PROGRAMSKO INŽENJERSTVO

Uvodno predavanje

Sadržaj

- Organizacija kolegija
- O programskom inženjerstvu

Predavanja

- Tematske cjeline:
  - *1. dio: metodologije programskog inženjerstva*
    - *Programsko inženjerstvo*
    - *Programski proizvod*
    - *Agilne metodologije razvoja*
  - *2. dio: oblikovanje, implementacija i testiranje*
    - *Agilno planiranje*
    - *Specifikacija zahtjeva*
    - *Procjenjivanje trajanja implementacije*
    - *Modeliranje UML dijagramima*
    - *Clean code i najbolje prakse prilikom programiranja*

Laboratorijske vježbe

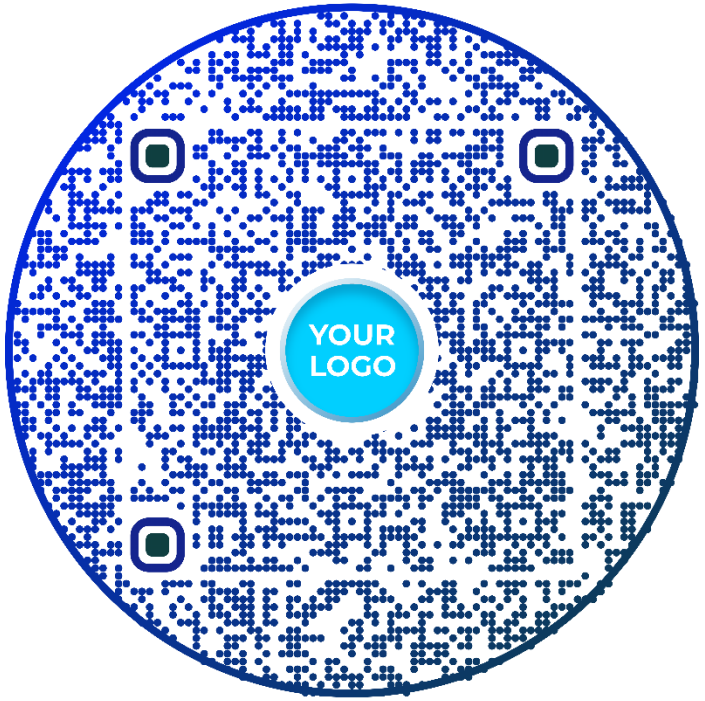
- 1. dio: Metodologije programskog inženjerstva
  - Programsko inženjerstvo
  - Programski proizvod
  - Agilne metodologije razvoja programskog proizvoda
- 2. dio: Oblikovanje, implementacija, testiranje
  - Zahtjevi i specifikacija programskog proizvoda
  - Oblikovanje programskog proizvoda
  - Testiranje programskog proizvoda
  - *Clean Code*

Laboratorijske vježbe

- Praktični zadaci:
  - *Agilno planiranje*
  - *Poslovna analiza*
  - *Specificiranje zahtjeva*
  - *Modeliranje arhitekture*
  - *Testiranje*
  - *Korištenje najboljih praksi prilikom programiranja*
- Korištene tehnologije:
  - *Open source*
  - *Java, Spring, C#, .NET, Python, Django, NextJs…*

O programskom inženjerstvu

- Što ćete raditi kad završite studij?



https://docs.google.com/forms/d/17rLSDIR9Se_441xxBdQ8l8rHFDgRV48R1cPvO8BukPc/edit#responses

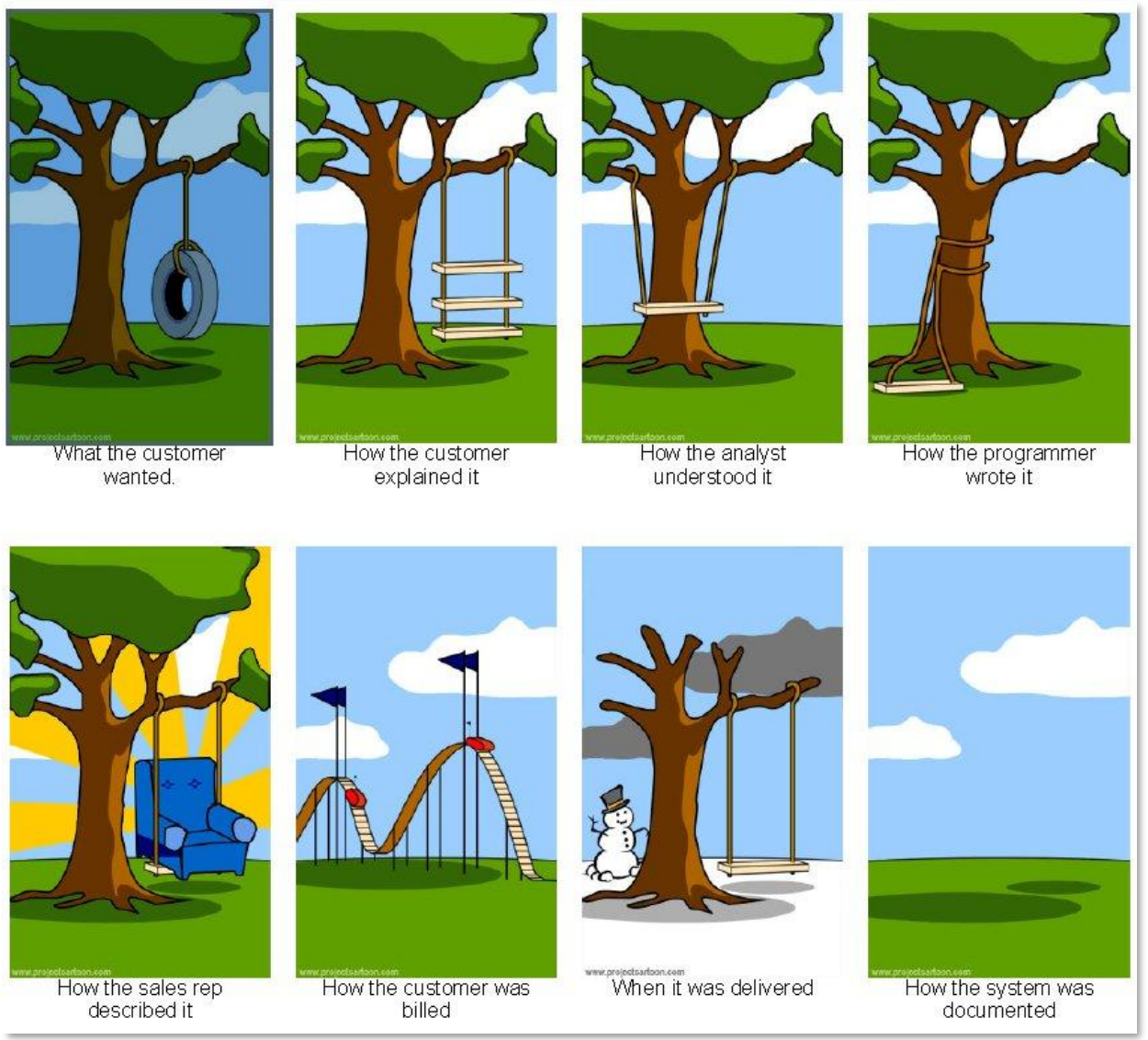
O programskom inženjerstvu

- Koji su izazovi posla koji ćete raditi?

https://docs.google.com/forms/d/1sKBixNxOlQzJUHsro77pWeNk4HBlvBTue5pt1b6EtJ0/edit

ALGEBRA BERNAYS SVEUČILIŠTE

O programskom inženjerstvu



O programskom inženjerstvu

- Troškovi razvoja softvera
- Specifičnosti softvera
- Je li programiranje dovoljno za uspjeh u karijeri?
- Tko je sve uključen u razvojni projekt?
- Što sve obuhvaća jedan softverski produkt?
- Što je programsko inženjerstvo?
- Koji je utjecaj AI-a na programsko inženjerstvo?

Troškovi vezani uz softver

- Što je skuplje, hardverski dio ili softverski dio?

- https://techjdi.com/blog/software-development-cost-breakdown
- https://www.cleveroad.com/blog/software-development-cost/
- https://idealink.tech/blog/software-development-maintenance-true-cost-equation

Troškovi vezani uz softver

- Što su vezani troškovi uz razvoj softvera?

- https://techpulsion.com/hr/tro%C5%A1ak-razvoja-softvera/
- https://duplico.io/razvoj-softvera-unutar-vlastite-tvrtke-ili-pronalazak-vanjskog-tima-za-razvoj/

Troškovi vezani uz softver

- Razvoj softvera

- Troškovi licenci za softverske platforme
- Troškovi analize i specifikacije
- Troškovi dizajna
- Troškovi testiranja i validacije
- Troškovi *deploymenta*
- Troškovi održavanja

Troškovi vezani uz softver

- Novi trendovi razvoja softvera
  - Umjetna inteligencija kao standard alata za razvoj
  - *Cloud-native* arhitekture i mikroservisi
  - Novi programski jezici (Rust, WebAssembly)
  - Poboljšanje iskustva i produktivnosti programera (DevEx)
  - Sigurnost i DevSecOps (*https://www.youtube.com/watch?v=-8BZfEaGb-w*)
  - *Low-code*/*no-code* platforme
  - *IoT* i *Edge computing*
  - *Quantum computing*

Utjecaj AI na razvoj softvera

- The Future of Software Development (2025 & Beyond): https://www.youtube.com/watch?v=KATAoOtxSqQ
- Top 8 Software & IT Trends to Watch in 2025: https://www.youtube.com/watch?v=UYQYlzEt3ps
- Learning Software Engineering During the Era of AI - https://www.youtube.com/watch?v=w4rG5GY9IIA
- What Software Engineering Jobs Will Look Like in 2025 (https://www.youtube.com/watch?v=5YgYHee6EQ4

Posebnosti softvera kao proizvoda

- Kompleksnost
- Visoko-tehnološki
- Logički (nematerijalni) proizvod
- Jedinstven i često individualno prilagođeni korisnicima ili tržištu
- Integriran u sve elemente društva, poslovanja i života
- Izrazita važnost u društvu

Posebnosti softvera kao proizvoda

- Kompleksnost
- Visoko-tehnološki
- Logički (nematerijalni) proizvod
- Jedinstven i često individualno prilagođeni korisnicima ili tržištu
- Integriran u sve elemente društva, poslovanja i života
- Izrazita važnost u društvu

Tehnologije koje se koriste kod razvoja softvera

- **Razvojni alati i okviri**: Visual Studio, IntelliJ IDEA, Eclipse, JetBrains alati, React, Angular, Spring, Django, .NET, Node.js i mnogi drugi za brzu i efikasnu implementaciju.
- **Programski jezici**: Java, Python, JavaScript, C#, C++, Rust, Go, Kotlin i drugi, prilagođeni različitim projektima i domenama.
- **Sustavi za upravljanje verzijama**: Git (GitHub, GitLab, Bitbucket), koji omogućuju suradnju, praćenje promjena i kontrolu verzija koda.
- **Alati za kontinuiranu integraciju i kontinuiranu isporuku (CI/CD)**: Jenkins, Travis CI, CircleCI, GitHub Actions, koje automatiziraju build, testiranje i deployment.
- **Sustavi za upravljanje projektima i suradnju**: Jira, Trello, Asana, Confluence, Slack, Microsoft Teams – za planiranje, praćenje zadataka i komunikaciju tima.

Tehnologije koje se koriste kod razvoja softvera

- **Testni alati**: Selenium, JUnit, TestNG, Postman, Cucumber, SonarQube za automatsko testiranje, integracijsko testiranje, testiranje API-ja i provjeru kvalitete koda.
- **Tehnologije za upravljanje bazama podataka**: SQL baze (MySQL, PostgreSQL, MS SQL), NoSQL baze (MongoDB, Cassandra), alati za dizajn i migraciju podataka.
- **Cloud platforme**: AWS, Azure, Google Cloud za hosting, skaliranje i upravljanje infrastrukturom.
- **Alati za virtualizaciju i kontejnerizaciju**: Docker, Kubernetes za upravljanje kontejnerima i orkestraciju.
- **Sigurnosni alati**: alati za statičku i dinamičku analizu sigurnosti, kao i alati za praćenje ranjivosti, autentifikaciju i autorizaciju.
- **Alati za upravljanje konfiguracijom**: Ansible, Chef, Puppet za automatizaciju postavki infrastrukture.
- **Razvoj softvera s umjetnom inteligencijom**: AI asistenti u kodiranju (GitHub Copilot) i alati za automatizaciju testiranja i analizu podataka.

Uzroci raznovrsnosti softverskih proizvoda

- **Različite domene primjene**: Softver se razvija za različite industrije i zadatke, poput financija, zdravstva, obrazovanja, zabave, industrijske automatizacije, telekomunikacija, vojne i svemirske tehnologije što generira različite vrste softverskih proizvoda.
- **Varijacija platformi i uređaja**: Softver se razvija za desktop, web, mobilne uređaje, ugrađene sustave i IoT, što zahtijeva različite tehnologije i pristupe razvoju.
- **Različiti korisnički zahtjevi i očekivanja**: Svaka grupa korisnika ima svoje specifične funkcionalnosti, performanse, sigurnosne i UX zahtjeve, što utječe na dizajn i implementaciju softvera.

Uzroci raznovrsnosti softverskih proizvoda

- **Tehnološki razvoj**: Pojava novih programskih jezika, okvira, alata, kao i promjene u infrastrukturi (npr. cloud, edge computing) doprinose razvoju različitih vrsta i stilova softverskih proizvoda.
- **Različiti modeli isporuke i održavanja**: Softver može biti komercijalni paket, *open-source*, prilagođeni softver po narudžbi ili SaaS (*software-as-a-service*), što zahtijeva različite pristupe u razvoju i distribuciji.
- **Globalizacija i tržišni zahtjevi**: Softver se koristi u različitim zakonodavnim, jezičnim i kulturnim okruženjima, što također povećava raznovrsnost softverskih rješenja.

Karakteristike dobrog softverskog proizvoda

- **Koristan je** – ispunjava konkretne potrebe i zahtjeve korisnika, pruža funkcionalnosti koje korisnici doista trebaju.
- **Pouzdan i siguran** – sustav radi predvidivo, stabilno i sigurno, bez neželjenih grešaka koje bi mogle izazvati štetu korisnicima ili poslovanju.
- **Efikasan** – optimalno koristi resurse sustava (memoriju, procesorsku snagu, mrežu itd.) te pruža zadovoljavajuće performanse.
- **Lagan za korištenje (upotrebljiv)** – ima intuitivan i pristupačan korisnički sučelje koje smanjuje napore i frustracije korisnika.

Karakteristike dobrog softverskog proizvoda

- **Održiv i fleksibilan** – lako se može mijenjati, prilagođavati i nadograđivati novim zahtjevima i tehnologijama.
- **Testabilan** – omogućava pouzdano testiranje svih komponenti kako bi se osigurala kvaliteta kroz cijeli životni ciklus.
- **Prenosiv** – može se koristiti u različitim okruženjima i platformama bez značajnih prilagodbi.
- **Pridržava** se standarda i normi – poštuje propise i standarde koji su relevantni za domen softverskog proizvoda.
- **Razvijen u planiranim okvirima** – razvoj je dovršen unutar rokova i budgeta

Razlozi neuspjeha softverskog projekta

- **Nedostatak jasno definiranih ciljeva** i opsega projekta, što dovodi do nesporazuma i lošeg upravljanja očekivanjima.
- **Loša komunikacija** između projektnog tima, krajnjih korisnika i uprave, što rezultira pogrešnim razumijevanjem zahtjeva i potrebe korisnika.
- **Nedovoljna podrška uprave** i menadžmenta koja može rezultirati nedostatkom resursa i odlučujuće podrške.

Razlozi neuspjeha softverskog projekta

- **Neadekvatno upravljanje resursima** i preopterećenje članova tima, osobito vanjskih konzultanata koji mogu raditi na više projekata istovremeno.
- **Nedostatak ažurne i kvalitetne dokumentacije**, kao i loše razumijevanje i modeliranje poslovnih procesa.
- **Nerealni rokovi i proračuni** koje se ne uspijevaju poštovati zbog tehničkih poteškoća i složenosti projekta.
- **Neadekvatna primjena metodologija razvoja** i loše upravljanje projektom, uz nepridržavanje standarda i praksi.

Razlozi neuspjeha softverskog projekta

“When projects fail, it’s rarely technical.”

Jim Johnson, The Standish Group

“Software costs more to maintain than it does to develop.”

„Roughly 60% of SW costs are development costs, 40% are testing costs.”

Ian Somerville

“Key software challenges: coping with increased diversity, demands for reduced delivery times and developing trustworthy software.”

Ian Somerville

Kako prevenirati najčešće rizike u IT projektima?

- **Proaktivna identifikacija rizika** – kontinuirano prepoznavanje i analiza potencijalnih rizika kroz cijeli projekat koristeći metode poput brainstorming-a, kontrolnih lista i analiza iskustava iz prethodnih projekata.
- **Primjena agilnih metoda** – agilni razvoj softvera omogućava fleksibilnost, brzu prilagodbu promjenama i učestale iteracije, čime se smanjuje opasnost da se isporuči neadekvatan proizvod ili da projekt zakasni.
- **Jasno definirani ciljevi i očekivanja** – osiguravanje da su zahtjevi, ciljevi i opseg projekta dobro razumljivi svim sudionicima, uključujući i zainteresirane strane (stakeholdere).
- **Efektivna komunikacija i suradnja** – uspostavljanje redovnih sastanaka, transparentna komunikacija i timska suradnja između svih članova projekta i korisnika.

Kako prevenirati najčešće rizike u IT projektima?

- **Planiranje rezervi** – uključivanje vremenskih i financijskih rezervi u plan projekta za nepredviđene situacije ili tehničke izazove.
- **Kontinuirani nadzor i kontrola napretka** – praćenje svih faza projekta u odnosu na plan, pravovremeno reagiranje na odstupanja i rizike.
- **Korištenje alata za upravljanje projektima i rizicima** – sustavi poput Jira, Trello, ili specijalizirani alati za praćenje rizika koji pomažu u organizaciji i evidenciji.
- **Obuka i angažman tima** – osiguravanje da svi članovi razumiju svoje uloge u upravljanju rizicima te da su educirani za rad u dinamičnom okruženju.

Povijest programskog inženjerstva

- Naziv "programsko inženjerstvo" (engl. *software engineering*) populariziran je na NATO-ovoj konferenciji 1968. godine u Garmischu, kao odgovor na "softversku krizu" koja je tada prijetila.
- Tokom 1960-ih službeno je formalizirano programsko inženjerstvo, s naglaskom na metodologije i procese koji bi pomogli rješavati rastuću složenost softvera.
- Razvoj programskih jezika visokog nivoa (poput C-a 1970-ih) i pojava UNIX operativnog sustava, postavili su temelje za moderni softverski razvoj.
- U 1990-ima internet i otvoreni izvorni kod izrazito su promijenili način na koji se softver razvija, potičući suradnju i brži razvoj.

- History of Programming Languages: https://www.youtube.com/watch?v=wDAxXIrZW3M

Naučene lekcije kroz povijest

- **Nova znanja**:
  - Automatizacija procesa kroz **CI/CD alate** (GitHub Actions, GitLab CI, Jenkins)
  - Integracija **AI alata** u razvoj (npr. Copilot, ChatGPT, Codeium)
  - Upotreba ***Cloud Development Environmenta*** (Gitpod, Codespaces)
  - Standardizacija i ponovno iskorištavanje koda putem „*packet managera*” (npm, pip, Maven)
- **Dobre prakse**:
  - „Shift-left testing” – s testiranjem se počinje već u fazi dizajna
  - „Infrastructure as Code” – konfiguracija i postavljanje sustava kroz izvorni kod
  - „Documentation as Code” – verzionirana dokumentacija uz izvorni kod

Naučene lekcije kroz povijest

- **CASE (*Computer Aided Software Engineering*) alati** :
  - **UML i modeliranje**: Visual Paradigm, Lucidchart, Draw.io
  - **Projektno upravljanje**: Atlassian Jira, Trello, ClickUp
  - **Automatizirano testiranje**: Selenium, Postman, Cypress
  - **DevOps i CI/CD**: GitHub Actions, Jenkins, Docker
- **Dobre prakse**:
  - „Shift-left testing” – s testiranjem se počinje već u fazi dizajna
  - „Infrastructure as Code” – konfiguracija i postavljanje sustava kroz izvorni kod
  - „Documentation as Code” – verzionirana dokumentacija uz izvorni kod

- What is DevOps: https://www.youtube.com/watch?v=0yWAtQ6wYNM

Inženjerstvo - primjer

- **Usporedba građevinskog inženjerstva i programskog inženjerstva:**

Faza / Pitanje	**Građevinarstvo** 🏗️	**Programsko inženjerstvo** 💻
Analiza zahtjeva	Arhitekt izrađuje nacrt i plan	Analitičar izrađuje specifikaciju zahtjeva
Dizajn	Inženjer definira strukturu	Dizajner sustava i UX/UI dizajner definiraju arhitekturu *backenda* i *frontenda*
Izgradnja	Građevinari i izvođači grade	Programeri implementiraju kod
Testiranje	Tehnički pregled	Testiranje i validacija

Inženjerski principi

Princip	Značenje	Primjer u praksi
Definiranost i mjerljivost	Svaki zahtjev mora biti jasan, testabilan i mjerljiv.	„Login funkcija mora omogućiti prijavu unutar 2 sekunde.”
Konzistentnost i ponovljivost	Korištenje standardnih procesa daje stabilne rezultate.	CI/CD pipeline uvijek reproducira isti build.
Predvidljivost	Planiranje vremena i troškova na temelju iskustva i metrika.	Agile sprint review predviđa opseg idućeg sprinta.

Principi programskog inženjerstva

- 🧪 Testabilnost → Unit testovi, CI/CD integracija
- 🔁 Konzistentnost → Git verzioniranje, *code review*
- 📊 Predvidljivost → Sprint *velocity*, *burndown* grafovi
- 🧩 Specifikacija → Jasan plan

- Kombiniranje metoda → DevOps kultura, hibridni pristupi (Scrum i Kanban)

- Principi programskog inženjerstva osiguravaju isporuku ispravnog softverskog proizvoda

Bitni pojmovi i definicije

- 🔺 Metodologija programskog inženjerstva ⚙️ → način i pravila kako se taj proced provodi
- 🔷 Programsko inženjerstvo 🧠 → disciplina i praksa koja vodi do proizvoda
- 🔶 Programski proizvod 💻 → rezultat rada (softver, dokumentacija, podaci, upute)

Pojam	Definicija	Primjer
Programski proizvod	Kombinacija programskog koda, dokumentacije i podataka koji čine funkcionalan sustav.	Mobilna aplikacija za naručivanje hrane.
Programsko inženjerstvo	Skup principa, metoda i alata za sustavno stvaranje kvalitetnog softvera.	Primjena Scrum-a, Git-a i CI/CD-a.
Metodologija programskog inženjerstva	Organizirani pristup koji definira *kako* se softver razvija.	Scrum, Kanban, XP, Waterfall.

Programski proizvod – više od samog koda

- Upute za instalaciju i korištenje
- Specifikacija okoline (OS, DB, API, Cloud)
- Podaci i konfiguracije
- Dokumentacija sustava i zahtjeva
- Programska logika (kod)

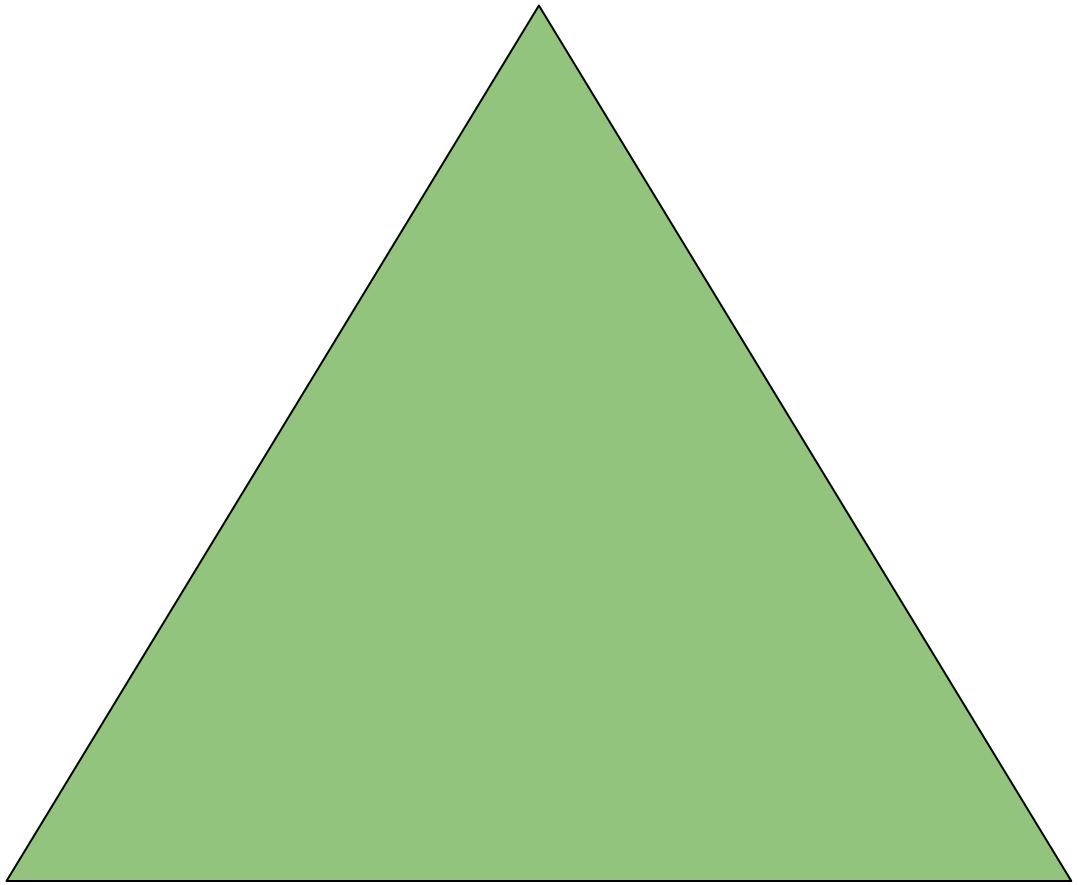
Kategorija	Svrha	Primjer
Kod	Funkcionalnost sustava	Python backend
Dokumentacija	Razumijevanje i održavanje	README, UML
Podaci	Konfiguracija i sadržaj	Baza podataka
Okruženje	Uvjeti izvođenja	Docker, AWS
Upute	Postavljanje i korištenje	Install guide

Programski proizvod – više od samog koda

- Upute za instalaciju i korištenje
- Specifikacija okoline (OS, DB, API, Cloud)
- Podaci i konfiguracije
- Dokumentacija sustava i zahtjeva
- Programska logika (kod)

Kategorija	Svrha	Primjer
Kod	Funkcionalnost sustava	Python backend
Dokumentacija	Razumijevanje i održavanje	README, UML
Podaci	Konfiguracija i sadržaj	Baza podataka
Okruženje	Uvjeti izvođenja	Docker, AWS
Upute	Postavljanje i korištenje	Install guide

Softverski produkt



Softverski produkt

Računalni programi

Softvjerski kod koji pruža traženu funkcionalnost i performanse - mozak cijelog sustava koji izvršava logiku i algoritme.

> 💡 ***Zanimljivost:*** *Prosječna web aplikacija ima ~50,000 linija koda, dok operativni sustav poput Windowsa ima preko 50 milijuna!*

Strukture podataka

Podaci koji omogućuju ispravan rad programa - without data, software je samo prazna ljuska. Organizacija podataka je ključna!

> 💡 ***Zanimljivost:*** *~90% modernih aplikacija zapravo upravlja podacima više nego što izvršava kompleksne kalkulacije!*

Dokumentacija

Dokumenti koji opisuju rad i korištenje programa - često zanemarena, ali kritična komponenta za održavanje i razvoj softvera.

> 💡 ***Zanimljivost:*** *Programeri provode 60% svog vremena čitajući kod i dokumentaciju, a samo 40% pišući novi kod!*

Programsko inženjerstvo

Software Engineering

PI SE

🎯

Znanstvena i stručna disciplina koja se bavi **svim aspektima proizvodnje softvera** - od početne ideje do finalnog proizvoda i njegovog održavanja.

🏛️ Tri stupa programskog inženjerstva

Procesi

Strukturirani koraci razvoja softvera: planiranje, analiza, dizajn, implementacija, testiranje i održavanje.

Agile, Scrum, DevOps

Metodologije

Sistemski pristupi i best practices koje osiguravaju kvalitetu, efikasnost i organizaciju tima kroz cijeli projekt.

Waterfall, Kanban, XP

Alati

Softverski tools koji olakšavaju razvoj: IDE, version control, CI/CD, testing frameworks, project management.

Git, Docker, Jenkins

BITNI POJMOVI I DEFINICIJE

Proces programskog inženjerstva

Metodologija

Method, Methodology

Preskriptivan način dolaska do cilja - definira **KAKO** se dolazi do rezultata na konzistentan, ponovljiv način

✨ Ključna karakteristika

Metodologija daje **jasne upute, korake i pravila** koje treba slijediti. To je recept za uspjeh!

🎯 Fokus: KAKO radimo

VS

Proces

Process, Framework

U širem smislu - krovni naziv za **skup aktivnosti** u razvoju softvera

✨ Ključna karakteristika

U užem kontekstu: Ne definira način KAKO se dolazi do rezultata - to je fleksibilniji okvir rada

🎯 Fokus: ŠTO radimo

🔍 Vizualna usporedba

📔 Metodologija = Recept

Kao recept za kolač: koristi 200g brašna, dodaj 3 jajeta, miješaj 5 minuta, peci na 180°C...

Primjer: Scrum točno definira daily standup, sprint planning, retrospective

⚙️ Proces = Framework

Kao plan: trebamo sastojke, miješanje, pečenje - ali TI odlučuješ kako i kojim redoslijedom

Primjer: "Razvijamo softver iterativno" - ali ne definira kako točno

🎯 Metodologija je stroga

Ima specifične korake, uloge, artefakte i pravila koja se moraju poštovati

Karakteristika: Mala fleksibilnost, velika konzistentnost

🔄 Proces je fleksibilan

Daje okvir za organizaciju rada, ali timovi mogu prilagoditi pristup

Karakteristika: Velika fleksibilnost, adaptivnost

💼 Real-world primjeri

📋 Scrum (Metodologija)

Točno definirani sprint-ovi (2-4 tjedna), daily standup (15 min), uloge (Product Owner, Scrum Master), eventi (planning, review, retro)

⚙️ Agile (Proces/Framework)

Iterativni razvoj, kontinuirana isporuka, adaptivnost - ALI ne kaže KAKO točno to postići

📋 Kanban (Metodologija)

Specifične kolone (To Do, In Progress, Done), WIP limiti, vizualni board, pull sistem

⚙️ DevOps (Proces/Framework)

Integracija dev-a i ops-a, kontinuirana isporuka - ALI nije preskriptivan o alatima ili koracima

BITNI POJMOVI I DEFINICIJE

Proces programskog inženjerstva

⚠️ **Proces u užem smislu VS metodologija**

Ne definira KAKO se dolazi do rezultata

CMM Model zrelosti razvojnog procesa

Capability Maturity Model (CMM)

Model koji mjeri **učinkovitost i zrelost** softverskog razvojnog procesa. Omogućava organizacijama da ocijene gdje se nalaze i kamo trebaju ići.

1980-ih: US Air Force podizvođači s isporukom

Američka vojska imala je **ogroman problem** - softverski projekti konstantno kasne, prelaze budžet i ne ispunjavaju zahtjeve. Trebalo je rješenje!

Software Engineering Institute (SEI)

Osnovan da **istraži problem** i razvije standardizirani pristup procjeni i poboljšanju softverskih procesa u obrambenom sektoru.

Definiran CMM (1991)

SEI razvija **Capability Maturity Model** - revolucionarni framework s 5 razina zrelosti koji omogućava mjerenje i poboljšanje procesa.

🎯 Svrha CMM modela

Mjerenje zrelosti

Objektivno procjenjuje koliko je razvojni proces organizacije razvijen, dokumentiran i standardiziran

Kontinuirano poboljšanje

Daje jasan put kako prijeći s jedne razine na drugu - što točno treba implementirati

Zajednički jezik

Universalni standard za sve metodologije i procese - omogućava usporedbu različitih organizacija

Smanjenje rizika

Organizacije s višom razinom zrelosti imaju predvidljivije rezultate, manje grešaka i bolju kvalitetu

📊 5 razina zrelosti CMM modela

1

Initial (Početna)

Kaos i ad-hoc pristup. Nema definiranih procesa, uspjeh ovisi o individualnim heroima. Projekti nepredvidivi, često prekoračuju budžet i deadline.

2

Repeatable (Ponovljiva)

Osnovni project management. Projekti se planiraju i prate, ali procesi nisu formalizirani. Timovi mogu ponoviti prethodne uspjehe.

3

Defined (Definirana)

Dokumentirani i standardizirani procesi. Svi projekti slijede iste procedure, postoji organizacijski standard koji se koristi i poboljšava.

4

Managed (Upravljana)

Mjerenje i kontrola. Procesi se kvantitativno mjere, organizacija koristi metrike za donošenje odluka i predviđanje performansi.

5

Optimizing (Optimizirajuća)

Kontinuirano poboljšanje. Organizacija fokusirana na inovacije, automatizaciju i proaktivno identificira i implementira poboljšanja procesa.

🔍 Detaljno kroz razine

🔴 1. Početni

Karakteristike: Bez formalnih procesa, nepredvidivi rezultati, uspjeh ovisi o pojedincima.

Primjer: Startup koji razvija MVP - sve je u glavama developera, nema dokumentacije.

🟠 2. Ponovljivi

Karakteristike: Uspostavljeni osnovni project management procesi, moguće ponoviti prethodne uspjehe.

Primjer: Tim počinje koristiti Jira za tracking taskova i definira osnovne procedure.

🟡 3. Definirani

Karakteristike: Standardizirani procesi za cijelu organizaciju, sve je dokumentirano.

Primjer: Kompanija ima Software Development Manual koji svi timovi koriste.

🟢 4. Upravljani

Karakteristike: Kvantitativno mjerenje procesa, predvidljive performanse pomoću metrika.

Primjer: Microsoft prati bug density, code coverage, velocity - i koristi to za predviđanja.

🔵 5. Optimizirani

Karakteristike: Fokus na kontinuiranom poboljšanju, proaktivno identificiranje problema.

Primjer: Netflix - automatizacija svega, culture of experimentation, constant innovation.

Ključne razlike između razina

Početni	Ponovljivi	Definirani	Upravljani	Optimizirani
✗ Nepredvidivo	✓ Praćenje projekata	✓ Standardizacija	✓ Metrike	✓ Automatizacija
✗ Reaktivno	✓ Basic planning	✓ Dokumentacija	✓ Predvidivost	✓ Inovacije
✗ Ovisi o ljudima	~ Djelomična kontrola	✓ Konzistentnost	✓ Data-driven	✓ Proaktivnost

Dva skupa metodologija

Tradicionalne metodologije

Traditional / Plan-Driven

Ključne karakteristike

- **Ekstenzivna dokumentacija** - sve mora biti detaljno zapisano
- **Rigidna struktura** - jasno definirane faze i koraci
- **Up-front planning** - sve se planira na početku projekta
- **Formalni procesi** - stroga pravila i procedure
- **Dugi ciklusi** - projekt traje mjesecima ili godinama
- **Teško mijenjanje** - promjene su skupe i kompleksne

Primjeri metodologija

Waterfall | V-Model | Spiral | RUP

VS

Agile metodologije

Agile / Lightweight

Ključne karakteristike

- **Minimalna dokumentacija** - working software over docs
- **Fleksibilnost** - prilagodba promjenama je laka
- **Iterativni razvoj** - kraći ciklusi i brze isporuke
- **Suradnja** - bliska komunikacija s klijentom
- **Kratki sprint-ovi** - 1-4 tjedna po iteraciji
- **Embrace change** - promjene su dobrodošle

Primjeri metodologija

Scrum | Kanban | XP | Lean

🔍 Detaljna usporedba

Planiranje

🏛 Tradicionalne

Sve se planira unaprijed. Detaljni plan za cijeli projekt. Teško mijenjanje plana.

🚀 Agile

Adaptivno planiranje. Sprint po sprint. Plan se prilagođava potrebama.

Dokumentacija

🏛 Tradicionalne

Ekstenzivna i formalna. Sve mora biti dokumentirano. Dokumentacija je ključna.

🚀 Agile

Minimalna i po potrebi. Fokus na working software. Just enough documentation.

Promjene

🏛 Tradicionalne

Promjene su rizik. Skupe i kompleksne. Change requests proces je težak.

🚀 Agile

Promjene su dobrodošle. Lako prilagodljive. Respond to change over following plan.

Klijent

🏛 Tradicionalne

Ograničena interakcija. Uglavnom na početku i kraju. Contract negotiation.

🚀 Agile

Kontinuirana suradnja. Klijent je dio tima. Customer collaboration.

Isporuka

🏛 Tradicionalne

Jedna velika isporuka na kraju. Mjeseci/godine čekanja. Big bang release.

🚀 Agile

Česte inkrementalne isporuke. Svaki sprint daje vrijednost. Continuous delivery.

Kada koristiti?

🏛 Tradicionalne

Stabilni zahtjevi. Regulirani projekti. Veliki, kompleksni sustavi (npr. banka, avijacija).

🚀 Agile

Promjenjivi zahtjevi. Inovativni produkti. Startup, web/mobile apps.

Proces programskog inženjerstva

Zajedničke faze svih metodologija

Bez obzira na metodologiju (Waterfall, Agile, Scrum...), sve prolaze kroz ove temeljne faze!

1

Specifikacija

Specification

Definiranje funkcionalnosti i ograničenja rada sustava

2

Oblikovanje

Design

Kreiranje arhitekture i dizajna sustava prema specifikaciji

3

Implementacija

Implementation

Ostvarenje softvera prema dogovorenoj specifikaciji i dizajnu

4

Validacija

Validation / Testing

Provjera ispravnosti i zadovoljavanje zahtjeva naručitelja

5

Evolucija

Evolution / Maintenance

Izmjene i poboljšanja po potrebi kroz vrijeme upotrebe

🔍 Detaljnije kroz faze

Specifikacija

Prva i ključna faza gdje se definira ŠTO softver treba raditi. Razgovor s klijentom, analiza potreba, definiranje funkcionalnih i nefunkcionalnih zahtjeva.

Aktivnosti:

- ✓ Prikupljanje zahtjeva od stakeholdera
- ✓ Analiza poslovnih procesa
- ✓ Definiranje use case-eva
- ✓ Kreiranje Requirements Document
- ✓ Prioritizacija feature-a

Oblikovanje i Implementacija

Dizajn arhitekture sustava i konkretno kodiranje. KAKO ćemo to implementirati? Koja arhitektura, koji design patterns, koja tehnologija?

Aktivnosti:

- ✓ System architecture design
- ✓ Database schema design
- ✓ UI/UX dizajn
- ✓ Pisanje koda
- ✓ Code review

Validacija i Testiranje

Osiguravanje da softver radi kako treba i zadovoljava zahtjeve. Pronalaženje i otklanjanje bugova prije nego što softver stigne do korisnika.

Aktivnosti:

- ✓ Unit testing
- ✓ Integration testing
- ✓ System testing
- ✓ User acceptance testing (UAT)
- ✓ Performance & security testing

Evolucija i Održavanje

Softver nikad nije "gotov"! Kontinuirano poboljšavanje, dodavanje novih feature-a, ispravljanje bugova, adaptacija na nove tehnologije i potrebe.

Aktivnosti:

- ✓ Bug fixing i hotfixes
- ✓ Dodavanje novih feature-a
- ✓ Performance optimization
- ✓ Security updates
- ✓ Refactoring legacy koda

PROCES PROGRAMSKOG INŽENJERSTVA

Aktivnosti procesa

Zajedničke aktivnosti (faze) svih metodologija

Linearni slijed

Specifikacija → **Oblikovanje** → **Ostvarenje** → **Validacija**

⚠️ Smislimo neku svoju metodologiju...

Mogu biti i isprepletene faze - ne mora uvijek biti strogo linearan slijed!

🌐 Utjecaj Web-a

Web je velik, jer je puno korišten i lako dostupan - promijenio je način kako razvijamo softver!

💡 Primjeri linearnog vs. iterativnog pristupa

Waterfall metodologija

Klasičan linearni pristup

Strogo linearan slijed gdje svaka faza mora biti potpuno završena prije nego što se krene na sljedeću. Nema vraćanja unazad!

Karakteristike:

- ✓ Specifikacija → Oblikovanje → Implementacija → Testiranje
- ✓ Svaka faza završava s dokumentom/outputom
- ✓ Teško vratiti se na prethodnu fazu
- ✓ Idealno za projekte sa stabilnim zahtjevima
- ✓ Primjer: Izgradnja bankovnog sustava

Agile/Scrum metodologija

Iterativni pristup - isprepletene faze

Faze se ponavljaju u kratkim ciklusima (sprint-ovima). Specifikacija, dizajn, implementacija i testiranje se dešavaju paralelno!

Karakteristike:

- ✓ Sprint-ovi od 1-4 tjedna
- ✓ Sve faze se dešavaju u svakom sprintu
- ✓ Kontinuirana isporuka radnog softvera
- ✓ Lako prilagodljivo promjenama
- ✓ Primjer: Razvoj mobilne aplikacije

🔍 Linearni vs. Iterativni - Kada koristiti?

📐 Linearni pristup (Waterfall)

Kada koristiti:
- Stabilni i jasno definirani zahtjevi
- Regulirani projekti (medicina, avijacija)
- Veliki, kompleksni sustavi
- Projekti sa fiksnim budžetom i rokom
- Klijent ne može biti često dostupan

Prednosti: Predvidljivost, jasna struktura, dobra dokumentacija
Nedostaci: Nefleksibilno, kasno otkrivanje problema

🔄 Iterativni pristup (Agile)

Kada koristiti:
- Promjenjivi zahtjevi
- Inovativni projekti
- Web/mobile aplikacije
- Startup okruženje
- Bliska suradnja s klijentom

Prednosti: Fleksibilnost, rana isporuka, prilagodljivost
Nedostaci: Manje predvidivo, zahtijeva disciplinu

🎯 Linearni - Primjer iz prakse

Projekt: Bankovni sustav za obradu transakcija

1. **Specifikacija (3 mj):** Detaljno definiranje svih zahtjeva, compliance, security
2. **Dizajn (2 mj):** Arhitektura, database schema, API design
3. **Implementacija (6 mj):** Kodiranje cijelog sustava
4. **Testiranje (2 mj):** Intenzivno testiranje, UAT
5. **Deployment:** Jedna velika isporuka

Trajanje: 13+ mjeseci

⚡ Iterativni - Primjer iz prakse

Projekt: E-commerce mobile app

Sprint 1 (2 tj): Login i registracija
Sprint 2 (2 tj): Product listing
Sprint 3 (2 tj): Shopping cart
Sprint 4 (2 tj): Payment integration
Sprint 5 (2 tj): Order tracking

Svaki sprint ima sve faze: spec → design → code → test

Trajanje: 10 tjedana, ali radna verzija nakon svakog sprinta!

Web i programsko inženjerstvo

Revolucija u distribuciji softvera

🌐 Prednosti Web-om isporučenih programskih proizvoda (sustava)

💰 Drugačiji business model

Pay-as-you-go & Subscription

- ✓ **Korisnici ne kupuju programski proizvod** - već plaćaju mjesečnu/godišnju pretplatu (SaaS model)
- ✓ **Koriste ga i plaćaju korištenje** - pay only for what you use, fleksibilna naplata
- ✓ **Niži inicijalni troškovi** - nema potrebe za velikim ulaganjem odjednom
- ✓ **Predvidljivi prihodi** - recurring revenue za developera, lakše planiranje budžeta za korisnike

🚀 Jednostavnije postavljanje

Zero Installation

- ✓ **Ne postoji instalacija na strani korisnika** - samo otvori preglednik i kreni!
- ✓ **Svi sustavi dostupni kroz preglednik** - Chrome, Firefox, Safari - radi na svemu!
- ✓ **Cross-platform kompatibilnost** - Windows, Mac, Linux, mobile - sve radi!
- ✓ **Instant access** - registriraj se i odmah počni koristiti, bez čekanja

🔄 Jednostavnije održavanje

Automatic Updates

- ✓ **Nova verzija ili zakrpa (patch)** - deploy jednom, svi korisnici odmah imaju novu verziju!
- ✓ **Korisnik ne mora ponovno prolaziti instalaciju** - sve se dešava "iza zavjese"
- ✓ **Centralizirano održavanje** - jedan server, svi korisnici, lakše za developer
- ✓ **Brže fixing bugova** - nađi bug ujutro, deploy fix popodne, svi imaju fix!

Desktop vs Web aplikacije

Desktop aplikacije (stari način)

Distribucija: Korisnik downloaduje .exe/.dmg file, instalira na svoje računalo

Problemi:
- Različite verzije za Windows/Mac/Linux
- Korisnik mora ručno downloadati update-e
- Compatibility issues sa različitim OS verzijama
- Zauzima prostor na hard disku
- Licenciranje je kompleksno

Primjer: Microsoft Office (tradicionalna verzija)

Web aplikacije (moderni način)

Distribucija: Korisnik otvori URL u pregledniku - that's it!

Prednosti:
- Jedna verzija za sve platforme
- Automatic updates - uvijek najnovija verzija
- Radi na svemu što ima preglednik
- Zero installation, zero storage
- Jednostavno licenciranje (subscription)

Primjer: Google Docs, Microsoft 365 (web verzija)

Business perspektiva

Za developera/kompaniju:
- Lako deploy nove verzije (CI/CD)
- Svi korisnici na istoj verziji - lakši support
- Predictable revenue (subscriptions)
- Lakše skaliranje (cloud infrastructure)
- Real-time monitoring i analytics

ROI: Niži troškovi razvoja i održavanja dugoročno

Korisničko iskustvo

Za korisnika:
- Pristup s bilo kojeg uređaja
- Automatski back-up podataka
- Collaborate in real-time
- Niži inicijalni trošak
- Uvijek najnovija verzija

Use case: Rad od kuće/ureda/putovanja - sve sinkronizirano!

🎯 Real-world primjeri Web aplikacija

📝 Google Docs

Word processing u pregledniku. Real-time collaboration, zero installation, free!

💼 Salesforce

CRM platforma. $150+ milijardi tržišna kap - sve web-based!

🎨 Figma

Dizajn tool u pregledniku. Zamijenio desktop Photoshop za mnoge!

📊 Notion

Note-taking i project management. Sve u cloud-u, sync svugdje!

🎵 Spotify

Music streaming. Milijuni pjesama, instant access, no download!

💬 Slack

Team communication. Real-time messaging, integrations, sve u web-u!

Web i programsko inženjerstvo

Doprinosi od razvoja Web sustava

Inkrementalna izgradnja

Iterative Development

Značajan doprinos razvoju agilnih metodologija - Web je pokazao da iterativni pristup funkcionira odlično!

💡 Kako je Web omogućio iterativni pristup?

🚀 Brzo deployment i testiranje

Web aplikacije mogu biti deployan-e i testirane u satima, ne mjesecima. Push to production i odmah vidiš rezultate!

📊 Real-time feedback od korisnika

Analytics, user behavior tracking, A/B testiranje - odmah vidiš što radi, a što ne. Prilagodi brzo!

🔧 Lako dodavanje novih feature-a

Deploy novu verziju, svi korisnici odmah imaju pristup. Rollback ako nešto krene po zlu - instant!

💰 Niži rizik i troškovi

Razvijaj u malim inkrementima, testiraj, prilagodi. Nije potrebno godinu dana čekati da vidiš hoće li funkcionirati.

Fokus na komponente

Component-Based Architecture

Skratiti vrijeme izgradnje - ponovno korištenje postojećih, testiranih komponenti umjesto pisanja svega ispočetka!

🎯 Prednosti component-based pristupa

♻️ Reusability - ponovno korištenje

Napravi button komponentu jednom, koristi je na 100 mjesta. Change once, update everywhere!

🧪 Lakše testiranje

Testiraj svaku komponentu zasebno (unit testing). Lakše pronaći i fixati bugove. Isolation je key!

👥 Bolja suradnja u timu

Developer 1 radi na header-u, Developer 2 na sidebar-u - neovisno! Manje merge conflicts.

📦 Ecosystem komponenti

npm, component libraries (Material-UI, Bootstrap) - tisuće ready-made komponenti. Don't reinvent the wheel!

🎯 Kako je Web utjecao na metodologije

Agile metodologije

Web je pokazao da kontinuirana isporuka i iterativni razvoj funkcioniraju! Scrum sprint-ovi su direktna posljedica Web-a.

- → 2-tjedni sprint-ovi umjesto 6-mjesečnih projekata
- → Continuous deployment i integration
- → MVP (Minimum Viable Product) pristup
- → Fail fast, learn faster mentalitet

DevOps kultura

Web je zahtijevao bržu integraciju developmenta i operations-a. CI/CD pipelines su postali norma.

- → Automated testing i deployment
- → Infrastructure as Code (Docker, Kubernetes)
- → Monitoring i observability
- → Dev i Ops rade zajedno, ne odvojeno

Component-Driven Development

React, Vue, Angular - svi promiču component-based architecture. Atomic design principles.

- → Reusable UI components
- → Storybook za component development
- → Design systems (Material, Ant Design)
- → Faster development, consistent UI

Lean Startup

Web je omogućio brzo testiranje ideja i pivot-iranje. Build-Measure-Learn loop.

- → MVP u tjednima, ne godinama
- → A/B testing i user feedback
- → Pivot based on data
- → Validate assumptions quickly

📊 Prije Web-a vs. Poslije Web-a

⏳ Prije Web-a (1980s-1990s)

Development cycle: 12-24 mjeseca
Deployment: CD-ROM distribucija
Updates: Jednom godišnje, ako i toliko
Testing: Beta testiranje s odabranim korisnicima
Feedback: Sporom poštom ili na konferencijama
Architecture: Monolitna aplikacija

Primjer: Microsoft Office 95 - 3 godine razvoja

🚀 Poslije Web-a (2000+)

Development cycle: 2-4 tjedna (sprint-ovi)
Deployment: Instant, cloud-based
Updates: Continuous, multiple times daily
Testing: A/B testing sa svim korisnicima
Feedback: Real-time analytics i metrics
Architecture: Microservices, komponente

Primjer: Google Docs - continuous updates

⏳ Tradicionalni pristup

Proces: Waterfall - sve odjednom
Komponente: Pišemo sve od nule
Rizik: Visok - bug se otkrije kasno
Cost of change: Jako visok - teško mijenjati
Time to market: Dugo čekanje

Problem: By the time you ship, requirements su se promijenili!

⚡ Web-enabled pristup

Proces: Agile - iterativno, inkrementalno
Komponente: React, npm packages - reuse everything
Rizik: Nizak - testiramo constantly
Cost of change: Nizak - deploy fix instantly
Time to market: Brzo - MVP u tjednima

Prednost: Respond to change quickly!

Procesni modeli programskog inženjerstva

Teorija i praksa

Planski (tradicionalni) proces

Plan-Driven / Waterfall Approach

Unaprijed se planiraju aktivnosti

Sve faze razvoja su detaljno definirane na početku projekta. Requirements, dizajn, implementacija - sve je isplanirano prije nego što se počne kodirati.

Napredak se mjeri u odnosu na plan

Projekt se prati kroz milestones i deliverables. "Jesmo li u planu?" je ključno pitanje. Gantt charts, project timeline-ovi.

Agilni proces

Agile / Iterative Approach

Planiranje je inkrementalno

Ne planiramo cijeli projekt unaprijed. Planiramo sprint po sprint, iteraciju po iteraciju. Adapt as you go!

Lakša izmjena procesa prema promjeni u naručiteljevim zahtjevima

Klijent je promijenio requirements? No problem! Prilagodimo se u sljedećem sprintu. Flexibility is key.

U praksi: Hibridni pristup

Hybrid / Best of Both Worlds

Kombiniraju se elementi planskog i agilnog procesa

Većina kompanija koristi "Agile-ish" pristup. Imaju sprint-ove (Agile), ali i roadmap za godinu dana (Plan-driven). Uzmi najbolje iz oba svijeta!

Detaljno kroz procesne modele

Planski proces

Waterfall metodologija

- **Kada koristiti:** Stabilni zahtjevi, regulirane industrije
- **Planiranje:** Sve unaprijed, detaljno
- **Dokumentacija:** Ekstenzivna
- **Promjene:** Teške i skupe
- **Isporuka:** Jedna velika na kraju
- **Primjer:** NASA software, medicinski uređaji

Agilni proces

Scrum, Kanban

- **Kada koristiti:** Promjenjivi zahtjevi, inovativni projekti
- **Planiranje:** Sprint po sprint
- **Dokumentacija:** Just enough
- **Promjene:** Dobrodošle
- **Isporuka:** Kontinuirana
- **Primjer:** Web apps, startup produkti

Hibridni pristup

Agile + Planning

- **Kada koristiti:** Većina real-world projekata
- **Planiranje:** Roadmap + sprint-ovi
- **Dokumentacija:** Balanced
- **Promjene:** Kontrolirane
- **Isporuka:** Inkrementalna s milestones
- **Primjer:** Enterprise software

Real-world scenariji

Planski proces u akciji

Projekt: Bankovni sustav za obradu transakcija

Zahtjevi: Stabilni, regulirani, compliance

Pristup: Sve se planira 12 mjeseci unaprijed

Mjesec 1-3: Requirements gathering
Mjesec 4-6: System design
Mjesec 7-10: Implementation
Mjesec 11-12: Testing & deployment

Agilni proces u akciji

Projekt: Social media startup app

Zahtjevi: Promjenjivi, inovativni

Pristup: 2-tjedni sprint-ovi

Sprint 1: User authentication
Sprint 2: Profile creation
Sprint 3: Feed feature
Sprint 4: Messaging (based on feedback)

Hibridni pristup u akciji

Projekt: E-commerce platforma

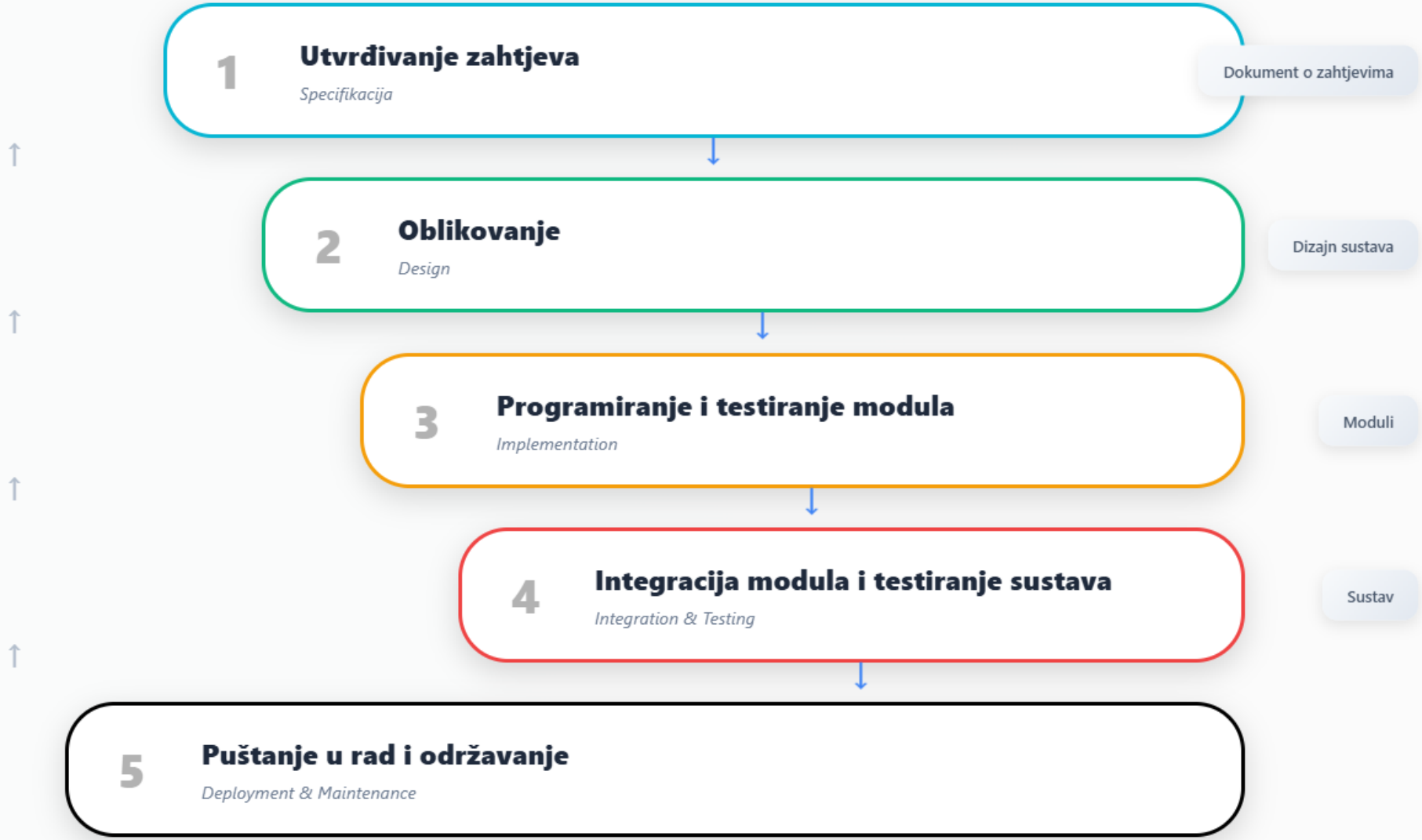
Zahtjevi: Miješani - core je stabilan, features mijenjaju

Pristup: Roadmap + Agile

Q1 Roadmap: Payment system (planirano)
Q2 Roadmap: Inventory management
Svaki kvartal: Agile sprint-ovi za features
Flexibility za nove ideje

Procesni modeli programskog inženjerstva

Vodopadni model



Ključne karakteristike Vodopadnog modela

📋 Sekvencijalni pristup

Svaka faza mora biti potpuno završena prije nego što se krene na sljedeću. Nema paralelnog rada na fazama.

📄 Ekstenzivna dokumentacija

Svaka faza proizvodi detaljnu dokumentaciju koja je input za sljedeću fazu. Requirements → Design → Code.

↩️ Povratna veza

Isprekidane linije pokazuju da je moguće vratiti se na prethodnu fazu, ali je to skupo i kompleksno.

⏱️ Dugi ciklusi

Projekti traju mjesecima ili godinama. Jedna velika isporuka na kraju, ne inkrementalno.

Detaljno kroz faze

Prednosti modela

✓ **Jednostavnost:** Lako razumjeti i implementirati

✓ **Struktura:** Jasno definirane faze i milestones

✓ **Dokumentacija:** Sve je dokumentirano, lako za prenos znanja

✓ **Stabilnost:** Dobro za projekte sa stabilnim zahtjevima

Nedostaci modela

✗ **Nefleksibilnost:** Teško prilagoditi se promjenama

✗ **Kasno testiranje:** Problemi se otkrivaju kasno u procesu

✗ **Nema working softvera:** Korisnik vidi proizvod tek na kraju

✗ **Visok rizik:** Ako requirements nisu dobri, cijeli projekt je u problemu

Kada koristiti?

1. Stabilni zahtjevi: Requirements su jasni i neće se mijenjati

2. Regulirane industrije: Medicina, avijacija, obrana

3. Veliki sustavi: Kompleksni projekti sa puno dokumentacije

4. Poznata tehnologija: Nema rizika, sve je već korišteno

Real-world primjer

Projekt: Sustav za upravljanje zračnim prometom

Zašto Waterfall?

- Requirements su regulirani (FAA)
- Safety critical - mora biti perfektno
- Ekstenzivna dokumentacija obavezna
- Ne može se "iterirati" - mora raditi prvo put!

Procesni modeli programskog inženjerstva

Vodopadni model

✅ Prednosti

📋 Detaljno planiranje

Sve faze su jasno definirane na početku. Plan je detaljan, milestones su jasni, svi znaju što se očekuje.

📐 Linearni slijed aktivnosti

Jednostavno razumjeti i pratiti. Jedna faza završava, druga počinje. Nema konfuzije o tome što se radi kada.

🎯 Prikladno za dobro definirane projekte

Kada su zahtjevi fiksni i dobro poznati doseg projekta - Waterfall je odličan izbor. Npr. regulirane industrije.

📊 Potrebno aktivno praćenje napretka

Formalni voditelj projekta, izvještaji, tracking. Sve je pod kontrolom i transparentno za management.

📚 Klasično upravljanje temeljeno na planu i dokumentaciji

Ekstenzivna dokumentacija olakšava prenos znanja, onboarding novih članova tima, i održavanje.

VS

⚠️ Mane

🔒 Naknadno otkrivanje grešaka

Testiranje je na kraju! Ako requirements nisu bili dobri, otkrit ćeš to prekasno. Skupo fixanje.

🔄 Teško vraćanje

Vratiti se na prethodnu fazu je kompleksno i skupo. "Ah, requirements su bili krivi" = restart projekta.

⚠️ Izrazito važna specifikacija!

Ako requirements document nije perfektan, cijeli projekt je u problemima. Garbage in, garbage out.

⏰ Dugo vremena do isporuke

Korisnik ne vidi working software mjesecima ili godinama. Nema early feedback. High risk!

Kada koristiti Vodopadni model?

Idealno za:

Stabilne zahtjeve: Requirements su jasni, neće se mijenjati

Regulirane industrije: Medicina, avijacija, obrana - gdje je dokumentacija obavezna

Fiksni budžet i rok: Kada je sve unaprijed dogovoreno

Poznata tehnologija: Nema tehničkog rizika

Primjer: Software za medicinske uređaje, bankovni core sustavi

Izbjegavati za:

Promjenjivi zahtjevi: Kada klijent ne zna što točno želi

Inovativne projekte: Gdje učimo kroz iteracije

Brze time-to-market potrebe: Kada treba brzo ispustiti MVP

Startup okruženja: Gdje je pivot normalan

Primjer: Web aplikacije, mobile apps, SaaS produkti

Ključna lekcija:

Vodopadni model nije "loš" - on je jednostavno **prikladan za specifične situacije**.

Problem nastaje kada se koristi u krivom kontekstu. Primjena Waterfalla na inovativnom projektu s promjenjivim zahtjevima = recept za katastrofu.

Bottom line: Koristi Waterfall kada su requirements stabilni, dokumentacija važna, i promjene skupe. Za sve ostalo - razmisli o Agile-u.

Real-world realnost:

U praksi, **čisti Waterfall je rijedak**. Većina kompanija koristi "modified Waterfall" ili hibridni pristup.

Npr. mogu imati Waterfall faze, ali s iteracijama unutar faza. Ili koriste Waterfall za core sustav, ali Agile za features.

Lesson learned: Budi pragmatičan, ne dogmatičan. Odaberi metodologiju koja paše projektu, ne ideologiji.

Procesni modeli programskog inženjerstva

Primjeri (Detaljno planiranje)

Softver za lansiranje rakete

Software koji kontrolira lansiranje rakete mora biti apsolutno precizan i pouzdan. Nema prostora za greške - jedna linija lošeg koda može značiti katastrofu.

Zašto Waterfall?

- ✓ Safety critical - mora biti perfektno
- ✓ Requirements su fiksni (fizika ne mijenja)
- ✓ Ekstenzivna dokumentacija obavezna
- ✓ Regulacije i compliance (NASA, ESA)
- ✓ Testiranje mora biti iscrpno
- ✓ Ne može se "iterirati" u produkciji

Softver za Mars Rover

Rover koji istražuje površinu Marsa - milijarde dolara investicije, nemoguće fizički popraviti ili ažurirati hardware nakon lansiranja.

Zašto Waterfall?

- ✓ Jedan pokušaj - nema drugog šansa
- ✓ Autonoman rad u ekstremnim uvjetima
- ✓ Godišnje planiranje (komunikacija 20+ min delay)
- ✓ Sve mora raditi prije lansiranja
- ✓ Simulacije i testiranje godine dana
- ✓ Primjer: NASA Mars 2020 Perseverance

Softver za robote - inspekcija nuklearne elektrane

Robotski sustavi koji ulaze u radioaktivne zone nuklearnih elektrana za inspekciju. Mora biti ekstremno pouzdan jer ljudski život ovisi o njemu.

Zašto Waterfall?

- ✓ Safety critical aplikacija
- ✓ Regulacije nuklearne sigurnosti
- ✓ Formalna verifikacija koda potrebna
- ✓ Ekstenzivna dokumentacija za compliance
- ✓ Testiranje u simuliranim uvjetima
- ✓ Link: INETEC Robotics

Softver za eHealth u RH

Nacionalni zdravstveni informacijski sustav - integracija svih bolnica, klinika, ljekarna u Hrvatskoj. Milijuni pacijenata, kritična medicinska dokumentacija.

Zašto Waterfall?

- ✓ Nacionalni projekt - fiksni budžet i rok
- ✓ GDPR i medicinski compliance obavezni
- ✓ Integracija s postojećim sustavima
- ✓ Ekstenzivna dokumentacija za ministarstvo
- ✓ Formalno testiranje i prihvaćanje
- ✓ Stabilni requirements (zakoni i propisi)

Što imaju zajedničko?

Svi ovi projekti dijele zajedničke karakteristike koje ih čine idealnim kandidatima za Waterfall metodologiju. Ključno je da su svi safety-critical ili mission-critical projekti gdje greške mogu imati katastrofalne posljedice.

Safety Critical

Ljudski životi ili velika materijalna šteta ovise o pravilnom radu softvera. Nema prostora za eksperimentiranje.

Fiksni Requirements

Zahtjevi su jasni, stabilni, i neće se mijenjati. Često su regulirani zakonima ili fizikalnim zakonima.

Ekstenzivna dokumentacija

Sve mora biti dokumentirano za compliance, audit, ili transfer znanja. Documentation is not optional.

Formalno testiranje

Testing mora biti sistematičan, iscrpan, i formalno verificiran. Unit, integration, system, acceptance tests.

Visok trošak promjena

Promjene nakon deployments su ekstremno skupe ili nemoguće (Mars Rover, raketa).

Regulirano okruženje

Industrije poput aerospace, nuclear, healthcare imaju stroge propise koji zahtijevaju waterfall pristup.

Procesni modeli programskog inženjerstva

Inkrementalni (iterativni) razvoj

Definiraj okvire zahtjeve

Rasporedi zahtjeve na inkremente

Oblikuj arhitekturu sustava

↓

Iterativni ciklus (ponavljaj dok sustav nije dovršen)

Specificiraj, oblikuj i implementiraj novi inkrement

Razvijaj sljedeći increment funkcionalnosti

Verificiraj i validiraj novi inkrement

Testiraj da radi ispravno

Integriraj novi inkrement

Dodaj u postojeći sustav

← sustav je nedovršen (vrati se na početak ciklusa)

↓

Konačni sustav

Svi inkrementi integrirani i sustav je spreman za produkciju

Ključne karakteristike

Prednosti iterativnog razvoja

- ✓ Rana isporuka radnog softvera - korisnik vidi rezultate brzo
- ✓ Feedback nakon svakog inkrementa - prilagodba je moguća
- ✓ Manji rizik - problemi se otkrivaju rano
- ✓ Fleksibilnost - lako prilagoditi promjenama
- ✓ Kontinuirana integracija i testiranje
- ✓ Bolje za projekte s promjenjivim zahtjevima

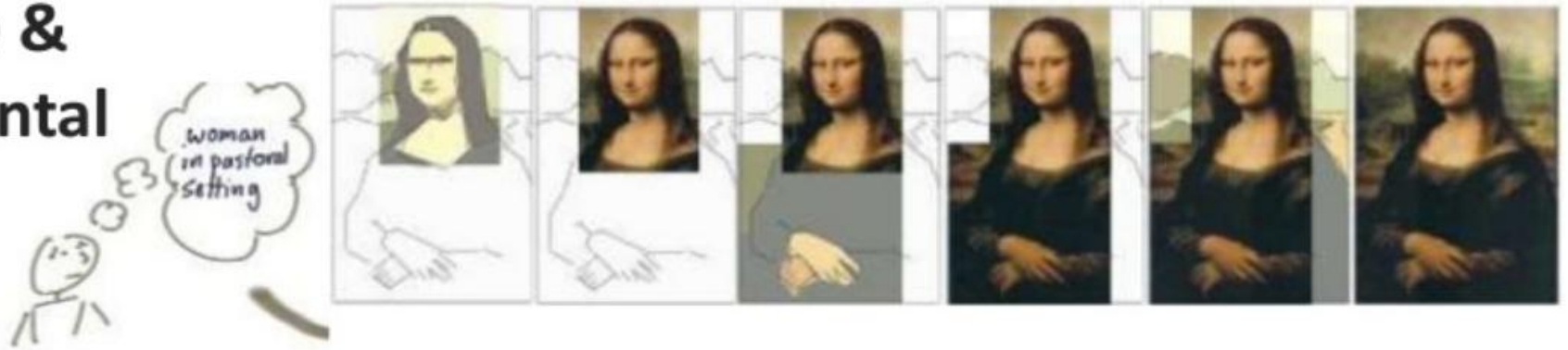
Kako funkcionira?

- ✓ Podijeli sustav na inkremente (module/feature-e)
- ✓ Svaki inkrement dodaje novu funkcionalnost
- ✓ Razvijaj, testiraj i integriraj jedan po jedan
- ✓ Svaki inkrement je working software
- ✓ Korisnik može početi koristiti rano
- ✓ Prilagodi sljedeće inkremente prema feedbacku

Iterative

Incremental

Iterative & Incremental



Procesni modeli programskog inženjerstva

Prototipiranje

Horizontalno

Breadth-first prototyping

Zastupljene sve funkcionalnosti

Prototip pokriva sve planirane feature-e sustava, ali na površnoj razini. Korisnik vidi cijelu širinu aplikacije.

Svaka funkcionalnost izgrađena u maloj mjeri

Ni jedna funkcionalnost nije potpuno razvijena. Mock-up sučelja, hardcoded podaci, osnovni flow bez detalja.

VS

Vertikalno

Depth-first prototyping

Zastupljena jedna odabrana funkcionalnost

Fokus na jednu ključnu funkcionalnost ili modul. Ostale funkcionalnosti se ne prikazuju u prototipu.

Funkcionalnost izgrađena u velikoj mjeri

Ta jedna funkcionalnost je gotovo potpuno implementirana. Sve detalje, edge case-ovi, integracije - sve je tu.

Detaljnije objašnjenje

Horizontalno prototipiranje

Svrha: Pokazati cijeli UI/UX flow i scope projekta

Što sadržava:

- Sve stranice/ekrane
- Navigaciju između njih
- Osnovni layout i dizajn
- Mock-up podatke (hardcoded)

Što NE sadržava:

- Stvarnu backend logiku
- Database integraciju
- Error handling
- Edge case-ove

Primjer: Figma mockup sa svim ekranima aplikacije, ali bez funkcionalne logike

Vertikalno prototipiranje

Svrha: Testirati tehnološku izvedivost jedne funkcionalnosti

Što sadržava:

- Kompletno implementiran jedan feature
- Frontend + Backend + Database
- Sve edge case-ove
- Error handling

Što NE sadržava:

- Ostale funkcionalnosti
- Kompletni UI
- Navigaciju kroz cijelu app

Primjer: Potpuno funkcionalni login/registracija sistem, ali bez ostalih feature-a aplikacije

Kada koristiti horizontalno prototipiranje?

Idealno za:

- Early-stage projekti
- Prezentacija klijentu/investitorima
- UX/UI validacija
- Feedback na scope i flow

Prednosti:

- Brzo napraviti
- Daje overview cijelog projekta
- Lako mijenjati dizajn

Nedostaci:

- Ne testira tehničku izvedivost
- Može biti obmanjujuće - izgleda gotovo, ali nije

Kada koristiti vertikalno prototipiranje?

Idealno za:

- Testiranje kompleksnih feature-a
- Proof of concept
- Tehnološka validacija
- Arhitekturalne odluke

Prednosti:

- Otkriva tehnološke probleme rano
- Working software za ključni feature
- Realna procjena vremena razvoja

Nedostaci:

- Sporije za napraviti
- Ne vidiš cijeli sistem

Procesni modeli programskog inženjerstva

Inkrementalni (iterativni) razvoj

Prednosti inkrementalnog spram vodopadnog

✅ Prednosti

🚀 **Rana isporuka prvog inkrementa**

Naručitelj dobiva prvu "vrijednost" puno prije završetka projekta. Može početi koristiti osnovne funkcionalnosti odmah, ne čeka mjesecima ili godinama.

📊 **Jasan napredak projekta**

Svaki inkrement je vidljiv rezultat. Management vidi working software, ne samo PowerPoint prezentacije. Napredak je opipljiv i mjerljiv.

💰 **Mogućnost naplate u fazama**

Klijent plaća inkrement po inkrement. Manji finansijski rizik za obje strane. Nije potrebno sve platiti unaprijed, a developer dobiva novac kontinuirano.

VS

⚠️ Mane

🔍 **Nije lako dobro definirati inkremente**

Kako podijeliti sustav na smislene inkremente? Što ide prvo, što drugo? Loša podjela može značiti da rani inkrementi nisu korisni ili da se kasnije sve mora refaktorirati.

🎯 **Nije lako definirati prioritete**

Idealno ako se može prebaciti na naručitelja - ali što ako klijent ne zna što je prioritet? Ili ako različiti stakeholderi imaju različite prioritete? Conflicts!

Detaljnije objašnjenje

Rana isporuka = Rana vrijednost

Waterfall: Čekaš 12 mjeseci da vidiš bilo što
Incremental: Nakon 2 mjeseca već imaš login i dashboard

Primjer: E-commerce platforma
- Inkrement 1 (1 mj): Product listing + Search
- Inkrement 2 (1 mj): Shopping cart
- Inkrement 3 (1 mj): Payment
- Inkrement 4 (1 mj): Order tracking

Nakon 1 mjeseca klijent već može pokazati investitorima working prototype sa searchom!

Jasan napredak projekta

Waterfall: "We're 60% done with coding phase" - što to znači?
Incremental: "We've completed login, dashboard, and search. Next is shopping cart."

Benefit:
- Management vidi stvarni napredak
- Lakše trackati rokove
- Demo nakon svakog inkrementa
- Transparentnost prema klijentu

Nije moguće "90% gotovo" fenomen koji traje mjesecima!

Problem definiranja inkremenata

Challenge: Kako podijeliti sustav?

Loš pristup:
- Inkrement 1: Cijeli backend
- Inkrement 2: Cijeli frontend
- Problem: Ništa ne radi dok nije sve gotovo!

Dobar pristup:
- Inkrement 1: Login (frontend + backend + DB)
- Inkrement 2: User profile
- Svaki inkrement = vertical slice = working feature

Requires careful planning and architecture design!

Problem prioritizacije

Ideal scenario: Klijent zna što je prioritet
Reality: Često ne zna, ili ima konfliktne prioritete

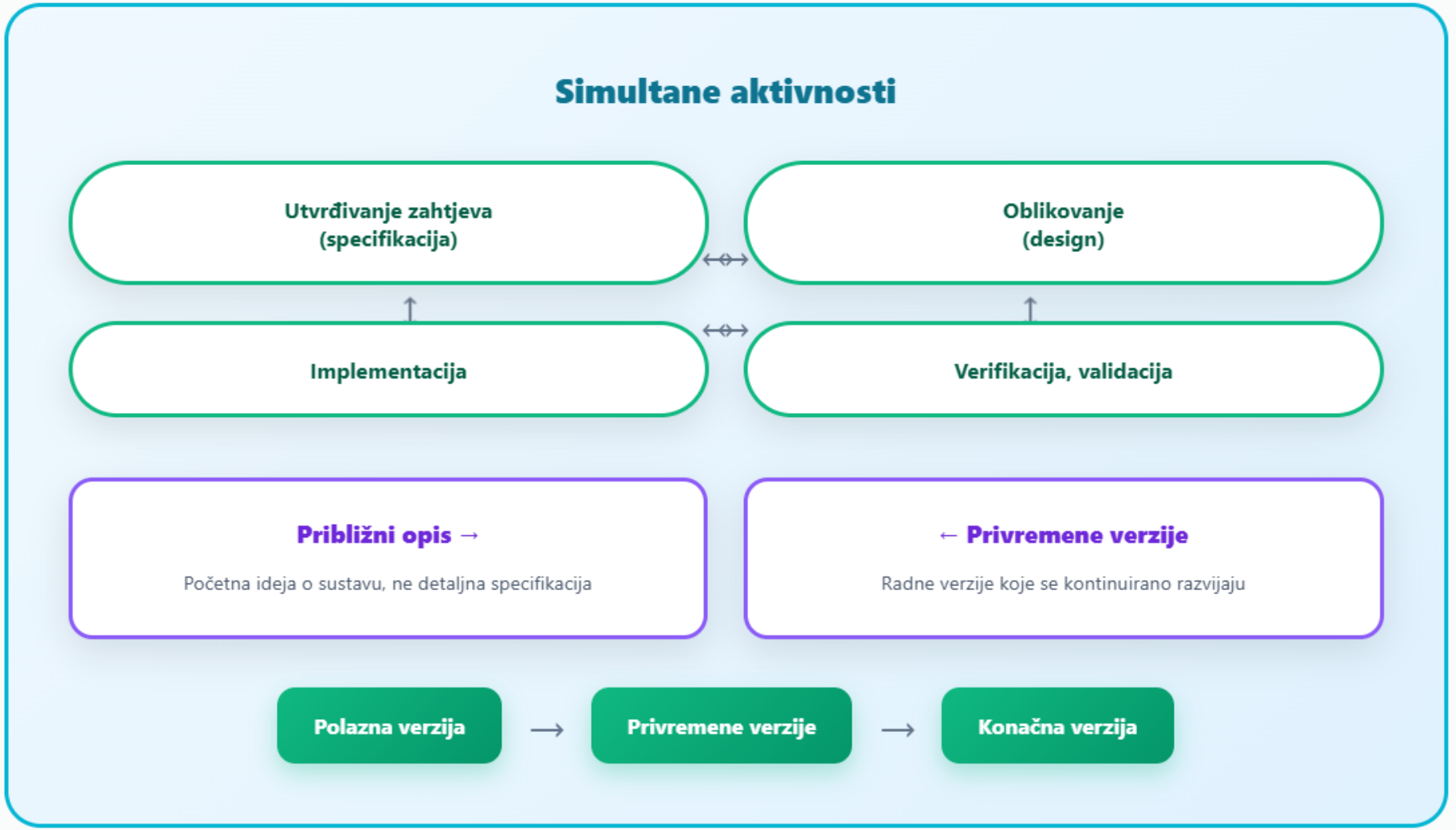
Tipične situacije:
- "Sve je važno!" - nema pomoći
- Sales kaže jedno, Tech drugi, Management treće
- Prioriteti se mijenjaju svaki tjedan

Rješenje:
- Product Owner odlučuje (Scrum model)
- MoSCoW method (Must, Should, Could, Won't)
- Business value vs development effort matrix

Procesni modeli programskog inženjerstva

Evolucijski razvoj

Podvrsta inkrementalnog



Ključne karakteristike

Simultane aktivnosti

Za razliku od Waterfall-a gdje faze idu jedna za drugom, ovdje se **sve dešava paralelno**!

Dok radiš implementaciju, istovremeno:
- Prikupljaš nove zahtjeve
- Prilagođavaš dizajn
- Testiraš postojeće feature-e

Sve aktivnosti se preklapaju i djeluju jedna na drugu. Strelice u dijagramu pokazuju da sve interaktira sa svime!

Približni opis - ne detaljan

Ne počinješ sa 100-stranicnim requirements documentom!

Početak:
"Trebamo e-commerce platformu sa košaricom, plaćanjem i order trackingom"

To je dovoljno! Detalje otkrivamo kroz razvoj.

Kao što evolucija u prirodi ne zna točno gdje će završiti - tako ni ovdje nemamo detaljan plan na početku.

Privremene verzije

Razvijaš sustav kroz niz **privremenih verzija** (throwaway prototypes ili evolutionary prototypes).

Svaka verzija:
- Dodaje novu funkcionalnost
- Poboljšava postojeću
- Odgovara na feedback
- Približava te konačnoj verziji

Nije kao incremental gdje je svaki inkrement "finalan" - ovdje se sve može promijeniti!

Evolucija = kontinuirana prilagodba

Kao u prirodi - **survival of the fittest ideas**!

Proces:
1. Razvij verziju s nekim feature-ima
2. Pokaži korisniku
3. Dobij feedback
4. Prilagodi (evolve) - možda čak i obriši neke feature-e
5. Ponovi

Requirements "evolvira" kroz razvoj - ne znaš točno što ćeš dobiti na kraju!

Procesni modeli programskog inženjerstva

Model ponovne upotrebe (Reuse oriented)

Ključne faze procesa

1. Specifikacija zahtjeva

Početna faza gdje definiramo što sustav treba raditi. Ali za razliku od tradicionalnog pristupa, odmah razmišljamo - **koje dijelove možemo ponovno upotrijebiti?**

2. Analiza raspoloživih dijelova

Ključna faza! Pretražujemo libraries, frameworks, komponente, API-je - što već postoji i što možemo iskoristiti? npm, GitHub, internal company libraries - sve pregledavamo.

3. Redefiniranje zahtjeva

Game changer! Umjesto da prilagođavamo komponente zahtjevima, prilagođavamo zahtjeve komponentama. "Možemo li koristiti ovaj library ako malo promijenimo requirements?"

4. Konfiguriranje pr. kladnih dijelova

Postojeće komponente config-uriramo za naše potrebe. Bootstrap tema, API parametri, database settings - sve što možemo iskoristiti "as is" uz konfiguraciju.

5. Modificiranje iskorištenih dijelova

Neki dijelovi skoro pašu, ali trebaju male izmjene. Fork-aj library, extend class, dodaj feature - malo modifikacije i ready to go!

6. Razvoj novih dijelova

Samo ono što ne možemo naći ili prilagoditi razvijamo od nule. Ovdje pišemo custom kod, ali to je **minimum potrebnog posla**.

7. Integracija

Spajamo sve dijelove zajedno - stare komponente, modificirane, i novo napisane. Integration je ključan - mora sve raditi zajedno.

8. Validacija i testiranje

Testiranje cijelog sustava. Posebna pažnja na integration points - da li sve komponente rade zajedno kako treba?

Filozofija: Don't reinvent the wheel

Zašto pisati authentication od nule? Koristi Auth0, Firebase Auth, Passport.js!

Zašto pisati payment? Stripe API već postoji!

Fokusiraj se na **business logic** koji je jedinstven za tvoj projekt, ostalo reuse!

Real-world primjer

Projekt: E-commerce platforma

Reuse:

- Frontend: React + Material-UI
- Authentication: Firebase Auth
- Payment: Stripe
- Database: MongoDB Atlas
- Hosting: Vercel

Custom razvoj: Samo business logic za product management i order processing!

Umjesto 6 mjeseci - gotovo za 6 tjedana!

Prednosti

✓ **Brži razvoj** - ne pišeš sve od nule

✓ **Niži troškovi** - manje dev sati

✓ **Veća kvaliteta** - koristi testirane komponente

✓ **Manje bugova** - reused kod je već debuggan

✓ **Fokus na inovaciju** - ne na osnove

Nedostaci

✗ **Vendor lock-in** - ovisnost o 3rd party

✗ **Licence** - legal issues s nekim libraries

✗ **Security rizici** - ako library ima bug

✗ **Kompromisi** - requirements se prilagođavaju

✗ **Dependency hell** - npm install problema

Procesni modeli programskog inženjerstva

Model ponovne upotrebe (Reuse oriented)

Značaj

Svi projekti koriste ovaj model

U manjoj ili većoj mjeri - nemoguće je danas razvijati softver bez korištenja postojećih komponenti, libraries i frameworks.

Čak i "from scratch" projekti koriste bar OS libraries, compiler, i programming language frameworks!

Skraćenje vremena isporuke

Glavni razlog popularnosti - umjesto 12 mjeseci, projekt može biti gotov za 3 mjeseca jer se 70-80% funkcionalnosti reuse-a.

Time to market je kritičan u modernom poslovanju!

Elementi ponovne upotrebe

Radni okviri (frameworks)

Struktura aplikacije je već definirana. React, Angular, Vue za frontend. Django, Spring, Express za backend.

Framework ti daje arhitekturu - ti samo "punish" business logic!

Biblioteke (third party libraries)

Gotove funkcionalnosti koje možeš uključiti u projekt. npm install, pip install, Maven dependency - milijuni libraries dostupno!

Primjeri: Lodash, Moment.js, Axios, NumPy, Pandas

Programski isječci (code snippets)

Mali dijelovi koda koji rješavaju specifične probleme. Stack Overflow, GitHub Gist, internal company wiki - copy/paste i adapt!

"Good developers write code, great developers reuse code"

🎯 Doprinos

Zajednica otvorenog kOda (Open source)

GitHub, GitLab, npm, PyPI - milijuni open source projekata dostupno besplatno. Contributors iz cijelog svijeta razvijaju i održavaju kod.

React (Meta), TensorFlow (Google), Linux kernel - sve open source!

Solidarnost programera

Developer community dijeli znanje i kod. Stack Overflow, GitHub discussions, dev.to, Reddit - pomažu jedni drugima. Open source contributions, dokumentacija, tutorials - svi doprinose.

"Standing on the shoulders of giants" - koristimo rad milijuna developera prije nas!

Detaljnije objašnjenje

Frameworks - Temelj moderne razvoja

Što je framework? Skelet aplikacije koji definira strukturu i flow.

Primjeri:

- React: UI komponente, state management
- Django: MTV pattern, ORM, admin panel
- Spring Boot: DI, REST, security out of the box

Prednost: Ne razmišljaš o arhitekturi, već o business logic-u!

Libraries - Building blocks

Razlika framework vs library:

Framework zove tvoj kod, ti zoveš library!

Top libraries:

- Lodash: utility functions
- Axios: HTTP requests
- Moment.js: date manipulation
- Chart.js: visualizacije

npm ima 2+ million packages - za sve postoji library!

Code Snippets - Brzina razvoja

Gdje naći?

- Stack Overflow - odgovori na pitanja
- GitHub Gist - dijeljeni kod
- CodePen - frontend snippets
- Company wiki - internal knowledge

Best practice: Razumi kod prije copy/paste! Prilagodi svojim potrebama.

Open Source revolucija

Zašto je važan?

90% modernih aplikacija koristi open source komponente!

Ekosistem:

- GitHub: 100M+ repositories
- npm: 2M+ packages
- PyPI: 400K+ packages

Licenciranje: MIT, Apache, GPL - pazi na uvjete!

Developer Community

Platforme:

- Stack Overflow: 20M+ pitanja
- GitHub Discussions: kolaboracija
- dev.to: članovi i tutorijali
- Reddit r/programming: news i diskusije

Kultura: "Pay it forward" - ako ti je netko pomogao, pomozi drugima!

Real-world statistika

Istraživanja pokazuju:

- 70-95% koda u modernim app-ovima je reused
- Developers provode 19% vremena tražeći reusable kod
- Prosječan web projekt koristi 100+ npm packages

Bottom line: Reuse nije opcija, već standard!

Procesni modeli programskog inženjerstva

Model ponovne upotrebe (Reuse oriented)

Primjer odabira komponenti iz prakse

Android Maps

Prilagodba zahtjeva korisnika

Umjesto razvijanja custom map sistema od nule, projekt prilagođava requirements kako bi koristio Google Maps Android API.

Što dobivamo?

Gotove funkcionalnosti:

- Prikaz mapa (satellite, terrain, hybrid views)
- Geolokacija i GPS tracking
- Routing i navigation
- Markers i custom overlays
- Street View integracija

Ušteda: Mjeseci razvoja → par dana integracije!

Real-world primjer

Uber, Bolt, Wolt - sve koriste Google Maps API. Ne razvijaju svoj mapping system, već fokus stavljaju na business logic (matching drivers, pricing, routing optimization).

Android HTTP Library

Prilagodba procesa prema zahtjevima naručitelja

Umjesto pisanja vlastitog HTTP client koda, koristimo etablirane libraries poput Retrofit ili OkHttp i prilagođavamo naš development process.

Što dobivamo?

Gotove funkcionalnosti:

- HTTP requests (GET, POST, PUT, DELETE)
- JSON parsing i serialization
- Error handling i retry logic
- Authentication headers
- Request/Response interceptors
- Connection pooling i caching

Ušteda: Tjedni debugging → battle-tested solution!

Real-world primjer

Gotovo svaka Android aplikacija koja komunicira sa API-jem koristi Retrofit ili OkHttp. Instagram, Twitter, Reddit apps - sve koriste iste libraries za networking.

Android Maps

Prilagodba zahtjeva korisnika

Umjesto razvijanja custom map sistema od nule, projekt prilagođava requirements kako bi koristio Google Maps Android API.

Što dobivamo?

Gotove funkcionalnosti:

- Prikaz mapa (satellite, terrain, hybrid views)
- Geolokacija i GPS tracking
- Routing i navigation
- Markers i custom overlays
- Street View integracija

Ušteda: Mjeseci razvoja → par dana integracije!

Real-world primjer

Uber, Bolt, Wolt - sve koriste Google Maps API. Ne razvijaju svoj mapping system, već fokus stavljaju na business logic (matching drivers, pricing, routing optimization).

Android HTTP Library

Prilagodba procesa prema zahtjevima naručitelja

Umjesto pisanja vlastitog HTTP client koda, koristimo etablirane libraries poput Retrofit ili OkHttp i prilagođavamo naš development process.

Što dobivamo?

Gotove funkcionalnosti:

- HTTP requests (GET, POST, PUT, DELETE)
- JSON parsing i serialization
- Error handling i retry logic
- Authentication headers
- Request/Response interceptors
- Connection pooling i caching

Ušteda: Tjedni debugging → battle-tested solution!

Real-world primjer

Gotovo svaka Android aplikacija koja komunicira sa API-jem koristi Retrofit ili OkHttp. Instagram, Twitter, Reddit apps - sve koriste iste libraries za networking.

Prednosti reuse pristupa u praksi

Brzina razvoja

Bez reuse: 3-6 mjeseci za map sistem
Sa reuse: 2-3 dana za integraciju Google Maps

Time to market je ključan - app može biti na tržištu 90% brže!

Kvaliteta i stabilnost

Google Maps API koristi milijarde ljudi dnevno - testirano u svim edge case-ovima.

Custom map sistem bi imao bugove koje bi trebalo mjesecima ispravljati. Reused komponente su production-ready!

Fokus na business logic

Ne trošiš vrijeme rješavajući već riješene probleme (maps, HTTP).

Fokusiraš se na ono što čini tvoj app jedinstvenim - matching algoritam, pricing logic, user experience.

Automatski updates

Google kontinuirano poboljšava Maps API - nove zemlje, bolje routing, nova feature-i.

Ti automatski dobivaš sva poboljšanja - besplatno! Nema potrebe za održavanjem map sistema.

Dokumentacija i community

Popularne libraries imaju odličnu dokumentaciju, tutorials, Stack Overflow odgovore.

Problem? Google it - vjerojatno je već netko riješio i podijelio rješenje!

Niži troškovi

Custom development: 3 developera x 3 mjeseca = $50,000+
Reuse pristup: 1 developer x 3 dana = $2,000

ROI je očigledan - 96% uštede!

Procesni modeli programskog inženjerstva

Odabir metodologije o čemu ovisi

Posebnosti softvera

Nema "one size fits all" metodologije! Odabir ovisi o brojnim faktorima vezanim za projekt, tim, klijenta i okolinu.

Opseg problema

Koliko je projekt velik i kompleksan? Mali projekti mogu koristiti Agile, veliki enterprise sistemi često zahtijevaju više strukture.

Mali: Mobile app → Scrum
Srednji: E-commerce → Hibridni
Veliki: Banking core → Modified Waterfall

Brzina i učestalost isporuke

Koliko brzo treba isporučiti prvi increment? Hoće li biti česte isporuke ili jedna velika na kraju?

Brzo: Startup MVP → Agile (2-tjedni sprint)
Sporo: Medical device → Waterfall (godinu dana)

Definiranost problema

Koliko jasno znamo što treba napraviti? Stabilni zahtjevi favoriziraju Waterfall, promjenjivi zahtjevi favoriziraju Agile.

Jasno definirano: Replika postojećeg sistema
Nejasno: Inovativni produkt koji se razvija

Vrsta naručitelja

Tko je klijent? Vlasnička vlada, javna uprava, korporacija, startup? Svaki ima različite zahtjeve i očekivanja.

Vlada/regulacija: Waterfall (compliance)
Startup: Agile (brzina i fleksibilnost)

Kritičnost neispravnog rada

Što se događa ako softver ima bug? Da li ugrožava ljudski život ili financije?

Safety-critical: Medicinski uređaji → Waterfall
Mission-critical: Banking → Hybrid
Non-critical: Social media → Agile

Veličina scope doseg primjene u broju korisnika

Koliko korisnika će koristiti sistem? Milijun korisnika zahtijeva drugačiji pristup od 100 korisnika.

Milijuni: Facebook scale → DevOps + Agile
Tisuće: Enterprise → Hibridni
Stotine: Internal tool → Flexible

Interakcija/integracija s okolinom i tehnologija izvedbe

Koliko integracija s drugim sustavima? Koja tehnologija se koristi? Legacy sistemi zahtijevaju pažljiviji pristup.

Puno integracija: Enterprise → Waterfall planning
Moderna tech: Cloud-native → Agile + DevOps

Kritični faktori odluke

Da li ugrožava ljudski život?

DA → Waterfall obavezan

Medical devices, avijacija, nuklearne elektrane - nema mjesta za "move fast and break things".

Ekstenzivna dokumentacija, formalna verifikacija, rigorozno testiranje. FDA, FAA, nuclear regulatori zahtijevaju Waterfall pristup.

Da li ugrožava financije?

DA → Waterfall ili Hibridni

Banking core systems, trading platforms, payment processors - bug može koštati milijune.

Potrebna je balance između brzine (Agile) i sigurnosti (Waterfall). Hibridni pristup često najbolji.

Promjenjivi zahtjevi?

DA → Agile obavezan

Startup produkti, inovativni projekti, competitive markets - requirements će se mijenjati.

Waterfall bi bio katastrofa - trebala bi fleksibilnost i brze iteracije.

Brzi time to market?

DA → Agile + Reuse

Competitive advantage ovisi o brzini - first mover advantage.

Koristi frameworks, libraries, cloud services. MVP u tjednima, ne mjesecima. Iterate based on feedback.

Vođenje softverskih projekata

Elementi koji se metodologijom definiraju

Uloge (roles, actors) koje su uključene

Tko radi što u projektu

Organizacija odgovornosti

Svaka metodologija definira jasne uloge i odgovornosti svakog člana tima. Tko donosi odluke? Tko piše kod? Tko komunicira s klijentom? Tko testira?

Vođenje softverskih projekata

Produkti koji nastaju

Što se isporučuje u pojedinim fazama

Mjerljivi izlazi u pojedinim fazama

Što konkretno nastaje na kraju svake faze? Requirements document? Design diagrams? Working software? Test reports? Svaka metodologija ima specifične deliverables.

Primjeri po metodologijama

Waterfall - Uloge

Project Manager: Vodi cijeli projekt, planira, tracka

Business Analyst: Prikuplja requirements

System Architect: Dizajnira arhitekturu

Developers: Pišu kod

QA Team: Testiraju na kraju

Technical Writer: Dokumentacija

Uloge su strogo odvojene - nema preklapanja!

Waterfall - Produkti

Requirements Phase: SRS Document (100+ stranica)

Design Phase: Architecture diagrams, ERD, Class diagrams

Implementation: Source code, Unit tests

Testing: Test plans, Test reports, Bug lists

Deployment: User manuals, Installation guides

Sve je dokumentirano i formalno!

Scrum - Uloge

Product Owner: Definira što se razvija, prioriteti

Scrum Master: Facilitira proces, uklanja prepreke

Development Team: Cross-functional (dev+test+design)

Team je self-organizing i cross-functional. Nema "qa person" ili "backend person" - svi rade sve što je potrebno za sprint goal!

Scrum - Produkti

Sprint Planning: Sprint backlog, Sprint goal
Daily Scrum: Updated task board
Sprint Development: Increment of working software
Sprint Review: Demonstrated features, Feedback
Sprint Retrospective: Process improvements

Fokus na working software, minimalna dokumentacija!

Kanban - Uloge

Team Members: Self-organizing, pull tasks
Service Delivery Manager: Optimizira flow
Service Request Manager: Upravlja backlogom

Uloge su fluidne - važan je flow, ne uloge. WIP (Work In Progress) limiti određuju tko radi što.

Kanban - Produkti

Kanban Board: Vizualizacija toka rada
Metrics: Lead time, Cycle time, Throughput
Continuous Delivery: Features kad god su gotovi

Nema fiksnih sprint-ova - continuous flow of deliverables. Produkti su ongoing, ne vezani za specifične datume.

Proces programskog inženjerstva

Uloge i odgovornosti

Informativno

Microsoftova podjela prema MSF CMMI

Najčešće uloge učesnika, dionika (stakeholders) u procesu

📦 Vodstvo proizvoda

Product Management

Odgovornosti: Definira viziju proizvoda, prioritete i roadmap. Odlučuje ŠTO se razvija.

Ključne aktivnosti: Prikupljanje zahtjeva, komunikacija s klijentima, prioritizacija feature-a, product backlog.

Primjer: Product Owner u Scrum-u

📋 Vodstvo programa

Program Management

Odgovornosti: Koordinacija između timova, planiranje i tracking napretka. Osigurava da projekt ide prema planu.

Ključne aktivnosti: Scheduling, resource management, risk management, status reporting.

Primjer: Scrum Master, Project Manager

💻 Razvojni tim

Development

Odgovornosti: Dizajn i implementacija softvera. Piše kod, kreira arhitekturu, rješava tehničke probleme.

Ključne aktivnosti: Coding, code review, system design, technical documentation, unit testing.

Primjer: Frontend dev, Backend dev, Full-stack dev

Testeri

Test / QA

Odgovornosti: Osiguravanje kvalitete softvera. Pronalazi bugove prije nego što stignu do korisnika.

Ključne aktivnosti: Test planning, manual testing, automation testing, bug reporting, regression testing.

Primjer: QA Engineer, Test Automation Engineer

Produkcijski tim

Release Management

Odgovornosti: Deployment softvera u produkciju. Osigurava da release prođe glatko i bez downtime-a.

Ključne aktivnosti: Release planning, deployment automation (CI/CD), monitoring, rollback procedures.

Primjer: DevOps Engineer, Release Manager

Korisnička podrška

User Experience

Odgovornosti: Fokus na korisničko iskustvo. Osigurava da softver bude intuitivan i user-friendly.

Ključne aktivnosti: UX research, wireframing, prototyping, usability testing, user feedback analysis.

Primjer: UX Designer, UI Designer, UX Researcher

Proces programskog inženjerstva

Uloge i odgovornosti

Uloge u procesu

Argumentirajte potrebu za ulogom poslovnog analitičara, potkrijepite primjerom.

Poslovni analitičar (Business Analyst)

Most između biznis strane i tehničke strane projekta. BA razumije business potrebe klijenta i prevodi ih u tehničke zahtjeve koje developers mogu implementirati.

Uloga koja pripada na stranu developera

BA je dio development tima, ne klijenta. Radi za development organizaciju, ali komunicira s klijentom da razumije njihove potrebe.

Zahtijeva razumijevanje domene primjene proizvoda

BA mora razumjeti industriju u kojoj klijent posluje. Npr. za banking app, BA mora razumjeti banking procese, regulacije, compliance. Bez domain knowledge, ne može postaviti prava pitanja.

Pripada više u vodstvo projekta ako je odvojeno zbog više projekata u programu

U velikim organizacijama gdje je Program Management odvojen od pojedinih projekata, BA često ide pod Program Management jer koordinira requirements kroz više projekata.

Ako postoji osoba u toj ulozi, znači li to da se ne radi o vlastitom proizvodu?

Ne, može biti domena u kojoj se želi proizvod ali nije domena za koju postoje kompetencije. Npr. tech kompanija želi napraviti medical app - trebaju BA sa medical domain knowledge čak i ako je to njihov vlastiti proizvod!

Zašto je BA potreban? Primjer:

Scenario: Tech startup želi napraviti aplikaciju za upravljanje lancima opskrbe u logističkoj industriji.

Problem: Developers razumiju tehnologiju, ali nemaju pojma o logističkim procesima, warehouse management, inventory optimization, shipping regulations.

Rješenje - Business Analyst:

1. Domain expertise: BA sa background-om u logistici razumije terminologiju, procese, pain points.

2. Requirements gathering: Kada klijent kaže "trebamo real-time tracking", BA zna pitati prava pitanja: GPS tracking ili RFID? Tracking po pošiljci ili po vozilu? Kako integrirati sa WMS sistemom?

3. Prevođenje u tech requirements: BA pretvara business potrebe u user stories koje developers razumiju: "Kao warehouse manager, želim vidjeti trenutnu lokaciju svih vozila na mapi da mogu optimizirati rute."

4. Sprječavanje scope creep-a: BA filtrira nerealne zahtjeve i fokusira projekt na ono što je stvarno potrebno.

Rezultat: Developers dobivaju jasne, implementabilne zahtjeve umjesto nejasnih biznis želja. Projekt uspijeva jer postoji netko tko razumije OBJE strane.

🚀 Uloge i odgovornosti u timu

Proces programskog inženjerstva

86%

timova koristi kombinaciju pristupa

5-9

idealnih članova tima

2x

brža isporuka u ravnopravnim timovima

Hijerarhijski tim

- ✓ Jasna struktura izvještavanja
- ✓ Razvoj, testiranje, produkcija - odvojeni timovi
- ✓ Vodstvo programa
- ✓ Jasno definirane uloge i odgovornosti
- ✓ Struktura komandnog lanca

⚠ **Izazov:** Može biti sporiji proces odlučivanja

Ravnopravni tim

- ✓ Svi imaju istu ulogu
- ✓ Zajedničke odluke (konsenzus)
- ✓ Demokratska organizacija
- ✓ Kohezivnost i konstruktivnost
- ✓ Kreativnost na prvom mjestu

⚠ **Izazov:** Problem donošenja odluka u velikim timovima

📋 Ključne karakteristike NIJE specifično za MSF CMMI

💡 Ovi principi organizacije tima nisu ograničeni samo na Microsoft Solutions Framework (MSF) CMMI metodologiju. Primjenjivi su na **Agile, Scrum, Kanban** i druge moderne metodologije razvoja softvera.

🎯 Izbor strukture ovisi o:

- Veličini projekta
- Kompleksnosti zadatka
- Kulturi organizacije
- Iskustvu tima

⚡ Prednosti ravnopravnog tima:

- Brže reagiranje na promjene
- Veća motivacija članova
- Bolja komunikacija
- Inovativniji pristupi

📊 Prednosti hijerarhijskog tima:

- Jasna odgovornost
- Lakše skaliranje
- Predvidljivost procesa
- Strukturirana eskalacija

👑 Hijerarhijska organizacija tima

Proces programskog inženjerstva

📊 Struktura tima

👨‍💼 Vođa tima (Team Lead)

🎯 Senior
Iskusan developer

🎯 Senior
Iskusan developer

🌱 Junior
Novi developer

🌱 Junior
Novi developer

Jasna hijerarhija | Različite razine iskustva | Mentorstvo

🤔 Kad hierarhija ima smisla?

❌ Kad ravnopravnost ne funkcionira

Kada demokratsko odlučivanje i samoorganiziranje dovodi do sporog napretka, nejasnih odluka ili konflikata u timu.

📏 Iskustva variraju

Broj članova i struktura tima ovise o specifičnostima projekta, industriji i kulturi tvrtke. Nema univerzalnog rješenja!

🎯 Tip projekta

Vrsta tima i posao koji obavljaju ključno određuju potrebnu strukturu. Kompleksnost zadataka je glavni faktor.

💡 **Ključni uvid:** Hierarhija je alat, a ne cilj. Koristi se kada donosi vrijednost - jasniju komunikaciju, bržu eskalaciju problema i bolju koordinaciju.

⚖️ Usporedba: Razvojni tim vs Call centar

💻 Razvojni tim

Kompleksnost: Visoka

- ✓ Kreativni problem solving
- ✓ Različite tehnologije
- ✓ Arhitektonske odluke
- ✓ Code review i mentorstvo
- ✓ Dinamičke tehničke prilike

📞 Call centar tim

Kompleksnost: Niska-Srednja

- ✓ Standardizirani procesi
- ✓ Skripta za komunikaciju
- ✓ Mjerljivi KPI-ji
- ✓ Ponavljajući zadaci
- ✓ Jasne procedure

🌟 Poseban slučaj: Team Lead koji još razvija

Kada vođa tima **aktivno doprinosi razvoju koda**, tim postaje još zahtjevniji za upravljanje. Team lead mora balansirati između:

- **Tehničkog rada** - pisanje koda, review, arhitektura
- **Upravljanja ljudima** - mentorstvo, feedback, koordinacija
- **Strategije** - planiranje, prioritizacija, komunikacija sa stakeholderima

📊 **Podatak:** Studije pokazuju da team leadovi koji pišu kod provode 60% vremena na tehničkim zadacima i 40% na menadžmentu, dok pure manageri mogu više fokusa staviti na strategiju i razvoj tima.

🎯 Zaključak

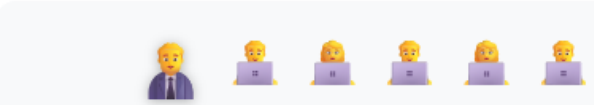
Hierarhija nije uvijek potrebna, ali je vrlo korisna kada:

- 📈 Tim raste i postaje složeniji (više od 5-7 članova)
- 🎓 Postoje različite razine iskustva koje zahtijevaju mentorstvo
- ⚡ Potrebno je brzo donošenje odluka i jasna odgovornost
- 🔧 Tehnička kompleksnost zahtijeva koordinaciju i arhitektonske odluke

👥 Optimalna veličina tima

Organizacija tima - Proces programskog inženjerstva

💻 Razvojni tim



Optimalno

5-6

članova tima

🔄 **Vođa također razvija**

Vođa timskog učestvuje dijelom vremena u razvoju koda

👥 **Više od dvoje po vođi**

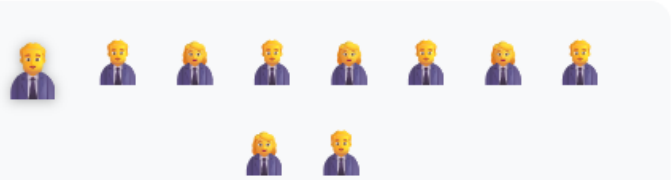
Minimalno 3+ člana da bi imalo smisla

⚠️ **Gornja granica**

Preporuka: Maksimalno 9 ljudi

Preko 9 članova, komunikacija postaje previše složena i efikasnost opada

🏢 Ostali timovi



Optimalno

5-15+

članova tima

👔 **Vođa samo vodi**

Vođa je posvećen isključivo vođenju ljudi, bez operativnog rada

📈 **Veći kapacitet**

Brojke su malo veće jer vođa ima puno fokusa na menadžment

✅ **Fleksibilnost**

Nekad i više od 15

Ovisno o prirodi posla, može biti i veći tim (npr. call centar, prodaja, podrška)

Vizualna usporedba veličine timova

Razvojni tim - Optimalno

Razvojni tim - Maksimum

Ostali timovi - Standardno

Ostali timovi - Prošireno

💡 Ključni uvidi

🎯 Zašto postoji razlika?

Razvojni timovi zahtijevaju više komunikacije, koordinacije i tehničkog usklađivanja. Vođa koji aktivno razvija ima manje vremena za menadžment.

📞 Kompleksnost zadataka

Call centar ili prodajni timovi često imaju standardiziranije procese, što omogućuje vođi da upravlja s više ljudi.

🔄 Komunikacijske linije

U timu od 6 ljudi postoji 15 komunikacijskih linija. U timu od 12 ljudi - čak 66! Eksponencijalni rast kompleksnosti.

⚖️ "Two pizza rule"

Amazon pravilo: Tim ne bi trebao biti veći od onoga što se može nahraniti s dvije pizze (obično 5-8 ljudi).

🚀 Agile preporuke

Scrum preporučuje timove od 5-9 članova. Manje od 5 - nedovoljan capacity. Više od 9 - previše kompleksnosti.

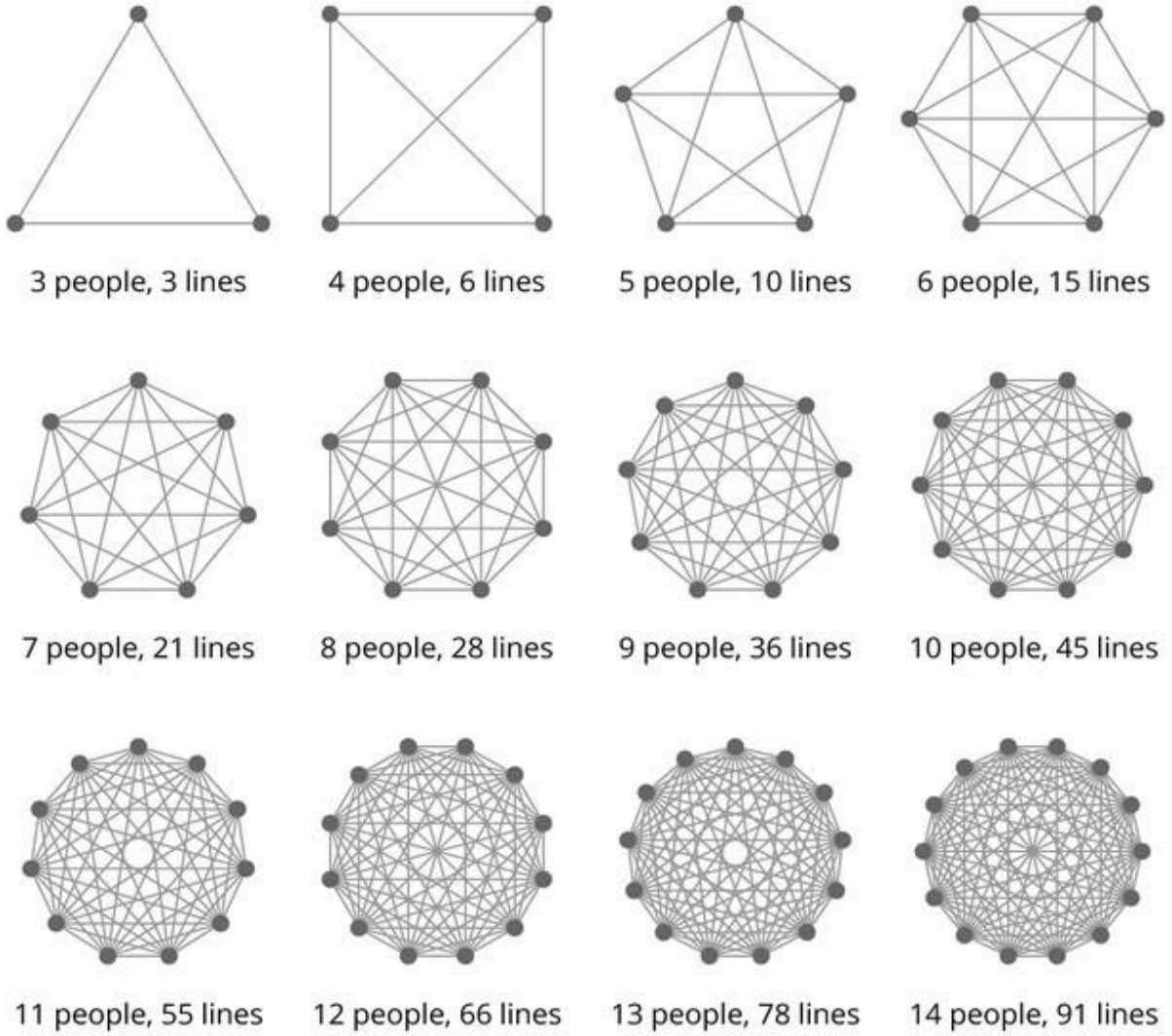
🎓 Istraživanja pokazuju

Studije potvrđuju da produktivnost po osobi opada kada tim premaši 7-8 članova zbog komunikacijskih overhead-a.

🎯 **Zlatno pravilo:** Manji tim = bolja komunikacija, brže odluke, veća agilnost.

Ako tim raste iznad optimalne veličine, razmotri podjelu na dva manja tima!

Porast složenosti interakcija u timu



Stvaranje klanova u timu

Porast složenosti interakcija u timu

Što su "klanovi" u timu?

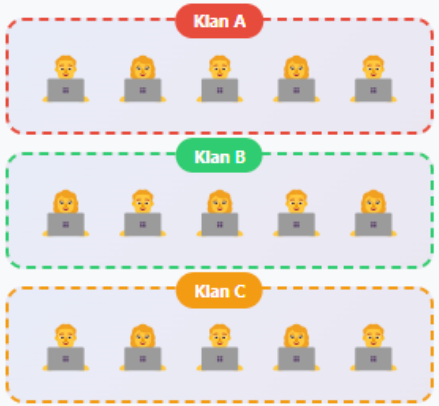
Kada tim raste, ljudi prirodno formiraju **mikro-timove** ili "klanova" - manje grupe unutar većeg tima s kojima najčešće surađuju. Ovo nije problem, već **prirodan proces** koji može biti koristan ako se pravilno upravlja!

Vizualizacija: Kako se formiraju klanovi?

Mali tim (6 ljudi)

Svi međusobno dobro komuniciraju

Veliki tim (15 ljudi)



Formiraju se prirodne podgrupe

Prirodno

Mikro-timovi se formiraju **spontano** unutar većeg tima. To je psihološki prirodan proces - ljudi se gravitiraju prema onima s kojima najčešće rade.

Nije negativno

Klanova **nisu loša stvar!** Mogu povećati efikasnost, omogućiti specijalizaciju i brže donošenje odluka unutar podgrupe.

4-6 ljudi

Klanova obično imaju **4-6 članova** - što je, ne slučajno, optimalna veličina tima! To je prirodna ljudska sklonost ka manjim grupama.

💡 Zaključak

4-6

Veličina klana je **prirodna mjera optimalne veličine tima**.
Čak i u velikim timovima, ljudi formiraju grupe od 4-6 osoba!

🎯 Zašto klanovi mogu biti korisni?

⚡ Brža komunikacija

Manji broj ljudi znači brže dogovaranje i manje komunikacijskih overhead-a unutar klana.

🎓 Specijalizacija

Svaki klan može se fokusirati na specifične aspekte projekta (frontend, backend, testiranje, itd.).

🤝 Jači timski duh

Članovi klana razvijaju bliskije radne odnose i bolje razumijevanje.

🔄 Fleksibilnost

Lakše koordinirati manji broj ljudi za specifične zadatke ili sprint ciljeve.

📊 Jasnije odgovornosti

Svaki klan može imati svoje zone odgovornosti unutar većeg projekta.

💪 Autonomija

Klanova mogu samostalno donositi odluke u svom domenu bez čekanja velikog tima.

⚠️ Pažnja: Potencijalni izazovi

Iako su klanova prirodni i često korisni, mogu stvoriti probleme ako nisu pravilno upravljani:

🚧 Silosi

Klanova mogu postati izolirani i prestati komunicirati s drugim klanovima, što usporava cjelokupni projekt.

⏳ Konkurencija

Može se razviti nezdrava konkurencija između klanova umjesto suradnje.

🧩 Fragmentacija znanja

Informacije mogu ostati zarobljene unutar jednog klana, što smanjuje transparentnost.

🎯 Gubitak fokusa

Klanova mogu težiti svojim ciljevima umjesto zajedničkim ciljevima tima.

🎓 Za zapamtiti:

Stvaranje klanova je **prirodan psihološki proces** kada tim preraste 7-9 članova.
Umjesto da to sprječavate, **iskoristite to** kroz pravilnu organizaciju i jasne komunikacijske kanale između klanova!

💡 Optimalan tim = 4-6 ljudi. Ako imate više, planirajte za klanova!

🛠️ Kako upravljati klanovima?

🎯 Cross-functional sastanci

Održavajte redovite sastanke gdje se svi klanova povezuju i dijele napredak.

🔄 Rotacija članova

Povremeno izmjenjujte ljude između klanova za bolju razmjenu znanja.

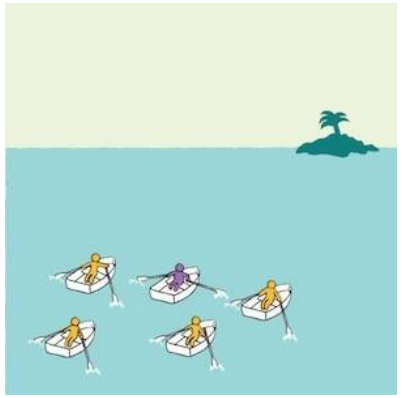
📢 Transparentna komunikacija

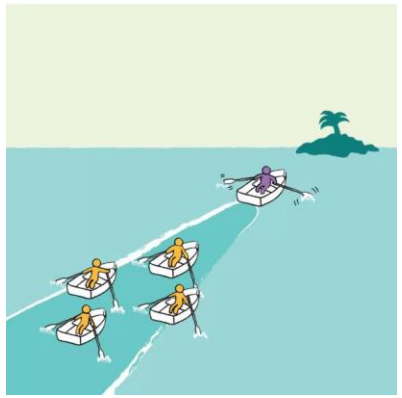
Koristite alate i procese koji omogućuju svim klanovima da vide što drugi rade.

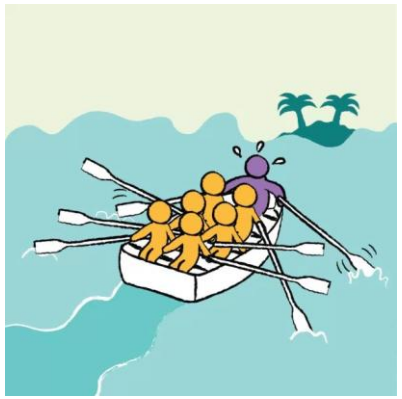
🤝 Team building

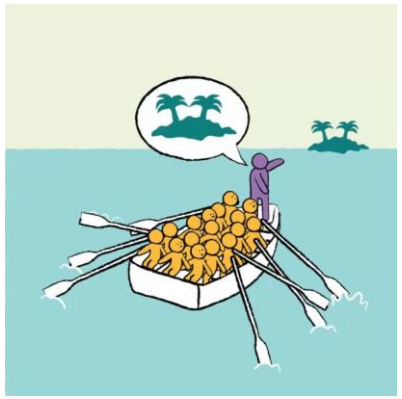
Organizirajte aktivnosti koje spajaju sve klanova i jačaju osjećaj pripadnosti većem timu.

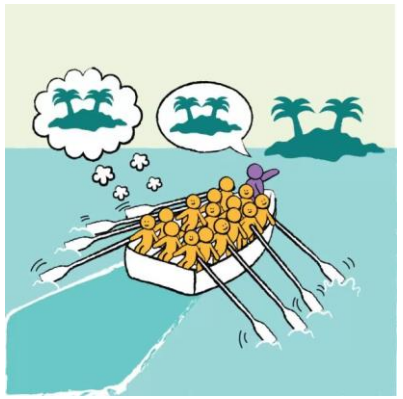
Porast složenosti interakcija u timu











🚣 Prijenos odgovornosti u timu

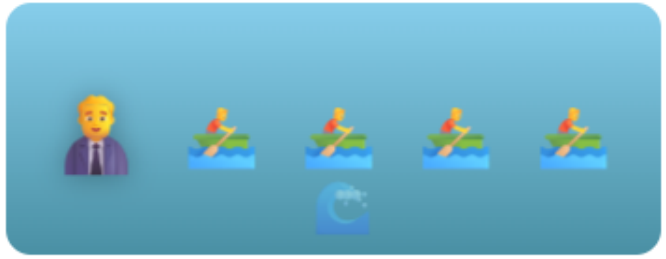
Kao veslački tim - svi moraju biti sinkronizirani!

🌊 Metafora veslača

Softverski tim je kao veslački čamac - svi moraju veslati u istom smjeru i ritmu da bi stigli do cilja. Ako jedan veslač ide svojim putem ili prestane veslati, cijeli čamac uspori ili skrene s kursa!

1

🎯 Koordiniran tim

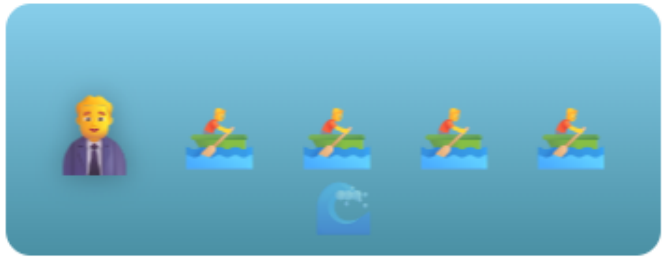


Idealna situacija: Svi veslači veslaju u istom smjeru, pod vodstvom kormilara. Jasne uloge, sinkronizacija i zajednički cilj.

✅ Optimalno

2

🔄 Delegiranje zadataka

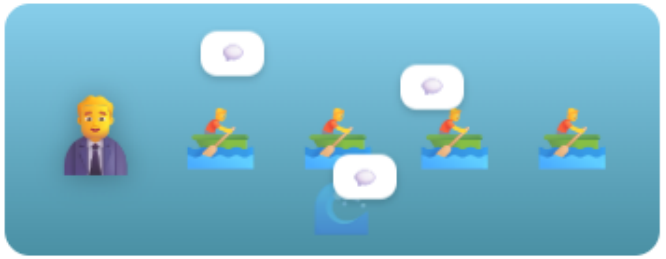


Prijenos odgovornosti: Neki veslači dobivaju specifične zadatke i autonomiju. Vođa delegira, ali zadržava koordinaciju.

⚠️ Zahtijeva koordinaciju

3

💬 Povećana komunikacija

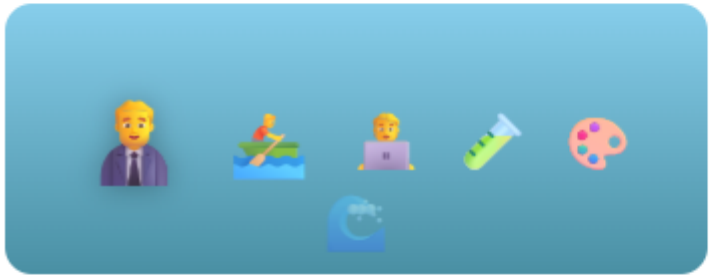


Više autonomije = **više komunikacije:** Kako ljudi preuzimaju odgovornosti, moraju više komunicirati međusobno kako bi ostali sinkronizirani.

✅ Zdrava dinamika

4

🎭 Različite odgovornosti

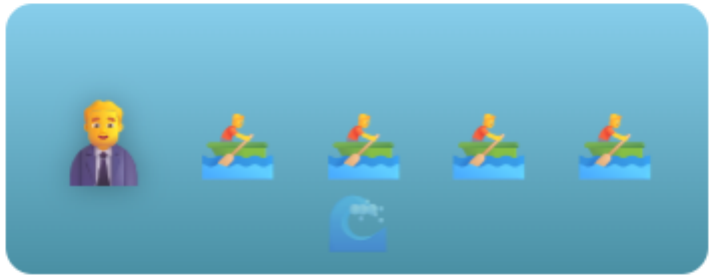


Specijalizacija uloga: Frontend, backend, tester, designer - svaki ima svoju ulogu, ali svi moraju raditi zajedno prema istom cilju.

✅ **Multidisciplinarni tim**

5

⚠️ Gubljenje sinkronizacije



Loša praksa: Ako svatko ide svojim putem bez koordinacije, čamac se vrti u krug i ne napreduje. Prijenos odgovornosti bez komunikacije!

❌ **Kaos**

6

🔥 Preopterećeni vođa

Greška: Vođa pokušava sve sam raditi umjesto da delegira odgovornosti. Tim ne radi puno kapaciteta, vođa je bottleneck.

❌ **Neučinkovito**

🎯 Ključni principi uspješnog prijenosa odgovornosti

🎯 Jasne odgovornosti

Svaka osoba mora točno znati što je njezina odgovornost i gdje su granice. Kao u veslačkom čamcu - svaki veslač zna svoju poziciju.

💬 Konstantna komunikacija

Više autonomije = više potrebe za komunikacijom. Tim mora redovito usklađivati napredak i probleme.

🤝 Međuovisnost

Priznati da smo ovisni jedni o drugima. Frontend ne može bez backend-a, razvoj ne može bez testiranja.

🏅 Povjerenje i autonomija

Vođa mora vjerovati timu i dati im slobodu donošenja odluka u svom domenu.

👁️ Vidljivost napretka

Svi moraju vidjeti što drugi rade. Koristite alate kao što su Jira, Trello, ili Scrum boardove.

🎯 Zajednički cilj

Bez obzira na različite uloge, svi moraju težiti istom cilju - kao veslači prema cilju!

⚖️ Dobra vs. Loša praksa

✅ Dobra praksa

- ✅ Jasno definirane uloge i odgovornosti
- ✅ Redoviti stand-up meetingi (dnevna sinkronizacija)
- ✅ Transparentni alati za praćenje (Jira, GitHub)
- ✅ Code review i peer feedback
- ✅ Retrospektive za kontinuirano poboljšanje
- ✅ Dokumentacija odluka i arhitekture
- ✅ Delegiranje s povjerenjem
- ✅ Sprint planiranja svi sudjeluju

❌ Loša praksa

- ❌ Nejasne ili preklapajuće odgovornosti
- ❌ Rijetka ili nikakva komunikacija
- ❌ Rad u silozima bez međusobne suradnje
- ❌ Vođa donosi sve odluke sam (mikromenadžment)
- ❌ Skrivanje informacija ili problema
- ❌ Prebacivanje krivnje umjesto rješavanja problema
- ❌ Nema dokumentacije ili znanje samo u glavama
- ❌ Pojedinci blokiraju napredak cijelog tima

💡 **Zaključak: Ravnoteža je ključ!**

Kao u veslačkom čamcu, tim mora pronaći savršenu ravnotežu između:

Autonomije (svaki veslač kontrolira svoje veslo) i **Koordinacije** (svi veslaju u istom ritmu).

🎯 Prijenos odgovornosti nije "prepuštanje kontrole" - to je **koordinirano dijeljenje moći** gdje svatko zna svoju ulogu i kako doprinosi zajedničkom cilju!

🏢 Google - Primjer organizacije tima

Kako jedna od najvećih tech kompanija organizira svoje timove

Google

📊 Google pristup organizaciji tima

U Google-u, **vođa proizvoda (Product Manager)** vodi napremu inženjera. Međutim, veličina tima varira ovisno o fazi i kompleksnosti projekta!

Google prosječan omjer

1:7

1 vođa na 7 inženjera

(Ali raspon ide od 1:2 do 1:20 ovisno o projektu!)

🔄 Različiti omjeri za različite faze projekta

🚀 Novi proizvod

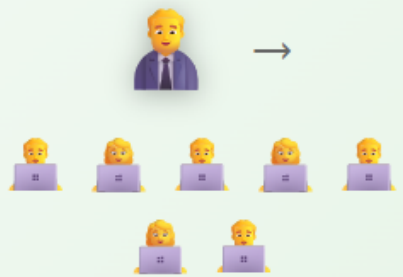
1:2

Kada? Novi proizvod u ranoj fazi razvoja

💡 **Zašto tako mali tim?**
- Puno učenja i eksperimentiranja
- Brze promjene smjera
- Intenzivna komunikacija potrebna
- Nisu još definirani procesi

📈 Prosječan projekt

1:7

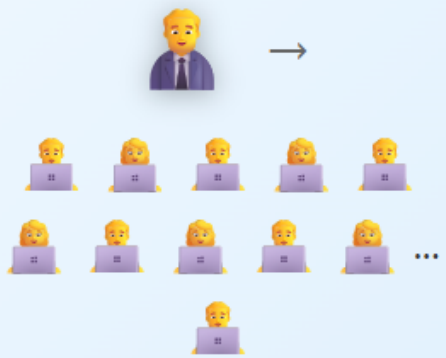


Kada? Standardni proizvod u aktivnom razvoju

💡 **Optimalna veličina!**
- Dovoljno ljudi za velocity
- Još uvijek upravljivu
- Dobra ravnoteža autonomije i koordinacije
- Jasni procesi i odgovornosti

⚙️ Uhodan proizvod

1:20



Kada? Zreli proizvod s etabliranim procesima

💡 **Primjer: Google Search**
- Procesi su jasni i uhodani
- Tim je samostalan
- Manje potrebe za intenzivnim nadzorom
- Infrastruktura i alati su zreli

📊 Detaljna usporedba

Faza projekta	Karakteristike	Zašto taj omjer?
🚀 **Novi (1:2)**	• Nepoznato područje • Brze iteracije • Često mijenjanje smjera	Vođa mora biti blizu svakog inženjera, puno učenja i eksperimentiranja
📈 **Prosječan (1:7)**	• Definirani ciljevi • Jasni procesi • Balans autonomije	Optimalna veličina za produktivnost i koordinaciju
⚙️ **Uhodan (1:20)**	• Stabilni procesi • Iskusan tim • Predvidljiv rad	Tim je samostalan, vođa više koordinira nego upravlja svaki dan

🎓 Što možemo naučiti od Google-a?

📏 Veličina ovisi o kontekstu

Nema univerzalnog pravila - prilagodi veličinu tima fazi projekta, kompleksnosti i iskustvu tima.

🚀 Počni s malim

Za nove projekte, mali tim omogućuje brže odluke, eksperimente i pivote bez velike koordinacije.

📈 Rasti postepeno

Kako projekt sazrijeva i procesi postaju jasni, tim može rasti bez gubljenja efikasnosti.

⚙️ Autonomija = veći tim

Uhodani proizvodi s jasnim procesima mogu imati veće timove jer je manje dnevne koordinacije.

👥 Mentorstvo zahtijeva mali tim

Kada novi ljudi dolaze ili treba puno učenja, drži tim malim za kvalitetno mentorstvo.

🎯 Kvaliteta > Kvantiteta

Google preferira male, fokusirane timove visokih performansi nego velike timove s lošom koordinacijom.

🔍 Kada tim treba biti manji?

📚 Kada treba puno učenja

Nova tehnologija, nova domena ili novi pristup - vođa mora biti blizu svakog člana za podršku i mentorstvo.

🌱 Novi ljudi dolaze u tim

Junior developeri ili ljudi novi u projektu trebaju više pažnje, pitanja i vodstva. Mali tim omogućuje kvalitetno onboarding.

🎯 Nisu samostalni

Ako članovi tima ne mogu samostalno donositi odluke i trebaju česte provjere, vođa može upravljati s manje ljudi.

🚀 Visoka kompleksnost

Arhitektonski izazovni projekti zahtijevaju blisko vođenje i kontinuirano usklađivanje - manji tim je efikasniji.

⚡ Brze promjene

Projekti gdje se smjer često mijenja (pivot) zahtijevaju malu, agilnu grupu koja može brzo reagirati.

🔬 Istraživanje i inovacija

R&D projekti s puno nepoznanica i eksperimenata funkcioniraju bolje s malim, fleksibilnim timom.

💡 **Ključna lekcija iz Google primjera:**

Ne postoji "magic number" za veličinu tima. Google varira od **1:2 do 1:20** ovisno o:

🎯 **Fazi projekta** (novi vs. uhodan)

📚 **Potrebi za učenjem** (puno učenja = manji tim)

👥 **Iskustvu tima** (junior = manji tim, senior = veći tim)

⚙️ **Zrelosti procesa** (kaotično = manji tim, strukturirano = veći tim)

🎓 Savjet: Počni s malim timom, rasti samo kada imaš dobre razloge!

📋 Projektni prijedlog

Projekt programskog inženjerstva

🎯 Što je projektni prijedlog?

Projektni prijedlog je ključni dokument koji definira što želimo postići, kako ćemo to učiniti, što će koštati i koji su rizici. To je "blueprint" cijelog projekta - **mapa puta** od ideje do realizacije!

🎯 FOKUS: Vizija i Doseg (Vision & Scope)

Vizija i Doseg

Najvažnija komponenta! Definira što gradimo i zašto.

- ✓ Koji problem rješavamo?
- ✓ Za koga gradimo proizvod?
- ✓ Što je u scopeu, a što nije?
- ✓ Koje su glavne funkcionalnosti?

Troškovi

Procjena resursa i budžeta potrebnog za realizaciju projekta.

- ✓ Ljudski resursi (developeri, dizajneri)
- ✓ Infrastruktura (serveri, alati)
- ✓ Licencije i servisi
- ✓ Vremenski okvir

Radni Plan

Timeline i faze projekta s ključnim milestoneovima.

- ✓ Faze razvoja (Sprint planovi)
- ✓ Ključni milestoneovi
- ✓ Deadlinei i deliverablesi
- ✓ Zavisnosti između taskova

Rizici

Identifikacija potencijalnih problema i strategije za njihovo ublažavanje.

- ✓ Tehnički rizici (skalabilnost, sigurnost)
- ✓ Resursni rizici (ljudi, budžet)
- ✓ Tržišni rizici (konkurencija)
- ✓ Plan za mitigaciju rizika

Izbor Suradnika

Definiranje tima i uloga potrebnih za uspjeh projekta.

- ✓ Uloge u timu (dev, QA, PM, dizajner)
- ✓ Potrebne vještine i iskustvo
- ✓ Interna vs. eksterna pomoć
- ✓ Struktura tima i komunikacija

🔭 Vizija i Doseg - Detaljnije

👁️ Vizija (Vision)

Vizija je "Zašto?" - inspiracija iza projekta.

- → **Problem:** Koji problem rješavamo?
- → **Korisnici:** Za koga je proizvod?
- → **Vrijednost:** Koja je korist za korisnike?
- → **Uspjeh:** Kako izgleda uspješan projekt?
- → **Dugoročni cilj:** Gdje želimo biti za 1-2 godine?

Primjer: "Stvoriti najjednostavniju aplikaciju za organizaciju timskih sastanaka koja štedi vrijeme i povećava produktivnost timova."

🎯 Doseg (Scope)

Scope je "Što?" - konkretne granice projekta.

- → **IN scope:** Što GRADIMO (funkcionalnosti)
- → **OUT of scope:** Što NE GRADIMO (granice)
- → **Prioriteti:** Must-have vs. Nice-to-have
- → **Ograničenja:** Tehnička, vremenska, budžetska
- → **MVP:** Minimum Viable Product - prva verzija

Primjer: "IN: Kreiranje sastanaka, kalendar, notifikacije. OUT: Video konferencije (integracija s Zoom), AI asistent."

⚠️ Najčešća greška:

Previše širok scope! Timovi često pokušavaju uraditi sve odjednom. Ključ uspjeha je **jasno reći ŠTO NE RADITE** u prvoj verziji. Bolje manje funkcionalnosti koje rade savršeno, nego mnogo polu-gotovih!

🔄 Proces kreiranja projektnog prijedloga

1. **🔭 Vizija**
   Definiraj problem i zašto ga rješavaš
2. **🎯 Scope**
   Odredi što ćeš graditi i što ne
3. **🗓️ Plan**
   Napravi timeline i milestoneove
4. **💰 Troškovi**
   Procijeni resurse i budžet
5. **⚠️ Rizici**
   Identificiraj i planiraj za rizike

💡 Savjeti za odličan projektni prijedlog

🎯 Budi specifičan

Izbjegavaj opće fraze. Umjesto "korisnicima će biti lakše", reci "korisnici će uštedjeti 30 minuta dnevno na organizaciji sastanaka".

✂️ Manje je više

Bolje 3 funkcionalnosti koje rade savršeno nego 10 polu-gotovih. MVP prvo, sve ostalo kasnije!

👥 Razgovaraj s korisnicima

Ne pretpostavljaj što ljudi trebaju - pitaj ih! Napravi istraživanje, intervjue, ankete.

📊 Koristi podatke

Potkrijepite tvrdnje s podacima: "85% timova se žali na kaos oko sastanaka" zvuči puno jače nego "ljudi imaju problema".

🔄 Budi realan

Nemoj obećavati nemoguće. Realni ciljevi i timeline grade povjerenje i omogućuju uspjeh.

📝 Dokumentiraj odluke

Objasni ZAŠTO si odlučio nešto ne raditi. To štiti tim od "feature creep-a" kasnije.

🎓 **Zaključak:**

Projektni prijedlog je **kompas** vašeg projekta. Bez njega, tim luta u mraku.

🔭 **Vizija** daje inspiraciju i smjer - "Zašto to radimo?"

🎯 **Scope** postavlja granice - "Što radimo i što NE radimo?"

🗓️ **Plan** organizira izvršenje - "Kako ćemo to uraditi?"

💰 **Troškovi** osiguravaju realnost - "Što nam treba?"

⚠️ **Rizici** pripremaju na probleme - "Što može poći krivo?"

💡 Zlatno pravilo: Bolje dan provedeni u planiranju nego tjedan u ispravljanju grešaka!

🔭 Vizija i Doseg

Temelj svakog uspješnog projekta

👁️ Vizija

*"**ZAŠTO?**" - Opisuje bit projekta i definira poslovne ciljeve*

📊 Trenutno stanje

Gdje se nalazimo sada? Koja je trenutna situacija i kontekst?

❓ Problem koji se rješava

Koji specifični problem pokušavamo riješiti? Zašto je to važno?

💡 Napuci rješenja

Kako ćemo riješiti problem? Koji je naš pristup?

🎯 Zahtjevi (high-level)

Što mora proizvod raditi na visokoj razini? Glavne funkcionalnosti bez detalja.

- → Ne ulazimo u tehničke detalje
- → Fokus na "što", ne "kako"
- → Iz perspektive korisnika

⚠️ Glavni rizici (high-level)

Što bi moglo poći po zlu? Najveće prijetnje uspjehu projekta.

- → Tehnički rizici
- → Tržišni rizici
- → Resursni rizici

🎯 Doseg

*"**ŠTO?**" - Definira granice projekta i fokusira resurse*

🚫 Ograničenja

Što NE ulazi u projekt? Jasno postavljanje granica.

- → Koje funkcionalnosti nisu uključene
- → Što ćemo raditi u budućnosti
- → Tehnologije koje ne koristimo

👥 Ciljna skupina korisnika

Za koga je proizvod? Fokusiranje na specifičnu skupinu, ne na sve!

- → Primarne persone
- → Sekundarne persone
- → Koga eksplicitno NE ciljamo

🗺️ Pokrivenost tržišta

Koji dio tržišta pokrivamo, a koji ne?

- → Geografska ograničenja
- → Industrije/sektori
- → Veličina tvrtki (mali, srednji, veliki)

🔄 Planiranje proširivanja

Kako ćemo proširivati doseg u nekoliko krugova iteracija?

- → MVP → V2.0 → V3.0
- → Prioritizacija funkcionalnosti
- → Roadmap proširenja

Praktični primjer: Aplikacija za upravljanje projektima

Vizija

Trenutno stanje: Mali timovi (5-10 ljudi) koriste Excel i email za organizaciju projekata, što je kaotično i neefikasno.

Problem: 60% vremena se gubi na traženje informacija i koordinaciju.

Rješenje: Jednostavna web aplikacija za task management s fokusom na jednostavnost.

High-level zahtjevi:

- Kreiranje i dodjela taskova
- Praćenje napretka
- Notifikacije
- Jednostavan kalendar

Doseg (Scope)

IN scope (MVP): Task management, kalendar, notifikacije, basic reporting

OUT of scope (MVP):

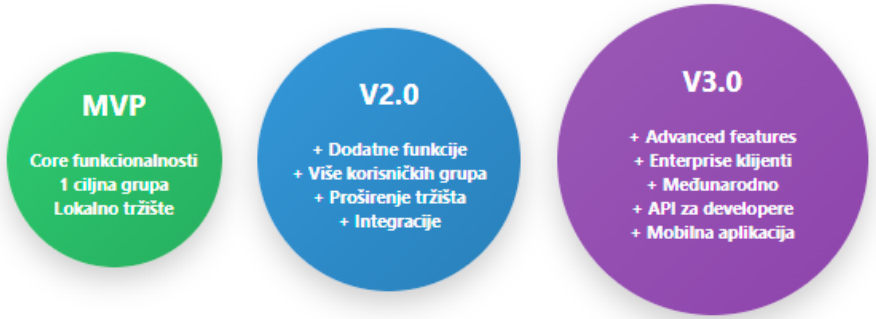
- ❌ Time tracking
- ❌ Integracije (Slack, Teams)
- ❌ Napredni analytics
- ❌ Mobilna aplikacija

Ciljna skupina: Startupi i male tech tvrtke s 5-15 ljudi

NE ciljamo: Velike korporacije, non-tech industrije

Proširivanje dosega kroz iteracije

Ne radite sve odjednom! Planirajte proširivanje u fazama.



💡 Svaki krug dodaje novi sloj funkcionalnosti i širi ciljanu skupinu.
Bolje savršen MVP nego prosječan full-featured proizvod!

⚖️ Vizija vs. Doseg - Ključne razlike

👁️ Vizija

Fokus:
ZAŠTO radimo projekt

Razina:
Visoka razina (30,000 ft view)

Pitanja:
Koji problem? Zašto važno? Koji cilj?

Trajanje:
Dugoročno - rijetko se mijenja

Publika:
Stakeholderi, investitori, cijela tvrtka

Primjer:
"Omogućiti malim timovima da rade efikasnije"

🎯 Doseg

Fokus:
ŠTO radimo (i što NE radimo)

Razina:
Konkretno - jasne granice

Pitanja:
Što je u scopeu? Za koga? Što nije?

Trajanje:
Kratkoročno - mijenja se sa verzijama

Publika:
Razvojni tim, project manager

Primjer:
"MVP uključuje task management, NE uključuje time tracking"

💡 Savjeti za pisanje Vizije i Dosega

🎯 Vizija: Inspirativna, ne tehnička

Vizija treba inspirirati tim i stakeholdere. Izbjegavaj tehničke detalje - fokusiraj se na "zašto" i poslovnu vrijednost.

✂️ Scope: Reci ŠTO NE radiš

Najvažniji dio scopea je lista "OUT of scope". To štiti tim od feature creep-a i drži fokus.

👥 Definiraj ciljanu skupinu usko

Ne možeš zadovoljiti sve. Bolje savršeno za 1000 ljudi nego osrednje za milijun.

📊 Koristi podatke

"60% timova gubi vrijeme" zvuči jače od "timovi gube vrijeme". Potkrijepite viziju s podacima.

🔄 Planiraj iteracije

Ne pokušavaj sve u MVP-u. Napravi roadmap kako ćeš proširivati scope kroz verzije.

⚠️ Budi realan s rizicima

Ne sakrivaj rizike. Bolje ih adresirati rano nego biti iznenađen kasnije.

🎓 **Ključna lekcija:**

Vizija daje inspiraciju i smjer - "Putujemo u Pariz!"

Doseg definira kako ćemo stići tamo - "Autom, ne avionom; kroz Austriju, ne Italiju"

👁️ Vizija je konstantna - cilj ne mijenjamo

🎯 Scope je fleksibilan - put do cilja prilagođavamo

💡 Najbolji projekti imaju jasnu viziju ali fleksibilan scope koji se proširuje kroz iteracije!

Hvala na pažnji!