моделовање софтверских процеса има за циљ да се концептуално сагледају, опишу и дефинишу софтверски процеси, што би требало да омогући креирање процеса који се могу понављати, али и рационалније управљање софтверским пројектима. Треба јасно нагласити разлику између модела животног циклуса софтвера и модела софтверских процеса, па се у складу са тим модел софтверског процеса дефинише као скуп активности, објеката, трансформација и догађаја који омогућују еволуцију софтвера. Према томе, модел софтверског процеса представља формални апстрактни опис архитектуре и дизајна елемената и активности процеса софтвера применом одговарајућег језика за моделовање процеса.

Модел процеса је апстракција која се односи на парцијалну и поједностављену представу појединих аспеката процеса. Врло често се због тога модел реализује као скуп подмодела где сваки подмодел описује један поглед на софтверски процес. Моделовање софтверских процеса има следећу намену:

- **Документовање**. Најчешће се модел креира да би се обезбедило јасно размевање намене и карактеристика процеса.
- **Анализа и побољшање модела**. Анализа ефикасности и ефективности модела процеса се може искористити као основа за побољшање карактеристика модела. Анализа модела се најчешће врши симулацијом процеса.
- **Побољшање процеса**. Подаци добијени креирањем модела и анализом његових карактеристика се могу искористити за побољшање самих процеса. Побољшање се може реализовати применом стандарда или водича добре праксе, али и једноставним отклањањем уочених недостатака.
- **Извршавање процеса**. Модели процеса се могу користити и за реализацију процеса у пракси, што се користи код процеса са јасно дефинисаним током извршења (*workflow-oriented style*). У таквим

случајевим је могуће аутоматизовати поједине активности у процесу или цео процес.

- **Мониторинг и контрола процеса**. Модели треба јасно да укажу на начин мониторинга и контроле извршавања процеса, употребу ресурса, контролу ризика и контролу излаза.

Избор модела процеса зависи од контекста софтверске организације, специфичности пројекта, клијената, као и од избора метода и алата које се користе. Због тога не постоји јединствен и универзалан начин моделовања који се може применити у свим ситуацијама. На пример, моделовање софтвера који интензивно користи базе података (на пример софтвер за организацију магацина за робу) је значајно другачије од моделовања софтвера који се користи у уграђеним системима (на пример у аутомобилу или веш машини). Због тога се за сваку специфичну ситуацију бира најпогоднији модел процеса. Модел софтверског процеса је опис софтверског процеса, а може се представити помоћу различитих записа (текстуални, графички) на различитим нивоима апстракције. Основни елементи модела софтверског процеса су:

- Опис активности која се може јединствено идентификовати, или опис групе активности.
- Опис тока производа што укључује опис улаза и излаза процеса.
- Опис контроле тока процеса, тј. секвенце извршавања активности у оквиру процеса.
- Опис техника, метода и алата који се користе у процесу.
- Опис улога и њихових међусобних релација у процесу.

Класификација модела софтверских процеса се најчешће врши на основу примене одређеног језика за моделовање, али ако се у обзир узме циљ моделовања процеса могу се разликовати два типа модела процеса:

> **Прескриптивни модели** прописују како нешто треба да буде урађено (извршено). Ови модели имају намену да укажу на све релевантне елементе које процес треба да обухвати, и заправо представљу модел идеалног процеса базираног на најбољој пракси. Прескриптивни модел представља жељено и препоручено извршавање процеса.
>
> **Дескриптивни модели** описују како се нешто ради (извршава) у стварном окружењу. Ови модели се креирају посматрањем процеса који се реализују и представљају опис имплементације процеса у посматраном окружењу.

Оба типа модела се могу применити у истој софтверској организацији, а разлика је у њиховој намени, а не у садржају. Дескриптивни модели описују шта и како људи раде, док прексриптивни модели упућују људе како треба да раде. Прескриптивни модели врло често подразумевају промену навика људи, што резултира избегавањем њихове примене или импровизацијом приликом примене.

Са повећаним захтевом за флексибилношћу и адаптивношћу софтверских процеса, подржану развојем савремених метода и алата, као и са трендом континуиране испоруке софтвера, појављује се све већа потреба за универзалним нотацијама за моделовање, симулацију и извршавање процеса. Иако су предложене различите методе и нотације за моделовање процеса и даље не постоји стандардизована парадигма и језик који се примењују за моделовање у индустријској пракси.

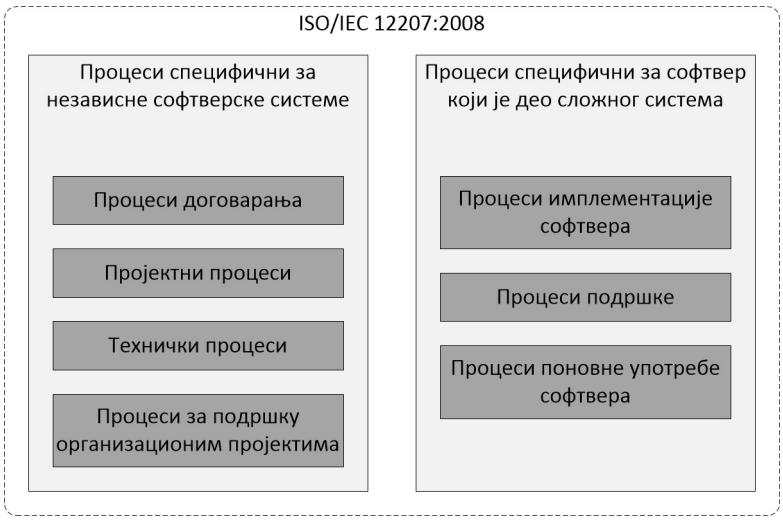
4.4.1 Прескриптивни модели

Прескриптивни модели описују захтевани начин имплементације процеса, тј. прецизно дефинишу како неки процес треба реализовати (спецификација захтева, дизајн архитектуре, тестирање итд.). Основни проблеми у имплементацији прескриптивних модела су:

- **Опсег валидне примене модела се мора знати**, што често није случај у софтверским организацијама због различитих контекста примене (потпуно је другачији контекст у малој софтверској фирми која развија веб апликације и у великој софтверској фирми која развија софтвер за ауто индустрију).
- **Утицај примене модела мора бити познат**, што подразумева сазнање о ефектима и побољшањима које примена доноси (смањење броја дефеката, боље искоришћење људских ресурса итд.). Ако се не могу сагледати утицај и ефекти примене прескриптивних модела могу се појавити непредвиђене ситуације и деградација појединих перформанси процеса који се тренутно имплементирају. Сазнања о ефектима примене модела су врло специфична за сваку софтверску организацију.
- **Ниво поверења мора бити познат за имплементацију модела**, што подразумева податке о евалуацији успешности имплементације модела. За сваку софтверску организацију са специфичним радним окружењем се морају прикупити подаци о евалуацији примене модела да би се добили показатељи нивоа поверења у прескриптивни модел.
- **Модел мора бити прилагодљив**. Пошто је прескриптивни модел приказ идеалне слике процеса која се ретко може применити у софтверским организацијама, неопходно је модел прилагодити специфичностима сваке софтверске организације (специфичност људског фактора, коришћене методе и алати, пословне стратегије и организација фирме итд.).

Прескриптивни модели процеса се најчешће базирају на стандардима, а циљ је да се процеси усагласе са препорукама које се могу наћи у стандардима. Стандард који се најчешће користи за прескрипцију софтверских процеса је *ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes*.

Међународни стандард *ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes* предлаже радни оквир за процесе у животном циклусу софтвера (од појављивања прве идеје за развој новог



Слика 4.6: *Преглед група процеса у животном циклусу софтвера према ISO/IEC 12207:2008 стандарду*

софтвера до повлачења софтвера из употребе). Радни оквир садржи јасно дефинисану терминологију и препоруке за примену у индустрији. Стандард групише типичне процесне активности у седам група, што је приказано на слици 4.6. На слици се могу уочити групе процеса које се односе на независне софтверске системе, и групе процеса које се односе на процесе специфичне за софтвер у склопу сложенијих система.

Процеси специфични за независне софтверске системе. *Процеси договарања* је група процеса који су неопходни за постизање договора између организација, а ту се налазе процеси аквизиције и испоруке. Аквизиција се односи на набавку потребних софтверских компоненти, алата, стандарда и других елемената потребних за развој софтвера, док се процес испоруке односи на испоруку софтвера клијентима. *Пројектни процеси* се односе на класичне послове управљања пројектима, као што су планирање, управљање ризицима, управљање информацијама, управљање конфигурацијом и мерење. *Процеси за подршку организационим пројектима* су процеси који обезбеђују подршку реализацији пројеката на стратешком нивоу, где спадају управљање инфраструктуром, управљање портфолиом пројеката, управљање људским ресурсима и управљање квалитетом. *Технички процеси* су процеси који се односе на развој софтвера, а укључују процес прикупљања захтева, процес дизајна архитектуре софтвера, имплементацију, интеграцију, тестирање, испоруку и одржавање софтвера.

Процеси специфични за софтвер који је део сложног система. *Процеси имплементације софтвера* описују активности које је неопходно спровести да би се креирао специфични елемент система који се имплементира као софтвер. Овде спадају процеси анализе софтверских захтева, дизајна архитектуре, имплементације, интегрције и тестирања. *Процеси подршке* су процеси који се односе на управљање документацијом, управљање конфигурацијом, обезбеђење квалитета, верификацију и

валидацију, прегледе и провере, као и решавање проблема. *Процеси поновне употребе софтвера* се односе на процес поновне употребе софтверских компоненти и инжењеринг домена (домени употребе софтвера).

Референтни модел процеса представљен на слици 4.6 не представља приступ конкретне имплементације процеса, не прописује одређени модел животног циклуса софтвера или избор технологија и метода. Референтни модел треба свака софтверска организација да усвоји на бази својих потреба, интерне организације и домена пословања.

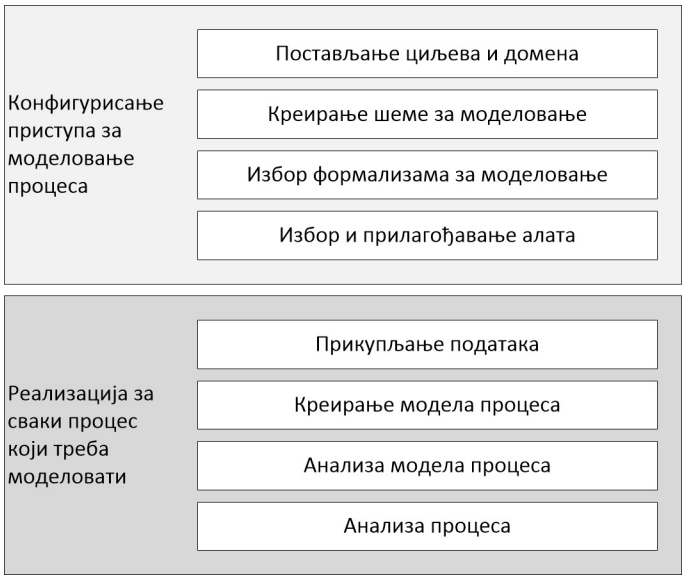
4.4.2 Дескриптивни модели

Успешно пословање великог броја софтверских фирми је базирано на специфичним процесима које су развили у пракси. Специфични процеси, који нису у складу са препорукама и стандардима, одсликавају реално стање праксе и доносе предности тим фирмама на тржишту. Дескриптивни описи процеса омогућују њихово боље разумевање, конзистентнију и поузданију имплементацију и побољшање идентификованих критичних сегмената процеса.

Дескриптивни модели могу бити неформални и формални, што обично зависи од намене моделовања. *Неформални модели* се обично користе за разумевање процеса а са циљем процењивања и побољшавања процеса или аутоматизације процеса или појединих делова процеса. *Формални модели* се користе за процењивање и предикцију процеса, што подразумева квантификацију одређених параметара процеса. Предикција процеса се односи на анализу модела процеса са циљем предикције будућег понашања процеса што се може искористити за планирање у софтверској организацији. Процењивање софтверских процеса се користи за анализу и поређење појединих активности у процесу (нпр. њихова ефикасност, стабилност и поузданост), а подаци добијени процењивањем постају основа за побољшавање процеса.

Креирање дескриптивног модела почиње прикупљањем информација о процесу, тј. информација о имплементацији процеса. Прикупљање информација се реализује кроз интервјуе, посматрање праксе, анализу докумената или производа. Када се прикупи довољно информација о процесу прелази се на моделовање процеса. Моделовање се најчешће реализује применом графичких језика који подржавају комуникацију између различитих интересних страна (програмери, менаџери, аналитичари итд.). Цео поступак је итеративан, што значи да се након моделовања може захтевати прикупљање додатних података о процесу. Итеративни поступак се може понављати све док се не стигне до задовољавајућег модела процеса кроз поступак валидације. Поступак дескриптивног моделовања процеса је приказан на слици 4.7.

Дескриптивно моделовање се према слици 4.7 састоји од две фазе. Прва фаза је дефинисање и конфигурисање приступа за моделовање софтвера, током које се организација припрема за опис процеса. Прва фаза подразумева следеће активности које је неопходно спровести пре него што се започне са моделовањем:



Слика 4.7: *Поступак дескриптивног моделовања процеса*

- **Постављање циљева и домена** које обухвата модел (ко моделује, који процеси се мподелују, која су очекивања од модела, итд). Ова активност такође укључује утврђивање детаља процеса (актери, ограничења, интеракције са окружењем, итд.) које треба моделовати
- **Креирање шеме за моделовање** се односи на креирање или дефинисање скупа концепата помоћу којих ће се процес моделовати. Примери концепата који се најчешће користе за моделовање процеса су: активност, производ, улоге, ток производа између активности итд. Скуп концепата који се користе за моделовање се обично назива **шема моделовања**. За шему моделовања је битно да обухвати све аспекте процеса које је потребно моделовати, да обезбеди одговарајући ниво детаља приликом моделовања, и да резултат моделовања буде јасно приказан и употребљив за што шири круг корисника.
- **Избор формализама за моделовање** подразумева дефинисање нотације којом се јасно приказују концепти који чине шему моделовања. Нотације за моделовање се могу класификовати према више димензија: графичке или текстуалне, формалне или неформалне, детаљне или мање детаљне и предодређене (фиксне) или проширive. Избор нотације зависи од конкретног пројекта моделовања процеса.
- **Избор и прилагођавање алата** треба да обезбеди следеће: подршку за одабрану нотацију за моделовање, чување и поновна употреба модела, креирање алтернативних репрезентација модела које су намењене различитим заинтересованим странама у моделовању (менаџери, програмери, аналитичари, корисници, итд.).

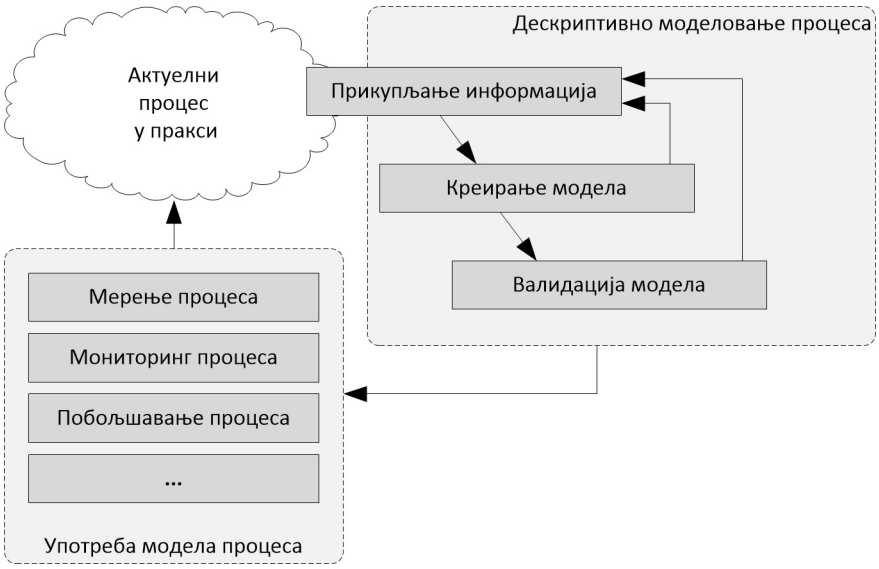
Фаза реализације моделовања се према слици 4.7 примењује за сваки процес који је одабран приликом дефинисања циљева и домена моделовања. Активности у овој фази су:

- **Прикупљање података** се односи на прикупљање информација које су неопходне за моделовање одабраних процеса. Ове информације се односе на: елементе процеса (ентитети, активности, улоге, производи, итд.), релације међу елементима процеса и својствена понашања елемената процеса. Прикупљање информација се може реализовати раличитим техникама: интервјуи, фокусне групе, посматрање праксе, анализа постојећих производа, анализа историјских података доступних у базама или документима.
- **Креирање модела процеса** се базира на прикупљеним информацијама о актуелним процесима. Моделовање треба започети са моделовањем производа, наставити са моделовањем активности, затим моделовати улоге људи и на крају применити ограничења на елементе модела.
- **Анализа модела процеса** је важна активност са циљем да се смањи ризик од грешака у моделу, што је посебно важно за моделе комплексних процеса. Типичне грешке које се јављају у моделима су: референцирање непостојећег ентитета у моделу, неконзистентност предуслова за извршавање процеса или појединих активности, неконзистентна употреба имена у моделу (референцирање једног објекта помоћу различитих имена) и непотпуни описи. Анализа модела подразумева анализу *конзистентности* (нема контрадикције у моделу), *комплетности* (све релевантне информације су у моделу) и *динамичности* (потенцијални проблеми приликом извршавања модела).
- **Анализа процеса** подразумева поређење модела процеса са актуелним процесом у пракси. Циљ је да се евидентирају недостаци модела и његова усаглашеност са актуелним процесом.

Дескриптивно моделовање процеса је итеративан поступак који је приказан на слици 4.8. Итеративност поступка моделовања омогућује побољшање (прочишћавање) модела тако да он што тачније приказује актуелни процес. Са друге стране, модел се може користити као основа за креирање плана мерења и мониторинга извршавања процеса, али и за планирање побољшавања процеса. У основи дескриптивног моделовања је прикупљање информација о процесима, моделовање процеса и потом преглед модела од стране људи који реализују процес у пракси, чиме се врши валидација модела.

Ограничења које се јављају приликом дескриптивног моделовања процеса су:

- **Ограничења у доступности људи** се односи на доступност процесних инжењера који раде моделовање процеса (експерти, консултанти, истраживачи са академије), али и људи који реализују процес у пракси (заузети свакодневним пословима). Најчешће су људи који реализују процес доступни само одређено време.

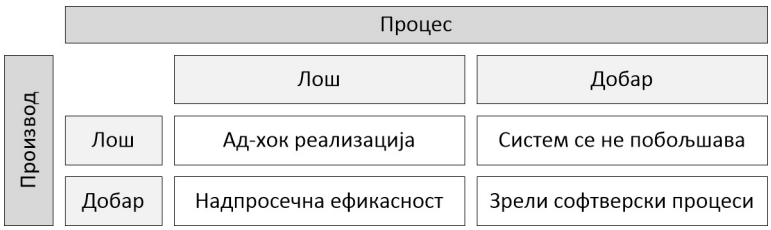


Слика 4.8: *Итеративан поступак дескриптивног моделовања процеса*

- **Географска ограничења** се односи на организације у којима су процеси имплементирани на различитим локацијама, па се често у тим случајевима користе методе базиране на савременим комуникационим технологијама (видео конференције, скајп сесије, електронска пошта, итд.).
- **Неусаглашеност процеса са документацијом** се често дешава пошто се процес имплементира на начин који није у складу са документацијом, а у зависности од људи који га имплементирају.
- **Поверљивост информација** се односи на поверљивост појединих информација које су потребне за моделовање процеса, што укључује специфичне информације о самој организацији или о људима који реализују процес. Овакве информације се морају третирати на посебан начин, који је безбедан за организацију и људе.
- **Артикулација информација** се односи на начин изношења информација од стране људи који реализују процес (употреба речи, конструкција реченица, специфична терминологија, итд.). Представљање информација треба да обезбеди разумевање модела процеса, а и самог процеса за све заинтересоване стране без обзира на њихово знање о детаљима процеса.

4.5 Процењивање и побољшање софтверских процеса

Побољшање софтверских процеса (*Software Process Improvement (SPI)*) је веома значајно за евалуацију и унапређење перформанси софтверских



Слика 4.9: *Однос квалитета софтверских процеса и производа*

организација. Захтеви тржишта за испоруком јефтиних и квалитетних софтверских производа у све краћим роковима подстичу софтверске организације да побољшају своју праксу, што се у већини случајева реализује побољшањем постојећих процеса. Побољшање праксе у софтверским организацијама се може остварити и увођењем нових, савремених технологија, што је у принципу значајно скупље и компликованије од побољшања постојећих процеса. Усавршавање запослених је такође начин побољшања праксе, али је ту проблем велика флуктуација радне снаге у софтверској индустрији, па инвестирање у том правцу може бити неисплативо. Дугорочно посматрано, побољшање процеса је најпоузданији и најисплативији начин побољшања праксе у софтверским организацијама.

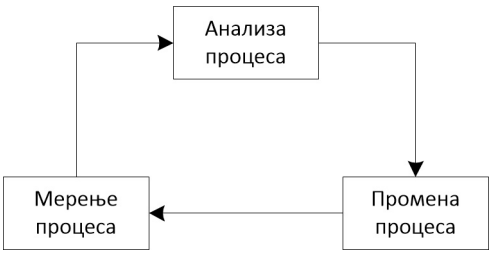
Побољшање процеса подразумева пре свега разумевање постојећих процеса и проналажење аспеката који се могу побољшати. Ако се побољшање процеса посматра као процес учења и делања у оквиру организације, тада се откривају стварни узроци проблема у процесима и они отклањају, што и јесте циљ побољшања процеса. На крају имплементације побољшања процеса, треба извршити анализу колико и на који начин побољшања имају ефекте у организацији, тако да стечена искуства остају доступна за наредне пројекте.

С обзиром да су софтверске организације базиране на тимовима и пројектима веома је битно да се побољшање процеса базира на активном учешћу запослених, и да се као основа за побољшање користи њихово искуство и знање. С обзиром да су процеси зависни од људи који у њима учествују, промене процеса могу унети значајне промене и трошкове у софтверској организацији. Због тога се побољшању процеса мора приступити плански и систематично.

Побољшање процеса је неопходно када организација нема дефинисане процесе, или има проблематичне процеси који утичу на квалитет пословања (квалитет производа и услуга). Лоше дефинисани и имплементирани процеси могу направити веће трошкове и негативно утицати на односе са клијентима. Однос квалитета софтверских процеса и производа је приказан на слици 4.9. Основна карактеристика софтверских организација које имају лош квалитет процеса и производа је ад-хок реализација процеса, што значи да се процес прилагођава свакој ситуацији која настане у организацији. Квалитетан производ са лошим процесима је могуће остварити само уз додатни (прековремени рад) и ако су запослени изузетни инжењери. Међутим да би се обезбедило трајно креирање квалитетних производа неопходно је имати квалитетне процесе, што указује да је неопходно процесе процењивати и побољшавати.

Утицај квалитета процеса на квалитет софтверских производа у великој мери зависи од величине софтверске организације. У великим организацијама које могу имати више географски дистрибуираних тимова квалитет процеса је од пресудног значаја, а посебно важни сегменти праксе су комуникација и интеграција производа које креирају различити тимови. У оваквим организацијама пројекти врло често трају дуго па се и тимови могу мењати, што потврђује пресудну улогу квалитетно реализованих процеса. У мањим пројектима које обично реализују мале софтверске организације, тим има већи значај од процеса, па се у оваквим организацијама обично имплементирају агилне методе. Тим који чине искусни и стручни чланови ће произвести одличан производ без обзира на квалитет процеса. С обзиром да имплементација процеса у великој мери зависи од величине и типа софтверске организације, имплементација и побољшање процеса се мора посматрати у односу на одређене атрибуте процеса. Атрибути процеса који се могу побољшати су:

- **Разумљивост** (*Understandability*) се односи на то колико је јасно процес дефинисан и колико лако се може разумети од стране запослених у софтверској организацији, али и од стране клијената (ако клијент разуме процес може имати више поверења у квалитет производа и услуге).
- **Стандардизованост** (*Standardization*) се односи на ниво усклађености са стандардним генеричким процесом или са препорукама најбоље праксе из стандарда. Чест је случај да клијенти захтевају да софтверске организације имају процесе усклађене са међународним стандардима (на пример ISO 9000 или ISO/IEC/IEEE 12207-2008).
- **Видљивост** (*Visibility*) се односи на начин дефинисања активности у процесу које резултирају јасним резултатима, тако да се и прогрес процеса и резултати споља могу лако пратити и евидентирати.
- **Мерљивост** (*Measurability*) процеса омогућује да се прикупе подаци о реализацији процеса, што обезбеђује да се одређене карактеристике процеса могу мерити.
- **Подршка** (*Supportability*) се односи на ниво подржаности процеса софтверским алатима који обезбеђују праћење реализације процеса и аутоматизацију појединих активности.
- **Прихватљивост** (*Acceptability*) се односи на прихватљивост и применљивост процеса у пракси коју спроводе инжењери у производњи софтвера. Добро дефинисани процеси се лако прихватају и реализују у свакодневној пракси.
- **Поузданост** (*Reliability*) се односи на начин дизајнирања процеса тако да се избегну процесне грешке које могу довести до грешака у софтверским производима.
- **Робусност** (*Robustness*) у дизајну процеса обезбеђује да се реализација процеса може наставити иако се појаве проблеми у његовој реализацији.
- **Лакоћа одржавања** (*Maintainability*) се односи на могућност процеса да се мења (еволуира) у складду са променама у софтверској



Слика 4.10: *Циклус побољшања софтверских процеса*

организацији или у складу са идентификованим побољшањима која треба имплементирати.

- **Брзина** (*Rapidity*) се односи на брзину коју дизајн процеса обезбеђује у активностима животног циклуса софтвера.

У пракси је тешко остварити истовремено побољшање свих атрибута процеса. Врло често побољшање појединих атрибута води до погоршања других, па је потребно пронаћи баланс да се ипак оствари побољшање процеса. На пример, повећање брзине процеса обично смањује видљивост процеса. Исто тако, повећање способности одржавања и поузданости често се базирају на увођење локалних процедура и нестандардних алата у реализацију процеса, чиме се нарушава усклађеност процеса са дефинисаним стандардима.

Процес побољшања процеса је цикличан и укључује подпроцесе мерења, анализе и промене као што је приказано на слици 4.10. Мерење процеса треба да обезбеди да се кључни атрибути процеса мере у складу са циљевима организације или потребама специфичног пројекта, а то представља основу за сагледавање да ли су имплементирана побољшања ефикасна. Анализа подразумева процењивање постојећих процеса и идентификовање недостатак и слабости, на основу чега се идентификују потенцијална побољшања према постављеним циљевима. На основу анализе врши се измена постојећег процеса тако да се елиминишу недостаци.

Побољшање процеса увек треба да буде базирано на подацима који се мере на стварним процесима, чиме се добија увид у реално стање, а након имплементације побољшања мерењем се такође може одредити ниво успешности побољшања. Приликом дизајна пројекта побољшања процеса треба водити рачуна о следећем:

- Основни циљ је побољшање производа и услуга, а побољшање процеса је само начин да се то постигне.
- Усаглашеност пројекта побољшања процеса са пословном стратегијом и оријентацијом организације је од кључне важности за успех побољшања процеса.
- Рад на побољшању процеса треба посматрати као циклус континуираног учења у организацији у чему учествују сви запослени.

- Резултате и искуства стечена током пројекта побољшања процеса треба анализирати и систематизовати тако да буду доступни за будуће пројекте.

Пројекти побољшања процеса обично се реализују у три фазе: иницијализација, извршавање и затварање. Иницијализација подразумева да се сагледају сви аспекти везани за организацију и да се идентификују процеси које треба побољшати.

У фази иницијализације се јасно дефинише циљ пројекта, одлучује се о временском распореду активности, идентификују сви учесници у пројекту (у организацији и потенцијално ван организације ако је то потребно), и сагледавају се сви елементи контекста у којем се пројекат реализује (технички, организациони, финансијски, итд.). Такође се идентификују потенцијална ограничења (финансијска, временска, експертиза) која могу утицати на ток пројекта и његову успешност.

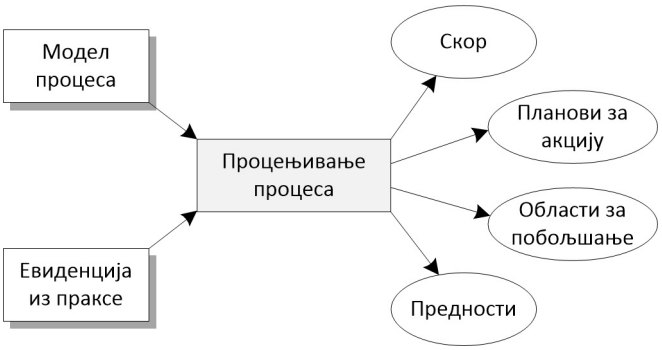
У фази извршавања пројекта се спроводе планиране активности по принципу планирај-ради-провери (*plan- do-check*). То подразумева да се након сваке активности изврши анализа и утврди шта је научено, а технике које се могу применити да се то реализује јесу радни састанци на којим се достављају повратне информације одговарајућим особама у организацији. Овакав приступ обезбеђује да се процес побољшања усаглашава са тренутним стањем и да се бирају увек најбоље опције за наставак рада.

У фази затварања пројекта се сумирају резултати, искуства и све што је научено током реализације пројекта. Техника која се најчешће користи у овој фази је пост мортем анализа (*post mortem analysis*), а која се реализује као састанак свих релевантних учесника у пројекту. Поред тога, може се користити и метода ретроспективе успешности реализације пројекта, која се базира на употреби одговарајућих дијаграма и структурних листи које описују реализацију и резултате пројекта.

4.5.1 Модели за процењивање софтверских процеса

Побољшање софтверских процеса се ослања на налазе који се добијају процењивањем процеса тако што се идентификују промене које ће се најбоље уклопити у опште пословне циљеве оргранизације. Иако се побољшање процеса може реализовати и без процењивања процеса, формално процењивање процеса омогућује да се идентификују проблеми у процесима, као и узроци који доводе до тих проблема. На тај начин, процењивање процеса обезбеђује да се побољшање фокусира на кључне проблеме у процесима. Процењивање процеса је за софтверску организацију прилика да створи јасну слику о стању процеса и свакодневне праксе, што омогућује да се циљеви побољшања процеса усагласе са стратегијом и циљевима пословања организације.

Процењивање софтверских процеса (*software process assessment*) је у софтверској индустрији препознато као фаза пројекта побољшања процеса која је суштински битна за разумевање процеса и идентификовање кључних елемената процеса које треба побољшати. Концептуални модел процењивања процеса је приказан на слици 4.11. За успешно процењивање процеса је потребно да у организацији постоји модел или дефиниција процеса, мада то



Слика 4.11: *Концептуални модел процењивања софтверских процеса*

често није случај и процеси се реализују на ад-хок начин (прилагођавају се тренутним потребама). Такође, успешно процењивање процеса се мора базирати на евиденцији из праксе о текућој имплементацији процеса, што подразумева да се прикупе подаци о тренутном стању процеса у организаццији.

Процењивање процеса атреба да обезбеди следеће информације о стању процеса:

- **Скор** представља квантитативну евиденцију о процењеним вредностима атрибута процеса који су мерљиви (временско трајање појединих активности, усаглашеност са стандадима или интерним документима, комплексност задатака, итд.).
- **Предности** се односе на позитивне карактеристике процеса које се требају очувати и након модификације која се спроводи у оквиру побољшања процеса.
- **Области за побољшавање** се односе на идентификоване сегменте или елементе процеса које треба побољшати (активности, артифакти, организациона питања, итд.)
- **Планови за акцију** представљају листу активности које треба спровести у оквиру побољшања процеса (ко, шта, када, како, чиме, итд.), као и листу ограничења која могу утицати на побољшање процеса.

Активности процењивања процеса у великој мери зависе од контекста организације где се спроводе, а посебно је значајан социјални аспект пошто све процесе имплементирају појединци или групе (тимови). Процењивање процеса укључује повратне информације (*feedback*) као суштински део процењивања, који обезбеђује да се информације о стању процењивања врате релевантним појединицима или групама у оквиру организације (програмер, лидер тима, менаџер, или тим). Повратне информације су обично део типичног редоследа активности, који укључује прикупљање, анализу и интерпретацију података у оквиру итеративног поступка процењивања процеса. Повратне информације имају значајну улогу у презентовању резултата процењивања, усмеравању и одржавању мотивације и посвећености актера укључених у процењивање, као и у организационом учењу кроз процењивање процеса.

Постоје два општа приступа или модела у индустријској пракси за процењивање и побољшање софтверских процеса:

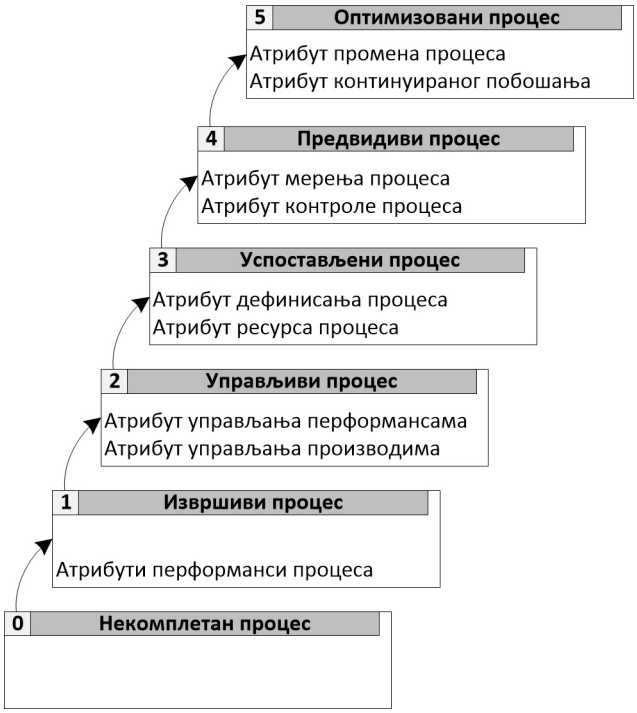
- **Прескриптивни или "одозго-према-доле" модели** (*prescriptive, top-down models*) су свеобухватни и ригорозни модели за процењивање и побољшање процеса, који се због тога називају и "тешки модели" (*heavyweights models*). Ови модели су скупи за имплементацију и захтевају значајне људске и материјалне ресурсе па их примењују само велике софтверске организације.
- **Дескриптивни или "одоздо-према-горе" модели** (*descriptive, bottom-up models*) су модели који су настали из потребе да се са минималним трошковима реализује процењивање и побољшање процеса у малим и средњим организацијама. Ови модели се ослањају на организациони контекст где се прилагођавају стварном стању, па се због тога називају "лагани модели" (*lightweights models*).

Прескриптивни модели за процењивање процеса

Прескриптивни модели су засновани на стандардима које су прописале водеће међународне организације International Organization for Standardization (ISO), International Electrotechnical Commission (IEC) и Institute of Electrical and Electronics Engineers (IEEE), а настали су као препорука најбоље праксе од стране експерата из ових организација, софтверске индустрије и академске заједнице. С обзиром да су ови стандарди врло строги и захтевају значајне ресурсе и време за имплементацију, у пракси су најчешће примењивани у великом софтверским организацијама, док их мање и средње организације прилагођавају или делимично користе за развој специфичних радних оквира за процењивање и побољшање процеса.

ISO/IEC 15504 Information technology - Process assessment је скуп докумената са међународним стандардима који дају препоруке за имплементацију процењивања процеса у контексту побољшања процеса или одређивања способности (карактеристика) процеса. У пракси се овај стандард такође назива Стандард за побољшање и утврђивање способности софтверских процеса (*Software Process Improvement and Capability Determination (SPICE)*). Одређивање способности процеса служи да се идентификују области са недостацима и ризицима, што указује где је потребно извршити побољшање процеса. Пословна пракса целе организације се може анализирати на основу процењивања одабраног скупа процеса. Анализом се утврђује ефикасност процеса са аспекта остваривања циљева. Приоритизација процеса које треба побољшати се врши на основу процењеног стања одабраних процеса у односу на одабрани профил способности. Стандард **ISO/IEC 15504** дефинише референтни модел процеса и његових способности, што представља основу за процењивање процеса у организацији. Процеси у оквиру референтног модела су груписани у следећих 5 категорија:

- **Клијент-добављач** (*Customer-Supplier*). Овде спадају процеси који директно утичу на клијенте, а укључују развој софтвера, испоруку софтвера и подршку за исправно функционисање софтвера.



Слика 4.12: *Нивои способности процеса према стандарду ISO/IEC 15504*

- **Инжењеринг** (*Engineering*). Овде спадају процеси који се односе на спецификацију, дизајн, конструкцију, инсталирање и одржавање софтвера, релације софтвера са социо-техничким системом у који се интегрише, и документацију за клијенте.
- **Подршка** (*Support*). Овде спадају процеси који пружају услуге другим процесима у оквиру животног циклуса софтвера.
- **Управљање** (*Management*). Овде спадају процеси који дефинишу опште процесе управљања процесима и пројектима у животном циклусу софтвера.
- **Организација** (*Organization*).Овде спадају процеси који омогућују постављање пословних циљева у организацији, као и управљање процесима, производима и услугама који омогућују остваривање постављених циљева.

Атрибута процеса који су груписани по нивоима способности се користе за представљање еволуције способности процеса. Модел даје упутство како повећати способности сваког процеса који се процењује, а сваки виши ниво представља инкременталну еволуцију у управљању и контроли процеса. Способности процеса се дефинишу скалом са 6 нивоа, као што је приказано на слици 4.12, при чему се на сваком вишем нивоу додају нови индикатори способности процеса.

Табела 4.1: *CMMI нивои способности и зрелости*

Ниво	**Ниво способности**	**Ниво зрелости**
Ниво 0	Непотпун	Н/П
Ниво 1	Изводљив	Иницијални
Ниво 2	Управљив	Управљив
Ниво 3	Дефинисан	Дефинисан
Ниво 4	Квантитативно управљив	Квантитативно управљив
Ниво 5	Оптимизован	Оптимизован

Евиденција оцена способности процеса је неопходна да би се обезбедила поновљивост, поузданост и конзистентност процеса процењивања. Ова евиденција се реализује помоћу индикатора перформанси и способности процеса који треба да објективно представе карактеристике процењиваних процеса (*Key Performance Indicator (KPI)*).

Capability Maturity Model Integration (CMMI) је модел процењивања процеса базиран на моделу зрелости процеса, који је развијен на *Software Engineering Institute*, *Carnegie Mellon University*, Pittsburgh, PA, USA. Модел обухвата управљање пројектима, управљање процесима, системски инћењеринг, хардверски и софтверски инжењеринг, а омогућује да се побољшању и процењивању процеса приступи на следећа два начина:

- **Континуирано побољшање** (*continuous improvement*). Прво се бира област у којој се жели извршити побољшање процеса. Процеси се потом побољшавају, а мерење побољшања се врши у односу на претходно побољшање, чиме се добија континуирано побољшање одабраних процеса. На овај начин је различите процесе могуће побољшати до различитих нивоа зрелости процеса.
- **Побољшање по стањима** (*staged improvement*). Користи де предефинисани скуп процесних области и потом се прати ниво зрелости организације у тим областима, а побољшање се врши по стањимау одређеним временским периодима.

За праћење побољшања процеса у организацији се користе нивои. Код континуираног приступа побољшању нивои се односе на способности/својства (*capability levels*) у специфичној процесној области, док се код побољшања по стањима нивои посматрају на нивоу целе организације и односе се на нивое зрелости (*maturity levels*). Нивои способности процеса се могу засебно дефинисати за сваку процесну област у организацији, што омогућује да свака процесна област засебно прати и побољшава. Нивои способности и нивои зрелости за континуирани приступ и приступ по стањима су приказани у табели 4.1.Код нивоа зрелости организације није дефинисан ниво 0, што је означено вредносћу Н/П (није применљиво). Процењивање процеса у организацији применом CMMI се реализује применом методе *Standard CMMI Appraisal Method for Process Improvement (SCAMPI)*, која омогућује процењивање способности специфичних процесних области или процењивање зрелости целе организације

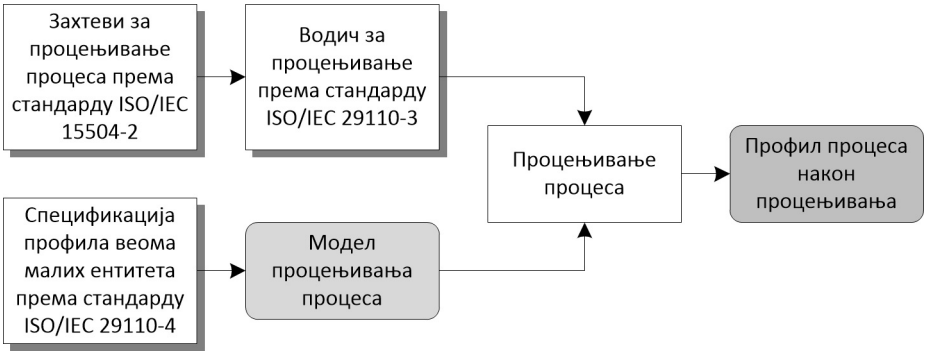
На *иницијалном* нивоу зрелости, процеси су нестабилни, непредвидиви и слабо или никако контролисани. На *управљивом* нивоу зрелости, процеси су дефинисани, контролисани и мерљиви само у одређеном сегменту

организације или на нивоу пројекта. На *дефинисаном* нивоу зрелости процеси су дефинисани и добро успостављени на нивоу целе организације. На *квантитативно управљивом* нивоу зрелости процеси су контролисани и мерљиви на нивоу целе организације. На *оптимизованом* нивоу зрелости врши се континуирано праћење и побољшање процеса. Примена CMMI подразумева институционализацију процеса, тј. дефинисање и успостављање процеса на конзистентан начин у целој организацији. Дефинисање процеса на нивоу организације подразумева следеће атрибуте: сврха, улази, полазни критеријуми и ограничења, активности, улоге, мере, метод верификације, излази и завршни критеријуми.

Основна предност CMMI модела је компатибилност модела процењивања са другим познатим стандардима и моделима управљања квалитетом, као што су ISO, *Six Sigma* и *Lean*. CMMI је такође компатибилан са ISO 15288:2008 и 12207:2008 стандардима. Могућност примене CMMI модела на одабраном пројекту, делу организације или у целој организацији, са избором границе примене модела, је такође предност овог модела.

ISO/IEC 29110 је серија стандарда и техничких извештаја која дефинише карактеристике и захтеве за процесе у животном циклусу софтвера за веома мале пословне организације (тим, департман, организација) до 25 чланова. Према стандарду ISO/IEC 29110, ове организације се називају веома мали ентитети (*Very Small Entity (VSE)*), а веома су важни пошто мале софтверске организације (фирме) чине значајан удео у софтверској индустрији широм света (подаци указују да је у већини земаља тај удео између 65 и 95 процената). Прегледи и техничка упутства су у оквиру ISO/IEC 29110 публиковани као технички извештаји (*Technical Report*, *TR*), а профили су публиковани као међународни стандарди (*International Standard*, *IS*). ISO/IEC 29110 серија стандарда се може употребити независно од тога да ли је модел животног циклуса водопад, итеративни, инкрементални, еволутивни или агилни. Делови серије стандарда ISO/IEC 29110 су:

- **ISO/IEC TR 29110-1** дефинише термине који су типични за профиле веома малих ентитета, уводи процесе, животни циклус и концепте стандардизације, и образлаже профиле, документе и водиче који се користе.
- **ISO/IEC 29110-2** уводи концепте за стандардне профиле софтверског инжењерства за веома мале ентитете и дефинише терминологију и таксономију свих профила.
- **ISO/IEC TR 29110-3** дефинише водич за процењивање процеса и захтеве за усаглашеност веома малих ентитета. Такође садржи информације потребне за развој метода и алата за процењивање, а који су потребни експертима директно укљученим у процењивање процеса.
- **ISO/IEC 29110-4-1** даје спецификацију свих генеричких профила базираних на елементима стандарда.
- **ISO/IEC 29110-5-m-n** даје менаџерске и инжењерске препоруке за примену профила описаних у делу стандарда ISO/IEC 29110-4-m.



Слика 4.13: *Елементи процењивања провеса веома малих софтверских организација према стандарду ISO/IEC TR 29110*

Део стандарда, публикован као технички извештај *ISO/IEC TR 29110-3 Software engineering - Lifecycle profiles for Very Small Entities (VSEs) - Part 3-1: Assessment guide* садржи препоруке и упутства за процењивање процеса. У овом техничком извештају су препоруке за методе и алате који се могу користити за процењивање процеса. ISO/IEC TR 29110-3 дефинише шеме за сертификацију, смернице за оцењивање, захтеве за усаглашеност приликом процене способности процеса и самопроцене процеса у пројектима побољшања процеса. Водич за процењивање садржи упутства за имплементацију која обезбеђује постизање одређеног нивоа зрелости. Водич такође садржи препоручене активности, мерења, технике, моделе и методе. На основу процене процеса, софтверска организација може добити профил способности имплементирани процеса и ниво зрелости саме организације.

Процењивање процеса применом ISO/IEC TR 29110-3 се спроводи у складу са процесом процењивања препорученим у ISO/IEC 15504-2, што је приказано на слици 4.13. Резултат процењивања процеса се добија као скуп оцена за атрибуте процеса, што представља профил процеса (*process profile*). Стандард ISO/IEC 15504-2 поставља минималне захтеве за спровођење процењивања тако да се обезбеди конзистентност и поновљивост оцењивања процеса.

Дескриптивни или индуктивни модели за процењивање процеса

Дескриптивни модели за процењивање процеса се могу ослањати на стандарде и препоруке, али у основи се базирају на креирању приступа који је потпуно прилагођен организацији која спроводи процењивање и побољшање процеса. Циљ је да се идентификује стање одабраних процеса и потенцијална побољшања која се могу потом имплементирати, а не да се оствари усаглашеност са постојећим стандардима или препорукама. Овакви приступи се још називају и **лагани** (*lightweight*), или **приступи од доле** (*bottom-up*), пошто се дизајнирају у складу са потребама и расположивим ресурсима организације која процењује процесе. Основна одлика ових приступа је да су по природи **индуктивни** (*inductive*) пошто се у процењивању процеса полази од стварне праксе у организацији, а предложена побољшања су утемељена у

начину рада и потребама организације. Аспекти добре индустријске праксе процењивања софтверских процеса применом *лаганих* и *индуктивних* метода за процењивање су:

- **Метод за процењивање**. Односи се на дизај метода процењивања, што укључује референце на документе, моделе или стандарде чији се сегменти користе, избор метода за прикупљање и анализу података и избор процеса који се процењују у оквиру пројекта.
- **Помоћни алати**. Односи се на софтверске алате који се користе за аутоматизацију послова процењивања. Овде спадају софтверски алати за аутоматску екстракцију података (на пример, подаци из дигиталних база података или подаци из докумената) и софтверски алати за трансформацију (на пример, транскрипција интервјуа) и анализу података (на пример, статистичке анализе података).
- **Процедура**. Односи се на дефинисање активности које је потребно реализовати у оквиру процењивања процеса, при чему за сваку активност треба одабрати најприкладнију технику за реализацију задатака у оквиру активности (на пример, избор одговарајуће технике или методе за анализу нумеричких података).
- **Документација**. Односи се на материјале са информацијама које су потребне свим учесницима у пројекту процењивања процеса. Документација може бити припремљена приликом дизајнирања пројекта процењивања, али и током имплементације пројекта.
- **Учесници**. Односи си се на људе који су укључени у пројекат процењивања процеса - експерти који процењују процесе, менаџеери и запослени у организацији која спроводи процењивање процеса. Овај аспект добре праксе треба да обезбеди да потребне вештине и одговорности за одређене улоге у пројекту процењивања процеса буду јасно дефинисане (на пример, која одговорности има руководилац тима или менаџер целе организације).

Ови приступи процењивању процеса су базирани на разумевању организационог контекстаи реалних потреба организације, па се према томе могу лако адаптирати. Основне карактеристике ових приступа су:

- **Дизајнирају се (кроје) према потребама организације**. Сви циљеви се изводе из стварних потреба организације пошто се откривају кроз присуство у организацији и праћење свакодневних активности.
- **Не прате препоруке стандарда, радних оквира и водича добре праксе**. Стандарди, радни оквири и препоруке указују шта и како да се процењује и побољша, што у великом броју случајева не одговара организацијама (посебно малим и микро софтверским предузећима).
- **Приступ процењивању се дизајнира кроз заједнички рад експерата и запослених у организацији**. Најпоузданије и најтачније информације о пракси и проблемима у организацији се могу добити од запослених, па се и приступ дизајнира тако да најбоље одговара организацији, што је веома битно пошто свака организација има своје специфичности.

- **Организација сама одлучује шта ће процењивати и побољшавати**. Избор процеса који се процењују и побољшавају је одлука руководства и запослених који се активно укључују у све активности процењивања и побољшавања процеса.
- **Фокус је на најкритичнијим и најбитнијим сегментима праксе**. Процењивањем процеса се идентификују потенцијална побољшања, а руководство организације одлучује шта ће имплементирати на основу критичности за пословање и на основу стварних потреба. Учешће руководства и запослених у доношењу одлука доприноси успеху процењивања и побољшавања процеса, а у складу са пословним циљевима и стратегијама организације.
- **Пружају подршку за организационо учење и дељење знања**. Кроз заједнички рад експерата (истраживача, консултаната) и запослених се прикупљају релевантне информације, што обезбеђује да се прећутно знање (*tacit knowledge*) преведе у експлицитно знање (*explicit knowledge*) доступно свим запосленима у организацији. На тај начин се подржава организационо учење на индивидуалном нивоу, нивоу тимова и нивоу целе организације.
- **Подесни су за мале организације**. Мале организације немају ресурсе (финансијске и људске) за спровођење процењивања на бази стандарда, па је за њих најбољи избор да заједно са одабраним експертима реализују процењивање фокусирано на свакодневну праксу.

Основни елементи ових приступа су: индуктивно резоновање, употреба више извора података и метода за анализу података, повратне информације ка организацији и организационо учење.

Индуктивно резоновање (*inductive reasoning*) и индуктивно размишљање је у основи индуктивних приступа процењивању процеса, а основна карактеристика је генерализација специфичног искуства и посматрања праксе, што води до генерализованих резултата. Индуктивно резоновање је базирано на индуктивном и интерпретативном извођењу закључака, што води до идентификације типичних шаблона кроз детектовање сличности и разлика. У основи индуктивног резоновања су когнитивни процеси засновани на аналогији, класификацији и категоризацији. Индуктивним резоновањем се долази до предлога побољшања процеса који су утемељени у искуству из свакодневне праксе.

Употреба више извора података и метода за анализу података (*triangulation of data sources and methods for data analysis*) је неопходна за за успешно процењивање процеса применом индуктивних метода. Кључни извор података су софтверски инжењери који реализује послове у свакодневној пракси. Прикупљени подаци могу бити квантитативни (нумерички) и квалитативни (најчешће неструктуриран текст), што подразумева употребу различитих извора података, метода за прикупљање података и потом метода за анализу података (триангулација), што је приказано на слици 4.14. Триангулација обезбеђује већу валидност резултата процењивања процеса, па се може сматрати да су побољшања утемељена у стварној пракси и искуству и знању запослених.

Слика 4.14: *Триангулација података и метода за прикупљање и анализу података у индуктивним приступима за процењивање процеса*

Повратне информације ка организацији *feedback to organization*) су основа континуираног побољшавања перформанси организације. Враћање информација организацији која спроводи процењивање процеса обезбеђује да информације стигну до релевантних људи, смањује могућност појављивања грешака и подржава организационо учење. Повратне информације су битне у процесу доношења одлука и тражења најбољег решења у датом контексту. Током процењивања софтверских процеса повратне информације су од пресудног значаја за управљање пошто обезбеђује да резултати буду доступни и видљиви током целог пројекта процењивања и побољшања процеса, позитивно утичу на одржавање радне атмосфере и фокуса, и помажу у мотивисању запослених. Праћењем повратних информација у сваком тренутку се може разумети тренутно стање пројекта процењивања и побољшања процеса, и може се сагледати које наредне активности треба спровести.

Организационо учење (*organizational learning*) је саставни део процењивања и побољшања процеса. Иницијатива процењивања процеса подржава идентификацију и систематизацију знања које постоји у организацији за употребу у наредним пројектима. Организационо учење је у индуктивним приступима процењивању процеса подржано активним учешћем запослених, што обезбеђује учење на основу искуства и идентификацију и имплементацију најрелевантнијих побољшања праксе. Знање о процесима у софтверској организацији се налази у главама (умовима) запослених, треба га прикупити, прочистити и систематизовати а потом и ускладиштити током активности процењивања процеса. На овај начин сво идентификовано и систематизовано знање постаје доступно свим запосленима, што је кључно за

ефикасно управљање знањем у организацији. Поред пословних добити за организацију, организационо учење помаже софтверској организацији да унапреди и сам процес процењивања и побољшања процеса.

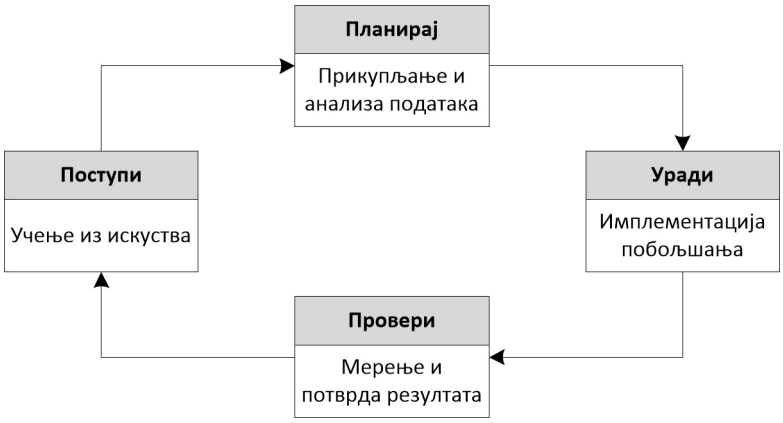
4.5.2 Модели за побољшање софтверских процеса

Софтверске организације најчешће немају довољно добро дефинисане процесе, или користе проблематичне и ад-хок осмишљене процесе, што резултира проблемима у квалитету производа и услуга, али и у комуникацији унутар организације и са окружењем (пре свега са клијентима). Проблеми који могу настати због лоше дефинисаних процеса могу бити финансијске природе, а често доводе и до губитка клијената. Због тога је од посебног значаја да се пажња посвети квалитету процеса и да се одаберу одговарајући приступи са побољшање процеса. У пракси се најчешће користе два приступа побољшавању процеса:

- **Континуирано побољшавање**, које се још и назива приступ оријентисан ка проблемима у пракси. Овакв приступ полази од стварног стања у пракси, тј. од реалног проблема, па се још назива и приступ одоздо (*bottom-up approach*). Због тога се приступ са континуираним побољшавањем користи за решавање специфичних проблема у софтверским организацијама.
- **Приступ базиран на моделима**, који се још назива и приступ оријентисан ка решењима која су прописана моделима. С обзиром да полази од модела и процесе пореди са препорученом праксом, овакав приступ се назива још и приступ од горе (*top-down approach*). Приступ базиран на моделима се због тога користи за поређење постојећих процеса са референтним моделима процеса и идентификовање потенцијалних области за побољшавање.

Приступи континуираног побољшања процеса се базирају на итеративном циклусу побољшања по принципу **планирај-уради-провери-поступи** (*Plan-Do-Check-Act (PDCA)*), који се генерално користи за побољшавање пословних процеса. То је у основи приступ решавању проблема у четири фазе, као што је приказано на слици 4.15. У фази планирања се дефинишу мерљиви циљеви побољшања процеса, активности које треба спровести, као и очекивани резултати након успешне имплементације планираног побољшања. Побољшања се реализују у малим корацима и у малом обиму, тако да се могу лако пратити и мерити. У фази мерења се прикупљају и анализирају резултати имплементације побољшања, а резултати се пореде са планираним. У последњој фази циклуса се анализира разлика између планираних и остварених резултата, на основу чега се сагледавају наредне активности планира нови цилус побољшања.

Приступи побољшању процеса базирани на моделима користе моделе који су настали из најбоље праксе организација које су успешно процењивале и побољшавале своје процесе. Најчешће коришћени модели су ISO/IEC 15504 (SPICE) и CMMI. Ови модели се базирају на поређењу процеса у организацији са референтним моделима процеса који су потврђени и стандардизовани, а служе за утврђивање и побољшање нивоа способности



Слика 4.15: *Континуирано побољшање процеса применом принципа планирај-уради-провери-поступи*

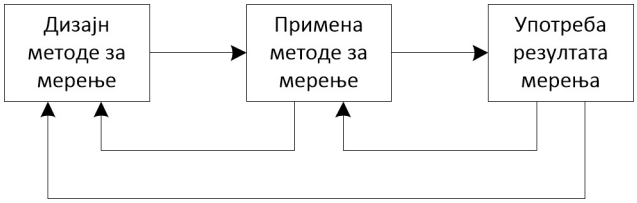
(*capability*) и зрелости (*maturity*) процеса и саме организације која имплементира процесе. Код приступа базираних на моделима, процењивање и побољшање процеса раде независни експерти или организације, тако да се тиме остварује већа објективност побољшања процеса и подстиче пракса управљања процесима и квалитетом у организацији. Недостаци примене модел базираних приступа су:

- Спровођење процењивања и побољшања процеса реализују специјализоване консултантске фирме које учествују и у креирању модела, па се поставља оправдано питање да ли је креирање и примена ових модела подређена остваривању већег пословног успеха консултантских фирми или софтверских организација које побољшавају процесе.
- Примена ових приступа служи да се процеси усагласе са референтним моделима, а питање је да ли је то директно усаглашено са пословним циљевима и стратегијама софтверске организације која спроводи побољшање процеса.
- Ови приступи су врло захтевни и скупи за имплементацију, што је посебно неповољно за мале софтверске организације које чине већину у софтверској индустрији.

У пракси се врло често ова два приступа допуњују. На пример, организација може применити приступ базиран на моделима да идентификује процесе које треба побољшати, а да потом примени приступ са континуираним побољшавањем одабраних процеса.

4.6 Мерење софтверских процеса

Циљ побољшања процеса је да се унапреде перформансе софтверске организације, што најчешће подразумева смањење трошкова, повећање ефикасности рада и повећање квалитета производа и услуга. Пре спровођења



Слика 4.16: *Контекст мерења софтверских процеса*

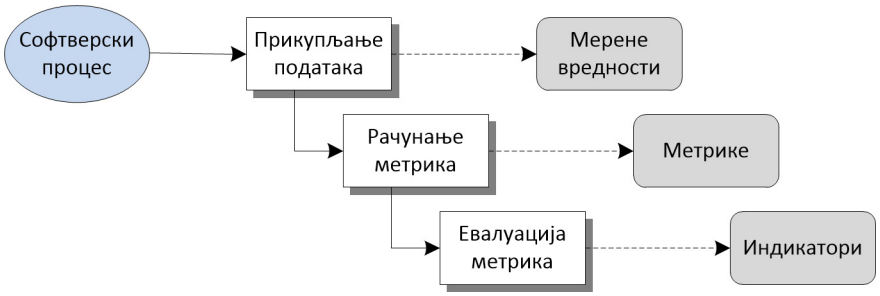
побољшања, али и након имплементације одабраних побољшања, потребно је мерити одабране параметре процеса да би се извршила евалуација текуће праксе и успешности имплементираних побољшања. Основни разлози за спровођење мерења софтверских процеса су:

- Разумевање и документовање текућег стања процеса да би се формирала основа за поређење током будућих процењивања и побољшавања.
- Утврђивање статуса процеса у односу на претходно постављене циљеве и планове.
- Постизање разумевања односа између процеса и производа, што представља основу за формирање модела и предикција праксе у будућности.
- Побољшање перформанси процеса на основу идентификованих проблема у текућој реализацији процеса.

Мерење софтверских процеса се спроводи у три фазе, а контекст мерења је приказан на слици 4.16. У првој фази се бира метода мерења међу постојећим, или се дизајнира нова у складу са контекстом мерења у организацији. Одабрана метода мерења се потом у другој фази примењује у датом контексту са циљем да се добију резултати мерења, док се у трећој фази на основу добијених резултата мерења креира модел и идентификују наредне акције (на пример, имплементација неког од идентификованих побољшања процеса). Најчешће се креирају квантитативни модели (модели процењивања, модели квалитета процеса и производа) на основу резултата мерења, а они служе као основа за доношење одлука и селекцију наредних побољшања процеса. Имплементацијом активности у другој и трећој фази се могу сагледати могућности за побољшање методе мерења и њене примене у датом контексту, што је приказано повратним везама на слици 4.16.

Дизајн методе мерења треба да обезбеди испуњење циља мерења, а као улазе има скуп концепата и техника које ће се користити приликом мерења. За идентификоване мерљиве концепте и конструкте се дефинишу принципи мерења и правила придруживања нумеричких вредности одабраним концептима и конструктима.

Поступак мерења процеса је приказан на слици 4.17. Мерење се врши у контексту софтверске организације, а добијају се мерене вредности које се односе на неке атрибуте процеса. На основу мерених вредности се рачунају метрике које су предвиђене дизајном методе мерења. **Метрике** (*metrics*) су мере квантитативне процене које се користе за процену, поређење и праћење



Слика 4.17: *Поступак мерења софтверских процеса*

перформанси процеса или производа. Евалуацијом израчунатих метрика се добијају индикатори перформанси процеса, који се потом користе за доношење одлука (на пример, да ли је потребно побољшати неке параметре процеса).

Мерење софтверских процеса има за циљ да утврди ефикасност процеса, појединих активности и задатака у смислу утрошка ресурса. **Напор** (*Effort*), или одговарајући еквивалентан трошак је примарна мера ресурса за већину софтверских процеса, активности и задатака. Напор се мери јединицама као што су човек-сати (*person-hours*), човек-дани (*person-days*), особље-недеље (*staff-weeks*) или особље-месеци (*staff-months*).

Ефективност (*Effectiveness*) процеса се представља као однос стварног излаза процеса и очекиваног излаза процеса. На пример, може се мерити стваран број дефеката током тестирања софтвера и то поредити са вредностима заснованим на процени на основу историјских података за сличне пројекте у бази дефеката. Мерење ефективности софтверских процеса обично подразумева и мерење релевантних атрибута производа (на пример број дефеката у односу на величину и комплексност модула у којем се дефекти откривају). Поред ефективности, може се мерити и **ефикасност** (*efficiency*) која се базира на мерењу напора да се оствари резултат процеса.

Ефикасност софтверских процеса се мери индиректно кроз изведене метрике које се добијају на основу излаза процеса. Као излази процеса могу се мерити: грешке које нису откривене пре испоруке софтвера, грешке пријављене од стране корисника, напор људи ангажованих на задацима, време утрошено у процесу, временски оквири реализације активности, перформансе распоређивања задатака на запослене. Мерење се може односити на перформансе људи ангажованих у процесу, на производе, активности и задатке, временске оквире активности и задатака, или употребу специфичних инжењерских алата у процесу.

Литература

Aaen, I., Arent, J., Mathiassen, L., & Ngwenyama, O. (2001). A conceptual map of software process improvement. *Scandinavian Journal of Information Systems*, 13(1), Article 8.

Abran, A. (2010). *Software Metrics and Software Metrology*. Hoboken, NJ, USA: John Wiley & Sons. doi: 10.1002/9780470606834.

Acuña, S.T., & Ferré, X. (2001). Software Process Modelling. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics: Information Systems Development - Volume I* (p. 237-242). Orlando, FL, USA.

Anda, B.C.D. (2007). Assessing Software System Maintainability using Structural Measures and Expert Assessments. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007)* (p. 204-213). Paris, France. doi: 10.1109/ICSM.2007.4362633.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), 970-983. doi: 10.1109/TSE.2002.1041053.

April, A., & Abran, A. (2008). *Software Maintenance Management: Evaluation and Continuous Improvement*. Hoboken, NJ, USA: John Wiley & Sons.

Ayalew, Y., & Motlhala, K. (2014). An ISO/IEC 15504 based Software Process Assessment in Small Software Companies. *International Journal of Software Engineering and Its Applications*, 8(6), 121-138. doi: 10.14257/ijseia.2014.8.6.10.

Banker, R.D., & Slaughter, S.A. (1997). A field study of scale economies in software maintenance. *Management Science*, 43(12), 1709-1725. doi: 10.1287/mnsc.43.12.1709.

Beatty, J., & Chen, A. (2012). *Visual models for software requirements*. Redmond, Washington, USA: Microsoft Press.

Behnamghader P., & Boehm B. (2019) Towards Better Understanding of Software Maintainability Evolution. In: Adams S., Beling P., Lambert J., Scherer W., Fleming C. (eds) *Systems Engineering in Context* (p. 593-603). Springer, Cham. doi: 10.1007/978-3-030-00114-8_47

Bennett, K.H., Rajlich, V.T., & Wilde, N. (2002). Software Evolution and the Staged Model of the Software Lifecycle. *Advances in Computers*, 56, 1-54. doi: 10.1016/S0065-2458(02)80003-1.

Biggerstaff, T.J. (1989). Design recovery for maintenance and reuse. *Computer*, 22(7), 36-49. doi: 10.1109/2.30731.

Birchall C. (2016). *Re-Engineering Legacy Software*. Shelter Island, NY, USA: Manning Publications.

Bourque, P., & Fairley, R.E. (Editors) (2014). *Guide to the Software Engineering Body of Knowledge, Version 3.0, SWEBOK*. IEEE.

Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering, Anniversary Edition*. Boston, MA, USA: Addison Wesley Longman.

Burnay, C. (2016). Are stakeholders the only source of information for requirements engineers? Toward a taxonomy of elicitation information sources. *ACM Transactions on Management Information Systems*, 7(3), 8:1-8:29. doi: 10.1145/2965085

Carrizo, D., Dieste, O., & Juristo, N. (2014). Systematizing requirements elicitation technique selection. *Information and Software Technology*, 56(6), 644-669. doi: 10.1016/j.infsof.2014.01.009.

Chapin, N. (2000). Software Maintenance Types - A Fresh View. In *Proceedings of the International Conference on Software Maintenance* (p. 247-). San Jose, CA, USA. doi: 10.1109/ICSM.2000.883056.

Chen, C., Alfayez, R., Srisopha, K., Boehm, B., & Shi, L. (2017). Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It? In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (p. 377-378). Buenos Aires, Argentina. doi: 10.1109/ICSE-C.2017.75.

Chikofsky, E.J., & Cross, J.H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1), 13-17. doi: 10.1109/52.43044.

CMMI Product Team (2001). *Appraisal Requirements for CMMI, Version 1.1 (ARC, V1.1)*. Technical report CMU/SEI-2001-TR-034. Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA, USA.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Boston, MA, USA: Addison-Wesley Professional.

Demeyer, S., Ducasse, S., & Nierstrasz O. (2003). *Object-oriented reengineering patterns, 1st edition*. Morgan Kaufman.

Desouza, K.C., Dingsøyr, T., & Awazu, Y. (2005), Experiences with conducting project postmortems: reports versus stories. *Software Process: Improvement and Practice*, 10(2), 203-215. doi: 10.1002/spip.224.

Dumas, M., & Pfahl, D. (2016). Modeling Software Processes Using BPMN: When and When Not? In M. Kuhrmann. J. Münch, I. Richardson, A. Rausch & H. Zhang (Eds.), *Managing Software Process Evolution: Traditional, Agile and Beyond - How to Handle Process Change* (p. 165-183). Springer, Cham. doi: 10.1007/978-3-319-31545-4.

Dybå, T., Dingsøyr, T., & Moe, N.B. (2004). *Process Improvement in Practice: A Handbook for IT Companies*. International Series in Software Engineering, Vol. 9. Norwell, MA, USA: Kluwer Academic Publishers. doi: 10.1007/b116193.

Dingsøyr, T. (2005). Postmortem reviews: purpose and approaches in software engineering. *Information and Software Technology*, 47(5), 293-303. doi: 10.1016/j.infsof.2004.08.008.

European Commission (2015). *User guide to the SME Definition*. Enterprise and industry publications. Luxembourg: Publications Office of the European Union.

Fairley, R.E. (2009). *Managing and leading software projects*. Hoboken, New Jersey, USA: John Wiley & Sons. doi: 10.1002/9780470405697.

Feliz, T. (2012). Lightweight Software Process Assessment and Improvement. In *Proceedings of Thirtieth Annual Pacific Northwest Software Quality Conference, PNSQC 2012*, (p. 405-424). Portland, Oregon, US.

Fu, J., Bastani, F. B., & Yen, I.-L. (2008). Model-Driven Prototyping Based Requirements Elicitation. In B. Paech & C. Martell (Eds.), *Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs* (Vol. 5320, p. 43-61). Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-89778-1_7.

Fuentes-Fernandez, R., Gómez-Sanz, J. J., & Pavón, J. (2010). Understanding the Human Context in Requirements Elicitation. *Requirements Engineering*, 15(3), 267-283. doi: 10.1007/s00766-009-0087-7.

Galvis-Lista, E., & Sánchez-Torres, J.M. (2013). A Critical Review of Knowledge Management in Software Process Reference Models. *Journal of Information Systems and Technology Management*, 10(2), 323-338. doi: 10.4301/S1807-17752013000200008.

Gray, E., Sampaio, A., & Benediktsson, O. (2005). An Incremental Approach to Software Process Assessment and Improvement. *Software Quality Journal*, 13(1), 7-16. doi: 10.1007/s11219-004-5258-7.

Grubb, P., & Takang, A.A. (2003). *Software Maintenance: Concepts and Practice, 2nd edition*. Singapore: World Scientific Publishing Company.

Gruhn, V. (2000). Software process landscaping. *Software Process: Improvement and Practice*, 5(2-3), 111-120.

Hass, A.M.J. (2002). *Configuration management principles and practice*. Boston, MA, USA: Addison-Wesley Professional.

Heidrich, J., Münch, J., Riddle, W., & Rombach, D. (2006). People-oriented Capture, Display, and Use of Process Information. In S.T. Acuña & M.I. Sánchez-Segura (Eds.), *New Trends in Software Process Modeling* (p. 121-179). World Scientific Publishing Company. doi: 10.1142/9789812774460_0005.

Heikkilä, M. (2009). Learning and Organizational Change in SPI Initiatives. In In F. Bomarius, M. Oivo, P. Jaring, & P. Abrahamsson (Eds.), *Product-focused software process improvement* (Vol. 32, p. 216-230). Berlin, Germany: Springer Berlin Heidelberg. doi:10.1007/978-3-642-02152-7_17.

Humphrey, W.S. (1989). The Software Engineering Process: Definition and Scope. *ACM SIGSOFT Software Engineering Notes*, 14(4), 82-83. doi: 10.1145/75111.75122.

Ivarsson, M. (2010). *Experience driven software process assessment and improvement*. PhD thesis. Department of Computer Science & Engineering, Chalmers University of Technology. Göteborg, Sweden.

Jones, C. (2010). *Software engineering best practices: Lessons from successful projects in the top companies*. New York, NY, USA: McGraw-Hill.

Kajko-Mattsson, M. (2002). Problem management maturity within corrective maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(3), 197-227. doi: 10.1002/smr.252.

Kitchenham, B.A., Travassos, G.H., von Mayrhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., & Yang, H. (1999). Towards an ontology of software maintenance. *Journal of Software Maintenance: Research and Practice*, 11(6), 365-389.

Klauer, K.J., & Phye, G.D. (2008). Inductive reasoning: A training approach. *Review of Educational Research*, 78(1), 85–123. doi: 10.3102/0034654307313402.

Kuhrmann, M., Münch, J., Richardson, I., Rausch, A., & Zhang, H. (Editors) (2016). *Managing Software Process Evolution: Traditional, Agile and Beyond - How to Handle Process Change*. Switzerland: Springer International Publishing. doi: 10.1007/978-3-319-31545-4.

Land, S.K., Smith, D.B., & Walz, J.W. (2008). *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Hoboken, NJ, USA: John Wiley & Sons. doi:10.1002/9780470289969.

Laporte, C.Y., & O'Connor, R.V. (2014). A Systems Process Lifecycle Standard for Very Small Entities: Development and Pilot Trials. In *Proceedings of the 21st European Conference, EuroSPI 2014*. Systems, Software and Services Process Improvement, Communications in Computer and Information Science Volume 425, (p. 13-24). doi: 10.1007/978-3-662-43896-1_2.

Laporte C.Y., O'Connor R.V., Paucar L.H.G. (2016). The Implementation of ISO/IEC 29110 Software Engineering Standards and Guides in Very Small Entities. In Maciaszek L.A., Filipe J. (eds) *Evaluation of Novel Approaches to Software Engineering. ENASE 2015*. Communications in Computer and Information Science, vol 599. Springer, Cham. doi: 10.1007/978-3-319-30243-0_9.

Larrucea, X., O'Connor, R.V., Colomo-Palacios, R., & Laporte, C.Y. (2016). Software Process Improvement in Very Small Organizations. *IEEE Software*, 33(2), 85-89. doi: 10.1109/MS.2016.42.

Lauesen, S. (2002). *Software requirements: Styles and techniques*. London, UK: Pearson Education.

Leffingwell, D. (2011). *Agile software requirements: Lean requirements practices for teams, programs, and the enterprise*. Boston, MA, USA: Pearson Education.

Lehman, M.M., & Belady, L.A. (Editors) (1985). *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional.

Lehman, M.M. (1996). Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology* (p. 108-124). Nancy, France. doi: 10.1007/BFb0017737.

Lehman, M.M., & Ramil, J.F. (2003). Software evolution - Background, theory, practice. *Information Processing Letters*, 88(1-2), 33-44. doi: 10.1016/S0020-0190(03)00382-X.

Lehnert, S. (2011). A Taxonomy for Software Change Impact Analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution* (p. 41-50). Szeged, Hungary. doi: 10.1145/2024445.2024454.

Lehtinen, Mäntylä, M.V., Itkonen, J., & Vanhanen, J. (2015). Diagrams or structural lists in software project retrospectives – An experimental comparison. *Journal of Systems and Software*, 103, 17-35. doi: 10.1016/j.jss.2015.01.020.

Lepmets, M., McBride, T., & Ras, E. (2012). Goal alignment in process improvement. *Journal of Systems and Software*, 85(6), 1440-1452. doi: 10.1016/j.jss.2012.01.038.

Madhavji, N.H., Fernandez-Ramil, J., & Perry, D. (Editors) (2006). *Software Evolution and Feedback: Theory and Practice*. Chichester, UK: John Wiley & Sons.

Majthoub, M., Qutqut, M.H., & Odeh, Y. (2018). Software Re-engineering: An Overview. In *Proceedings of the 2018 8th International Conference on Computer Science and Information Technology (CSIT)* (p. 266-270). Amman, Jordan. doi: 10.1109/CSIT.2018.8486173.

McBride, T. (2010) Organisational theory perspective on process capability measurement scales. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(4), 243–254. doi: 10.1002/spip.440.

McChesney, I.R. (1995). Toward a classification scheme for software process modelling approaches. *Information and Software Technology*, 37(7), 363-374. doi: 10.1016/0950-5849(95)91492-I.

Medeiros, J., Vasconcelos, A., Silva, C., & Goulao, M. (2018). Quality of software requirements specification in agile projects: A cross-case analysis of six companies. *Journal of Systems and Software*, 142, 171-194. doi: 10.1016/j.jss.2018.04.064.

Messnarz, R., O'Suilleabhain, G., & Coughlan, R. (2006). From process improvement to learning organisations. *Software Process: Improvement and Practice*, 11(3), 287-294. doi: 10.1002/spip.272.

Miller, J. (2008). Triangulation as a basis for knowledge discovery in software engineering. *Empirical Software Engineering*, 13(2), 223–228. doi: 10.1007/s10664-008-9063-y.

Molnár, G., Greiff, S., Csapo, B. (2013). Inductive reasoning, domain specific and complex problem solving: Relations and development. *Thinking Skills and Creativity*, 9, 35–45. doi: 10.1016/j.tsc.2013.03.002.

Münch, J., Armbrust, O., Soto, M., & Kowalczyk, M. (2012). *Software Process Definition and Improvement*. The Fraunhofer IESE Series on Software and Systems Engineering. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-24291-5.

North, K., & Kumta, G. (2014). *Knowledge Management: Value Creation Through Organizational Learning*. Switzerland: Springer International Publishing. doi: 10.1007/978-3-319-03698-4.

Nuseibeh, B., & Easterbrook, S. (2000). Requirements engineering: A roadmap. In *Proceedings of the conference on the future of software engineering* (p. 35-46). Limerick, Ireland. doi: 10.1145/336512.336523.

O'Regan, G. (2011). *Introduction to Software Process Improvement*. London, UK: Springer London. doi: 10.1007/978-0-85729-172-1.

Patton, J., & Economy, P. (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product*. Sebastopol, CA, USA: O'Reilly Media.

Petersen, K., Wohlin, C., & Baca, D. (2009). The waterfall model in large-scale development. In F. Bomarius, M. Oivo, P. Jaring, & P. Abrahamsson (Eds.), *Product-focused software process improvement* (Vol. 32, p. 386-400). Berlin, Germany: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02152-7_29.

Pfleeger, S.L., & Atlee J.M. (2010). *Software Engineering: Theory and Practice, 4th edition*. Upper Saddle River, NJ, USA: Prentice Hall.

Pino, F.J., Pardo, C., García, F., & Piattini, M. (2010). Assessment methodology for software process improvement in small organizations. *Information and Software Technology*, 52(10), 1044-1061. doi: 10.1016/j.infsof.2010.04.004.

Pressman, R.S. (2010). *Software engineering: a practitioner's approach, 7th edition*. New York, NY, USA: McGraw-Hill.

Project Management Institute (2017). *A guide to the project management body of knowledge (PMBOK guide), 6th edition*. Newtown Square, PA, USA: Project Management Institute, Inc.

Rajlich, V. T., & Bennett, K. H. (2000). A staged model for the software life cycle. *Computer*, 33(7), 66-71. doi: 10.1109/2.869374.

Riaz, M., Mendes, E., & Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, (p. 367-377). doi: 10.1109/ESEM.2009.5314233.

Ribaud, V., & Saliou, P. (2010). Process assessment issues of the ISO/IEC 29110 emerging standard. In *Proceedings of the 11th International Conference on Product Focused Software (PROFES '10)*, (p. 24-27). doi: 10.1145/1961258.1961264.

Roebuck, C. (1996). Constructive feedback: Key to higher performance and commitment. *Long Range Planning*, 29(3), 328–336. doi: 10.1016/0024-6301(96)00028-3.

Ruparelia, N. B. (2010). Software development life cycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3), 8-13. doi: 10.1145/1764810.1764814.

Sánchez-Gordón, M-L., & O'Connor, R.V. (2016). Understanding the gap between software process practices and actual practice in very small companies. *Software Quality Journal*, 24(3), 549–570. doi: 10.1007/s11219-015-9282-6.

Scacchi, W. (2002). Process Models in Software Engineering. In J.J. Marciniak (Eds.), *Encyclopedia of Software Engineering. 2nd edition*. New York, NY, USA: John Wiley & Sons. doi: 10.1002/0471028959.sof250

SCAMPI Team (2013). *Standard CMMI Appraisal Method for Process Improvement (SCAMPI) Version 1.3a: Method Definition Document for SCAMPI A, B, and C*. Handbook, CMMI Institute-2013-HB-001. CMMI Institute. Carnegie Mellon University. Pittsburgh, PA, USA.

Seifert, T., & Pizka, M. (2003). Supporting software-evolution at the process level. *Net. ObjectDays 2003* (p. 1-8). Erfurt, Germany.

Sharma, P., & Sangal, A.L. (2018). Framework for empirical examination and modeling structural dependencies among inhibitors that impact SPI implementation initiatives in software SMEs. *Journal of Software: Evolution and Process*, 30(12), e1993. doi: 10.1002/smr.1993.

Shih, S.-P., Shaw, R.-S., Fu, T.-Y., & Cheng, C.-P. (2013). A Systematic Study of Change Management During CMMI Implementation: A Modified Activity Theory Perspective. *Project Management Journal*, 44(4), 84-100. doi: 10.1002/pmj.21358.

Schonberger, R.J. (2008). *Best Practices in Lean Six Sigma Process Improvement: A Deeper Look*. John Wiley & Sons, Hoboken, NJ, USA.

Sommerville, I. (2011) *Software Engineering, 9th edition*. Addison-Wesley, Boston, MA, USA.

Staples, M., Niazi, M., Jeffery, R., Abrahams, A., Byatt, P., & Murphy, R. (2007). An exploratory study of why organizations do not adopt CMMI. *Journal of Systems and Software*, 80(6), 883-895. doi: 10.1016/j.jss.2006.09.008.

Stojanov, Z., Dobrilovic, D., & Stojanov, J. (2013). Analyzing Trends for Maintenance Request Process Assessment: Empirical Investigation in a Very Small Software Company. *Theory and Applications of Mathematics & Computer Science*, 2(2), 59-74.

Stojanov, Z. (2015). Qualitative research on practice in small software companies. In M. Khosrow-Pour (Eds.), *Encyclopedia of Information Science and Technology, 3rd edition* (p. 650-658). Hershey, PA, USA: IGI Global. doi: 10.4018/978-1-4666-5888-2.ch062.

Stojanov, Z. (2016). Inductive Approaches in Software Process Assessment. In *Proceedings of the 6th International conference on Applied Internet and Information Technologies (AIIT2016)*, (p. I-XV). Bitola, North Macedonia. doi: 10.20544/AIIT2016.I01.

Stojanov, Z., & Dobrilovic, D. (2017). The Role of Feedback in Software Process Assessment. In M. Khosrow-Pour (Eds.), *Encyclopedia of Information Science and Technology, 4th edition* (p. 7514-7524). Hershey, PA, USA: IGI Global. doi: 10.4018/978-1-5225-2255-3.ch654.

Stojanov, Z., Dobrilovic, D., & Stojanov, J. (2018). Extending data-driven model of software with software change request service. *Enterprise Information Systems*, 12(8-9), 982-1006. doi: 10.1080/17517575.2018.1445296.

Stojanov, Z, Stojanov, J., & Dobrilovic, D. (2019). A lightweight inductive method for process assessment based on frequent feedback: A study in a micro software company. *Journal of Engineering Management and Competitiveness*, 9(2), 134-147. doi: 10.5937/jemc1902134s.

Stojanov, Z. (2019). Thematic knowledge framework on human factor in software maintenance practice: A study in a micro software company. *Journal of Software Engineering & Intelligent Systems*, 4(1), 41-57.

Sudhakar, G.P. (2012). A model of critical success factors for software projects. *Journal of Enterprise Information Management*, 25(6), 537-558. doi: 10.1108/17410391211272829.

Swanson, E.B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering* (p. 492-497). San Francisco, CA, USA.

Takeuchi, M., Kohtake, N., Shirasaka, S., Koishi, Y., & Shioya, K. (2014). Report on an assessment experience based on ISO/IEC 29110. *Journal of Software: Evolution and Process*, 26(3), 306-312. doi: 10.1002/smr.1591.

Tonella, P., Torchiano, M., Du Bois, B., & Systä, T. (2007). Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5), 551-571. doi: 10.1007/s10664-007-9037-5.

Tripathy, P., & Naik, K. (2015). *Software evolution and maintenance: a practitioner' s approach*. Hoboken, New Jersey, USA: John Wiley & Sons. doi: 10.1002/9781118964637.

van Vliet, H. (2008). *Software engineering: Principles and practice, 3rd edition*. Chichester, England: John Wiley & Sons.

von Wangenheim, C.G., Varkoi, T., & Salviano, C.F. (2006). Standard based software process assessments in small companies. *Software Process: Improvement and Practice*, 11(3), 329-335. doi: 10.1002/spip.276.

West, M. (2013). *Return On Process (ROP): Getting Real Performance Results from Process Improvement*. Boca Raton, FL, US: CRC Press. doi: 10.1201/b14053.

Wiegers, K., & Beatty, J. (2013). *Software requirements, 3rd edition*. Redmond, Washington, USA: Microsoft Press.

Yamamoto, S. (2017). An Evaluation of Requirements Specification Capability Index. *Procedia Computer Science*, 112, 998-1006. doi: 10.1016/j.procs.2017.08.080.

Zarour, M., Abran, A., Desharnais, J-M., & Alarifi, A. (2015). An investigation into the best practices for the successful design and implementation of lightweight software process assessment methods: A systematic literature review. *Journal of Systems and Software*, 101, 180-192. doi: 10.1016/j.jss.2014.11.041.

Zowghi, D., & Coulin, C. (2005). Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In A. Aurum & C. Wohlin (Eds.) *Engineering and managing software requirements*, (p. 19-46). Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. doi: 10.1007/3-540-28244-0_2.

Стандарди

У овом додатку је наведена листа међународних стандарда који се користе у софтверском инжењерству, а могу помоћи у успешнијем сагледавању и разумевању тема обрађених у овој књизи.

IEEE 12207:2008 - ISO/IEC/IEEE international standard - systems and software engineering - software life cycle processes. Software & Systems Engineering Standards Committee, IEEE Computer Society. New York, NY, USA. 2008. doi: 10.1109/IEEESTD.2008.4475826.

IEEE std. 610.12-1990, IEEE standard glossary of software engineering terminology. Standards coordinating committee of the Computer Society of the IEEE, IEEE Computer Society. New York, NY, USA. 1990. doi: 10.1109/IEEESTD.1990.101064.

IEEE Std 830-1998(R2009)(Revision of IEEE Std 830-1993), IEEE Recommended Practice for Software Requirements Specifications. Software Engineering Standards Committee, IEEE Computer Society. New York, NY, USA. 2009. doi: 10.1109/IEEESTD.1998.88286.

ISO/IEC/IEEE 29148:2011(E) International Standard. Systems and software engineering - Life cycle processes - Requirements engineering. IEEE Standards Activities Department. Institute of Electrical and Electronics Engineers (IEEE). Piscataway, NJ, USA. 2011. doi: 10.1109/IEEESTD.2011.6146379.

European Standard EN 13306:2017, Maintenance - Maintenance terminology. European Committee for Standardization. Bruxelles, Belgium. 2017.

IEEE Std 1219-1998, IEEE Standard for Software Maintenance. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 1998. doi: 10.1109/IEEESTD.1998.88278.

14764-2006 - ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes - Maintenance. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2006. doi: 10.1109/IEEESTD.2006.235774.

ISO/IEC 15504: Information technology – Process assessment. International Standards Organization. Geneve, Switzerland. 2004.

ISO/IEC/IEEE 15288:2015 Systems and software engineering - System life cycle processes. International Standards Organization. Geneve, Switzerland. 2015.

IEEE 1062-2015 - IEEE Recommended Practice for Software Acquisition. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2015. doi: 10.1109/IEEESTD.2016.7419835.

IEEE 1220-2005 - IEEE Standard for Application and Management of the Systems Engineering Process. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2005. doi: 10.1109/IEEESTD.2005.96469.

IEEE 1028-2008 - IEEE Standard for Software Reviews and Audits. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2008. doi: 10.1109/IEEESTD.2008.4601584.

IEEE 1074-2006 - IEEE Standard for Developing a Software Project Life Cycle Process. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2006. doi: 10.1109/IEEESTD.2006.219190.

IEEE 1063-2001 - IEEE Standard for Software User Documentation. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2001. doi: 10.1109/IEEESTD.2001.93368.

IEEE 828-2012 - IEEE Standard for Configuration Management in Systems and Software Engineering. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2012. doi: 10.1109/IEEESTD.2012.6170935.

IEEE 730-2014 - IEEE Standard for Software Quality Assurance Processes. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2014. doi: 10.1109/IEEESTD.2014.6835311.

IEEE 1044-2009 - IEEE Standard Classification for Software Anomalies. The Institute of Electrical and Electronics Engineers. New York, NY, USA. 2009. doi: 10.1109/IEEESTD.2010.5399061.

IEEE 829-2008 - IEEE Standard for Software and System Test Documentation. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2008. doi: 10.1109/IEEESTD.2008.4578383.

IEEE 1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2016. doi: 10.1109/IEEESTD.2017.8055462.

ISO/IEC/IEEE 15939:2017 Systems and software engineering - Measurement process. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2017. doi: 10.1109/IEEESTD.2017.7907158.

IEEE 1540-2001 - IEEE Standard for Software Life Cycle Processes - Risk Management. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2001. doi: 10.1109/IEEESTD.2001.92418.

16326-2019 - ISO/IEC/IEEE International Standard - Systems and software engineering - Life cycle processes - Project management. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2019. doi: 10.1109/IEEESTD.2019.8932690.

IEEE 1490-2003 - IEEE Guide Adoption of PMI Standard - A Guide to the Project Management Body of Knowledge. The Institute of Electrical and Electronics Engineers (IEEE). New York, NY, USA. 2003. doi: 10.1109/IEEESTD.2004.94565.

ISO/IEC TR 29110 Systems and software engineering - Lifecycle profiles for Very Small Entities (VSEs). International Standards Organization. Geneva, Switzerland. 2016.

ISO/IEC/IEEE 24748-1:2018 Systems and software engineering - Life cycle management - Part 1: Guidelines for life cycle management. International Standards Organization. Geneva, Switzerland. 2018.

ISO/IEC/IEEE 15289:2011 Systems and software engineering - Content of life-cycle information products (documentation). International Standards Organization. Geneva, Switzerland. 2011.

ISO/IEC 20000-1:2018 Information technology - Service management - Part 1: Service management system requirements. International Standards Organization. Geneva, Switzerland. 2018.