

Razvoj programske opreme

3.letnik TR
(zapiski predavanj)
delovna verzija

zapiski so namenjeni spremljanju predavanj

zbral in uredil: © Simon Pogorelčnik univ. dipl. inž.

Proces razvoja programske opreme - Od koncepta do vzdrževanja

Razvoj programske opreme je zapleten in večplasten proces, ki ni samo zgolj pisanje kode. Temelji na strukturiranem pristopu, znanem kot življenjski cikel razvoja programske opreme (SDLC). Ta vodi ekipe od začetnega koncepta do končnega izdelka in njegovega dolgoročnega vzdrževanja.

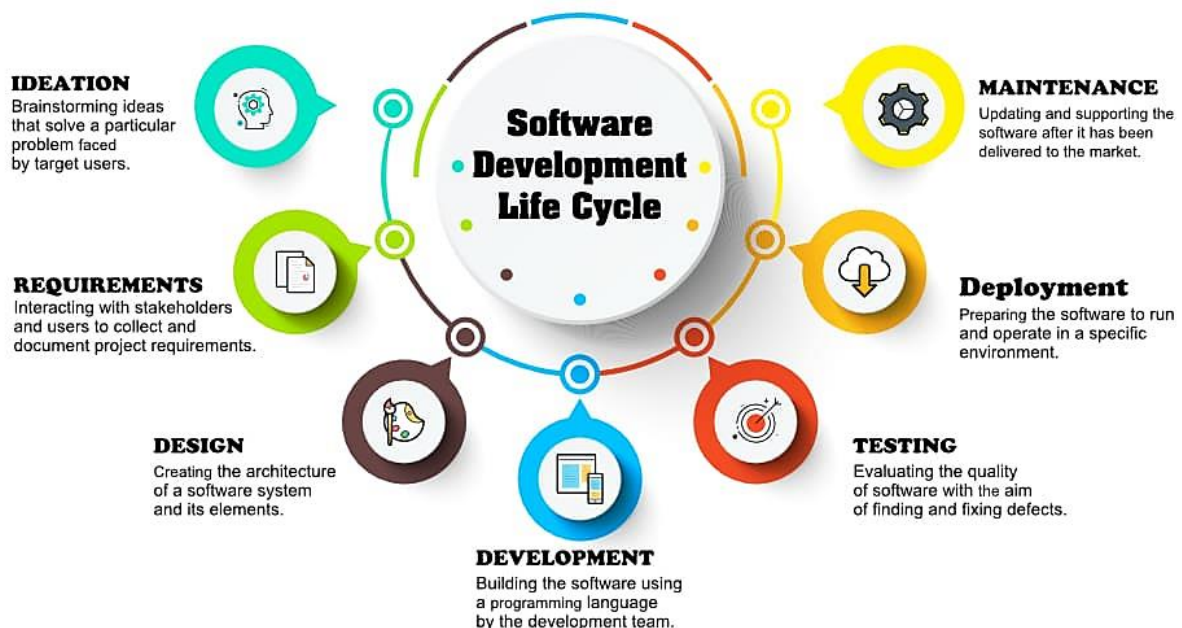
Sodobne metodologije, kot so agilni pristopi, so se razvile kot odgovor na omejitve tradicionalnih linearnih modelov, ker poudarjajo pomen hitrega razvoja (kratek čas med idejo in realizacijo) in prilagodljivost (nenehne spremembe programske opreme).

Uspeh projekta je danes odvisen od aktivnega sodelovanja različnih strokovnjakov: strokovnjaki s področja, ki ga aplikacija (program) obravnava, analitiki, oblikovalci, razvijalci in strokovnjaki za zagotavljanje kakovosti. Ključni element, ki omogoča hiter razvoj in zanesljivost, je avtomatizacija skozi procese, kot sta neprekinjena integracija in dostava (CI/CD). Analize kažejo, da je sodobni razvoj programske opreme povezava med klasičnim načinom oz. pristopom k razvoju programske opreme (SDLC), prilagodljivo filozofijo (Agilne metodologije) in napredno avtomatizacijo (CI/CD), kar omogoča učinkovito doseganje visokokakovostne programske opreme.

Razumevanje procesa razvoja programske opreme (SDLC)

Kaj je življenjski cikel razvoja programske opreme (SDLC)?

Življenjski cikel razvoja programske opreme (SDLC) je strukturiran proces, ki služi kot temelj industrije programske opreme. Predstavlja sistematičen okvir, ki razvijalce in organizacije vodi skozi celotno pot ustvarjanja, uvajanja in vzdrževanja programskih izdelkov. Namen SDLC je zagotoviti, da je vsaka faza razvoja skrbno načrtovana, izvedena in dokumentirana, s čimer se doseže izdelek, ki ustreza pričakovanjem strank in poslovnim ciljem.



Slika 1: življenjski cikel programa

Pomen SDLC je večplasten.

- zagotavlja strukturiran okvir, ki organizira razvojne faze v obvladljive korake, kar prinaša jasnost in zmanjšuje tveganja projekta. S tem se zagotovi, da so viri, roki in načrti učinkovito upravljeni.
- SDLC v svoj proces vključuje testiranje in validacijo, kar omogoča zgodnje odkrivanje in odpravljanje težav, posledično pa to vodi do visokokakovostne programske opreme.
- dobro opredeljen SDLC prispeva k učinkovitemu upravljanju stroškov, saj zmanjša napake, ponovno delo in preprečuje zamude, kar na koncu privede do prihrankov.
- SDLC spodbuja učinkovito komunikacijo in sodelovanje med vsemi deležniki – od razvojne ekipe in vodij projektov do strank – s čimer se zagotovi, da so cilji razvoja programske opreme usklajeni s splošnimi poslovnimi cilji.

Koraki razvoja programske opreme

Razvoj programske opreme poteka skozi več faz. Čeprav se pristopi lahko razlikujejo, večina projektov sledi splošnemu **življenjskemu ciklu razvoja programske opreme (SDLC - Software Development Life Cycle)**, ki vključuje naslednje ključne korake.

1. Načrtovanje in analiza zahtev

To je prva in ena najpomembnejših faz. Cilj je podrobno razumeti, kaj stranka ali uporabnik potrebuje.

- **Načrtovanje in brainstorming**
Prvi korak v življenjskem ciklu razvoja programske opreme je načrtovanje. To je faza, ko zberete ekipo za brainstorming, postavljanje ciljev in prepoznavanje tveganj. Na tej stopnji bo ekipa sodelovala pri oblikovanju niza poslovnih ciljev, zahtev, specifikacij in vseh tveganj visoke ravni, ki bi lahko ovirala uspeh projekta.
- **Zbiranje zahtev**
Ekipa se sreča z deležniki, da ugotovi, kaj mora programska oprema delovati. To vključuje funkcionalne zahteve (kaj mora sistem početi) in nefunkcionalne zahteve (kako hitro mora delovati, kako varen mora biti itd.).
- **Analiza zahtev**
Zbrane zahteve se analizirajo in dokumentirajo. Pripravi se podroben dokument, ki služi kot osnova za celoten projekt. Ta faza pomaga preprečiti nesporazume in morebitne napake v kasnejših fazah.

2. Oblikovanje (Design)

Ko so zahteve določene, se začne načrtovanje arhitekture programske opreme. Ko imate načrte oblikovanja, je čas za izdelavo žičnih modelov in maket (wireframe, mockup). Ta korak poudarja faze načrtovanja in definira naloge, ki jih morate opraviti v urniku razčlenitve dela. Na voljo je veliko orodij, kot sta Adobe XD ali InVision, ki ta postopek precej olajšajo.

- **Arhitekturno oblikovanje:** Določi se visoka raven strukture sistema, vključno s komponentami, moduli in interakcijami med njimi. Razmisli se o tehnologijah, ki se bodo uporabile, ter o splošni zasnovi sistema.
- **Podrobno oblikovanje:** Vsak modul se podrobneje načrtuje, vključno z logiko, podatkovnimi strukturami in algoritmi. Ta faza pogosto vključuje ustvarjanje diagramov (npr. UML), ki vizualno prikazujejo, kako bo sistem deloval.
- **Oblikovanje uporabniškega vmesnika (UI/UX):** Posebna pozornost se nameni temu, kako bo uporabnik interaktiral s programsko opremo. Pripravijo se osnutki (wireframes) in prototipi, ki se testirajo z uporabniki, da se zagotovi intuitivnost in enostavnost uporabe.

3. Implementacija (Kodiranje)

V tej fazi programerji pišejo kodo na podlagi prej pripravljenega načrta.

- **Pisanje kode:** Razvijalci prevajajo načrte v delujočo kodo, pri čemer upoštevajo standarde kodiranja in dobre prakse. To se pogosto izvaja v manjših delih (modulih).
- **Integracija:** Posamezni moduli se združujejo in preverja se, ali delujejo skupaj, kot je bilo načrtovano.
- **Sistem za nadzor različic (Git):** Ekipa uporablja orodja, kot je Git, da sledi spremembam v kodi, kar omogoča učinkovito sodelovanje med več razvijalci in enostavno vračanje na prejšnje različice, če pride do težav.

4. Testiranje

Testiranje je ključno za zagotavljanje kakovosti programske opreme in odkrivanje napak.

- **Testiranje enot (Unit Testing):** Posamezni deli kode se testirajo ločeno, da se preveri, ali delujejo pravilno.
- **Integracijsko testiranje:** Preverja se, ali različni moduli delujejo skupaj brez težav.
- **Sistemske testiranje:** Celoten sistem se testira, da se zagotovi, da izpolnjuje vse določene zahteve.
- **Sprejemno testiranje (Acceptance Testing):** Končni uporabniki ali stranka preizkusijo programsko opremo, da se prepričajo, da ustreza njihovim potrebam. Če je vse v redu, programsko opremo "sprejmejo".

5. Postavitev (Deployment)

V tej fazi se programska oprema postavi na produkcijsko okolje, kjer jo bodo uporabniki dejansko uporabljali.

- **Priprava okolja:** Pripravi se strežnik ali platforma, kjer bo programska oprema delovala.

- **Namestitev:** Koda in vsi potrebni viri se prenesejo na strežnik in konfigurirajo.
- **Preverjanje:** Po postavitvi se izvedejo še zadnji testi, da se zagotovi, da vse deluje, kot je treba.

6. Vzdrževanje in podpora

Razvojni cikel se ne konča s postavitvijo.

- **Popravljanje napak:** Vzdrževanje vključuje odpravljanje napak (bugov), ki se pojavijo po postavitvi.
- **Posodobitve in izboljšave:** Dodajajo se nove funkcionalnosti ali se izboljšajo obstoječe.
- **Podpora uporabnikom:** Zagotavlja se pomoč uporabnikom in reševanje njihovih težav.

Pomembno je poudariti, da se sodobni pristopi, kot je **agilna metodologija**, razlikujejo od tradicionalnega "slapnega modela" (Waterfall), kjer se vsaka faza izvaja strogo zaporedno.

Agilni razvoj poteka v kratkih, ponavljajočih se ciklih (imenovanih "sprinti"), kar omogoča hitro odzivanje na spremembe in hitrejšo dostavo delujočih delov programske opreme. Vseeno pa vsi ti pristopi pokrivajo zgoraj naštetih ključne faze.

Vsako od teh faz nastajanja programske opreme bomo natančno opisali.

Agilne metode - Filozofija hitrosti in prilagodljivosti

Agilne metodologije predstavljajo filozofijo in nabor vrednot, ki dajejo prednost hitrosti, sodelovanju in prilagodljivosti. Ta pristop je nastal kot odgovor na omejitve modela Waterfall in se osredotoča na nenehno dostavljanje delujoče programske opreme.

Temeljna načela agilne filozofije so definirana v **Agilnem manifestu**, ki določa štiri ključne vrednote :

- **Posamezniki in interakcije** so cenjeni bolj kot procesi in orodja.
- **Delujoča programska oprema** je cenjena bolj kot vseobsežna dokumentacija.
- **Sodelovanje s stranko** je cenjeno bolj kot pogodbeno pogajanje.
- **Odziv na spremembe** je cenjen bolj kot togo sledenje načrtom.

Te vrednote so osnova za dvanajst principov, ki služijo kot kompas za agilne ekipe.

Principi vključujejo: zgodnje in neprekinjeno izdajanje vredne programske opreme, sprejemanje sprememb zahtev tudi v poznih fazah razvoja, pogosto izdajanje delujoče programske opreme (od nekaj tednov do nekaj mesecev), in dnevno sodelovanje med poslovnimi in razvojnimi strokovnjaki.

Poudarjajo tudi pomen zaupanja motiviranim posameznikom, uporabo osebnega pogovora kot najučinkovitejšega načina komunikacije in merjenje napredka z delujočo programsko opremo. Pomemben princip je tudi nenehna težnja k tehnični odličnosti in dobremu načrtovanju, kar izboljša agilnost in zmožnost odzivanja.

Agilni principi niso zgolj idealistične izjave; neposredno so povezani s poslovno vrednostjo. Princip "Odziv na spremembe... v prid konkurenčnosti" in "Delujoča programska oprema je primarno merilo napredka" neposredno omogočata, da podjetja ostanejo konkurenčna na dinamičnem trgu. Namesto da bi prepozno ugotovila, da je izdelek nepravilen, agilni pristop omogoča, da se hitro obravnavajo nove zahteve, kar je bila ena glavnih pomanjkljivosti tradicionalnega modela Waterfall.

Hibridni pristopi in izbira metodologije

Izbira metodologije ni zgolj tehnična, ampak strateška odločitev, ki je odvisna od narave projekta.

Slapni model je primeren za projekte z jasnimi, nespremenljivimi zahtevami in fiksnimi roki. Nasprotno, agilni razvoj je nujen za kompleksne in dinamične projekte, kjer se zahteve nenehno spreminjajo, kot je to pogosto pri razvoju programske opreme. Odločitev med tema dvema pristopoma tako temelji na razumevanju tveganja, jasnosti zahtev, pričakovanj deležnikov in kulture organizacije.

Agilni model Cilj agilnega procesa razvoja programske opreme je zagotoviti visokokakovostno programsko opremo hitro, pogoste spremembe in poceni. Agilne metode dajejo prednost delujoči programski opremi pred obsežnim predhodnim načrtovanjem in dokumentacijo, kar lahko upočasni ustvarjalni proces. Gre za sodoben pristop s kratkimi fazami, ki dobro deluje, ko se bodo zahteve glede programske opreme verjetno pojavile na začetku procesa razvoja. Agilni model ponuja večjo prilagodljivost kot model Waterfall, vendar ni vedno primeren za obsežne projekte s kompleksnimi zahtevami, ker mu manjka začetna dokumentacija.

Iterativni model: Iterativni model organizira razvojni proces v majhne cikle namesto strogo linearnega napredovanja. To razvijalcem omogoča, da spremembe izvajajo postopoma in pogosto, tako da se učijo iz napak, preden postanejo drage. Razvijalci z iterativnim modelom skozi celoten proces prejemajo povratne informacije uporabnikov, zato je idealen za velike projekte z močno vodstveno ekipo.

Model v obliki črke V: Imenovan tudi model za preverjanje in validacijo, model v obliki črke V omogoča sočasen razvoj in testiranje. Tako kot model Waterfall tudi ta model sledi linearnemu napredovanju, vendar se na naslednjo fazo premaknete šele, ko ekipa konča prejšnjo. Model v obliki črke V se osredotoča na dokumentacijo in načrtovanje, zato je idealen za obsežne projekte z dolgimi časovnimi načrti. Vendar pa togost, vgrajena v sistem, omogoča le redke spremembe.

Model Big Bang: Big Bang ima manj strukture kot drugi modeli razvoja programske opreme. Pri tem modelu razvijalci začnejo delo z malo več kot razumevanjem zahtev projekta. Stvari morajo ugotavljati sproti, saj večino virov vložijo v fazo razvoja programske opreme. Big Bang se osredotoča na to, da nekaj hitro začene. Ta pristop dobro deluje pri majhnih projektih, kjer lahko en ali dva razvijalca sodelujeta pri določanju zahtev in rešitev med kodiranjem. Vendar pa je lahko pri velikih projektih drag in dolgotrajen.

Spiralni model : Spiralni model združuje elemente drugih modelov, in sicer slapnega in iterativnega. Razvijalci delajo v krajših ciklih, delo znotraj ciklov pa sledi linearnemu napredovanju. Po vsaki iteraciji se programska oprema postopoma izboljšuje. Ključna prednost tega modela je, da zelo učinkovito upravlja s tveganji, saj se osredotoča na majhne dele tveganja hkrati in uporablja različne pristope glede na profil tveganja v tej fazi. To razvijalcem omogoča prilagoditve, ne da bi pri tem ogrozili izid projekta. Ta pristop dobro deluje pri zelo kompleksnih, velikih in dragih projektih.

Kdo sodeluje pri razvoju programske opreme?

Ključne vloge in odgovornosti v razvojni ekipi

Uspešen razvoj programske opreme ni odvisen zgolj od veščin kodiranja, ampak od medsebojnega sodelovanja specializiranih vlog, ki združujejo poslovno, tehnično in uporabniško perspektivo. Te vloge delujejo v sinergiji, da zagotovijo nemoten in učinkovit razvojni proces.

1. Poslovni in sistemski analitik

Poslovni analitik deluje kot ključni "prevajalec" med poslovnimi uporabniki in razvijalci programske opreme. Njegova primarna naloga je zagotoviti, da informacijska tehnologija podpira in izboljšuje poslovne procese, medtem ko hkrati zadovoljuje potrebe končnih uporabnikov.

Glavne naloge analitika vključujejo zajem, analiziranje in upravljanje zahtev, kar je temeljen proces zagotavljanja kakovosti. V sodobnem okolju se njihova vloga razširja od zgolj dokumentiranja zahtev do razumevanja, zakaj so spremembe potrebne, in določanja želenih poslovnih ciljev, ki jih je treba doseči. Z uporabo ustreznih orodij za analizo podatkov in razvoj napovednih modelov, poslovni analitiki na podlagi preteklih podatkov napovedujejo prihodnje dogodke, kar pomaga pri načrtovanju in izboljšanju procesov. V agilnih okoljih so pogosto vključeni v analizo, oblikovanje, vodenje projekta in testiranje, odvisno od organizacije.

2. Oblikovalec uporabniške izkušnje in vmesnika (UX/UI)

Vloga oblikovalca uporabniške izkušnje (UX) in uporabniškega vmesnika (UI) je ustvariti intuitiven, učinkovit in privlačen digitalni izdelek. Čeprav se pogosto povezujeta, je UX zasnova proces razumevanja uporabnika in oblikovanja izkušnje z izdelkom, medtem ko se UI zasnova osredotoča na vizualne elemente, kot so postavitve, barve in tipografija.

UX/UI oblikovanje ni le estetska faza, ampak proces, ki se aplicira v vseh fazah SDLC. V fazi zbiranja zahtev sodelujejo pri razumevanju uporabniških potreb, med načrtovanjem ustvarjajo skice, žične mreže (wireframes) in prototipe, med implementacijo pa sodelujejo z razvijalci, da zagotovijo, da je končni izdelek vizualno skladen z oblikovalskimi specifikacijami.

Vloga UX/UI oblikovalca je ključna za zagotavljanje, da funkcionalnost, ki jo razvijajo inženirji, ne nastane na račun uporabnosti. S tem ustvarjajo most med funkcionalno in nefunkcionalno kakovostjo, kar je nujno za končno zadovoljstvo strank. Oblikovalci lahko vplivajo na tehnična področja, kot so struktura URL-jev, sporočila o napakah in odzivnost oblikovanja, s čimer pomagajo ustvariti uporabniku bolj prijazno izkušnjo.

3. Razvijalec programske opreme in arhitekt

Vloga razvijalca programske opreme se je močno razširila izven samega kodiranja. Danes razvijalci analizirajo, načrtujejo, programirajo, testirajo in vzdržujejo nove programske aplikacije. Njihove ključne odgovornosti vključujejo načrtovanje, razvoj in implementacijo programske opreme v skladu z najboljšimi praksami in specifikacijami.

Razvijalci so pogosto aktivno vključeni v razvojne projekte in sodelujejo z vodji projektov ter drugimi strokovnjaki. Arhitekti programske opreme imajo ključno vlogo pri načrtovanju tehnične rešitve projekta. Uporabljajo diagrame, kot je na primer UML (Unified Modeling Language), da vizualno predstavijo arhitekturo, zasnovo in izvedbo kompleksnih sistemov, še

preden se začne kodiranje. To pomaga pri odpravljanju nepotrebnih predelav in zagotavlja usklajenost med tehničnimi in netehničnimi člani ekipe.

4. Strokovnjak za zagotavljanje kakovosti (QA)

Zagotavljanje kakovosti (QA) je celosten proces, ki vključuje nadzor vseh inženirskih postopkov, metod in delovnih izdelkov, da se zagotovi skladnost z določenimi standardi. Testiranje je sicer ključni del QA, vendar je QA širši koncept, ki zagotavlja kakovost skozi celoten razvojni cikel.

Glavne naloge strokovnjaka za QA vključujejo vzpostavljanje standardov kakovosti, upravljanje zahtev, pregledovanje kode, načrtovanje testov in stalno izboljševanje procesov. Strokovnjaki za QA zagotavljajo, da je programska oprema zanesljiva in izpolnjuje pričakovanja strank, kar preprečuje izdajo okvarjenega izdelka. Z odkrivanjem napak v zgodnjih fazah prispevajo k prihrankom, saj je njihovo odpravljanje kasneje bistveno dražje.

5. Vodja projekta

Vodenje ekipe za razvoj programske opreme zahteva skrbno načrtovanje in učinkovito vodenje, pri čemer ima vodja projekta ključno vlogo pri določanju uspeha ali neuspeha projekta. Njihova primarna naloga je zagotoviti učinkovito komunikacijo in sodelovanje med člani ekipe, medtem ko se usklajujejo naloge, nadzoruje napredek in ohranjajo projekti v okviru določenih rokov.

Vloga vodje projekta je pretežno ne tehnična. Osredotočajo se na upravljanje ne tehničnih nalog in tveganj, kar razvijalcem omogoča, da se osredotočijo na svoje tehnične vloge. Uporabljajo programsko opremo za upravljanje projektov, da spremljajo napredek, dodeljujejo naloge, postavljajo roke in zagotavljajo, da je projekt na pravi poti. To poudarja, da uspeh projekta ni odvisen zgolj od kakovosti kode, ampak tudi od učinkovitega upravljanja celotnega procesa.

Vloga	Ključne naloge	Povezava z drugimi vlogami
Poslovni analitik	Zbiranje in analiza zahtev, dokumentacija, razumevanje poslovnih potreb, komunikacija med deležniki.	Prevaja poslovne zahteve razvijalcem, sodeluje z vodji projektov.
UX/UI Oblikovalec	Oblikovanje uporabniške izkušnje in vmesnika, ustvarjanje prototipov in vizualizacij, testiranje uporabnosti.	Sodeluje z razvijalci pri implementaciji, s poslovnimi analitiki pri razumevanju potreb.
Razvijalec	Analiza, načrtovanje, kodiranje, testiranje in vzdrževanje programske opreme, odpravljanje napak, dokumentacija.	Sodeluje z arhitekti pri načrtovanju, s strokovnjaki QA pri testiranju in z vodji projektov.
Strokovnjak QA (testiranje)	Vzpostavljanje standardov kakovosti, načrtovanje testov, odkrivanje napak , poročanje, avtomatizacija testiranja.	Tesno sodeluje z razvijalci med testiranjem in odpravljanjem napak, z vodjem QA pri dodeljevanju nalog.
Vodja projekta	Vodenje ekipe, usklajevanje nalog in virov, spremljanje napredka, upravljanje rokov in komunikacija z deležniki.	Dodeljuje naloge razvijalcem in testerjem, komunicira s strankami in drugimi deležniki.

Faze nastajanja programske opreme v praksi

Analiza in določanje zahtev

Prva faza v procesu razvoja je analiza in določanje zahtev. V tej fazi se zberejo zahteve strank, ciljnega trga in strokovnjakov iz industrije. Te informacije se nato uporabijo za izdelavo dokumentacije, kot sta dokument poslovnih specifikacij (BS) in dokument specifikacij programskih zahtev (SRS), ki podrobno določata, katere težave mora programska oprema rešiti. Ta faza je ključna, saj določa temelje celotnega projekta.

Načrtovanje in oblikovanje uporabniškega vmesnika (UI/UX)

Pri razvoju aplikacij je maketavisokozvestna, statična vizualna predstavitev uporabniškega vmesnika aplikacije, ki prikazuje končni videz zasnove, vključno z barvami, tipografijo, slikami in postavitvijo, vendar brez interaktivne funkcionalnosti. Služi kot "generalka" ali realističen posnetek izdelka za predstavitev konceptov, zbiranje povratnih informacij deležnikov in sprejemanje odločitev o vizualnem oblikovanju, preden se začne drag in dolgotrajen razvojni proces.

Kaj vključuje maketa

Vizualni elementi:

Vsi grafični elementi in estetske podrobnosti končnega izdelka, kot so ikone, gumbi, pisave, besedilna vsebina in barvne sheme.

Postavitev in struktura:

Splošna razporeditev in struktura strani ali zaslonov aplikacije.

Visoka realističnost:

Izpopolnjena, realistična upodobitev, ki je videti zelo blizu končnemu izdelku.

Česa maketa NE vključuje

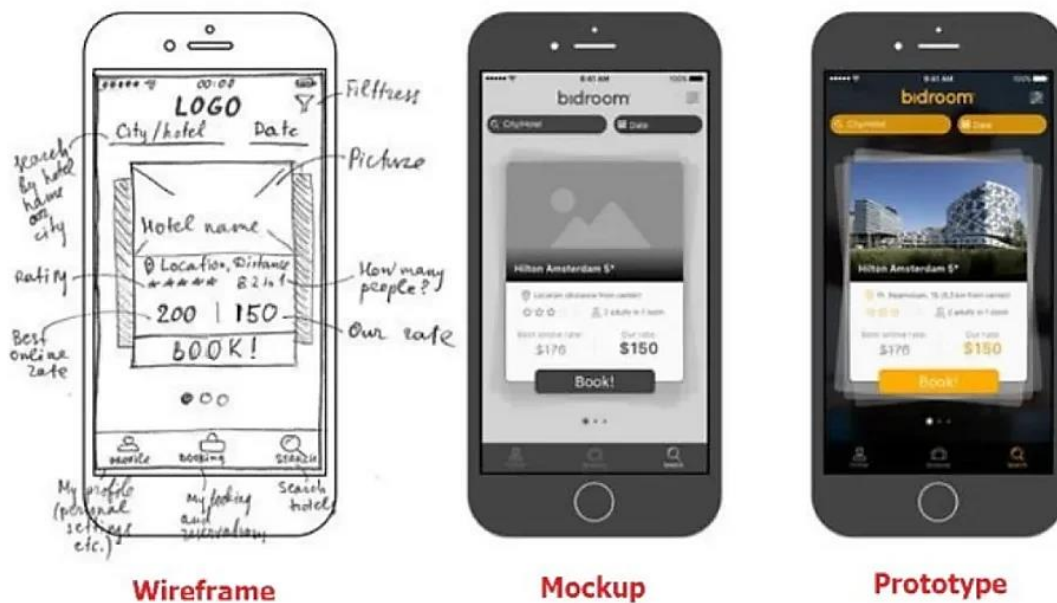
Funkcionalnost: Maketa je statična slika; nanjo ni mogoče klikniti ali z njo komunicirati.

Delujoče povezave: Povezave in animacije niso del makete.

Namen maket je **Vizualizacija idej**. Pomaga preoblikovati abstraktne ideje v oprijemljivo in razumljivo obliko. **Zbiranje povratnih informacij**, to omogoča oblikovalcem, da dobijo povratne informacije o vizualnem videzu od deležnikov in potencialnih uporabnikov. **Ponavljanje oblikovanja** omogoča ekipam, da pred začetkom razvoja prilagodijo in izboljšajo vizualno zasnovo. **Definiranje estetike** - odličen način za usklajevanje vizualne blagovne znamke in celotne estetike izdelka.

Makete v primerjavi s prototipi

- Makete so statične in se osredotočajo na videz in elemente vizualnega oblikovanja.
- Prototipi so interaktivni : prikazujejo, kako bo izdelek deloval, uporabnikom pa omogočajo, da kliknejo nanj in izkusijo potek uporabe.



Slika 2: tri vrste orodij za načrtovanje UI/UX

Pomembno orodje v tej fazi je UML (Unified Modeling Language), ki razvijalcem pomaga pri vizualni predstavitvi kompleksne arhitekture in zasnove. Uporaba diagramov, kot so na primer ti, poenostavi zapletene koncepte, omogoča razmišljanje in sodelovanje ter preprečuje nepotrebno predelavo v kasnejših fazah, saj vsem članom ekipe zagotavlja jasno razumevanje načrta.

Distribucija programske opreme

Distribucija programske opreme in sistemi za upravljanje kode

Distribucija programske opreme pomeni, da se razvite aplikacije, knjižnice ali moduli dostavijo končnim uporabnikom ali drugim razvijalcem. V preteklosti je bila distribucija preprosta – programer je končno datoteko prenesel na disketi, CD-ju ali pozneje na spletni strani. Kasneje so uporabljali sisteme za nadzor nad verzijami CVS (Control Version System). Kasneje pa Danes pa je proces veliko bolj kompleksen, saj programska oprema pogosto nastaja **s sodelovanjem več avtorjev in se nenehno posodablja**.

Razvoj skozi čas:

- 1980–1990: CVS(Control Version System)
- 2000–2010: SVN (Apache Subversion)
- 2010–danes: Git (in platforme, kot so GitHub, GitLab, Bitbucket)

<https://www.geeksforgeeks.org/git/difference-between-git-and-svn/>

<https://rhodecode.com/blog/156/version-control-systems-popularity-in-2025>

Za učinkovito distribucijo uporabljamo **sisteme za nadzor verzij in deljenje kode**, med katerimi je najpogostejše uporabljen **Git**.

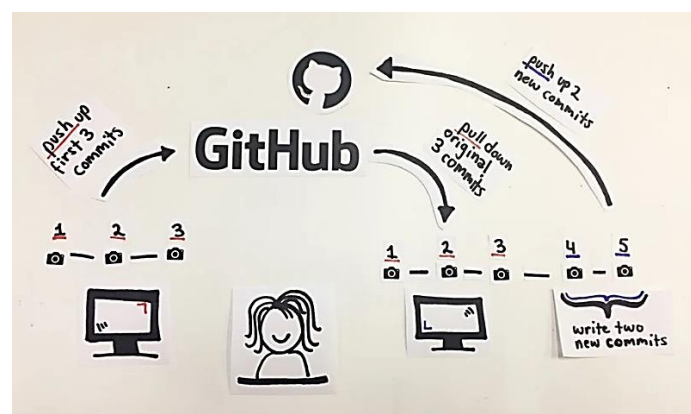
Git je distribuiran sistem za nadzor verzij, ki omogoča sledenje spremembam v kodi, sodelovanje več razvijalcev hkrati in upravljanje z različnimi vejami razvoja. Sam Git je orodje, medtem ko obstajajo različne spletne platforme, ki omogočajo gostovanje Git repozitorijev in lažjo distribucijo programske opreme.

GitHub

GitHub je najbolj znana spletna platforma za gostovanje repozitorijev Git. Poleg samega sistema za nadzor verzij nudi tudi vrsto dodatnih možnosti:

- deljenje kode med razvijalci,
- sledenje hroščem in upravljanje nalog,
- orodja za timsko delo in pregled kode (pull requests),
- avtomatizacijo (GitHub Actions),
- enostavno distribucijo odprtokodnih in zaprtokodnih projektov.

GitHub je postal standard v odprtokodni skupnosti, saj omogoča, da programerji po vsem svetu sodelujejo na skupnih projektih.



Slika 3: delovanje GitHub

Drugi sistemi in platforme

Čeprav je GitHub najpopularnejši, obstajajo tudi druge pomembne platforme, ki ponujajo podobne ali specializirane funkcionalnosti:

- **GitLab** – odprtokodna alternativa GitHubu. Omogoča lastno namestitev na strežnikih organizacije in s tem popoln nadzor nad kodo. Ima integrirano orodje CI/CD (Continuous Integration/Continuous Deployment), kar je posebej uporabno za profesionalno razvojno okolje.
- **Bitbucket** – platforma Atlassian, ki je tesno povezana z orodji, kot sta Jira in Trello. Podpira Git in Mercurial. Pogosto se uporablja v poslovnem okolju zaradi integracije s sistemi za vodenje projektov.
- **SourceForge** – ena najstarejših platform za distribucijo odprtokodne programske opreme. Danes je manj priljubljena kot GitHub, a še vedno pomembna v določenih skupnostih.
- **Azure DevOps Repos** – del Microsoftovega paketa za razvojno okolje. Ponuja gostovanje Git repozitorijev in tesno povezavo z drugimi orodji za vodenje razvojnega cikla.
- **Apache Allura** – odprtokodna platforma, ki jo uporablja Apache Software Foundation za gostovanje svojih projektov.

Vse o GitHub:

- <https://docs.github.com/en/get-started/start-your-journey/about-github-and-git>
- <https://www.youtube.com/watch?v=a9u2yZvsqHA>
- <https://docs.github.com/en/get-started/git-basics>
- <https://docs.github.com/en/get-started>

Primer uporabe GitHub-a

Vprašanje: Imamo predavanja za dijake in bi jim radi delili gradivo in vaje. Imamo različne predmete. Uporabljamo GitHub. Kako je najbolje? En repozitorij ali več različnih?

Odgovor: Če gledaš iz pedagoškega vidika in tudi iz praktičnega (za študente in za tvoje delo), imaš dve možnosti – vsaka ima prednosti in slabosti:

1. En repozitorij za vse predmete

Prednosti:

- Vse gradivo je na enem mestu – študentom daš samo eno povezavo.
- Lažje vzdrževanje, če imaš enoten sistem (npr. mapo za vsak predmet).
- Če želiš poudariti enotnost svojega predavanja ali povezave med predmeti, je to lep način.

Slabosti:

- Lahko postane hitro nepregledno, če je predmetov več.
- Če študente zanima le en predmet, bodo morali iskati znotraj map.
- Zgodovina sprememb (commiti, issue-ji) se zmeša med vse predmete.

2. Ločen repozitorij za vsak predmet

Prednosti:

- Večja preglednost – vsak predmet ima svoj repozitorij, svojo zgodovino sprememb, svoje vaje.
- Študentom, ki obiskujejo le en predmet, daš samo tisti repozitorij.
- Možno je nastaviti različno organizacijo: javno/skrito glede na predmet.
- Lažje je kasneje arhivirati predmet, ko se letnik zaključi.

Slabosti:

- Imaš več repozitorijev za vzdrževati.
- Če je kakšno skupno gradivo (npr. pravila, osnovni vodič), ga moraš podvajati ali pa narediti še poseben “skupni” repozitorij.


Tradicionalen nasvet (ki se sklada z univerzitetno prakso):

- **Za vsako disciplino oz. predmet svoj repozitorij.**
To je preglednejše za študente in bolj naravno za arhiviranje – vsako leto lahko npr. predmet skopiraš ali ustvariš “branch” za novo generacijo.
- Če imaš skupne materiale (npr. splošna pravila za seminarske naloge, navodila za uporabo GitHuba, osnove programiranja, ki veljajo pri več predmetih), naredi en ločen repozitorij z naslovom npr. *Skupni viri* in ga poveži iz vseh ostalih.

Kako to naredimo na GitHub-u

Registracijo (prijavo) sem opravil predhodno.

Quick setup — if you've done this kind of thing before


 Set up in Desktop

 or

HTTPS


SSH

https://github.com/simonpogo/skupni.git




Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line



```
echo "# skupni" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/simonpogo/skupni.git
git push -u origin main
```

...or push an existing repository from the command line



```
git remote add origin https://github.com/simonpogo/skupni.git
git branch -M main
git push -u origin main
```

Načrtovanje programov s pomočjo algoritmov

Danes so algoritmi nekoliko zapostavljeni, ker imamo sodobne programske jezike, orodja in knjižnice, a **klasično načrtovanje programov z algoritmi** je temelj računalništva in še vedno dragoceno za razumevanje logike programiranja.

Načrtovanje programov s pomočjo algoritmov

1. Opredelitev problema

Najprej se jasno določi, kaj želimo rešiti. Problem mora biti zapisan v razumljivi obliki – npr. »Izračunati povprečje ocen študenta«.

2. Analiza problema

Problem se razdeli na osnovne korake in podatkovne enote. Vprašamo se: kakšne podatke potrebujemo, kako jih bomo shranili, kateri postopki bodo uporabljeni in kakšni so pogoji ali omejitve.

3. Sestava algoritma

Algoritem je **zaporedje končno definiranih korakov**, ki vodijo od začetnega stanja do rešitve problema. Algoritem mora biti jasen, enoličen, končen in učinkovit.

Algoritmi se lahko predstavijo:

- v **psevdokodi** (bližnji jeziku programiranja, a razumljiv človeku),
- z **blok diagrami** (vizualni prikaz korakov),
- ali v **naravnem jeziku** (opis postopka v besedah).

4. Testiranje algoritma

Preden ga prevedemo v dejanski program, ga preverimo na papirju ali v mislih – ali res deluje za vse primere (tudi robne primere)?

5. Implementacija algoritma

Ko smo prepričani, da je algoritem pravilen, ga pretvorimo v konkreten programski jezik (C#, Python, Java ...).

6. Vrednotenje in optimizacija

Ko program deluje, se vprašamo, ali bi ga lahko zapisali bolj učinkovito – z manj koraki, manj spomina ali hitrejšim izvajanjem.

Primer: Izračun povprečja ocen študenta

Naravni jezik (opis algoritma)

1. Preberemo vse ocene.
2. Seštejemo vse ocene.
3. Vsoto delimo s številom ocen.
4. Izpišemo rezultat.

Algoritem Povprecje

Vhod: seznam ocen

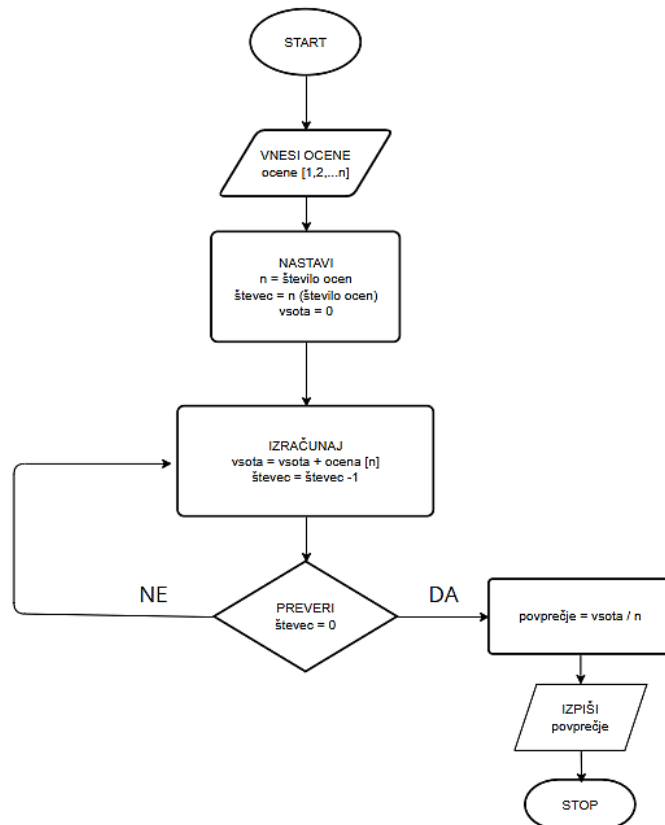
Izhod: povprecje

koraki:

```
vsota ← 0  
n ← dolžina(ocene)  
za vsako oceno v seznamu:  
    vsota ← vsota + ocena  
povprecje ← vsota / n  
izpiši povprecje
```

Konec

Slika 4: psevdokoda v yaml



Slika 5: algoritem zapisan z blokvnim diagramom

Implementacija in razvoj kode

Faza implementacije je tehnična faza, **kjer programerji pišejo kodo na podlagi podrobnih načrtov** in specifikacij projekta. V agilnih okoljih ta faza poteka iterativno v kratkih časovnih obdobjih. Kljub podrobni dokumentaciji je v tej fazi ključno nenehno sodelovanje, saj lahko razvijalci opazijo napake ali potrebe po spremembah, ki jih je treba preveriti z drugimi člani ekipe.

Testiranje in zagotavljanje kakovosti (QA)

Testiranje je bistveni del procesa razvoja, ki potrjuje, da programska oprema nima napak in izpolnjuje vse zahteve. Služi kot ključni del celostnega zagotavljanja kakovosti (QA).

Napake, ki se odkrijejo kasneje nas stanejo veliko več kot napake, ki se odkrijejo prej (pravilo potence števila 10).

Testiranje se deli na dve glavni kategoriji:

- **Funkcionalno testiranje:** Preverja, ali programska oprema deluje v skladu s specifikacijami. Vključuje **testiranje enot** (posameznih komponent), **integracijsko testiranje** (preverjanje delovanja komponent, ko so združene), **sistemsko testiranje** (testiranje celotnega sistema) in **testiranje uporabniškega sprejema (UAT)**, ki ga izvajajo končni uporabniki.
- **Nefunkcionalno testiranje:** Preverja pomembne elemente, ki niso povezani s funkcionalnostjo, kot so **testiranje zmogljivosti** (odzivnost pod obremenitvijo), **varnostno testiranje** (iskanje ranljivosti) in **testiranje združljivosti** (delovanje v različnih okoljih in na različnih napravah). Testiranje zmogljivosti je še posebej pomembno, saj pomaga prepoznati ozka grla in zagotavlja, da aplikacija izpolnjuje pričakovanja uporabnikov glede hitrosti in odzivnosti.

Naraščajoča vloga umetne inteligence bo ključno vplivala na to področje. Napovedi kažejo, da bodo programska orodja do leta 2028 lahko samodejno napisala okoli 70% testov programske opreme. To ne zmanjšuje, ampak povečuje potrebo po programerjih in QA strokovnjakih, saj jim omogoča, da se osredotočijo na kompleksnejše naloge, ki zahtevajo globoko strokovno znanje in človeško presojo.

Najpogostejša orodja za razvoj programske opreme

Pri razvoju programske opreme uporabljamo različna orodja, ki pomagajo pri pisanju, testiranju, uvajanju in vzdrževanju kode. Spodaj so naštetе glavne skupine orodij s kratkimi opisi.

Sistemi za nadzor različic (Version Control System)

Sistemi za nadzor različic, kot je **Git (GitHub)**, so postali standard za upravljanje sprememb kode in omogočanje sodelovanja med razvijalci. Omogočajo, da več ljudi hkrati dela na isti kodi, sledijo spremembam in preprečujejo izgubo dela. Omogočajo spremljanje vsake spremembe, kdo jo je naredil in kdaj, kar je ključnega pomena pri delu v ekipi.

Git ni samo orodje za sodelovanje, ampak tudi močno orodje za razhroščevanje (debugging) in iskanje napak. Funkcija `git blame`, na primer, prikaže, kateri avtor in katera potrditev (commit) sta nazadnje spremenila posamezno vrstico kode. To omogoča hitro prepoznavanje vzrokov za napake in olajša popravke. Uporaba Gitu je bistvena za ohranjanje urejenosti in transparentnosti vseh razvojnih procesov.

- **Git** – najbolj razširjen sistem za sledenje različicam kode.
- **GitHub, GitLab, Bitbucket** – spletne storitve, ki gostijo Git repozitorije in dodajo funkcije, kot so pregled kode, sledenje napakam in avtomatizacija (CI/CD).

Razvojna okolja (IDE – Integrated Development Environment)

Glavno orodje razvijalca, ki združuje urejevalnik kode, orodja za gradnjo, testiranje in dodatke.

- **Visual Studio Code** – lahek, hiter in zelo prilagodljiv urejevalnik.
- **IntelliJ IDEA, PyCharm, WebStorm, PhpStorm** – zmogljiva okolja, namenjena posameznim jezikom (Java, Python, JavaScript, PHP).
- **Visual Studio** – najprimernejši za razvoj v .NET in C++.
- **Eclipse, NetBeans** – pogosto uporabljena pri razvoju v Javi.

Sledenje napakam in vodenje projektov

Pomagajo organizirati delo ekipe, spremljati naloge in odpravljati napake v kodi.

- **Jira** – standard v večjih podjetjih za agilno vodenje projektov.
- **Trello, Kanban boards** – enostavnejša orodja s pregledom nalog v obliki kartic.
- **Asana, Monday.com** – splošna orodja za projekte, prilagodljiva tudi za razvoj.
- **GitHub/GitLab Issues** – vgrajeno sledenje napakam in nalogam znotraj teh platform.

Stalna integracija in dostava programske opreme (CI/CD)

Avtomatizirajo gradnjo kode, testiranje in namestitve aplikacij. Tako se napake odkrijejo hitreje, posodobitve pa pridejo do uporabnikov brez ročnega dela.

- **Jenkins** – odprtokodno orodje z ogromno možnostmi prilagajanja.
- **GitLab CI/CD, GitHub Actions** – avtomatizacija, neposredno vključena v platforme.
- **CircleCI, Travis CI** – oblačne storitve, ki omogočajo hitro nastavitve CI/CD procesov.

Testiranje

Zagotavljajo, da programska oprema deluje pravilno in brez napak.

- **Selenium** – omogoča avtomatsko testiranje spletnih aplikacij v brskalniku.
- **JUnit, pytest, Jest** – okvirji za testiranje posameznih delov kode (Java, Python, JavaScript).
- **Cypress, Playwright** – moderna orodja za testiranje celotnih spletnih aplikacij (od začetka do konca).
- **Postman, Insomnia** – za preverjanje delovanja API-jev.

Upravljanje odvisnosti in paketov

Olajšajo uporabo zunanjih knjižnic in paketov, ki jih projekt potrebuje.

- **npm, yarn** – za JavaScript in Node.js.
- **Maven, Gradle** – za Javo.
- **pip, Conda** – za Python.
- **NuGet** – za .NET.
- **Docker** – omogoča pakiranje vseh odvisnosti in aplikacije v enoten “kontejner”, ki deluje povsod enako.

Komunikacija in sodelovanje

Skrbijo za učinkovito komunikacijo in skupinsko delo.

- **Slack, Microsoft Teams** – pogovori, deljenje datotek in integracije z drugimi orodji.
- **Discord** – pogosto uporabljen v odprtokodnih in študijskih skupnostih.
- **Confluence** – za ustvarjanje in organizacijo tehnične dokumentacije.

Varnost

Preverjajo kodo in odvisnosti za morebitne ranljivosti.

- **SonarQube, Checkmarx** – pregled izvirne kode za varnostne napake.
- **Snyk, Dependabot** – obveščajo o nevarnih odvisnostih in pomagajo pri posodobitvah.

Uvajanje in razporejanje (Deployment & Orchestration)

Skrbijo, da aplikacija deluje v različnih okoljih (razvoj, test, produkcija) in da jo je mogoče enostavno razširiti.

- **Docker** – pakiranje aplikacij v kontejnarje za enotno delovanje.
- **Kubernetes (K8s)** – samodejno uvajanje, skaliranje in upravljanje aplikacij v oblaku.
- **Terraform, Ansible** – avtomatizacija nastavitve strežnikov in omrežij (infrastruktura kot koda).

Vzdrževanje programske opreme

Vzdrževanje ni le zadnja faza v življenjskem ciklu, temveč neprekinjen proces, ki je ključen za dolgoročno kakovost, uporabnost in vrednost programske opreme.

Poznamo štiri glavne vrste vzdrževanja :

- **Popravno (Corrective):** Vključuje odpravljanje napak, hroščev in drugih pomanjkljivosti, ki jih odkrijejo uporabniki ali notranje ekipe. Ta oblika vzdrževanja je nujna za zagotavljanje pravilnega delovanja programske opreme.
- **Prilagoditveno (Adaptive):** Nanaša se na spreminjanje programske opreme, da se prilagodi spremembam v zunanjem okolju. To lahko vključuje posodobitve operacijskih sistemov, spremembe v zakonodaji ali nadgradnje strojne opreme.
- **Izpopolnjevalno (Perfective):** Osredotoča se na izboljšanje funkcionalnosti, zmogljivosti in lažjega vzdrževanja programske opreme po izdaji. Vključuje lahko dodajanje novih funkcij, izboljšanje uporabniške izkušnje ali optimizacijo kode (refaktoring).
- **Preventivno (Preventive):** Je proaktivno vzdrževanje, ki vključuje spremembe programske opreme, da se preprečijo prihodnje težave. Z odpravljanjem manjših, skritih napak, ki bi lahko v prihodnosti postale resne, se zagotavlja dolgoročna stabilnost sistema.

Vzdrževanje predstavlja pomemben del celotnega življenjskega cikla izdelka in je pogosto povezano z visokimi stroški, ki pa so nujni za ohranjanje relevantnosti in zanesljivosti programske opreme.

Povzetek

Analiza procesa razvoja programske opreme potrjuje, da gre za dinamično in nenehno razvijajoče se področje. Življenjski cikel razvoja (SDLC) ostaja temeljen okvir, ki zagotavlja strukturo, medtem ko agilne metodologije, zlasti Scrum in Kanban, predstavljajo sodobno filozofijo, ki daje prednost hitrosti, prilagodljivosti in nenehnemu dostavljanju vrednosti. Uspešen razvoj je rezultat sinergije med specializiranimi vlogami – od poslovnih analitikov, ki razumejo potrebe, do razvijalcev in strokovnjakov za QA, ki zagotavljajo kakovost in zanesljivost. Največji preboj v zadnjih letih omogočajo orodja in procesi avtomatizacije, kot je CI/CD, ki so ključni za uresničevanje agilnih načel in omogočajo hitro, zanesljivo in pogosto izdajanje programske opreme.

Priporočila za izbiro pravega pristopa

Izbira metodologije mora biti strateška odločitev, ki se uskladi z naravo projekta in kulturo organizacije. Za projekte z jasnimi, stabilnimi zahtevami in fiksnimi proračuni je tradicionalni slapni model še vedno primeren, saj omogoča natančno načrtovanje in nadzor. Vendar je za večino sodobnih programskih projektov, kjer so zahteve nejasne, tržne spremembe pogoste in je pričakovanje strank visoko, agilni pristop nujen. Hibridni modeli ponujajo prilagodljivost in so lahko primeren prehod za organizacije, ki se postopoma premikajo k agilnim praksam.

Prihodnost razvoja programske opreme

Prihodnost razvoja je tesno povezana z avtomatizacijo in umetno inteligenco. Pričakuje se, da bo umetna inteligenca prevzela velik del rutinskega dela, kot je ustvarjanje in izvajanje testov, kar bo povečalo učinkovitost in zmanjšalo človeške napake.¹ To ne bo zmanjšalo potrebe po strokovnjakih, ampak bo preusmerilo njihovo delo na kompleksnejše in strateške naloge, kot so načrtovanje, oblikovanje in reševanje zapletenih problemov. To bo povečalo povpraševanje po strokovnjakih, ki so sposobni premostiti vrzeli med različnimi področji, kot so posel, oblikovanje in inženiring, in ki bodo sposobni voditi ekipe v nenehno spreminjajočem se digitalnem okolju.

Vrste moderne programske opreme

Sodobna programska oprema se neprestano razvija in prilagaja potrebam uporabnikov. Od klasičnih samostojnih programov do kompleksnih oblačnih rešitev je skupni cilj vedno enak – olajšati delo in izboljšati uporabniško izkušnjo.

Razumevanje različnih vrst aplikacij je ključno za razvijalce, podjetja in tudi uporabnike, saj vsak tip ponuja specifične prednosti in omejitve. V prihodnosti lahko pričakujemo še več povezovanja med vrstami programske opreme, še posebej na področju oblačnih storitev, mobilnih naprav in umetne inteligence.

Samostojne (stand-alone) aplikacije

Samostojne aplikacije so programi, ki se namestijo in uporabljajo neposredno na računalniku, brez potrebe po stalni povezavi z omrežjem.

Značilnosti:

- Tečejo lokalno na uporabnikovem računalniku.
- Običajno ne zahtevajo internetne povezave.
- Odvisne so od operacijskega sistema, na katerem tečejo.

Primeri: Microsoft Office (Word, Excel), Adobe Photoshop, predvajalniki glasbe ali videa programi in orodja za delo z datotekami, razna orodja, operacijski sistem...

Prednosti: hitrost, neodvisnost od povezave, stabilnost.

Slabosti: omejen dostop do podatkov na drugih napravah, odvisnost od operacijskega sistema, težje posodabljanje.

Omrežne (client-server) aplikacije

Gre za programe, ki so razdeljeni na dva dela: **odjemalca (client)** in **strežnik (server)**. Odjemalec pošilja zahteve, strežnik pa jih obdela in vrne rezultate. Večinoma se na strežniku nahaja podatkovna baza, ki hrani vse podatke.

Značilnosti:

- Temeljijo na komunikaciji prek omrežja, omrežje je za delovanje nujno.
- Uporabljajo se predvsem v podjetjih in večjih sistemih.
- Omogočajo sočasno delo več uporabnikom.

Primeri: bančni informacijski sistemi, sistemi za rezervacije letalskih kart, univerzitetni informacijski sistemi...

Prednosti: centralizirano shranjevanje podatkov, lažje vzdrževanje, varovanje podatkov in nadzor nad dostopom

Slabosti: odvisnost od strežnika in omrežja, potreba po zmogljivi infrastrukturi.

Spletne aplikacije

Spletne aplikacije delujejo znotraj spletnega brskalnika in ne zahtevajo namestitve na uporabnikovem računalniku.

Značilnosti:

- Dostopne prek interneta, običajno na različnih napravah.
- Redno se posodablja na strežniku, uporabnik sam ne skrbi za namestitve.

- Pogosto uporabljajo sodobne tehnologije, kot so HTML, CSS, JavaScript ter različni okvirji (React, Angular).

Primeri: Gmail, Google Docs, Office 365, spletne banke, e-trgovine (Amazon, eBay).

Prednosti: dostopnost z različnih naprav, preprosto vzdrževanje, sodelovanje v realnem času.

Slabosti: odvisnost od internetne povezave, varnostna tveganja, kompatibilnost s spletnimi pregledovalniki (Chrome, Edge, Opera, Firefox...), omejenost na spletne tehnologije (HTML, CSS, JS...).

Mobilne aplikacije

Mobilne aplikacije so posebej razvite za pametne telefone in tablice. Delujejo na platformah, kot sta **Android** in **iOS**.

Značilnosti:

- Namenjene mobilni uporabi, pogosto optimizirane za dotik.
- Povezujejo se s strojno opremo naprave (kamera, GPS, senzorji).
- Razdelimo jih na t.im. »**native**« **aplikacije** (pisane posebej za določen sistem), **hibridne aplikacije (združujejo spletne tehnologije)** in **progresivne spletne aplikacije (PWA)**.

Primeri: WhatsApp, Instagram, Uber, mobilne bančne aplikacije.

Prednosti: enostavna uporaba, stalna povezanost, prilagojenost mobilnim napravam (UI/UX).

Slabosti: odvisnost od platforme (Android, iOS...), potreba po rednem posodabljanju, večja poraba strojnih virov, varnost.

Porazdeljene in oblačne aplikacije

Sodobni trend razvoja so aplikacije, ki niso vezane na en računalnik ali strežnik, ampak delujejo porazdeljeno v oblaku.

Značilnosti:

- Podatki in storitve so shranjeni v oblačni infrastrukturi.
- Omogočajo visoko razpoložljivost in prilagodljivo skaliranje.
- Uporabljajo se pri velikih sistemih, kjer je pomembna hitrost in odzivnost.

Primeri: Dropbox, Google Drive, Microsoft 365, Netflix.

Prednosti: dostopnost od kjerkoli, avtomatske posodobitve, razširljivost.

Slabosti: odvisnost od internetne povezave, vprašanja zasebnosti in varnosti podatkov.

Vgrajena (embedded) programska oprema

To so programi, vgrajeni neposredno v naprave, kjer uporabnik pogosto niti ne ve, da delujejo.

Značilnosti:

- Tesno povezana s strojno opremo.
- Namenjena nadzoru naprav in sistemov.
- Običajno zelo stabilna in optimizirana.

Primeri: programska oprema v avtomobilih, pametnih urah, televizorjih, medicinskih napravah.

Prednosti: zanesljivost, optimizacija za specifične naloge.

Slabosti: omejena možnost nadgradenj, pogosto specifična strojna vezava.

Primerjalna tabela

Vrsta aplikacije	Prednosti	Slabosti	Primeri
Samostojne	Hitrost, stabilnost, delujejo brez interneta	Težko posodabljanje, omejen dostop drugje	Word, Photoshop
Omrežne (<i>client-server</i>)	Centralizirani podatki, lažje vzdrževanje	Odvisnost od strežnika in povezave	Bančni sistemi, univerzitetni portali
Spletne	Dostopne kjerkoli, enostavno vzdrževanje	Odvisnost od interneta, varnostna tveganja	Gmail, Google Docs, e-trgovine
Mobilne	Vedno pri roki, uporaba funkcij naprave	Odvisne od platforme, veliko posodobitev	WhatsApp, Instagram, Uber
Porazdeljene in oblačne	Razširljivost, dostop od kjerkoli	Varnostna vprašanja, odvisnost od interneta	Dropbox, Netflix, Google Drive
Vgrajene (<i>embedded</i>)	Zanesljivost, optimizacija	Omejena nadgradnja, vezane na napravo	Pametne ure, avtomobili, TV

Programski jeziki

Pregled programskih jezikov pri razvoju aplikacij

Razvoj programske opreme je dinamično in neprestano spreminjajoče se področje, pri čemer so programski jeziki temeljni gradniki, ki določajo arhitekturo, zmogljivost in obseg aplikacij.

Izbira pravega jezika je ključna, saj vpliva na hitrost razvoja, prilagodljivost in vzdrževanje končnega produkta. V današnjem svetu obstaja na stotine programskih jezikov, vendar se le peščica uveljavlja kot dominantna orodja v specifičnih sektorjih. Ti jeziki niso zgolj sintaktične strukture, temveč prinašajo s seboj bogate ekosisteme, knjižnice in skupnosti, ki podpirajo njihov razvoj in uporabo.

Ker smo v prejšnjem poglavju razdelili aplikacije glede na način delovanja (samostojna, odjemalec-strežnik, spletna, mobilna...) moramo tudi tukaj omeniti programske jezike, ki so primerni in se uporabljajo za razvoj aplikacij.

Jeziki za namizne in sistemske aplikacije

Razvoj namiznih in sistemskih aplikacij zahteva jezike, ki nudijo moč, zanesljivost, visoko učinkovitost in neposreden dostop do strojne opreme. **C++** in **C#** sta tukaj v ospredju. **C++** je še vedno neprekosljiv na področjih, kot so razvoj iger, visoko zmogljiva programska oprema, vdelani sistemi in gonilniki. Njegova kompleksnost se odtehta z izjemno kontrolo nad pomnilnikom in zmogljivostjo, kar je ključno za aplikacije, kjer je vsak milisekund pomemben.

C#, ki ga je razvil Microsoft, je osnova za okvir **.NET**. Je vsestranski jezik, ki se uporablja za razvoj namiznih aplikacij z okoljem **WPF** ali **Windows Forms**, pa tudi za razvoj iger v pogonu **Unity**, spletnih aplikacij z **ASP.NET** in celo za mobilne aplikacije prek platforme **MAUI**. C# združuje moč in preprostost, kar ga dela priljubljenega v poslovnem okolju.

Jeziki za spletne aplikacije

Spletni razvoj je verjetno najbolj živahen segment programiranja, kjer dominirajo trije ključni jeziki. **JavaScript** je nedvomno kralj spletnega razvoja. Sprva je bil zasnovan za interaktivnost na strani

odjemalca (brskalnika), a se je z razvojem okolja **Node.js** prebil tudi na strežniško stran. Ta dualnost mu omogoča, da se isti jezik uporablja za celotno aplikacijo (tako imenovani "full-stack development"), kar poenostavi delo in zmanjša preklapljanje kontekstov. Njegova prilagodljivost in ogromna zbirka ogrodij, kot so **React**, **Angular** in **Vue.js**, so ga utrdila kot nepogrešljiv del sodobnega spleta.

Za strežniški del aplikacij so poleg JavaScripta priljubljeni še **Python** in **PHP**. Veliko spletnih aplikacij je bilo v preteklosti razvitih tudi z **Microsoft ASP.NET**.

PHP je dolgo veljal za hrbtenico spleta, saj poganja ogromno spletnih strani, vključno z najbolj razširjenim sistemom za upravljanje vsebin, **WordPressom**. Čeprav se je soočal s kritikami glede varnosti in učinkovitosti, je z novejšimi različicami močno napredoval, kar mu omogoča, da ostaja relevanten. **Python**, znan po svoji berljivosti in preprosti sintaksi, se je uveljavil s spletnimi ogrodji, kot sta **Django** in **Flask**. Njegova univerzalnost – od spletnega razvoja do znanosti o podatkih – ga postavlja v edinstven položaj.

Jeziki za razvoj mobilnih aplikacij

Mobilni razvoj je izrazito razdeljen med dva glavna ekosistema: **Android** in **iOS**. Za **Android** je primarni jezik **Kotlin**, ki je v zadnjih letih zamenjal **Java** kot uradno priporočen jezik. Kotlin ponuja sodobno sintakso in izboljšave, ki poenostavljajo razvoj in zmanjšujejo število napak, hkrati pa ohranja polno združljivost z Java virtualnim strojem (JVM). **Java** ostaja pomembna, a se njen pomen na področju mobilnega razvoja zmanjšuje.

Na drugi strani, za **iOS** in celoten Appleov ekosistem, je glavni jezik **Swift**. Zasnovan je kot sodobna, varna in učinkovita alternativa svojemu predhodniku, **Objective-C**. S svojimi funkcijami, kot so avtomatsko upravljanje pomnilnika in močno tipiziranje, omogoča hitrejši in varnejši razvoj aplikacij za iPhone, iPad in Mac.

Jeziki za programiranje AI aplikacij (podatki, umetna inteligenca in avtomatizacija)

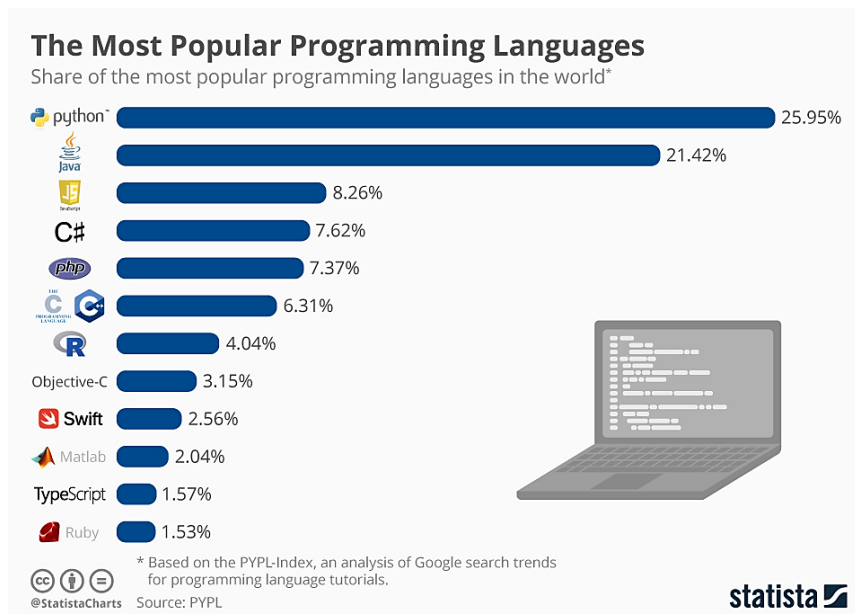
V zadnjih letih je **Python** postal de facto standard na področjih znanosti o podatkih in umetne inteligence. Zaradi bogate zbirke knjižnic, kot so **Numpy**, **Pandas**, **Matplotlib** in **TensorFlow**, je idealen za obdelavo podatkov, strojno učenje in vizualizacijo. Njegova preprosta sintaksa omogoča znanstvenikom in analitikom, da se osredotočijo na problem namesto na zapletenost kode.

Poleg Pythona se na tem področju uveljavljajo tudi **R**, ki je specializiran za statistično analizo, in **Julia**, ki je namenjena visoko zmogljivemu numeričnemu računanju.

Zaključek

Izbira programskega jezika je odvisna od konteksta. Medtem ko se nekateri jeziki, kot sta **JavaScript** in **Python**, lahko pohvalijo z izjemno vsestranskostjo, so drugi specializirani za določena področja, kot so **Swift** za iOS, **Kotlin** za Android ali **C++** za visoko zmogljive aplikacije. Uspešen razvijalec danes ne obvlada le enega jezika, temveč razume, kateri jezik je najprimernejši za reševanje določenega problema. Ta prilagodljivost in razumevanje sta ključna za učinkovit razvoj in ustvarjanje inovativnih rešitev v hitro spreminjajočem se digitalnem svetu.

Najbolj priljubljeni programski jeziki



Slika 6: najbolj popularni programski jeziki

Na gornji sliki so najbolj popularni programski jeziki po podatkih [Statista](#). Jaz jih bom naštel po poljubnem vrstnem redu

1. JavaScript (z ekosistemom Node.js, React, Angular, Vue)

Jezik, ki se prvotno uporabljal za pisanje skript v spletnih brskalnikih, danes pa je **neizogiben za sprednji konec (frontend)** skoraj vsake spletne aplikacije. Z platformo **Node.js** se pogosto uporablja tudi za **zadnji konec (backend)**.

Uporaba: Interaktivne spletne strani, spletne aplikacije, mobilne aplikacije (React Native), desktop aplikacije (Electron).

Zakaj je priljubljen: Je povsod. Deluje v vsakem brskalniku, ima ogromno skupnost in neizmerno število knjižnic (npr. React, Vue.js) in ogrodij.

2. Python

Jezik z naglasom na berljivosti kode in učinkovitosti razvijalcev. Izjemno vsestranski in prijazen za začetnike.

Uporaba: Podatkovna znanost in umetna inteligenca (AI/ML), **avtomatizacija**, znanstveni izračuni, razvoj backend-a (Django, Flask), scripting.

Zakaj je priljubljen: Preprosta sintaksa, ogromna skupnost in izjemno močne knjižnice za specializirana področja (npr. Pandas, NumPy, TensorFlow, Scikit-learn).

3. Java

Zrel, objektno orientiran jezik z motto "enkrat napiši, povsod poganjaj" (WORA) zaradi navidezne strojne naprave Java Virtual Machine (JVM).

Uporaba: Razvoj velikih podjetnih (enterprise) sistemov, **android aplikacije**, bančni sistemi, strežniške aplikacije.

Zakaj je priljubljen: Zelo stabilen, zanesljiv in ima ogromno ekosistem. Zahtevan na trgu dela, zlasti v velikih korporacijah.

4. C#

Močan objektno orientiran jezik razvit pri Microsoftu, prvotno kot konkurent Javi. Deluje na platformi .NET.

Uporaba: Razvoj igr (z uporabo ogrodja Unity), Windows desktop aplikacije, enterprise aplikacije, backend storitve.

Zakaj je priljubljen: Odličen jezik za razvoj na Windows platformi, glavni jezik za razvoj v Unity-ju, ki vladja na trgu indie iger.

5. TypeScript

Nadgradnja JavaScript-a, ki ji doda statično tipkanje in druge napredne funkcije. TS koda se prevede (transpilira) v JavaScript.

Uporaba: Pisanje velikih, kompleksnih spletnih aplikacij, kjer JavaScriptovo dinamično tipkanje postane nepredvidljivo.

Zakaj je priljubljen: Omogoča bolj varen in vzdrževan razvoj velikih aplikacij, hkrati pa je popolnoma združljiv z obstoječim ekosistemom JavaScript.

6. PHP

Skriptni jezik, zasnovan za splet. Kljub starosti je še vedno zelo relevanten.

Uporaba: Razvoj spletnih strani in aplikacij na strani strežnika (backend). Ogrodja kot so Laravel in Symfony so zelo priljubljena.

Zakaj je priljubljen: Poganja ogromen del spleta (vključno z WordPress, Facebookom v začetku, Wikipedia). Široko je razširjen in poceni za gostovanje.

7. Kotlin

Sodoben, jedrnat in varen jezik, ki teče na JVM. Uradno ga priporoča Google za razvoj **android aplikacij**.

Uporaba: Android razvoj, backend razvoj (z Java ogrodji ali Kotlin-specificnimi kot je Ktor).

Zakaj je priljubljen: Rešuje številne težave Jave (npr. preveč verbose sintaksa), je popolnoma interoperabilen z Javo, kar omogoča postopno uvajanje.

8. Swift

Sodoben, zmogljiv in prijazen jezik, ki ga je razvil Apple kot zamenjavo za Objective-C.

Uporaba: Razvoj aplikacij za **ekosistem Apple**: iOS, macOS, watchOS, tvOS.

Zakaj je priljubljen: Hitrejši in varnosti od Objective-C. Edina logična izbira za razvijalce, ki se osredotočajo na Appleove platforme.

9. Go (Golang)

Jezik, ki so ga razvili pri Googleu. Znan po svoji **enostavnosti, hitrosti in učinkovitem sočasnem programiranju**.

Uporaba: Sistemsko programiranje, cloud storitve, mikrostoritve, CLI orodja, visoko obremenjeni strežniki (npr. Docker, Kubernetes so napisani v Go-ju).

Zakaj je priljubljen: Hitra prevajanja, enostavna sintaksa, odlična podpora za sočasnost. Idealno za moderne, razpršene arhitekture.

10. Rust

Jezik, ki se osredotoča na **hitrost in varnost pomnilnika**, brez odlaganja smeti (garbage collection). Težko se ga naučiti, a zelo močan.

Uporaba: Sistemsko programiranje, razvoj operacijskih sistemov, pogonski programi, igralniki, kjer so zmogljivost in varnost kritične.

Zakaj je priljubljen: Rešuje probleme, ki so prisotni v jezikih kot sta C/C++ (npr. napake pri delu s pomnilnikom). Po mnenju razvijalcev je "najljubši" jezik že več let zapored (Stack Overflow Survey).

Pogosto je kombinacija jezika in ogrodja tista, ki določa priljubljenost:

- **Frontend Ogrodja:** React.js (JavaScript), Vue.js (JavaScript), Angular (TypeScript)
- **Backend Ogrodja:** Node.js (JavaScript), Spring (Java), Django/Flask (Python), Laravel (PHP), .NET (C#)
- **Mobile:** React Native (JavaScript), Flutter (Dart), Kotlin (Android), Swift (iOS)

Priljubljeni jeziki, ki podpirajo OOP

- Java
- C++
- Python
- C#
- PHP (od verzije 5 naprej)
- JavaScript (s prototipnim dedovanjem)
- Ruby

Povezave do vsebin

<https://playbackpress.com/books/pybook>

<https://www.freecodecamp.org/news/tag/python/>

Prevajanje in/ali tolmačenje - Kako pridemo do izvršne kode?

Tolmačenje in prevajanja Programske Opreme

V svetu razvoja programske opreme sta tolmačenje (interpretation) in prevajanje (compilation) dva temeljna procesa, ki omogočata pretvorbo človeku razumljive izvirne kode v obliko, ki jo lahko neposredno izvrši CPU (izvršna koda). Čeprav oba procesa dosežeta isti končni cilj – izvajanje programa – se njuna pot do tja bistveno razlikuje, kar prinaša različne prednosti in slabosti.

https://youtu.be/d7Qs-zHzQhc?si=jE7vZ6oKONrIBK_w&t=44

Prevajanje (Compilation)

Prevajanje je proces, pri katerem poseben program, imenovan **prevajalnik (compiler)**, v celoti analizira in pretvori izvirno kodo, napisano v višjem programskem jeziku (npr. C++, Java, Go), v strojno kodo, ki je neposredno izvršljiva s strani procesorja. Ta proces poteka pred samim zagonom programa in ga lahko razdelimo na več ključnih faz:

1. **Leksična Analiza (Lexical Analysis):** V tej prvi fazi prevajalnik prebere izvirno kodo kot zaporedje znakov in jo razdeli na osnovne gradnike, imenovane leksikalne enote ali *tokeni*. To so ključne besede (npr. if, while), identifikatorji (imena spremenljivk in funkcij), operatorji (+, -, *) in ločila.
2. **Sintaktična Analiza (Syntax Analysis):** Prevajalnik nato preveri, ali zaporedje tokenov ustreza slovničnim pravilom (sintaksi) programskega jezika. To fazo si lahko predstavljamo kot preverjanje, ali je "stavek" v programskem jeziku pravilno zgrajen. Rezultat je običajno drevesna struktura, imenovana sintaktično drevo.
3. **Semantična Analiza (Semantic Analysis):** V tej fazi se preverja pomen kode. Prevajalnik preveri, ali so spremenljivke pravilno deklarirane, ali se tipi podatkov ujemajo pri operacijah (npr. ali poskušamo sešteti število in besedilo) in ali so klici funkcij pravilni.
4. **Generiranje Vmesne Kode (Intermediate Code Generation):** Po uspešni analizi prevajalnik ustvari vmesno predstavitev programa, ki je neodvisna od specifične arhitekture procesorja. Ta koda je lažja za optimizacijo.
5. **Optimizacija Kode (Code Optimization):** Ta faza je ključna za učinkovitost končnega programa. Prevajalnik poskuša izboljšati vmesno kodo tako, da odstrani nepotrebne operacije, preuredi ukaze za hitrejšo izvajanje in zmanjša porabo pomnilnika.
6. **Generiranje Strojne Kode (Code Generation):** Na koncu prevajalnik prevede optimizirano vmesno kodo v strojno kodo, ki je specifična za ciljno arhitekturo procesorja (npr. x86, ARM). Rezultat tega procesa je izvršljiva datoteka (npr. .exe v sistemu Windows), ki jo lahko uporabnik zažene.

Glavne Značilnosti Prevajanja:

- Celotna koda se prevede naenkrat.
- Ustvari se samostojna izvršljiva datoteka.
- Hitrejšo izvajanje programa, saj je koda že prevedena v strojni jezik.
- Odkrivanje napak pred izvajanjem, kar pripomore k večji zanesljivosti.
- Manjša prenosljivost, saj je izvršljiva datoteka vezana na specifičen operacijski sistem in arhitekturo.

Primeri jezikov, ki se prevajajo: **C, C++, C#, Go, Rust.**

Tolmačenje (Interpretation): Sproten Pristop Med Izvajanjem

Tolmačenje je proces, kjer program, imenovan **tolmač (interpreter)**, bere izvorno kodo vrstico za vrstico, vsako vrstico sproti prevede v strojno kodo (ali vmesno obliko) in jo takoj zatem tudi izvede. Za razliko od prevajalnika, tolmač ne ustvari ločene izvršljive datoteke.

Proces tolmačenja je v osnovi cikel, ki se ponavlja za vsako vrstico ali ukaz v izvorni kodi:

1. **Branje:** Tolmač prebere naslednjo vrstico ali ukaz iz izvorne kode.
2. **Analiza:** Podobno kot prevajalnik, tudi tolmač izvede leksikalno, sintaktično in semantično analizo prebrane vrstice, da preveri njeno pravilnost.
3. **Prevod in Izvedba:** Če je vrstica pravilna, jo tolmač prevede v strojne ukaze in jih takoj posreduje procesorju v izvedbo. Ta prevod se ne shrani za kasnejšo uporabo.

Glavne značilnosti tolmačenja:

- Koda se izvaja vrstico za vrstico.
- Ne ustvari se ločena izvršljiva datoteka.
- Počasnejše izvajanje programa, saj proces prevajanja poteka med samim izvajanjem.
- Lažje in hitrejšo odpravljanje napak, saj tolmač takoj opozori na napako in njeno lokacijo.
- Večja prenosljivost, saj se ista izvorna koda lahko izvaja na kateremkoli sistemu, ki ima nameščen ustrezen tolmač.

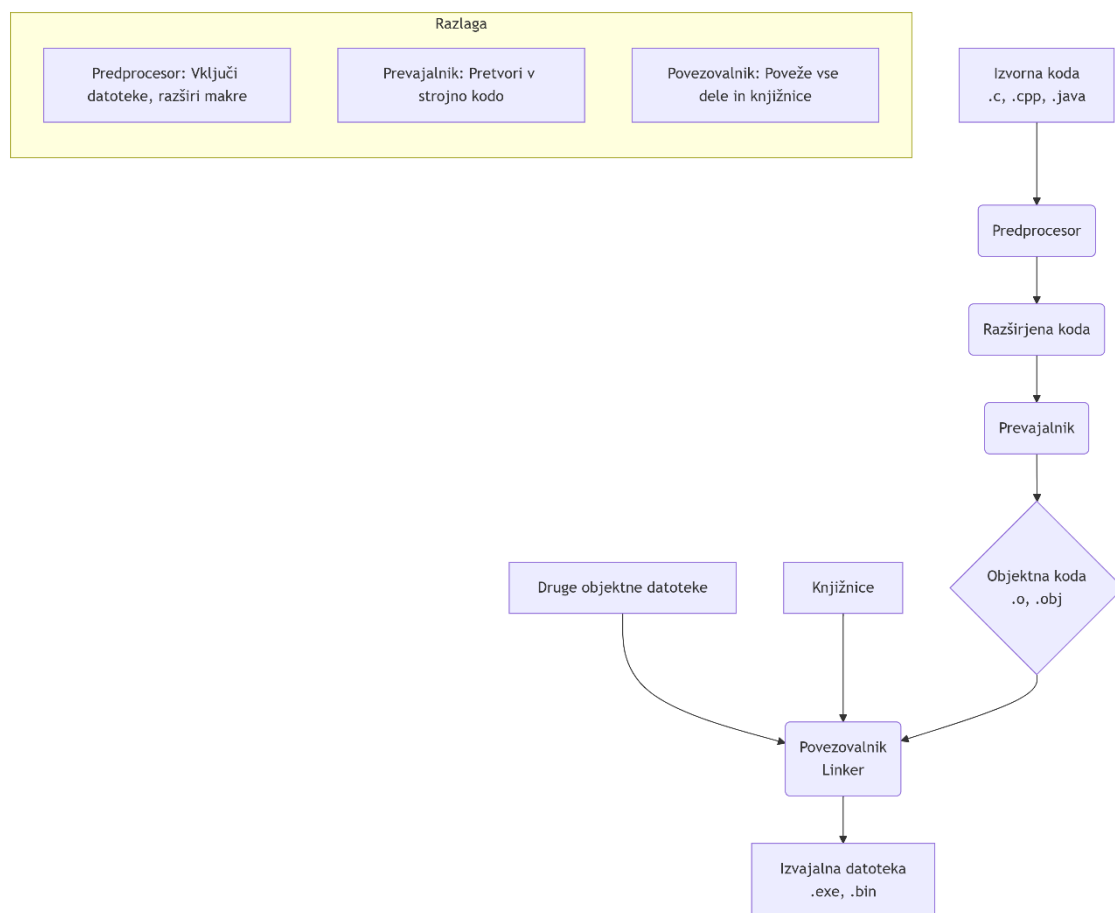
Primeri tolmačenih jezikov: Python, JavaScript, Ruby, PHP.

Hibridni Pristopi

Pomembno je omeniti, da meja med prevajanjem in tolmačenjem ni vedno ostra. Številni sodobni programski jeziki uporabljajo hibridni pristop. Tipičen primer je **Java**, ki se najprej prevede v vmesno kodo, imenovano *bytecode*. Ta bytecode ni neposredno izvršljiv s strani procesorja, ampak ga izvaja **Javin navidezni stroj (JVM)**, ki deluje kot tolmač za to vmesno kodo. Podoben pristop uporablja tudi **Python**, ki izvorno kodo najprej prevede v bytecode (.pyc datoteke), ki ga nato izvaja Pythonov navidezni stroj. Ta pristop združuje prenosljivost tolmačenja in določene optimizacije, ki jih omogoča predhodno prevajanje.

Primerjalna Tabela

Značilnost	Prevajalnik (Compiler)	Tolmač (Interpreter)
Kdaj poteka prevod?	Pred izvajanjem programa, v celoti.	Med izvajanjem programa, vrstico po vrstici.
Izhodni produkt	Samostojna izvršljiva datoteka.	Brez ločene izvršljive datoteke.
Hitrost izvajanja	Hitrejša.	Počasnejša.
Odkrivanje napak	Vse napake se odkrijejo pred zagonom.	Napake se odkrivajo sproti med izvajanjem.
Prenosljivost	Manjša (odvisna od platforme).	Večja (neodvisna od platforme).
Poraba pomnilnika	Večja (celoten program je v pomnilniku).	Manjša (le del kode je naenkrat v pomnilniku).
Primeri jezikov	C, C++, Go, Rust	Python, JavaScript, Ruby, PHP



Objektno orientirano programiranje (OOP) (s poudarkom na C#, Pythonu)

Kaj je objektno orientirano programiranje (OOP)?

Objektno orientirano programiranje je **način programiranja**, ki se je pojavil v 60. in 70. letih 20. stoletja kot odgovor na težave, ki so nastajale pri pisanju vse bolj kompleksnih programov. Prej so prevladovali **proceduralni jeziki** (npr. C, Pascal, Fortran), kjer je bil program sestavljen predvsem iz **funkcij in podatkovnih struktur**.

Težave:

- podatki in funkcije so bili ločeni,
- kompleksni sistemi so postajali nepregledni,
- težko je bilo ponovno uporabiti kodo.

Rešitev je bila ideja, da združimo **podatke in vedenje (funkcije) v objekte**.

Prve jezike z OOP pristopom (npr. **Simula** in **Smalltalk**) so razvili kot način, da bi programska logika posnemala **resnični svet**. Če imamo v resnici avtomobile, živali, osebe, zakaj ne bi v programu imeli »objektov«, ki predstavljajo te pojme?

Python je bil razvit kasneje (v začetku 90-ih) in že od samega začetka vsebuje močno podporo za OOP, hkrati pa omogoča tudi proceduralno in funkcijsko programiranje. Prav ta prilagodljivost je razlog, da je Python postal eden najbolj priljubljenih jezikov na svetu.

Osnovna načela OOP

Vsa OOP teorija se vrti okoli **štirih glavnih principov**. Če jih razumemo, razumemo jedro objektnega pristopa.

Abstrakcija

je načelo skrivanja notranjih podrobnosti objekta in zagotavljanja kontrole nad dostopom. Podatki so "zaprti" znotraj objekta, zunanji svet pa lahko komunicira z njim le prek jasno definiranih vmesnikov (metod). Abstrakcija pomeni, da od razreda oziroma objekta pokažemo samo tisto, kar je za uporabnika pomembno, skrijemo pa nepotrebne podrobnosti.

Primer:

če imamo razred Avto, nas kot uporabnika zanima metoda **vozi()**, ne pa podrobnosti, kako motor izgoreva gorivo. Ne vemo, kako natančno deluje motor, menjalnik ali sistem zaviranja (skrite podrobnosti). Komuniciramo z avtomobilom prek vmesnikov: **volana** (za smer), **pedala za plin** (za pospešek) in **zavore** (za ustavljanje). Če proizvajalec spremeni notranji del motorja, se vaš način vožnje ne spremeni. Abstrakcija nam omogoča, da razmišljamo na **višjem nivoju**.

Inkapsulacija

Inkapsulacija pomeni, da **združimo podatke in metode v eno enoto (razred)** ter omejimo dostop do notranjih podrobnosti.

- To preprečuje, da bi nekdo od zunaj spremenil občutljive podatke.
- V Pythonu nimamo popolnega sistema »private« in »public«, a smo se dogovorili, da:
 - `_ime` pomeni, da je spremenljivka notranja,
 - `__ime` pomeni močnejše skrita spremenljivka (Python jo preimenuje interno, da se zmanjša možnost zlorabe).

Dedovanje

Kaj je dedovanje? Dedovanje je sposobnost enega razreda (podrazreda/otroka), da podeduje lastnosti in metode drugega razreda (nadrazreda/starša). To omogoča ustvarjanje hierarhije razredov in ponovno uporabo kode. S tem se izognemo ponavljanju kode.

Dedovanje omogoča, da **nov razred prevzame lastnosti obstoječega razreda**.

Primer:

Vozilo je starševski razred, iz njega pa lahko dedujejo bolj specifična vozila **Avto, Ladja, Avion, Kolo**. Tako Avtomobil podeduje splošne lastnosti Vozila (npr. hitrost, barva, metoda `premikanje()`), in doda svoje specifične lastnosti (npr. število vrat, metoda `prižgiRadio()` ipd).

Izogremo se podvajanju kode. Skupno logiko napišemo v osnovni razred, specializirano logiko pa v podrazrede.

Polimorfizem

Kaj je Polimorfizem? Sposobnost objekta, da prevzame različne oblike. Polimorfizem (iz grščine: »več oblik«) pomeni, da lahko objekti različnih razredov uporabljajo isto metodo, a jo implementirajo vsak po svoje. Omogoča pisanje splošne in fleksibilne kode. Koda, ki dela z nad-razredom, bo avtomatsko delala tudi s katerimkoli njegovim pod-razredom.

Primer:

metoda `premakni()` je skupna za vsako Vozilo, a avto se premika po cesti, ladja po vodi, avion po zraku. Tako lahko z eno samo kodo upravljamo različne tipe objektov.

Primer:

Metoda `prikaziSliko()`. Če je klicana za objekt tipa PDF, bo prikazala stran dokumenta. Če je klicana za objekt tipa Slika, bo prikazala sliko. Ime metode je enako, a se njena notranjost obnaša drugače glede na kontekst.

Objektno orientirano programiranje je danes eden izmed **najbolj razširjenih pristopov k pisanju programske kode**. Python ga izvaja na pregleden in enostaven način, zaradi česar je primeren tudi za začetnike.

Če bi OOP primerjali z gradnjo hiše, bi pomenilo:

- **razred** je načrt,
- **objekt** je zgrajena hiša po tem načrtu,
- **metode** so funkcije hiše (odpri vrata, prižgi luč),
- **atributi** so lastnosti hiše (barva, površina, število nadstropij).

Prednosti OOP

- **Naravna povezanost z resničnim svetom** → razredi predstavljajo pojme (oseba, vozilo, račun).
- **Ponovna uporaba kode** → dedovanje omogoča, da ne pišemo iste stvari večkrat.
- **Večja preglednost in modularnost** → sistem razbijemo na manjše dele (objekte).
- **Lažje vzdrževanje in razširjanje** → nove funkcionalnosti dodamo z novimi razredi.
- **Podpora knjižnicam** → Pythonove knjižnice (npr. Django za splet, Tkinter za GUI, Pygame za igre) temeljijo na OOP.

Kdaj uporabiti OOP?

- Ko gradimo **večje in kompleksne projekte**.
- Ko želimo modelirati **resnične entitete** (uporabniki, naročila, izdelki).
- Ko potrebujemo **fleksibilnost** in **razširljivost** (aplikacije, ki se bodo razvijale).
- Ko delamo z **grafičnimi vmesniki, spletnimi okvirji** in **igralnimi pogoni**, kjer je OOP že vgrajen.

<https://anzeljg.github.io/rin2/book2/2401/index.html>

Proceduralni pristop

Za majhne in enostavne skripte pa je včasih enostavnejši **proceduralni pristop**.

Proceduralni pristop je tradicionalen način programiranja, ki se osredotoča na izvajanje zaporedja ukazov oziroma algoritmov za reševanje problema. V tem pristopu so podatki in funkcije ločeni; funkcije obdelujejo podatke, ki so pogosto globalni ali se jim posredujejo kot argumenti. Programerju je glavni cilj napisati logiko korak za korakom, pri čemer se podatki ne združujejo z metodami, ki jih obdelujejo. Proceduralni pristop je primeren za enostavne skripte ali programe, kjer kompleksna struktura ni potrebna, in ga najdemo v jezikih, kot so C, Pascal ali Fortran.

V nasprotju s tem objektno orientirano programiranje združuje podatke in metode v objekte, omogoča dedovanje, polimorfizem in boljšo modularnost. V proceduralnem pristopu so funkcije ponavadi osnovni gradniki, medtem ko so v OOP razredi in objekti, kar omogoča lažje vzdrževanje in ponovno uporabo kode v kompleksnih sistemih, kot so igre, grafični vmesniki ali večje poslovne aplikacije.

Proceduralni pristop (Procedural Programming)

- Program je sestavljen iz **funkcij in postopkov**, ki izvajajo določena dejanja nad podatki.
- Podatki in funkcije so **ločeni** – podatki so običajno globalni ali se prenašajo kot argumenti funkcij.
- Fokus je na **zaporedju ukazov** (algoritmu), ki rešuje problem korak za korakom.
- Primer jezika: **C, Pascal, Fortran**.

Primer proceduralnega pristopa v C#

```
using System;

class Program
{
    static double Povprecje(int[] ocene)
    {
        int vsota = 0;
        foreach (int o in ocene)
        {
            vsota += o;
        }
        return (double)vsota / ocene.Length;
    }

    static void Main()
    {
        int[] oceneAna = { 8, 9, 10 };
        int[] oceneMarko = { 7, 9 };

        Console.WriteLine("Ana: " + Povprecje(oceneAna));
        Console.WriteLine("Marko: " + Povprecje(oceneMarko));
    }
}
```

Tukaj so ocene ločene od funkcije, ki jih obdeluje. Ni razredov, ni dedovanja, ni polimorfizma.

Primer proceduralnega pristopa v Python-u

```
# Proceduralni način za izračun povprečja ocen
ocene_ana = [8, 9, 10]
ocene_marko = [7, 9]

def povprecje(ocene):
    return sum(ocene) / len(ocene)

print("Ana:", povprecje(ocene_ana))
print("Marko:", povprecje(ocene_marko))
```

Tukaj so ocene ločene od funkcije, ki jih obdeluje. Ni razredov, ni dedovanja, ni polimorfizma.

Objektno orientirano programiranje v Pythonu in C#

Primer objektno orientiranega pristopa v Pythonu

Python uveljavlja OOP na zelo prijazen in berljiv način. Vsak razred v Pythonu je definiran s ključno besedo **class**, objekti pa se ustvarjajo tako, da razred pokličemo kot funkcijo.

Razredi in objekti

Primer definicije razreda v Python-u

```
class Oseba:
    def __init__(self, ime, starost):
        self.ime = ime
        self.starost = starost

    def pozdravi(self):
        print(f"Živjo, moje ime je {self.ime} in star sem {self.starost} let.")

# ustvarimo objekt
ana = Oseba("Ana", 25)
ana.pozdravi()
```

Primer definicije razreda v C#

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string Ime { get; set; }
    public string Priimek { get; set; }
    private List<int> Ocene { get; set; }

    public Student(string ime, string priimek)
    {
        Ime = ime;
        Priimek = priimek;
        Ocene = new List<int>();
    }


    public void DodajOceno(int ocena)
    {
        if (ocena >= 1 && ocena <= 10)
            Ocene.Add(ocena);
        else
            Console.WriteLine("Ocena mora biti med 1 in 10.");
    }
}
```

V tem primeru imamo razred *Student*, ki združuje podatke (*ime*, *priimek*, *ocene*) in metode (*DodajOceno*, *Povprecje*, *PrikaziInfo*). To je **OOP pristop**, saj podatki in vedenje tvorijo skupno enoto.

Primer objektno orientiranega pristopa v C#

1. Razred `Student` (osnovni razred)


csharp

 Kopiraj kodo

```
class Student
{
    public string Ime { get; set; }
    public string Priimek { get; set; }
    private List<int> Ocene { get; set; }
```

- `Ime` in `Priimek` sta javni lastnosti, do katerih lahko dostopamo od zunaj.
- `Ocene` so zasebne (`private`) – to je inkapsulacija, saj so podatki skriti pred zunanjim svetom.


csharp

 Kopiraj kodo

```
public Student(string ime, string priimek)
{
    Ime = ime;
    Priimek = priimek;
    Ocene = new List<int>();
}
```

- Konstruktor nastavi ime, priimek in pripravi prazen seznam ocen.

csharp


 Kopiraj kodo

```
public virtual void DodajOceno(int ocena)
{
    if (ocena >= 1 && ocena <= 10)
        Ocene.Add(ocena);
    else
        Console.WriteLine("Ocena mora biti med 1 in 10.");
}
```



- `DodajOceno` doda oceno v seznam, če je veljavna.
- Ključna beseda `virtual` pomeni, da lahko podrazredi to metodo prepišejo (polimorfizem).

csharp

 Kopiraj kodo

```
public double Povprecje()
{
    return Ocene.Count > 0 ? Ocene.Average() : 0;
}
```

- `Povprecje` vrne povprečno vrednost vseh ocen, če obstajajo, sicer 0.

csharp

 Kopiraj kodo

```
public virtual void PrikaziInfo()
{
    Console.WriteLine($"{Ime} {Priimek}, Povprečje: {Povprecje():F2}");
}
}
```

- Metoda `PrikaziInfo` izpiše podatke o študentu.
- Tudi ta metoda je `virtual`, da jo lahko podrazredi prilagodijo.

2. Razred `IzredniStudent` (dedovanje + polimorfizem)

csharp

 Kopiraj kodo

```
class IzredniStudent : Student
{
    public IzredniStudent(string ime, string priimek) : base(ime, priimek) { }
```

- `IzredniStudent` **deduje** od `Student`.
- Konstruktor kliče konstruktor nadrazreda (`base`).

csharp

 Kopiraj kodo

```
public override void DodajOceno(int ocena)
{
    int bonusOcena = Math.Min(10, ocena + 1);
    base.DodajOceno(bonusOcena);
}
```

- Ta metoda **prepiše** (override) osnovno različico.
- Izredni študent dobi **bonus točko**, vendar največ 10.
- Nato pokliče osnovno metodo (`base.DodajOceno`) z novo oceno.

csharp


 Kopiraj kodo

```
public override void PrikaziInfo()
{
    Console.WriteLine($"[Izredni] {Ime} {Priimek}, Povprečje: {Povprecje():F2}");
}
```

- Izpis prilagodi tako, da je jasno označeno, da gre za izrednega študenta.
- To je **polimorfizem**: ista metoda `PrikaziInfo()` deluje različno za navadne in izredne študente.

3. Razred Univerza (abstrakcija sistema)


csharp

 Kopiraj kodo

```
class Univerza
{
    public string Ime { get; set; }
    private List<Student> Studenti { get; set; }
```

- Univerza ima ime in seznam študentov.


csharp

 Kopiraj kodo

```
public Univerza(string ime)
{
    Ime = ime;
    Studenti = new List<Student>();
}
```

- Konstruktor nastavi ime in pripravi prazen seznam študentov.


csharp

 Kopiraj kodo

```
public void DodajStudenta(Student s)
{
    Studenti.Add(s);
}
```

- Metoda za dodajanje študentov.
- Sprejme kateregakoli študenta (rednega ali izrednega) – to omogoča polimorfizem.


csharp

 Kopiraj kodo

```
public void PrikaziVse()
{
    Console.WriteLine($"Univerza: {Ime}");
    foreach (var s in Studenti)
    {
        s.PrikaziInfo();
    }
}
```

- Izpiše vse študente univerze.
- Ključni trenutek: klic `s.PrikaziInfo()` bo odvisno od vrste objekta uporabil ustrezno različico (navadni ali izredni študent).

csharp

 Kopiraj kodo

```
public double PovprecjeUniverze()
{
    return Studenti.Count > 0 ? Studenti.Average(s => s.Povprecje()) : 0;
}
}
```

- Izračuna povprečje vseh študentov na univerzi.

4. Glavni razred Program

csharp

 Kopiraj kodo

```
class Program
{
    static void Main()
    {
        Univerza uni = new Univerza("Univerza Ljubljana");

        Student ana = new Student("Ana", "Novak");
        ana.DodajOceno(8);
        ana.DodajOceno(9);

        IzredniStudent marko = new IzredniStudent("Marko", "Kovač");
        marko.DodajOceno(7);
        marko.DodajOceno(9);

        uni.DodajStudenta(ana);
        uni.DodajStudenta(marko);

        uni.PrikaziVse();

        Console.WriteLine($"Povprečje univerze: {uni.PovprecjeUniverze():F2}");
    }
}
```

V tem programu:

- Ustvarimo univerzo.
- Dodamo navadnega študenta Ana.
- Dodamo izrednega študenta Marka.
- Ob prikazu bo Ana imela ocene 8 in 9 → povprečje 8,5.
- Marko bo vpisal ocene 7 in 9, vendar se mu pretvorita v 8 in 10 (zaradi bonusa) → povprečje 9.
- Univerza nato prikaže oba študenta in izračuna skupno povprečje.

Kaj smo tukaj pokazali? Pokazali smo osnovne koncepte objektnega pristopa.

- **Abstrakcija:** razredi `Student`, `IzredniStudent` in `Univerza` modelirajo realne koncepte.
- **Inkapsulacija:** ocene so skrite in dosegljive le preko metod.
- **Dedovanje:** `IzredniStudent` podeduje lastnosti in metode od `Student`.
- **Polimorfizem:** metode `DodajOceno` in `PrikaziInfo` se obnašajo različno, odvisno od vrste študenta.

Osnovni pojmi (terminologija) pri OO razvoju aplikacij

Razred (Class)

Razred je nekakšen načrt ali predloga za ustvarjanje objektov. Določa lastnosti (atribute) in obnašanje (metode) objektov, ki bodo ustvarjeni iz njega. Sami po sebi razredi ne vsebujejo podatkov, le opisujejo, kakšni bodo objekti.

Primer: Razred `Avto` bi lahko imel attribute, kot so `barva`, `znamka` in `model`, ter metode, kot sta `pospeši()` in `zaviraj()`.

Objekt (Object)

Objekt je konkreten primerek razreda. Je del programske opreme, ki združuje podatke (attribute) in funkcionalnost (metode). Lahko si ga predstavljate kot fizično izvedbo načrta, ki ga določa razred.

Primer: `moj_avto` je objekt, ki pripada razredu `Avto`. Njegovi atributi so na primer `barva="rdeča"`, `znamka="Ford"` in `model="Focus"`.

Lastnost (Property ali Atribut)

Lastnost ali **atribut** je značilnost objekta, ki opisuje njegovo stanje. To so spremenljivke, ki so del razreda in določajo podatke, ki jih bo imel objekt.

Primer: V razredu `Avto` so `barva`, `znamka` in `model` lastnosti.

Metoda (Method)

Metoda je funkcija, ki je definirana znotraj razreda in opisuje obnašanje objekta. Metode izvajajo operacije znotraj objekta, dostopajo do njegovih atributov in jih spreminjajo.

Primer: V razredu `Avto` sta `pospeši()` in `zaviraj()` metodi.

Funkcija (Function)

Funkcija je blok kode, ki opravlja določeno nalogo. V nasprotju z **metodo** funkcija ne pripada nobenemu razredu ali objektu. Gre za splošen, neodvisen kos kode, ki ga lahko pokličemo s poljubnega mesta v programu.

Iz prejšnjega primera lahko v kodi prepoznamo tudi navadne funkcije, ki so del določenega razreda in jih imenujemo **metode**. Vendar pa lahko za primer **funkcije** (v smislu, da ni vezana na objekt) v C# napišemo statično metodo, ki pripada razredu *Program*, vendar ne potrebuje instance tega razreda. To pomeni, da jo lahko pokličemo direktno, brez ustvarjanja novega objekta.

Primer funkcije v C#

```
using System;

class Program
{
    // Glavna metoda
    static void Main()
    {
        int stevilo = 5;
        // Kličemo statično funkcijo brez ustvarjanja objekta
        long rezultat = IzracunajFaktorial(stevilo);
        Console.WriteLine($"Faktorial števila {stevilo} je {rezultat}.");
    }

    // Funkcija (statična metoda), ki ni vezana na nobeno instanco razreda
    public static long IzracunajFaktorial(int n)
    {
        if (n < 0)
        {
            throw new ArgumentException("Število mora biti nenegativno.");
        }
        if (n == 0)
        {
            return 1;
        }

        long faktorial = 1;
        for (int i = 1; i <= n; i++)
        {
            faktorial *= i;
        }
        return faktorial;
    }
}
```

V tem primeru je *IzracunajFaktorial* klasičen primer funkcije, ker:

- je statična, kar pomeni, da je vezana na razred in ne na objekt,
- je namenjena specifični nalogi (*izračun faktoriala*),
- lahko jo pokličemo neposredno iz glavne metode *Main* brez ustvarjanja instance razreda *Program*.

Če bi želeli vstaviti funkcijo v vašo obstoječo kodo, bi jo lahko dodali v razred *Program*, saj tam že obstajajo druge statične funkcije, vključno z glavno metodo *Main*. Na ta način bi bila funkcija dostopna vsem delom programa, ne da bi bila del specifičnega objekta, kot sta *Student* ali *Univerza*.

Modul (Module)

Modul je datoteka, ki vsebuje programsko kodo, kot so razredi, funkcije in spremenljivke. Moduli so namenjeni organizaciji kode in ponovni uporabi. Omogočajo, da kodo, ki je shranjena v eni datoteki, uvozimo v drugo.

Knjižnica (Library)

Knjižnica je zbirka modulov, razredov, funkcij in drugih virov, ki so vnaprej napisani, da bi programerjem olajšali razvoj. Namesto da bi pisali kodo od začetka, lahko preprosto uporabite kodo iz knjižnice. Npr.: Knjižnica za delo z grafikoni, ki vsebuje module in razrede za ustvarjanje različnih vrst grafikonov.