PS kūrimo procesas klausimai egzaminui (II teorijos dalis)

2024-04-16

1. **Versijų kontrolė, tipai, paskirtis**
2. **Versijų kontrolė GIT: terminai, veiksmai (commit, push ir kt.), konvencijos**
3. **Išsišakojimų konfiktų priežastys ir sprendimai**
4. **Testavimo proceso sąsaja su kitais PS procesais**

Testavimo procesas turi glaudžią sąsają su kitais programų sistemų (PS) procesais ir veiklomis. Ši sąsaja yra būtina siekiant užtikrinti, kad sistema būtų išbandyta visapusiškai ir kokybiškai prieš jos įdiegimą arba įvedimą į produkciją. Štai keli pagrindiniai aspektai, kurie atskleidžia testavimo proceso sąsają su kitais PS procesais:

1. **Reikalavimų analizė ir specifikavimas**: Testavimo procesas prasideda nuo reikalavimų analizės ir specifikavimo. Testavimo specialistai turi suprasti sistemos funkcinius ir nefunkcinius reikalavimus, kad galėtų sukurti atitinkamas testavimo scenarijus ir planus.
2. **Dizainas ir kūrimas**: Testavimo komandai reikia įsitraukti į PS dizaino ir kūrimo procesą, kad galėtų identifikuoti ankstyvųjų klaidų priežastis ir palaikyti kokybę jau nuo pradžios. Tai gali apimti įvairių rūšių peržiūras, kaip antai architektūros, kodavimo ir dizaino peržiūras.
3. **Kodavimas ir programavimas**: Testavimo procesas turi sąsają su kodavimo ir programavimo etapu. Testuotojai gali naudoti automatizuotus testavimo įrankius, kurie leidžia kurti testus ir juos vykdyti kartu su koduojančiais programuotojais, siekiant užtikrinti, kad kiekvienas kodavimo pokytis nepažeistų esamų funkcijų ir kad naujos funkcijos būtų išbandytos.
4. **Konfigūracijos valdymas**: Testavimo proceso sąsaja su konfigūracijos valdymo procesu užtikrina, kad testavimo aplinka būtų suderinta su PS konfigūracija. Tai apima versijų valdymą, konfigūracijos valdymą ir infrastruktūros valdymą, kad būtų galima atlikti korektiškus testus.
5. **Vidinės kokybės užtikrinimas**: Testavimo procesas taip pat turi sąsają su procesais, skirtais užtikrinti vidinę kokybę, pavyzdžiui, kokybės kontrolės ir garantijos procesais. Tai apima klaidų stebėjimo ir tvarkymo sistemų palaikymą, kad būtų užtikrinta, jog aptiktos klaidos būtų kuo greičiau taisomos.
6. **Diegimas ir palaikymas**: Prieš sistemos diegimą į produkciją, testavimo procesas turi užtikrinti, kad visas funkcijos ir veikla veiktų teisingai ir be klaidų. Be to, testavimas tęsiasi ir po diegimo, siekiant užtikrinti, kad naujos versijos palaikymas ir toliau garantuotų sistemos veikimą.

Šios sąsajos ir bendradarbiavimas su kitais PS procesais yra esminiai, siekiant užtikrinti, kad sistema būtų kokybiška, patikima ir atitiktų visus verslo ir vartotojų poreikius.

1. **Elgsena grįstas kūrimas (BBD). Pavyzdžiai. Paskirtis, taikymas. Gerkin sintaksė**

Elgsena grįstas kūrimas (BDD) yra metodas, kuriuo naudojamasi programinio kodo funkciniam elgesiui aprašyti ir išbandyti. Pagrindinė idėja yra ta, kad programos elgesys turėtų būti aprašytas iš išorinio naudotojo perspektyvos, nurodant, kaip sistema turėtų elgtis konkrečiose situacijose. Šių aprašymų pagrindu galima automatiškai generuoti testus.

Pavyzdžiui, jei kuriate internetinį pardavimų puslapį, vienas iš elgesio aprašymų gali būti: "Kai vartotojas paspaudžia 'Pridėti į krepšelį' mygtuką su prekės ID, sistema turėtų pridėti prekę į krepšelį ir atnaujinti bendrą krepšelio sumą."

BDD naudoja specialų kalbos sintaksę, vadinamą Gherkin, kad būtų galima aiškiai aprašyti šį elgesį. Gherkin yra paprasta, struktūruota kalba, kuri leidžia išreikšti elgesio scenarijus ir juos sujungti su programos kodu. Pagrindinės Gherkin sintaksės dalys apima:

1. **Funkcijos pavadinimas**: Apibrėžiama, kokia funkcija ar savybė bus testuojama.
2. **Scenario (Scenarijus)**: Tai konkrečios situacijos aprašymas, kaip sistema turėtų elgtis. Scenarijus prasideda žodžiu "Scenario" ir tada pateikiamas aprašymas, kaip turėtų vykti elgesys.
3. **Gyvendinimo pavyzdys (Example)**: Tai konkrečios situacijos pavyzdys, pateiktas lentelės formatu. Šis pavyzdys pateikia konkretų įvesties ir išvesties duomenų rinkinį, kuris turėtų būti išbandytas.
4. **Turi būti (Given)**, **Kai (When)**, **Tada (Then)**: Šios trys frazės naudojamos aprašant scenarijaus eigos sąlygas (Given), veiksmą, kurį naudotojas atlieka (When), ir tikėtą rezultatą (Then).
5. **TDD testavimas: paskirtis, procesas, ypatumai**

TDD (Test Driven Development) yra programavimo metodika, kurioje testai yra kūrimo proceso pagrindas. Prieš pradedant kurti programinį kodą, rašomi testai, kurie apibrėžia pageidaujamą programos elgesį. Toliau kodas yra rašomas taip, kad jis praeitų sukurtus testus. TDD turi kelis pagrindinius tikslus ir ypatumus:

**Paskirtis:**

1. **Kokybės užtikrinimas**: Testai padeda nustatyti norimą programos elgesį ir užtikrinti, kad kiekviena nauja funkcija arba pokytis neįtakos esamo kodo veikimo.
2. **Dokumentacija**: Testai tampa savotiška programos funkcionalumo dokumentacija, nes jie apibrėžia, kaip kiekviena dalis turėtų veikti.
3. **Pritraukimas ir detektyvavimas**: TDD padeda greitai aptikti klaidas ir defektus, nes naujai sukurti testai yra skirti patikrinti funkcijų teisingumą.

**Procesas:**

1. **Rašykite testą**: Pradėkite kurti testą, kuris apibrėžia norimą funkcionalumą arba elgesį. Šis testas turi nepavykti, nes dar nėra jokio kodo, kuris jį įvykdytų.
2. **Rašykite kodą**: Rašykite kodo dalį, kuri leidžia jūsų sukurtam testui sėkmingai įvykdyti.
3. **Paleiskite testą**: Paleiskite testą, kad įsitikintumėte, jog jis praeina.
4. **Refaktoringas**: Jei reikia, atlikite kodą tobulinimus (refaktoringą), kad jis būtų tvarkingesnis ir efektyvesnis, bet užtikrinant, kad testai vis dar sėkmingai praėjo.
5. **Kartokite**: Kartokite šį ciklą, kurdami naujus testus ir kodo dalis, kol pasiekiate norimą funkcionalumą.

**Ypatumai:**

1. **Greitas atsakymas**: TDD suteikia greitą atsakymą apie kodo kokybę ir galimus defektus, nes kiekviena kodo dalis yra nuolat testuojama.
2. **Tobulėjantis dizainas**: Kadangi kodo architektūra yra kuriami remiantis testais, tai padeda išvengti perkomplikuoto kodo ir skatina geresnį dizainą.
3. **Pasitikėjimas kodu**: TDD praktiką taikant, programuotojai gali būti tikri, kad kiekviena kodo dalis veikia kaip tikimasi, nes tai patvirtina testai.
4. **Stiprus testavimo fokusas**: TDD skatina susikoncentruoti ties tikslu, ką reikia pasiekti, nes kodas yra kuriamas siekiant praeiti testus, o ne tiesiog implementuoti funkcijas.

TDD yra labai veiksmingas būdas užtikrinti kodo kokybę, didinti produktivumą ir sumažinti klaidų skaičių. Tai yra populiarus metodas daugelyje programavimo bendruomenių.

1. **Vienetų testai, struktūra, paskirtis, esminės rašymo taisyklės**

Vienetų testai (angl. unit tests) yra programinės įrangos testavimo tipas, kuris skirtas tikrinti mažiausius įgyvendintų programos komponentų, paprastai funkcijų arba metodų, veikimo vienetus. Šie testai yra rašomi ir vykdomi programuotojų, dažniausiai naudojant testavimo bibliotekas, kad būtų patikrinta, ar kiekvienas programos veikimo vienetas veikia taip, kaip tikimasi. Štai pagrindinės vienetų testų struktūra, paskirtis ir esminės rašymo taisyklės:

### Struktūra:

1. **Pasiruošimas (Setup)**: Ši dalis apima reikiamų pradinės būsenos nustatymą testui, pavyzdžiui, kintamųjų inicijavimą ar objektų kūrimą.
2. **Veiksmas (Action)**: Čia yra vykdomas pats testuojamas veiksmas ar funkcija, sukurta programoje.
3. **Tikrinimas (Assertion)**: Ši dalis apima patikrinimą, ar gauti rezultatai yra tokie, kokie buvo tikėtasi. Tai dažniausiai atliekama naudojant įvairius patikrinimo metodus arba operatorius.
4. **Išardymas (Teardown)**: Paskutinis žingsnis, kuriame yra išvaloma arba atlaisvinama naudojama resursai, pavyzdžiui, duomenų bazės prisijungimas arba laikinieji failai.

### Paskirtis:

* **Patikrinti mažus veikimo vienetus**: Vienetų testai skirti tikrinti mažus programos veikimo vienetus, pavyzdžiui, funkcijas arba metodus, atskirai nuo visos sistemos.
* **Greitas aptikimas klaidų**: Šie testai leidžia greitai aptikti klaidas arba nesuderinamumus veikimo vienetuose, kad jos būtų ištaisytos ankstyvoje plėtros stadijoje.
* **Palaikyti programos stabilumą**: Vienetų testai padeda palaikyti programos stabilumą ir saugumą, patikrinant, ar kiekvienas komponentas veikia teisingai, netgi po pakeitimų.

### Esminės rašymo taisyklės:

1. **Kiekvienas testas turi būti izoliuotas ir nepriklausomas nuo kitų**: Kiekvienas vienetų testas turi būti nepriklausomas ir neturėti sąsajų su kitais testais ar išorinėmis priklausomybėmis.
2. **Rašykite testus, prieš rašydami kodą**: Idealiu atveju vienetų testai turėtų būti sukurti prieš pat įgyvendinant kodą, kad būtų aišku, ką reikia įgyvendinti.
3. **Vienetų testai turi būti nuolat palaikomi ir vykdomi**: Vienetų testai turi būti nuolat palaikomi ir vykdomi, ypač po kodo pakeitimų ar atnaujinimų.
4. **Rašykite aiškius ir suprantamus testus**: Testai turi būti lengvai suprantami ir aiškiai nurodyti, ką tikrinama ir kokį rezultatą tikimasi gauti.

Rašant vienetų testus svarbu laikytis gerų programavimo praktikų ir kokybės standartų, kad jie būtų patikimi ir naudingi projekto vystymosi procese.

1. **Integraciniai vienetų testai: paskirtis, skirtumai nuo vienetų testų**

Integraciniai vienetų testai ir vienetų testai yra dvi skirtingos testavimo strategijos, kurios yra naudojamos programinio kodo kokybei užtikrinti. Štai jų paskirtys ir pagrindiniai skirtumai:

### Integraciniai vienetų testai:

**Paskirtis**:

Integraciniai vienetų testai skirti patikrinti, kaip programos skirtingos dalys sąveikauja tarpusavyje. Šie testai tikrina, ar komponentai (dažniausiai moduliai arba klasės) veikia teisingai kartu, kai jie yra sujungti į didesnę sistemą. Integraciniai testai padeda užtikrinti, kad įvairios programos dalys bendrai veiktų ir komunikuotų pagal numatytą logiką.

**Skirtumai nuo vienetų testų**:

1. **Apimtis**: Integraciniai vienetų testai dažnai turi platesnę apimtį nei vienetų testai, nes jie patikrina, kaip skirtingi komponentai sąveikauja tarpusavyje, o ne tik kaip veikia atskirai kiekvienas komponentas.
2. **Priklausomybės**: Integraciniai vienetų testai gali turėti priklausomybes nuo išorinių sistemų arba duomenų šaltinių, kad būtų galima patikrinti, kaip komponentai sąveikauja su aplinka.
3. **Sudėtingumas**: Dažnai integraciniai vienetų testai gali būti sudėtingesni nei vienetų testai, nes reikia atsižvelgti į sąveiką tarp skirtingų komponentų.

### Vienetų testai:

**Paskirtis**:

Vienetų testai skirti patikrinti atskiras programos dalis (dažniausiai funkcijas arba metodus) izoliuotai, nepriklausomai nuo kitų programos dalių. Šie testai tikrina, ar kiekviena maža programos dalis veikia teisingai pagal savo specifikaciją.

**Skirtumai nuo integracinių vienetų testų**:

1. **Apimtis**: Vienetų testai yra skirti tikrinti atskiras programos dalis, todėl jų apimtis yra ribota, ir jie neturėtų turėti išorinių priklausomybių.
2. **Izoliacija**: Vienetų testai yra izoliuoti ir nepriklauso nuo išorinių veiksnių arba kitų komponentų būsenos. Jie turėtų tikrinti tik konkrečią funkciją ar metodą, nepriklausomai nuo kitų komponentų veikimo.
3. **Priežastys gedimams**: Jei vienetų testas nepavyksta, priežastis dažniausiai yra pačiame teste arba testuojamoje funkcijoje, o ne kitų komponentų sąveikos.

Todėl integraciniai vienetų testai ir vienetų testai yra papildantys vienas kitą strategijos, skirtos užtikrinti programinio kodo kokybę ir veikimą. Integraciniai vienetų testai tikrina, kaip skirtingos programos dalys veikia kartu, o vienetų testai tikrina, kaip atskiri komponentai veikia izoliuotai.

1. **Testinis testavimas: paskirtis, etapai**

Testinis testavimas yra svarbus programų kokybės užtikrinimo procesas, kurio metu testuojama sistema, siekiant nustatyti galimas klaidas, trūkumus arba nesėkmes, kurios gali turėti įtakos sistemos veikimui. Šis testavimo metodas skirtas patikrinti, ar sistema atitinka nustatytus reikalavimus ir ar ji veikia taip, kaip numatyta.

### Paskirtis:

1. **Kokybės užtikrinimas**: Tikrinama, ar sistema veikia taip, kaip tikėtasi, ir ar ji atitinka reikalavimus.
2. **Klaidų nustatymas**: Identifikuojamos klaidos, trūkumai arba nesėkmės, kurios gali turėti įtakos sistemos naudotojams arba jos veikimui.
3. **Patikimumo vertinimas**: Įvertinama, kiek patikimai sistema veikia ir kaip ji elgiasi skirtingose situacijose.

### Etapai:

1. **Planavimas**: Šiame etape nustatomos testavimo strategijos, tikslai, aplinka ir resursai, reikalingi testavimui atlikti. Taip pat paruošiamas testavimo planas.
2. **Dizainas ir kūrimas**: Sukuriami testavimo scenarijai ir testavimo atvejai, kurie padės patikrinti sistemos funkcionalumą ir našumą. Šiame etape taip pat rengiamos testavimo duomenų bazės ir kitos reikalingos priemonės.
3. **Vykdimas**: Testavimo atvejai vykdomi, o gauti rezultatai analizuojami. Šiame etape atliekama testavimo automatinės ar rankinės įrankiais.
4. **Rezultatų vertinimas**: Gautos testavimo rezultatų analizė padeda nustatyti, ar sistema veikia kaip tikėtasi, ir ar reikalingos papildomos pataisos arba tobulinimai.
5. **Ataskaitų pateikimas**: Baigus testavimą, sukuriama testavimo ataskaita, kurioje pateikiami visi testavimo rezultatai, rasta klaidos, jų priežastys ir rekomendacijos jas ištaisyti.
6. **Pataisos**: Jei testavimo metu aptiktos klaidos, jos turi būti ištaisytos ir vėl patikrintos. Šis procesas gali kartotis tol, kol sistema yra laikoma pakankamai patikima ir veiksminga.

Testinis testavimas yra nepakeičiamas žingsnis bet kurios programos kūrimo ir įgyvendinimo procese. Tai padeda užtikrinti, kad sistema veiktų patikimai ir naudotojams teiktų aukščiausią vertę.

1. **FIRST testų kokybė modelis**

FIRST testų kokybės modelis yra geras įrankis, kurį galima naudoti norint užtikrinti, kad testai būtų kokybiški ir veiktų efektyviai. Šis modelis, kuriame „FIRST“ yra akronimas, siūlo gaires, kuriomis vadovaujantis galima sukurti efektyvius ir patikimus testus. Štai kiekvieno „FIRST“ raidės reikšmės:

1. **Fast (Greitas)**: Testai turi būti greiti. Tai reiškia, kad jie turėtų būti lengvai paleidžiami ir vykdomi, nesvarbu, ar tai yra rankiniai, arba automatizuoti testai. Greitas testas yra svarbus, nes jis leidžia greitai aptikti klaidas ir juos pataisyti.
2. **Independent (Nepriklausomas)**: Testai turi būti nepriklausomi vienas nuo kito. Tai reiškia, kad jie neturėtų tapti priklausomi nuo kitų testų rezultatų arba eigos. Nepriklausomi testai yra lengviau palaikomi, suprantami ir perdirbami.
3. **Repeatable (Pakartojamas)**: Testai turi būti pakartojami. Tai reiškia, kad jie turėtų būti lengvai paleidžiami ir kartojami bet kuriuo metu, nepriklausomai nuo aplinkos ar sąlygų. Pakartojami testai yra svarbūs siekiant užtikrinti konsistentišką testavimo rezultatą.
4. **Self-Validating (Savęs patvirtinantis)**: Testai turi patvirtinti save, nes jie turi turėti aiškius ir konkretų pasiekiamą rezultatą. Tai reiškia, kad testas turėtų automatiškai nustatyti, ar funkcija veikia teisingai, be poreikio žmogaus įsikišimo.
5. **Timely (Laiku)**: Testai turi būti laiku sukurti. Tai reiškia, kad jie turėtų būti įtraukti į kodo kūrimo procesą nuo pat pradžių ir nuolat tobulinami. Laiku sukurti testai padeda anksti aptikti ir ištaisyti galimas klaidas.

Naudodamiesi „FIRST“ testų kokybės modeliu, komandos gali sukurti kokybiškus ir veiksmingus testus, kurie padeda užtikrinti programos funkcionalumą, patikimumą ir našumą. Tai svarbus įrankis kiekvienam, kuris siekia geriausios kokybės programų kūrimo procese.

1. **Verifikacijos ir validacijos procesai**

Verifikacija ir validacija (V&V) yra du skirtingi, bet susiję procesai, kurių tikslas yra užtikrinti, kad sukurtas produktas atitiktų numatytus reikalavimus ir būtų kokybiškas bei patikimas. Štai pagrindiniai skirtumai tarp verifikacijos ir validacijos:

### Verifikacija:

1. **Paskirtis**: Verifikacija yra procesas, skirtas patikrinti, ar sukurtas produktas atitinka nustatytus reikalavimus, specifikacijas arba standartus.
2. **Kas**: Verifikacija yra daugiau orientuota į tai, ar buvo teisingai sukurtas produktas. Ji tiriama programinio kodo, dokumentacijos ar kitų artefaktų, kad būtų įsitikinta, jog jie atitinka nustatytus kriterijus.
3. **Kada**: Verifikacija vykdoma dažniausiai ankstyvosiose produktų kūrimo stadijose, prieš produktą pradedant naudoti arba testuoti.
4. **Pavyzdžiai**: Verifikacija gali apimti statinės analizės procesus, kaip antai kodavimo peržiūras, dokumentacijos peržiūras, ir kitus procesus, skirtus patikrinti, ar sukurtas produktas atitinka numatytus reikalavimus.

### Validacija:

1. **Paskirtis**: Validacija yra procesas, skirtas patikrinti, ar sukurtas produktas atitinka galutinių naudotojų realius poreikius ir tikslus bei ar jis naudojamas ir veikia taip, kaip numatyta.
2. **Kas**: Validacija yra orientuota į tai, ar sukurtas produktas yra teisingas ir naudingas naudotojams. Ji dažniausiai atliekama su realiais naudotojais, kad būtų įsitikinta, jog produktas atitinka jų lūkesčius.
3. **Kada**: Validacija dažniausiai vykdoma vėlesnėse produktų kūrimo stadijose, kai produktas jau yra pasirengęs naudoti arba testuoti.
4. **Pavyzdžiai**: Validacija gali apimti vartotojo testavimą, beta testavimą, naudotojų grupeės apklausas ar kitus procesus, skirtus įvertinti, ar produktas patenkina realius naudotojų poreikius ir tikslus.

Verifikacija ir validacija yra būtini ir papildantys vienas kitą procesai, kurie padeda užtikrinti sukurtų produktų kokybę ir tinkamumą naudoti. Tai yra esminiai žingsniai kiekvieno produkto kūrimo cikle.

1. **CI procesas. Paskirtis, etapai, gerinimo gairės (šablonai)**

Nuolatinė integracija (CI) yra procesas, kuriuo kodo pakeitimai reguliariai ir automatiškai jungiami su bendra versija, siekiant anksti aptikti galimas klaidas ir užtikrinti produkto kokybę. Štai pagrindinės nuolatinės integracijos paskirtis, etapai ir gairės:

### Paskirtis:

1. **Ankstyvas klaidų aptikimas**: CI leidžia greitai aptikti klaidas arba nesuderinamumus tarp kodo pakeitimų, padėdamas išvengti jų plitimo ir užtikrinti aukštos kokybės produktą.
2. **Automatizavimas**: CI automatizuoja kodo integracijos, testavimo ir kokybės tikrinimo procesus, sumažindamas žmogiškųjų klaidų riziką ir pagreitinant plėtros ciklą.
3. **Suderinamumo užtikrinimas**: CI procesas užtikrina, kad visi kodo pakeitimai derėtų su bendru projektu ir nesukeltų nesuderinamumo arba konfliktų.

### Etapai:

1. **Kodo pakeitimų įtraukimas**: Nauji kodo pakeitimai yra įtraukiami į bendrą kodą, dažnai naudojant versijų kontrolės sistemą.
2. **Automatinis testavimas**: Automatiniai testai yra paleidžiami, siekiant patikrinti, ar nauji kodo pakeitimai neįtakojo esamo funkcionalumo ir ar jie veikia kaip tikimasi.
3. **Kodo integracija**: Kodo pakeitimai yra integruojami su bendra versija, o sukurtas naujas kodas yra automatiškai kompiliuojamas ir paleidžiamas.
4. **Rezultatų pranešimas**: Po kodo integracijos yra generuojami rezultatų pranešimai, kurie informuoja komandą apie testų rezultatus ir galimas klaidas.

### Gerinimo gairės:

1. **Automatizacija**: Automatizuokite kiekvieną įmanomą žingsnį, pradedant nuo kodo integracijos ir baigiant testavimo ir kokybės tikrinimo procesais.
2. **Daug kartų per dieną**: Vykdant CI, integracijos ir testavimo procesai turėtų būti vykdomi daug kartų per dieną, siekiant užtikrinti kuo greitesnį klaidų aptikimą ir taisymą.
3. **Svarbiausi testai pirma**: Paleiskite svarbiausius ir esminius testus pirma, siekiant greitai aptikti pagrindines klaidas ir užtikrinti pagrindinės funkcionalumo veikimą.
4. **Stabilumo užtikrinimas**: Visada užtikrinkite, kad kodo bazė būtų stabilus ir veikiantis prieš įtraukiant naujus pakeitimus, kad nebūtų iškreiptas bendras kodas.

CI procesas gali būti įgyvendintas naudojant įvairias priemones ir įrankius, tokius kaip Jenkins, GitLab CI/CD, Travis CI ir kt. Svarbu atsižvelgti į specifinius projekto poreikius ir kontekstą, siekiant tinkamai pritaikyti nuolatinės integracijos principus ir praktikas.

1. **CI/CD skirtumai, etapai**
2. **MLDevOps, DevOps procesas. Paskirtis, etapai**

MLDevOps yra susijęs su mašininio mokymosi (Machine Learning, ML) projektų įtraukimu į DevOps procesą. Tai pritaikytas DevOps metodų ir praktikų rinkinys, skirtas efektyviai valdyti mašininio mokymosi modelių kūrimą, testavimą, diegimą ir eksploatavimą. Štai pagrindinės MLDevOps paskirtis ir etapai:

### Paskirtis:

1. **Spartesnis modelių kūrimas**: MLDevOps siekia pagreitinti mašininio mokymosi modelių kūrimo procesą, integruodamas juos į vieningą kodo kūrimo ir diegimo srautą.
2. **Didesnė produktyvumas**: Ši metodologija padeda automatizuoti daugelį mašininio mokymosi procesų, kas leidžia komandoms daug greičiau kurti, testuoti ir diegti modelius.
3. **Geriausia kokybė ir stabilumas**: MLDevOps užtikrina, kad mašininio mokymosi modeliai būtų testuojami, validuojami ir diegiami su aukščiausiu kokybės ir stabilumo lygiu.

### Etapai:

1. **Duomenų surinkimas ir paruošimas**: Pirmasis etapas yra susijęs su duomenų surinkimu ir jų paruošimu modeliui mokytis. Tai apima duomenų analizę, valymą, normalizavimą ir transformaciją.
2. **Modelio kūrimas ir mokymas**: Šiame etape kuriama mašininio mokymosi modelio architektūra ir modelis yra mokomas su tinkamais duomenimis.
3. **Modelio testavimas ir vertinimas**: Sukurtas modelis yra testuojamas su atskirais testavimo duomenimis, o gauti rezultatai yra įvertinami, siekiant patikrinti jo efektyvumą ir tikslumą.
4. **Diegimas ir eksploatavimas**: Patvirtinus modelio veikimą ir kokybę, jis yra diegiamas į produkciją ir pradeda būti naudojamas realiame pasaulyje. Be to, būtina užtikrinti jo stabilų veikimą, atlikdami nuolatines sekas ir prireikus atnaujinimus.
5. **Automatizacija ir CI/CD**: MLDevOps taiko nuolatinės integracijos ir nuolatinės pristatymo principus, kuriant automatizuotus procesus, kurie leidžia greitai ir patikimai diegti modelius į produkciją.
6. **Monitoringas ir atnaujinimai**: Svarbus etapas yra modelio veikimo stebėjimas ir nuolatinis vertinimas, kad būtų galima identifikuoti problemas ir reikalingus atnaujinimus arba tobulinimus.

MLDevOps procesas padeda užtikrinti, kad mašininio mokymosi modeliai būtų kokybiški, stabilūs ir efektyviai naudojami produkcijoje, tuo pat metu optimizuojant visą kūrimo, testavimo ir diegimo ciklą.

1. **Brandos ir gebėjimo modeliai: modelių pavyzdžiai (Agile Maturity Model, Scrum Maturity Model, CMMI-DEV ir kt.)**

Šie modeliai, tokie kaip Agile Maturity Model, Scrum Maturity Model ir CMMI-DEV (Capability Maturity Model Integration for Development), yra naudojami organizacijose siekiant įvertinti jų gebėjimų lygį ir tobulinti darbo procesus. Štai pavyzdžiai kiekvieno modelio:

### Agile Maturity Model (Agile Brandos Modelis):

1. **Agile Fluency Model**: Šis modelis apibrėžia keturias skirtingas komandų gebėjimų brandos stadijas: Focusing, Delivering, Optimizing ir Strengthening. Kiekviena stadija apibrėžia skirtingą lygį, kuriuo komanda praktikuoja agilias principus ir praktikas.
2. **Agile Capability Model (ACM)**: Šis modelis pateikia skirtingus brandos lygius, pradedant nuo pradinio, kai organizacija dar tik pradeda įgyvendinti agilias praktikas, iki aukščiausio, kai organizacija yra visiškai pritaikė agilų modelį ir pasiekiama aukščiausia efektyvumo lygį.

### Scrum Maturity Model (Scrum Brandos Modelis):

1. **Scrum Capability Model**: Šis modelis apibrėžia penkis skirtingus Scrum brandos lygius: Forming, Norming, Storming, Performing ir Adjourning. Kiekvienas lygis atspindi skirtingą komandos būseną ir brandos laipsnį, kai praktikuojami Scrum metodai.
2. **Scrum@Scale**: Tai skalėjimo rėmas, kuris padeda organizacijoms pritaikyti Scrum praktikas ir procesus visoje organizacijoje, ne tik atskirose komandose.

### CMMI-DEV (Capability Maturity Model Integration for Development):

1. **Initial (Lygis 1)**: Pradinis lygis, kai procesai yra nevaldomi ir nenuoseklūs.
2. **Managed (Lygis 2)**: Procesai yra kontroliuojami ir planuojami, nors gali kilti nuoseklumo problemų.
3. **Defined (Lygis 3)**: Procesai yra gerai apibrėžti ir standartizuoti, organizacija vadovaujasi pripažintais geriausiais praktikais.
4. **Quantitatively Managed (Lygis 4)**: Procesai yra vertinami naudojant kiekybinius duomenis, siekiant užtikrinti jų efektyvumą ir efektyvumą.
5. **Optimizing (Lygis 5)**: Organizacija nuolat tobulina ir tobulina savo procesus, remdamasi kiekybiniais duomenimis ir geriausiomis praktikomis.

Šie modeliai padeda organizacijoms vertinti ir tobulinti jų gebėjimus bei procesus, siekiant pasiekti aukštesnį veiklos efektyvumą ir kokybę.

1. **Organizacijos brandos ir proceso gebėjimo samprata, lygmenys, priklausomybė, skirtumai**

Organizacijos branda ir proceso gebėjimas yra svarbūs sąvokos, susijusios su organizacijų gebėjimu efektyviai vykdyti savo veiklą ir pasiekti užsibrėžtus tikslus. Štai pagrindiniai skirtumai, lygmenys ir priklausomybė tarp šių sąvokų:

### Organizacijos branda:

1. **Samprata**: Organizacijos branda apibrėžia organizacijos gebėjimą efektyviai prisitaikyti, tobulėti ir išlikti konkurencinga savo rinkoje per laiką. Tai apima organizacijos kultūrą, vertybes, gebėjimus, procesus ir technologijas.
2. **Lygmenys**: Organizacijos branda gali būti vertinama skirtinguose lygmenyse, pradedant nuo individualių darbuotojų, komandų ir skyrių, iki visos organizacijos lygmens.
3. **Priklausomybė**: Organizacijos branda priklauso nuo daugelio veiksnių, įskaitant vadybą, kultūrą, inovacijas, procesus, gebėjimus ir adaptacijos galimybes.

### Proceso gebėjimas:

1. **Samprata**: Proceso gebėjimas apibrėžia organizacijos gebėjimą efektyviai planuoti, vykdyti, kontroliuoti ir tobulinti savo veiklą, naudojant standartizuotus arba pritaikytus procesus.
2. **Lygmenys**: Proceso gebėjimas taip pat gali būti vertinamas skirtinguose lygmenyse, pradedant nuo individualių darbo užduočių arba funkcijų iki visos organizacijos procesų.
3. **Priklausomybė**: Proceso gebėjimas priklauso nuo organizacijos pajėgumo efektyviai valdyti ir tobulinti savo procesus, atsižvelgiant į reikalavimus, standartus ir geriausias praktikas.

### Skirtumai:

1. **Apimtis**: Organizacijos branda apima daugiau nei tik procesų valdymą. Ji apima visą organizacijos veiklą, įskaitant vadybą, kultūrą, strategiją ir inovacijas. Proceso gebėjimas yra daugiau orientuotas į procesų valdymą ir tobulinimą.
2. **Lygmenys**: Organizacijos branda ir proceso gebėjimas gali būti vertinami skirtinguose lygmenyse, bet procesų gebėjimas dažniausiai yra daugiau susijęs su konkrečiais procesais arba veiklos sritimis, tuo tarpu organizacijos branda apima bendrą organizacijos veiklą.
3. **Poveikis**: Organizacijos branda tiesiogiai įtakoja procesų gebėjimą, nes stipri organizacijos kultūra ir vadovybė gali skatinti procesų tobulinimą ir inovacijas. Tačiau proceso gebėjimas gali turėti tiesioginį poveikį organizacijos brandai, nes efektyvūs ir efektyvūs procesai padeda organizacijai pasiekti savo tikslus ir išlikti konkurencinga.
4. **CMMI-Dev paskirtis, struktūra, lygmenis, procesai.**

CMMI-DEV (Capability Maturity Model Integration for Development) yra standartizuotas modelis, skirtas vertinti ir tobulinti organizacijų gebėjimą vykdyti ir valdyti programinės įrangos ar kitų produkto kūrimo procesus. Šis modelis padeda organizacijoms identifikuoti jų gebėjimų lygmenis, taip pat nustatyti, kuriose srityse reikia tobulėti. Štai pagrindinės CMMI-DEV paskirtis, struktūra, lygmenys ir procesai:

**Paskirtis:**

1. **Vertinti organizacijų brandą**: CMMI-DEV padeda organizacijoms įvertinti jų brandos lygį ir procesų gebėjimus, siekiant nustatyti stipriąsias ir silpnąsias sritis.
2. **Tobulinti procesus**: Modelis padeda organizacijoms tobulinti savo procesus, nustatant geriausias praktikas ir įgyvendinant jas.
3. **Užtikrinti kokybę ir efektyvumą**: CMMI-DEV padeda organizacijoms užtikrinti, kad jų procesai būtų kokybiški, efektyvūs ir pasiektų numatytus tikslus.

**Struktūra:**

CMMI-DEV modelis susideda iš penkių pagrindinių kategorijų (Process Areas), kurios yra suskirstytos į lygius, kurie atspindi proceso brandą ir gebėjimus. Šios kategorijos yra:

1. **Lygis 1: Initial**: Procesai yra nevaldomi, organizacija negali užtikrinti stabilumo ir kokybės.
2. **Lygis 2: Managed**: Pagrindiniai procesai yra valdomi ir gerinami, bet kokybė ir našumas gali kilti problemų.
3. **Lygis 3: Defined**: Procesai yra gerai apibrėžti ir standartizuoti, organizacija vadovaujasi pripažintais geriausiais praktikais.
4. **Lygis 4: Quantitatively Managed**: Procesai yra vertinami naudojant kiekybinius duomenis, siekiant užtikrinti jų efektyvumą ir efektyvumą.
5. **Lygis 5: Optimizing**: Organizacija nuolat tobulina ir tobulina savo procesus, remdamasi kiekybiniais duomenimis ir geriausiomis praktikomis.

**Procesai:**

Kiekvienas CMMI-DEV proceso lygis apima tam tikrą skaičių procesų sričių arba sritis, vadinamų „Process Areas“. Šios sritys apima veiklas, kurios turėtų būti įgyvendintos organizacijoje norint pasiekti tam tikrą brandos lygį. Pavyzdžiui, kai kurios iš pagrindinių „Process Areas“ apima:

* Requirements Management (Reikalavimų valdymas)
* Project Planning (Projekto planavimas)
* Monitoring and Control (Stebėjimas ir kontrolė)
* Measurement and Analysis (Matavimas ir analizė)
* Process and Product Quality Assurance (Proceso ir produkto kokybės užtikrinimas)

Kiekviena proceso sritis apibrėžia tam tikras praktikas ir veiklas, kurias organizacija turėtų vykdyti norint užtikrinti efektyvų ir kokybišką proceso eigą.

1. **PS proceso gerinimas. Kam skirtas? Pavyzdžiai iš konkrečių PS procesų (praktiniai darbai)**

Procesų gerinimas projektų vadovavime yra skirtas nuolat tobulinti ir optimizuoti projektų valdymo procesus, siekiant pagerinti veiklos efektyvumą, sumažinti riziką ir padidinti projektų sėkmės tikimybę. Tai gali būti taikoma bet kokiuose projektuose, nepriklausomai nuo jų pobūdžio ar dydžio. Štai keletas pavyzdžių iš konkrečių projektų vadovavimo procesų, kurie gali būti patobulinti:

1. **Projekto planavimas**: Procesas, skirtas sukurti detalią projekto planą, apimantį užduočių sąrašą, resursų paskirstymą, laiko grafiką ir rizikų valdymo planą. Gerinimas šioje srityje gali apimti geresnį komunikacijos suinteresuotaisiais subjektais, detaliau identifikuojant ir įvertinant rizikas bei atsižvelgiant į ankstesnių projektų patirtį.
2. **Reikalavimų valdymas**: Procesas, skirtas identifikuoti, dokumentuoti, vertinti ir tvarkyti projekto reikalavimus. Gerinimas šioje srityje gali būti siekiamas didesnio aiškumo ir nuoseklumo reikalavimų dokumentuose, taip pat reguliaraus jų peržiūrėjimo ir atnaujinimo.
3. **Projekto vykdymas ir kontrolė**: Procesas, skirtas vykdyti numatytus veiksmus pagal projekto planą, stebėti pažangą, identifikuoti problemas ir koreguoti vykdymo eigą. Gerinimas šioje srityje gali apimti efektyvesnį komandos bendradarbiavimą, naudojant tokius metodus kaip reguliarių susitikimų organizavimas, taip pat efektyvių stebėsenos ir ataskaitų sistemų diegimas.
4. **Rizikos valdymas**: Procesas, skirtas identifikuoti, analizuoti, vertinti ir valdyti rizikas, kurios gali paveikti projekto sėkmę. Gerinimas šioje srityje gali apimti geresnį rizikų identifikavimą, jų poveikio analizę ir reagavimo strategijų plėtojimą.
5. **Komunikacija ir susitarimų valdymas**: Procesas, skirtas užtikrinti tinkamą informacijos srautą tarp suinteresuotųjų šalių ir tvarkyti susitarimus bei konfliktus. Gerinimas šioje srityje gali apimti efektyvesnę komunikaciją, naudojant skaitmeninius komunikacijos kanalus ir įdiegiant konfliktų sprendimo mechanizmus.

Procesų gerinimas projektų vadovavime yra nuolatinis procesas, kuris skatina organizaciją nuolat tobulėti ir prisitaikyti prie kintančių sąlygų bei geriausios praktikos. Tai padeda organizacijai užtikrinti didesnę sėkmę savo projektuose ir ilgalaikę konkurencinę pranašumą.