

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Ivadas: Sistemų analizė ir projektavimas
kūrimo proceso kontekste

Prof. dr. Agnus Liutkevičius,

Kompiuterių katedra

Studentų g. 50-202, tel. 300394

Modulio temos

Nr.	Pavadinimas	Paskaitų val.	Pratybų val.	Laboratorinių darbų val.	Konsultacinių seminarų val.
1	Kompiuterinių sistemų kūrimo procesai	2	0	0	0
2	Modeliais pagrįstas kompiuterinių sistemų kūrimas	2	0	0	0
2.1	Sistemos poreikio analizė ir vizijos sudarymas	2	0	2	0
2.2	Programinės ir techninės įrangos analizė ir parinkimas	4	0	4	0
3	UML analizės ir projektavimo metodika	2	0	0	0
3.1	Sistemos reikalavimų analizė, modeliavimas ir specifikavimas	10	0	10	0
3.2	Sistemos architektūrinių modelių sudarymas	3	0	3	0
3.3	Sistemos elgsenos modelių sudarymas	3	0	3	0
3.4	Sistemos realizacijos ir diegimo modelių sudarymas	4	0	4	0
4	Kompiuterinės sistemos projekto parengimas ir pristatymas	0	0	6	0
Iš viso:		32	0	32	0

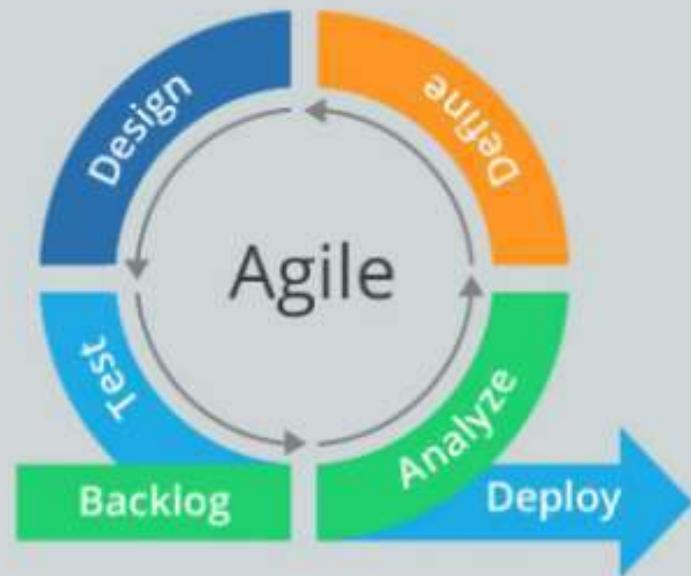
Atsiskaitymų tvarka

- **1 laboratorinio darbo gynimas** - **4 savaitė**. Pateikiama **ataskaita** su sistemos Vizija ir Programinės bei techninės įrangos analize. Perlaikymas - 5 savaitė.
- **2 laboratorinio darbo gynimas** - **9 savaitė**. Pateikiama **ataskaita** su sistemos Funkciniais reikalavimais (panaudojimo atvejais) ir nefunkciniais reikalavimais. Perlaikymas - 10 savaitė.
- **3 laboratorinio darbo gynimas** - **14 savaitė**. Pateikiama **ataskaita** su sistemos Architektūra (komponentų, klasių diagramos) ir Elgsena (sekų diagramos). Perlaikymas sutampa su galutinio projekto ataskaita - 15 savaitė.
- **Inžinerinis projektas / Žodinis iliustruotas pranešimas** - **15 savaitė**. Pateikiama galutinė projekto **ataskaita**, apjungianti ir papildanti 1 ir 2 laboratorinio darbo ataskaitas. Parengiamas trumpas projekto **pristatymas** (PowerPoint) formatu. Perlaikymas - 16 savaitė.
- **Egzaminas** - vyksta sesijos metu. Testas kompiuteriu iš teorinės ir praktinės medžiagos.

Atsiskaitymų rezultatai

Populiarių sistemų kūrimo procesų etapai

Waterfall vs. Agile



Šaltinis: <http://ouriken.com/blog/which-one-is-right-for-you-waterfall-or-agile/>

Modulio rezultatai ir suteikiamos kompetencijos

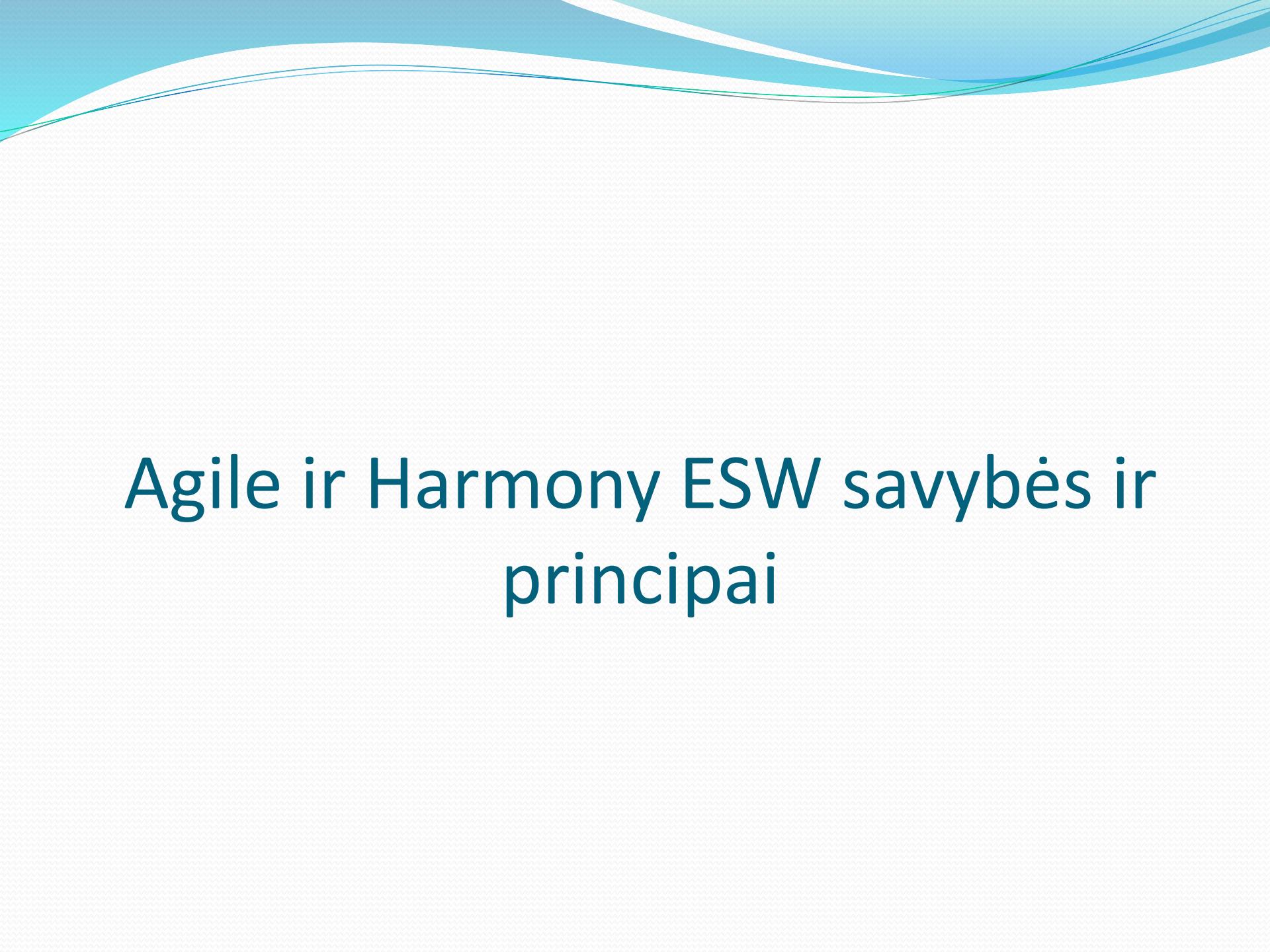
- Išklausę modulį, išsisavinsite UML pagrįstą projektavimo metodiką ir išmoksite:
 - Sudaryti kuriamos sistemos viziją;
 - Atlikti reikiamas programinės ir techninės įrangos analizę;
 - Sudaryti sistemos reikalavimų (funkcinių ir nefunkcinių) specifikaciją;
 - Suprojektuoti sistemos architektūrą ir elgseną (sudaryti atitinkamus UML modelius);
 - Parengti sistemos projekto dokumentaciją.

Kam to reikia?

- Didelio namo nepastatysi be brėžinių ir sudėtingo, daugialypio projekto, kuriuos sukuria architektai ir inžinieriai (nebent gaminsite inkilą ar šuns būdą ☺).
- Didelėms kompiuterinėms sistemoms taip pat reikia „brėžinių“ ir projektinės dokumentacijos, kurias sukuria kompiuterinių **sistemų architektai**.
- Kiekvienas rimtas IT specialistas (net ir paprastas programuotojas) privalo suprasti projektinę dokumentaciją bei ją kurti ir tobulinti.
- Baigiamojo studijų projekto 70 proc. yra sistemos projektas ir ataskaita ☺

Kūrimo procesas, kuri naudosime mokydamiesi projektuoti

- Nors **reikalavimų/analizės ir projektavimo** etapai yra bet kokiamė šiuolaikiniame sistemų kūrimo procese, kad būtų lengviau išsiavinti analizės ir projektavimo principus, naudosime Agile pagrįstą **Harmony ESW** sistemų kūrimo procesą, tinkantį ne tik įprastai programinei įrangai, bet ir įterptinėms sistemoms kurti:
 - ESW – Embedded Software (liet. įterptinių sistemų programinė įranga).



Agile ir Harmony ESW savybės ir
principai

Agile manifestas

- Manifestas tai Agile Alliance viešas pareiškimas, kuri pasirašė 17 žinomų sistemų kūrimo specialistų 2001 metais. <http://agilemanifesto.org/>
- 4 pagrindiniai prioritetai:
 - *Žmonės ir jų sąveika* svarbiau nei procesai ir įrankiai;
 - *Veikianti programinė įranga* svarbiau nei išsami dokumentacija;
 - *Bendradarbiavimas su užsakovu* svarbiau nei kontrakto derybos;
 - *Reagavimas į pasikeitimus* svarbiau nei plano sekimas.

12 Agile principų (1)

- Pagrindinis mūsų prioritetas yra užsakovo reikalavimų tenkinimas, reguliariai ir kuo anksčiau pateikiant vertingą programinę įrangą.
- Reikalavimų pasikeitimai yra sveikintini, net vėlyvuosiuose sistemos kūrimo etapuose. Agile procesai leidžia pasinaudoti pakeitimais tam, kad suteikti užsakovui konkurencinį pranašumą.
- Veikiantį produktą reikia išleisti kiek įmanoma dažniau, kelių savaičių ar mėnesių periodiškumu.
- Viso projekto metu užsakovai ir vykdytojai turi kasdien dirbtį kartu.

12 Agile principų (2)

- Projekte turi dirbt i motyvuoti profesionalai. Tam, kad darbas būtų atliktas, sukurkite salygas, suteikite palaikymą ir pilnai jais pasitikėkite.
- Tiesioginis bendravimas yra pats efektyviausias ir praktiskiausias informacijos apsikeitimo būdas tiek su pačia kūrėjų komanda, tiek komandos viduje.
- Veikiantis produktas – pagrindinis progreso rodiklis.
- Investitoriai, kūrėjai ir vartotojai privalo turėti galimybę nuolat palaikyti pastovų tempą. Agile leidžia pasiekti stabilų kūrimo procesą.

12 Agile principų (3)

- Nuolatinis dėmesys techniniams tobulumui ir projektavimo kokybei padidina projekto lankstumą.
- Paprastumas – menas minimizuoti nereikalingus darbus – yra esminis.
- Patys geriausi reikalavimai, architektūriniai ir techniniai sprendimai gimsta iš saviorganizuojančių komandų.
- Komanda turi reguliariai analizuoti galimus efektyvumo didinimo būdus ir atitinkamai koreguoti savo darbo stilių.

Agile ir įterptinės sistemas

- Agile metodai yra kilę iš ekstremalaus programavimo (angl. extreme programming).
- Tiek Agile, tiek XP yra smarkiai pagrįstos kodo rašymu ir nuolatiniu produkto tobulinimu.
- Dalyko rėmuose mes pritaikysim agile principus įterptinėms (realaus laiko) sistemoms.
- Taip pat naudosime modeliais pagrįsto kūrimo (angl. MDD – model-driven development) principus ir UML kalba.

Kodėl Agile? (1)

- Programinę įrangą kurti yra **sudėtinga**.
- Dauguma egzistuojančių PĮ kūrimo procesų yra sunkiai pakartojami ir dar sunkiau nuspėjami.
- “Nežinomo urvo ištyrimo alegorija”:
 - Įvertinti programinės įrangos kūrimo proceso trukmę ir kainą yra taip pat sudėtinga, kaip pasakyti kiek užtruks nežinomo urvo tyrinėjimas...
 - Tačiau vadybininkai ar užsakovai dažnai primygintinai reikalauja pasakyti konkretų įvertinimą.

Kodėl Agile? (2)

- Kuo toliau, tuo PĮ tampa sudėtingesnė, funkcionalesnė ir sunkiau suvokiamą.
- Pvz. DOS operacinė sistema tilpo į diskelį, o kiek dabar užima Windows 7? ☺
- PĮ kūrimas nėra mechaniškas ir šabloniškas, kiekvienu konkrečiu atveju tai **kūrybinis darbas**.
- Praktiškai PĮ kūrimas tai išradimų procesas.

Šiuolaikinės PĮ ir kūrimo projektų problemos

- Dauguma projektų žlunga nesulaukę pabaigos.
- Dauguma pabaigos sulaukusių projektų būna smarkiai viršiję biudžetą ir projekto laiką.
- Sukurta PĮ turi begalę klaidų ir defektų.
- Dauguma PĮ neatlieka visų numatytyų funkcijų.
- Nors mums normalu perkrauti savo kompiuterius ar telefonus, prieš kelis dešimtmečius tai skambėtų absurdiskai ir nesuprantamai...

Kaip kūrimo procesai bando susidoroti su šiomis problemomis

- Tvarkaraščiai;
 - Valdymo planai;
 - Progreso metrikos (pvz. parašyto kodo eilutės);
 - Kontrolė ir atlikto darbo peržiūra;
 - Progreso ataskaitos ir t.t.
-
- Tačiau kaip taisyklė tai mažai padeda, nes PĮ kūrimas yra nenuspējamas procesas...

Kaip Agile sprendžia PĮ kūrimo problemas

- Agile PĮ kūrimo **metodika** ir **procesas**:
 - skiria daugiausia dėmesio tiems dalykams ir funkcionalumui, kuris **sukuria pridėtinę vertę** ir **naudingas užsakovui**, žymiai mažiau dėmesio skiriant antraeiliams dalykams.
- Agile metodika visiškai atitinka Agile manifesto ideologiją.

Įterptinių realaus laiko sistemų savybės (1)

- PĮ kūrimas yra sudėtingas, o įterptinės PĮ kūrimas dar sudėtingesnis dėl specifinių realaus laiko ar kitokių **nefunkcinių** reikalavimų.
- Reikalinga optimizacija, nes ŠS veikia labai ribotų resursų aplinkoje.
 - Todėl tradicinis PĮ kūrimas nėra tinkamas.
- Dažnai tenka kurti tvarkyklių lygio PĮ, nes įterptinės sistemos valdo įvairią aparatinej įrangą, kuri tiesiog neturi tvarkyklių.
- O kur dar realaus laiko reikalavimai, patikimumas, saugumas ir t.t....

Įterptinių realaus laiko sistemų savybės (2)

- Vienas iš esminių skirtumų tarp paprastos PĮ ir IĮPĮ yra tai, kad paprasta PĮ kuriama toje pat platformoje, kurioje ir veiks.
- Įterptinės sistemos yra kuriamos visai kitoje platformoje nei diegiamos ir vėliau veikia:
 - Sunku derinti ir testuoti;
 - IĮS derinimo priemonės dažnai kur kas primityvesnės nei įprastos;
- Tradicines sistemas net ir po sukūrimo ir paleidimo galima tobulinti, taisyti klaidas.
- Deja, įterptinės sistemos privalo veikti kaip ir numatyta iškart, kai tik yra įdiegiamos ir paleidžiamos.

Kritinės sistemos ir jų patikimumas

- Patikimumas (angl. reliability) – kiek laiko sistema teikia savo paslaugas (procentai).
- Atsparumas klaidoms ir klaidos toleravimo laikas (angl. fault tolerance time) – kiek laiko sistema gali būti klaidos būsenoje, kol neįvyksta incidentas.

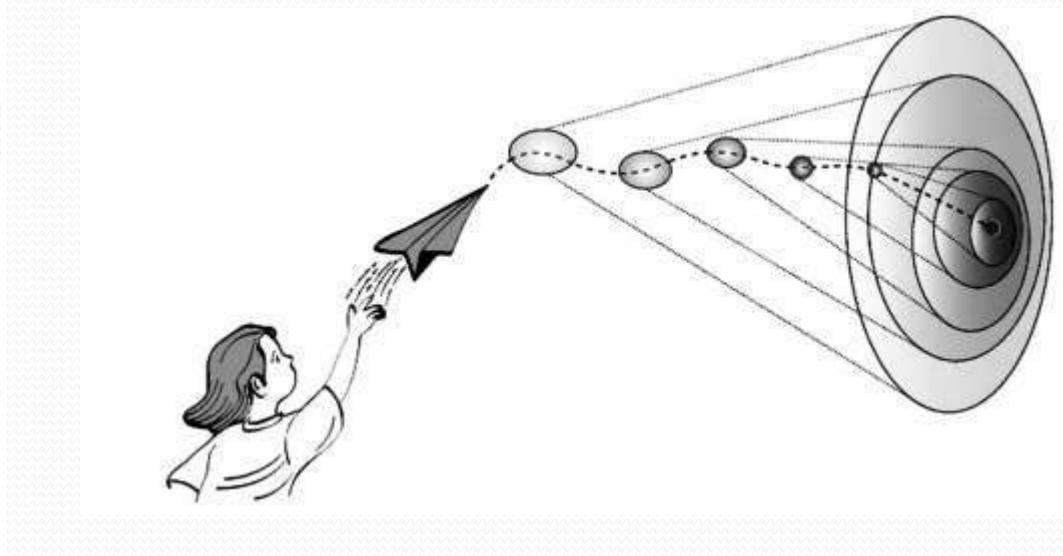
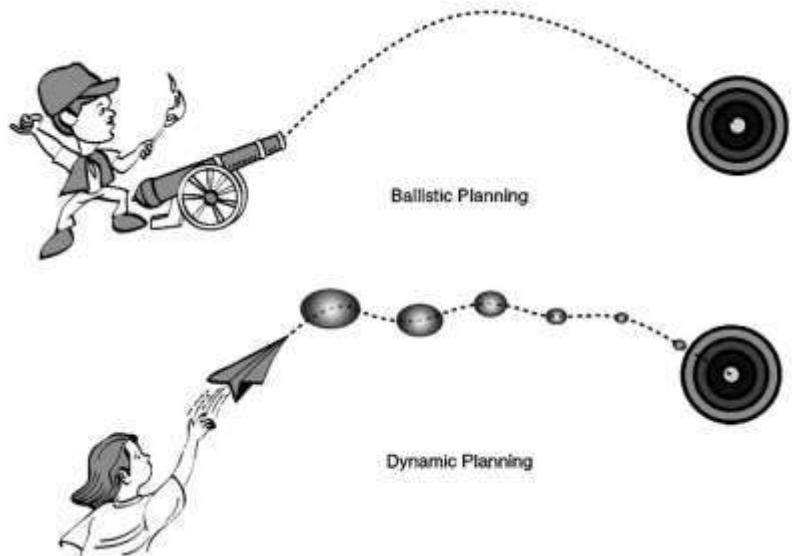
Agile metodų tikslas

- Pagrindinis tikslas – sukurti **veikiančią** programinę įrangą, kuri atitinka užsakovo ir vartotojų lūkesčius.
- Dokumentacija, nuolatinis bendravimas, darbo grafikai, vertinimo kriterijai ir kiti dalykai yra svarbūs ir reikalingi, bet jie yra antraeiliai ir turi būti atliekami tik tam, kad įvykdyti pagrindinį tikslą.
- Kūrimas vyksta iteratyviai. Sistemos versijos vis papildomos nauju funkcionalumu, kol nepasiekiamas galutinis rezultatas.

Agile metodų privalumai

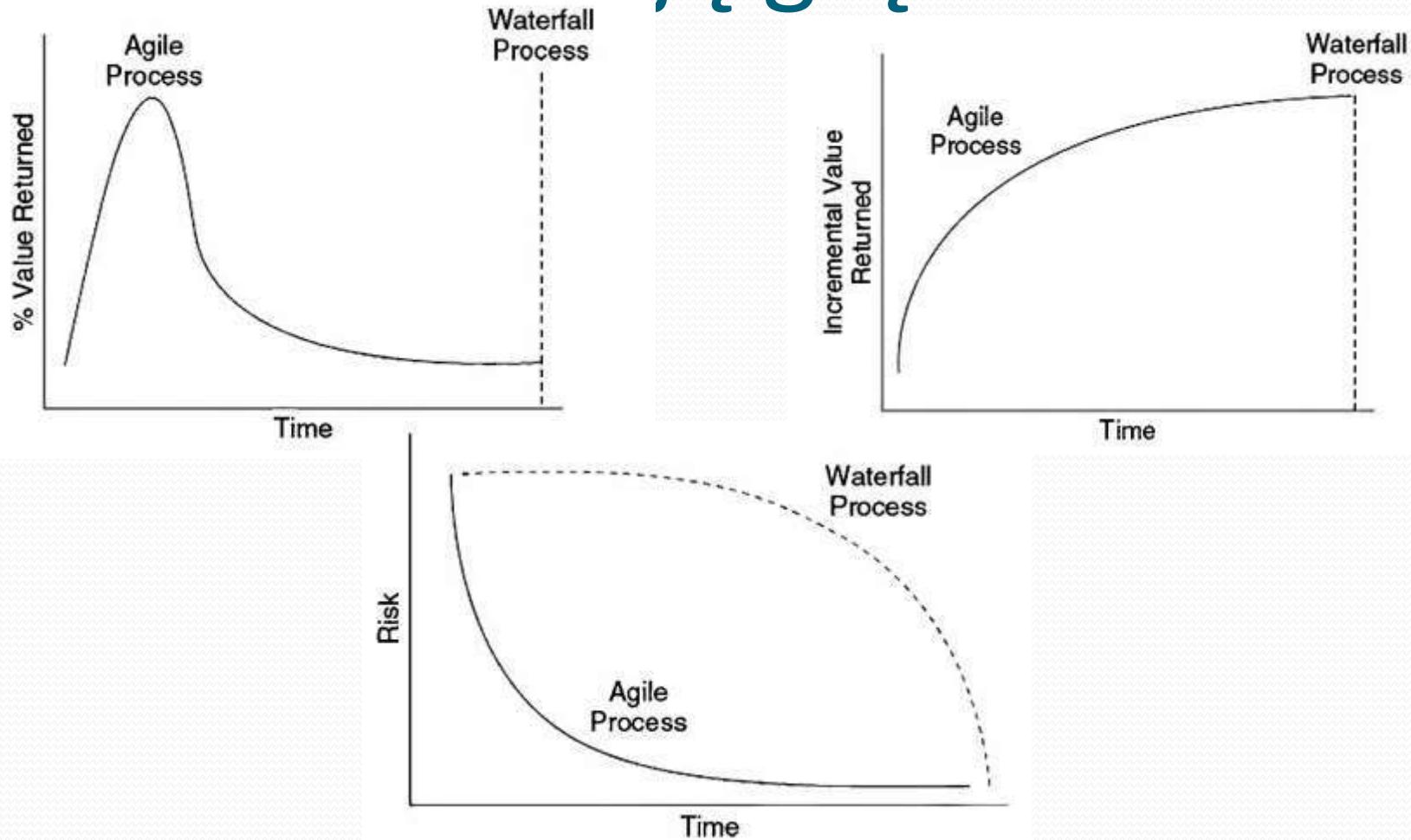
- Greitas projekto reikalavimų ir reikalingų technologijų išgryninimas;
- Greita investicijų grąža;
- Patenkinti užsakovai ir investitoriai;
- Geresnė proceso kontrolė;
- Greitas reagavimas į pasikeitimus;
- Ankstyvas ir didelis projekto rizikų sumažinimas;
- Efektyvus aukšto kokybės lygio kūrimas.

Greitas reikalavimų ir reikalingų technologijų išgryninimas



Šaltinis: Bruce Powel Douglass. Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development. Addison Wesley, 2009, 560 p. ISBN-10: 0321545494

Greita investicijų graža



Šaltinis: Bruce Powel Douglass. Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development. Addison Wesley, 2009, 560 p. ISBN-10: 0321545494

Patenkinti investoriai

- Agile metodai leidžia pateikti investoriams, užsakovams ir vartotojams funkcionuojančią sistemą labai anksti.
- Vartotojai gali ir net privalo dalyvauti kiekvienos papildytos ir pataisytos programos versijos validavime.
- Funkcionalumas realizuojamas iteratyviai, naudojant vieną iš strategijų:
 - Pirmiausiai rizikingiausias funkcionalumas;
 - Pirmiausiai svarbiausias/kritinis funkcionalumas;
 - Pirmiausiai infrastruktūra/karkasas;
 - Pirmiausiai geriausiai suprantamas funkcionalumas.

Geresnė proceso kontrolė (1)

- Daugumą projektų nepavyksta suvaldyti ($>80\%$).
 - Nepakanka suplanuoti – būtina **nuolat stebėti ir vertinti**, ar laikomasi plano.
 - Kodėl stebėti?
 - Kad pastebėti nukrypimus nuo plano ir persiorganizuoti.
 - Ką stebėti?
 - Kaina, laikas, resursai, klaidos.

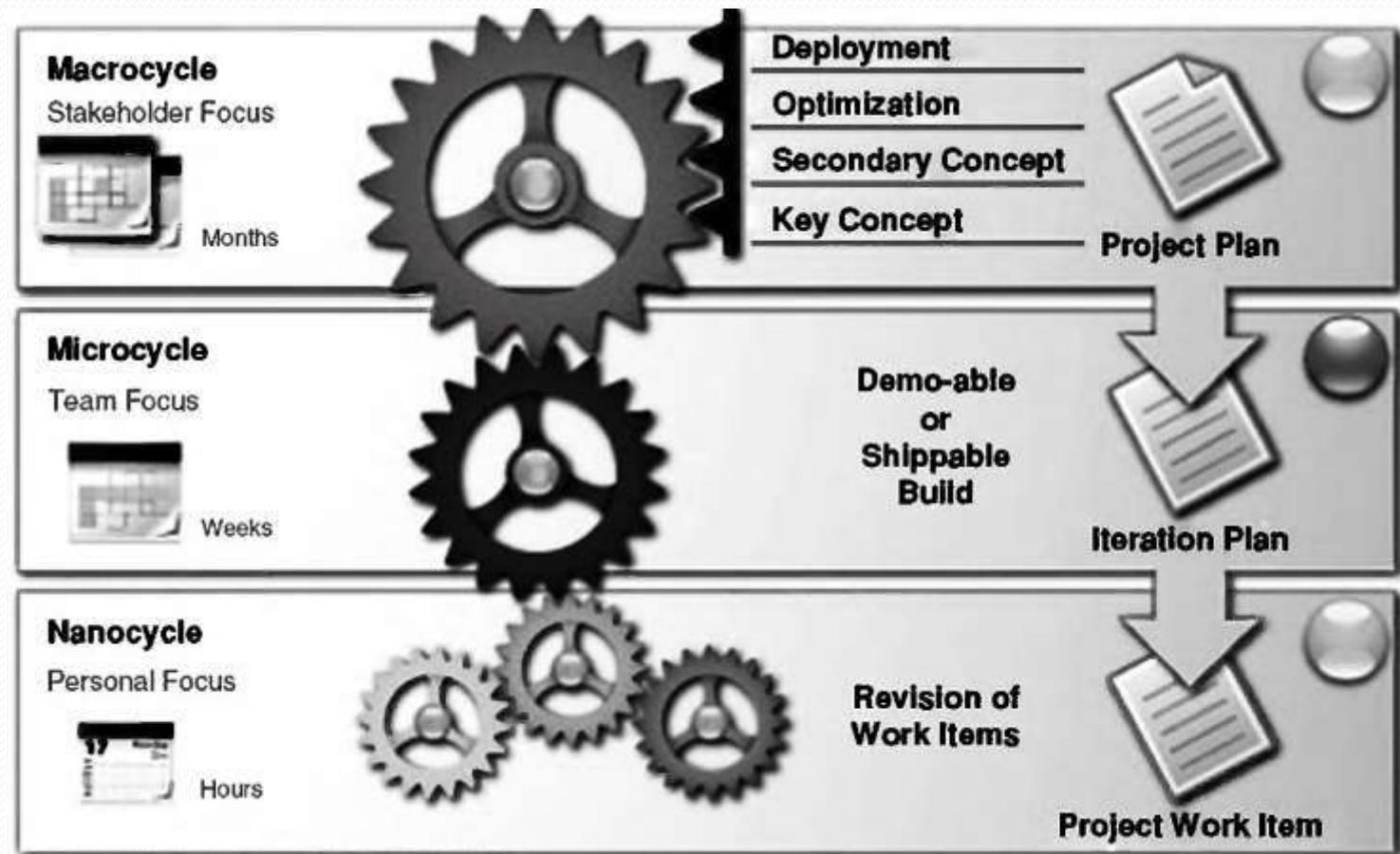
Geresnė proceso kontrolė (2)

- Kaip stebeti?
 - Jeigu matuojam išbaigtumą, stebim kiek PĮ reikalavimų įvykdyta, o ne skaičiuojam kodo eilutes.
 - Jeigu matuojam kokybę, tikrinam klaidų santykį, o ne ištaisytų klaidų kiekį ir t.t.
 - Stebim tuos dydžius, kurie tiesiogiai koreliuoja su mūsų projekto tikslu ir planu.
- Agile suteikia geriausią projekto vertinimo metriką – veikiantį ir validuotą funkcionalumą – anksti ir dažnai.

Greitas reagavimas į pasikeitimus

- Planai dažniausiai (tiesą sakant visada) griūna, nes neįmanoma visko numatyti.
 - Dažnai trūksta informacijos, gali nutrūkti įrangos gamyba, atsiranda daug klaidų, nuolat keičiasi reikalavimai, trūksta kvalifikuotų specialistų ir t.t.
- Vietoje “plan-and-pray” naudojame “plan-track-replan” metodą.
- Agile metodai iškart daro prielaidą, kad planai nėra korektiški ir leidžia juos koreguoti minimaliom sąnaudom ir nuostoliais.

Harmony/ESW proceso ciklai (1)



Harmony/ESW proceso ciklai (2)

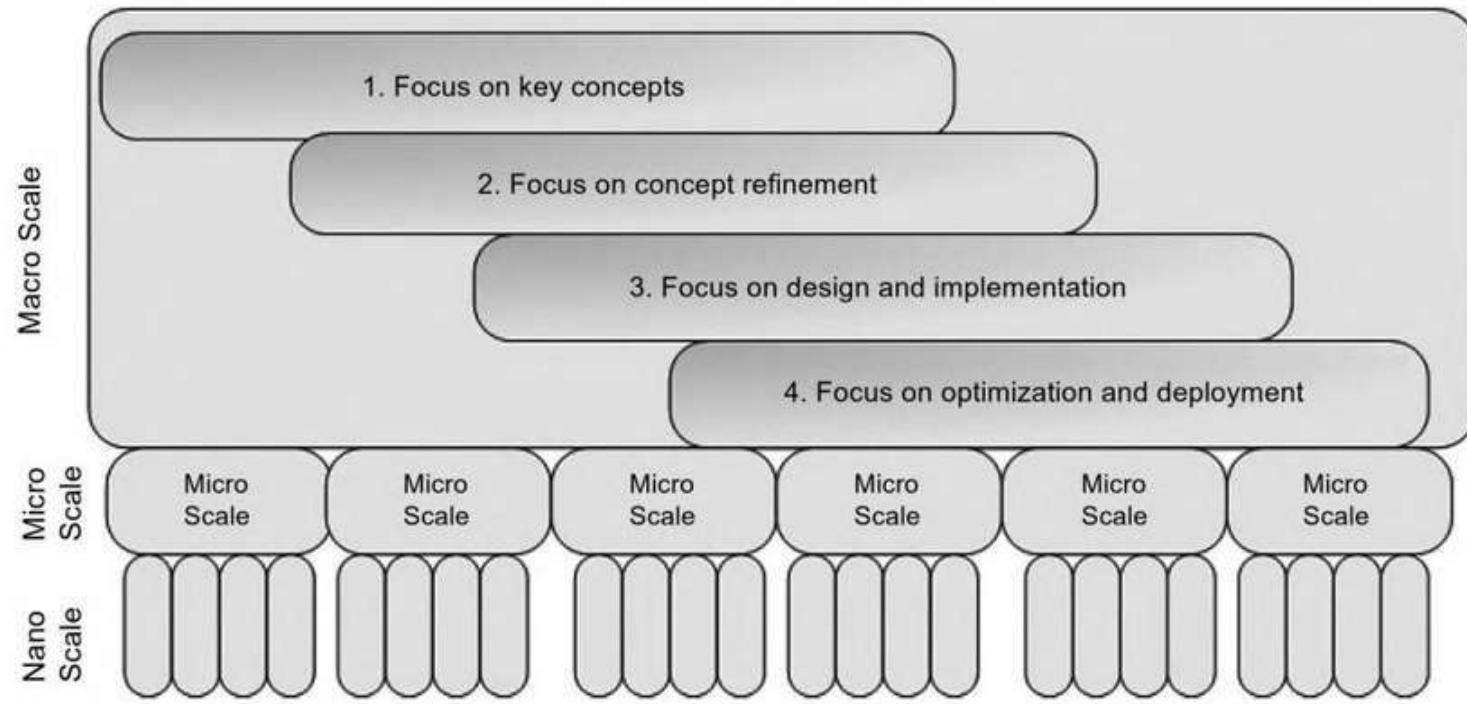


Figure 2.2
Harmony process time frames.

Harmony/ESW proceso ciklai (3)

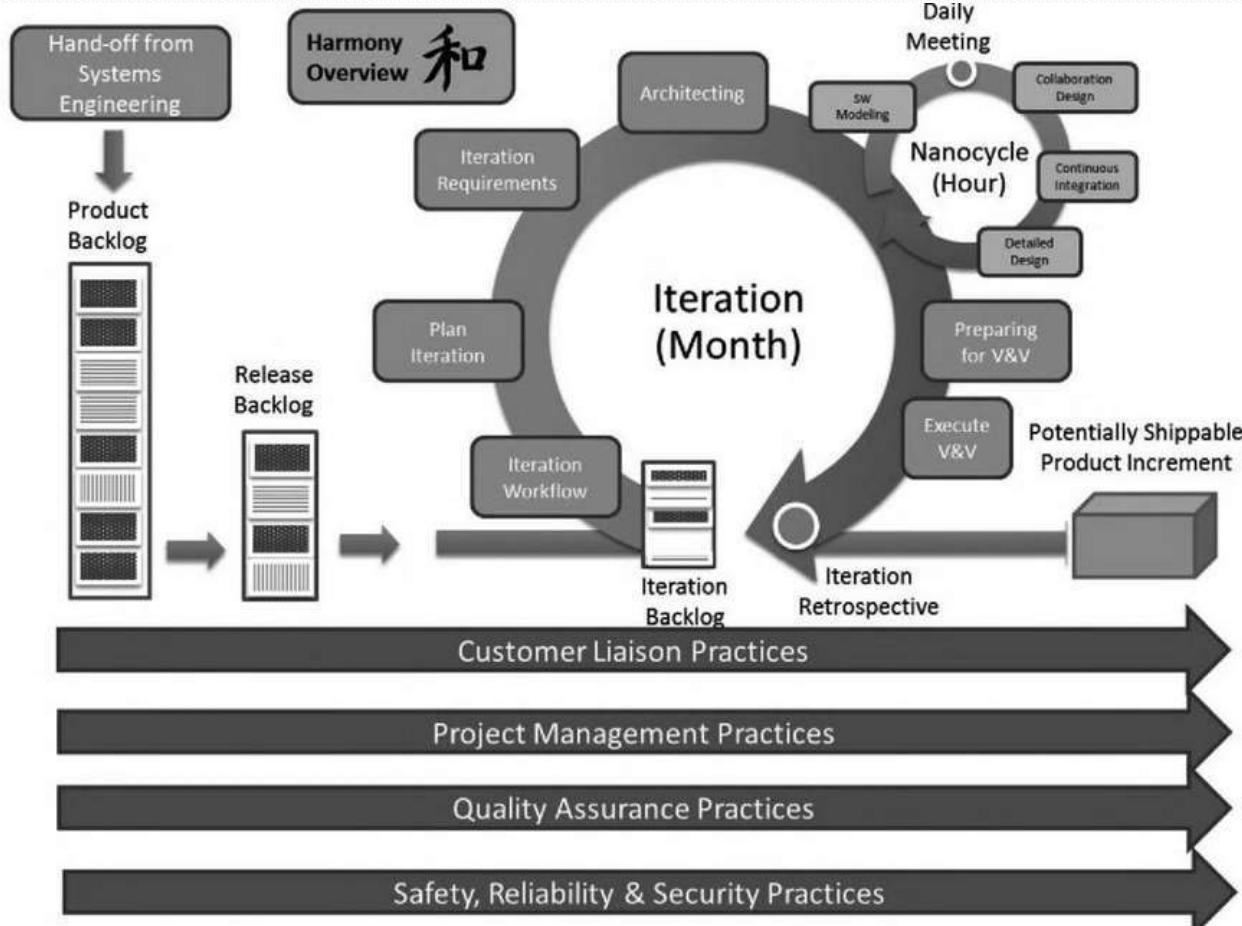


Figure 2.11
Harmony software process: agile view.

Ankstyvas ir didelis projekto rizikų sumažinimas

- Rizikų negalima ignoruoti.
- Rizikas būtina valdyti!
 - Rizikų sąrašas
 - Rizikų valdymo planas
- Jokių “tikėkimės, kad veiks” ☺

Efektyvus aukšto kokybės lygio kūrimas

- Skirtingai nuo tradicinių ne iteratyvių metodų (pvz. krioklio),
- Agile užtikrina aukštą kokybę **nuolatinio vykdymo, nuolatinio integravimo ir nuolatinio testavimo** priemonėmis.
- Šias veiklas reikia pradėti kiek įmanoma anksčiau!
- Agile principas:
 - Geriausias būdas neturėti klaidų sistemoje yra ne nuolatinis testavimas, bet tų klaidų nedarymas ☺

Kuo Agile skiriasi nuo tradicinių metodų?

- Nėra labai griežto pradinio suplanavimo;
- Testais/prototipais pagrįstas kūrimas;
- Atsparus pokyčiams.

Agile planavimas

- Neįmanoma suplanuoti kelerių metų projekto minučių tikslumu. Tai absurdė Šaka ☺
- Deja krioklio ar panašūs metodai verčia mus užsiimti tais absurdais...
- Visuose (absoliučiai visuose) projektuose įvyksta nenumatyta dalyku ir planai keičiasi ir griūna.
- Planuoti vis vien reikia visais atvejais, tik Agile planavimas yra lankstesnis, koncentruojantis į mažesnius vienos iteracijos tikslus.
- Iškart susitaikoma su tuo, kad planai yra netikslūs ir juos reiks koreguoti.
- **Dinaminis planavimas** – pagrindinis Agile metodų išskirtinumas.

Depth-First kūrimo principas

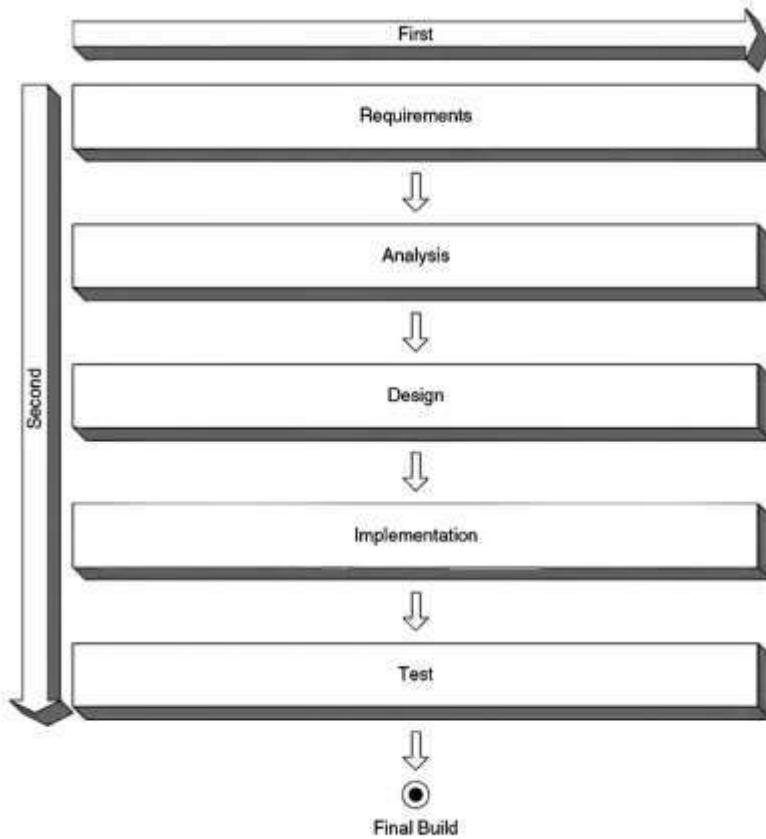


Figure 1.10 Waterfall lifecycle

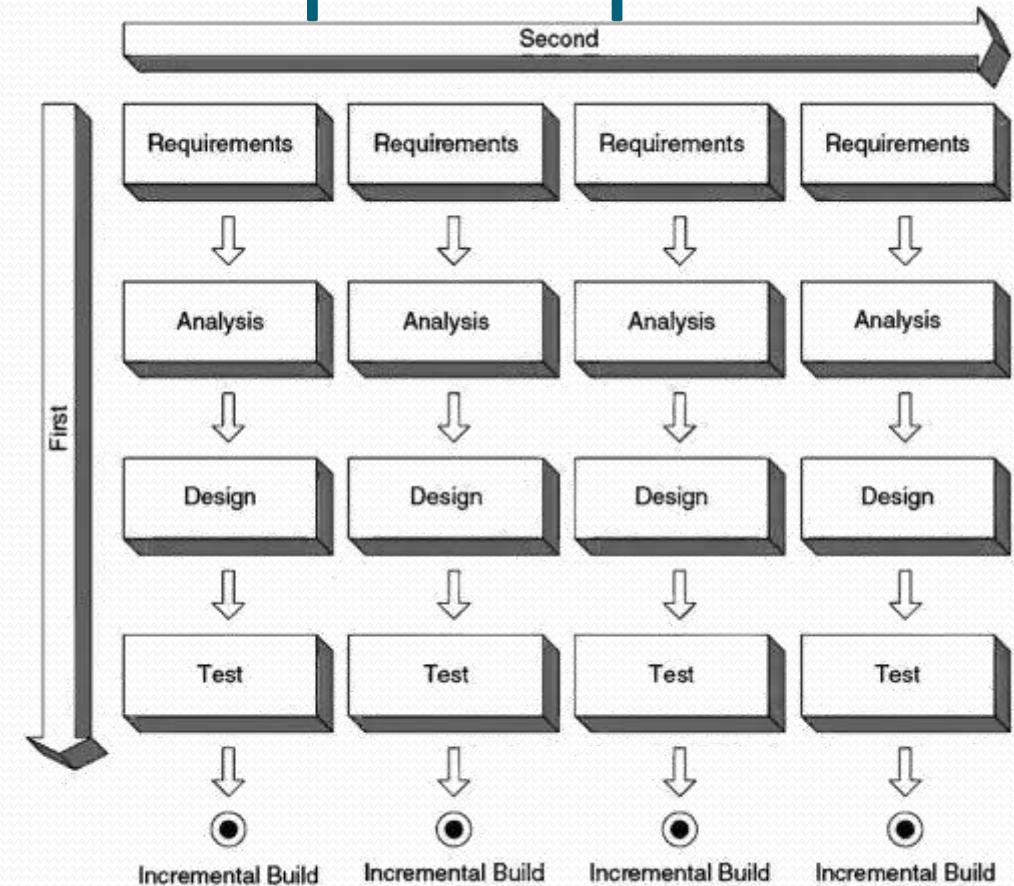


Figure 1.11 Incremental spiral lifecycle

Spiralinis kūrimo procesas

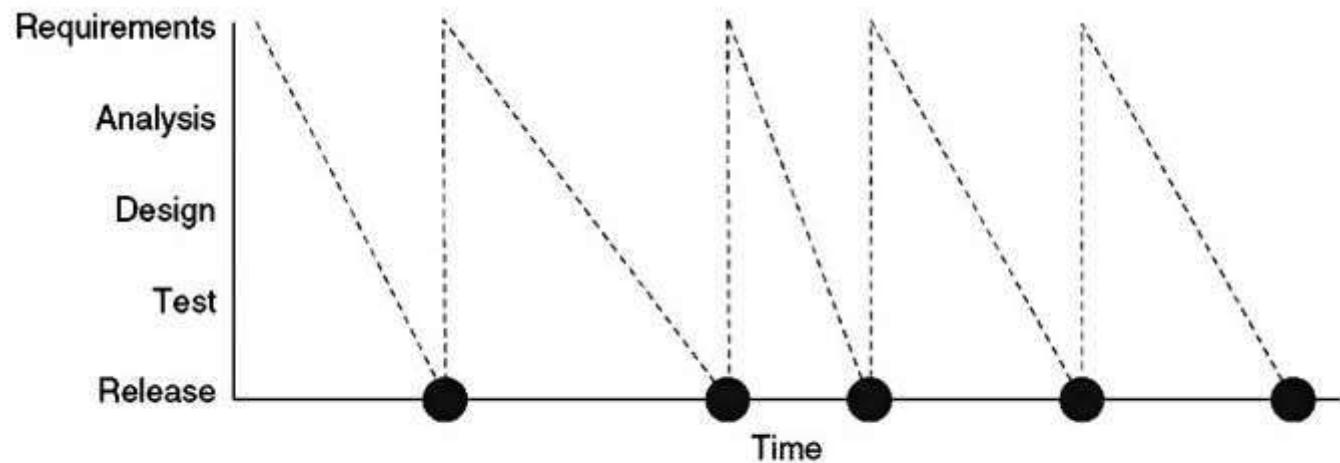


Figure 1.12 *Unrolling the spiral*

Testais pagrīstas kūrimas

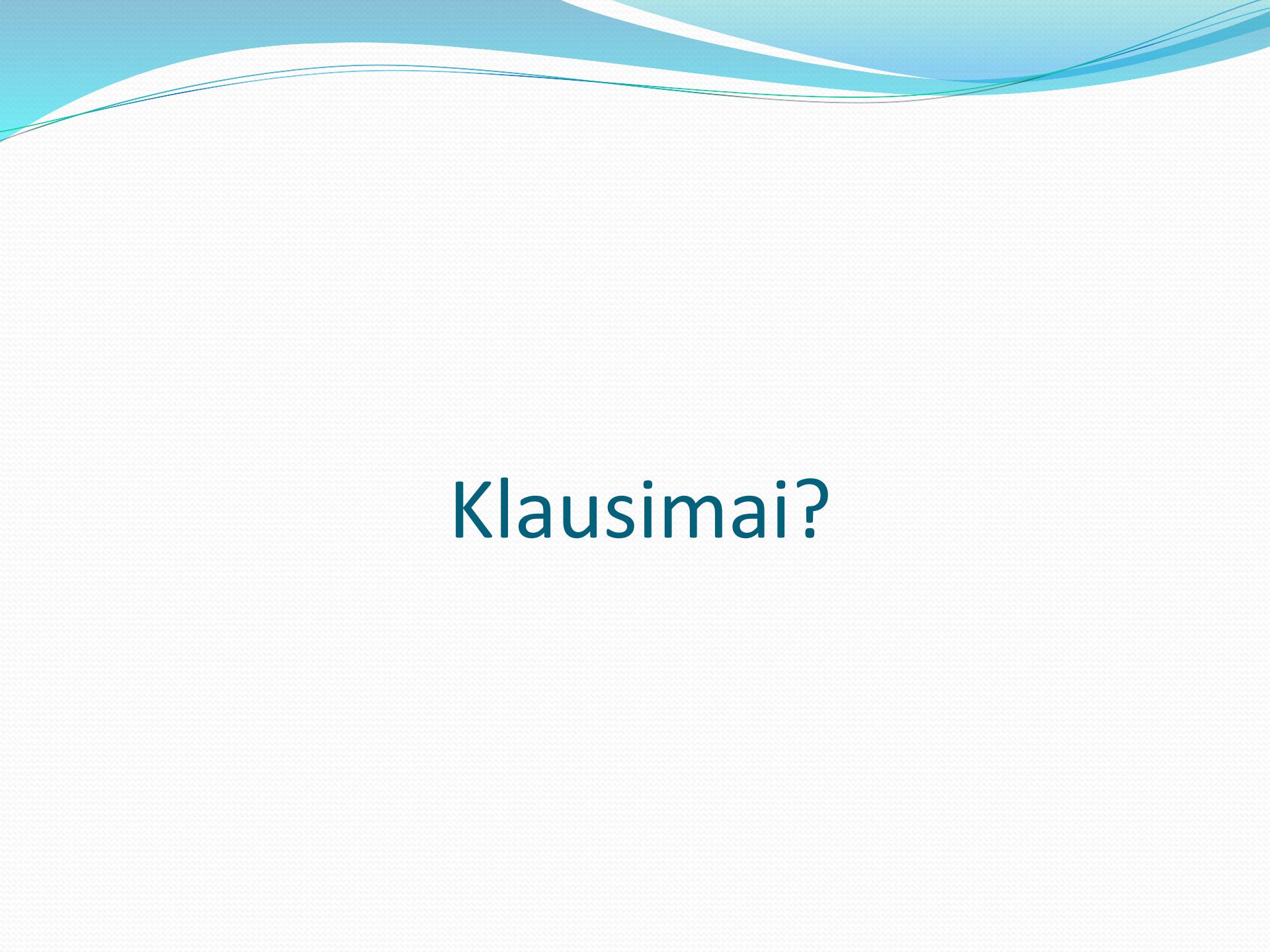
- Testavimas nėra tai, ką mes darome projekto pabaigoje, kad uždėti varnelę ☺
- Testavimas privalo būti vykdomas nuolat, įsitikinant, kad kuriamas produktas atitinka keliamus reikalavimus.
- Idealiu atveju, testavimo programos (angl. test cases) kuriamos prieš paties produkto kūrimą.

Atsparumas pokyčiams

- Keistis gali:
 - Situacija rinkoje;
 - Technologijos;
 - Konkurentų produktai ir paslaugos;
 - Kūrimo priemonės;
- Neseniai ir nekilnojamo turto rinka leido užsidirbtį didžiulius pinigus per labai trumpą laiką, tačiau jei tuo metu jūs padarėte prielaidą, kad taip bus amžinai ir sukūrėte ilgalaikę verslo planą ... šiandien mažų mažiausiai turite didelių bėdų...

Kaip įgyvendinti atsparumą pokyčiams?

- Reikia kurti tokius planus, kuriuos būtų galima nuolat
 - sekti
 - vertinti
- tam, kad kuo greičiau nustatyti, kad kažkas pasikeitė ir
 - identifikuoti pasikeitusias sąlygas
 - išanalizuoti pasikeitimus
 - atlikti plano pakeitimus ir adaptuoti jį prie naujų sąlygų
- Pasikeitimai savaime nėra baisūs, jeigu mes iš anksto planuojame ir numatome pokyčius bei įvertiname rizikas.



Klausimai?

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Sistemos vizijos (konceptcijos) dokumentas

Prof. dr. Agnus Liutkevičius,

Kompiuterių katedra

Studentų g. 50-202, tel. 300394

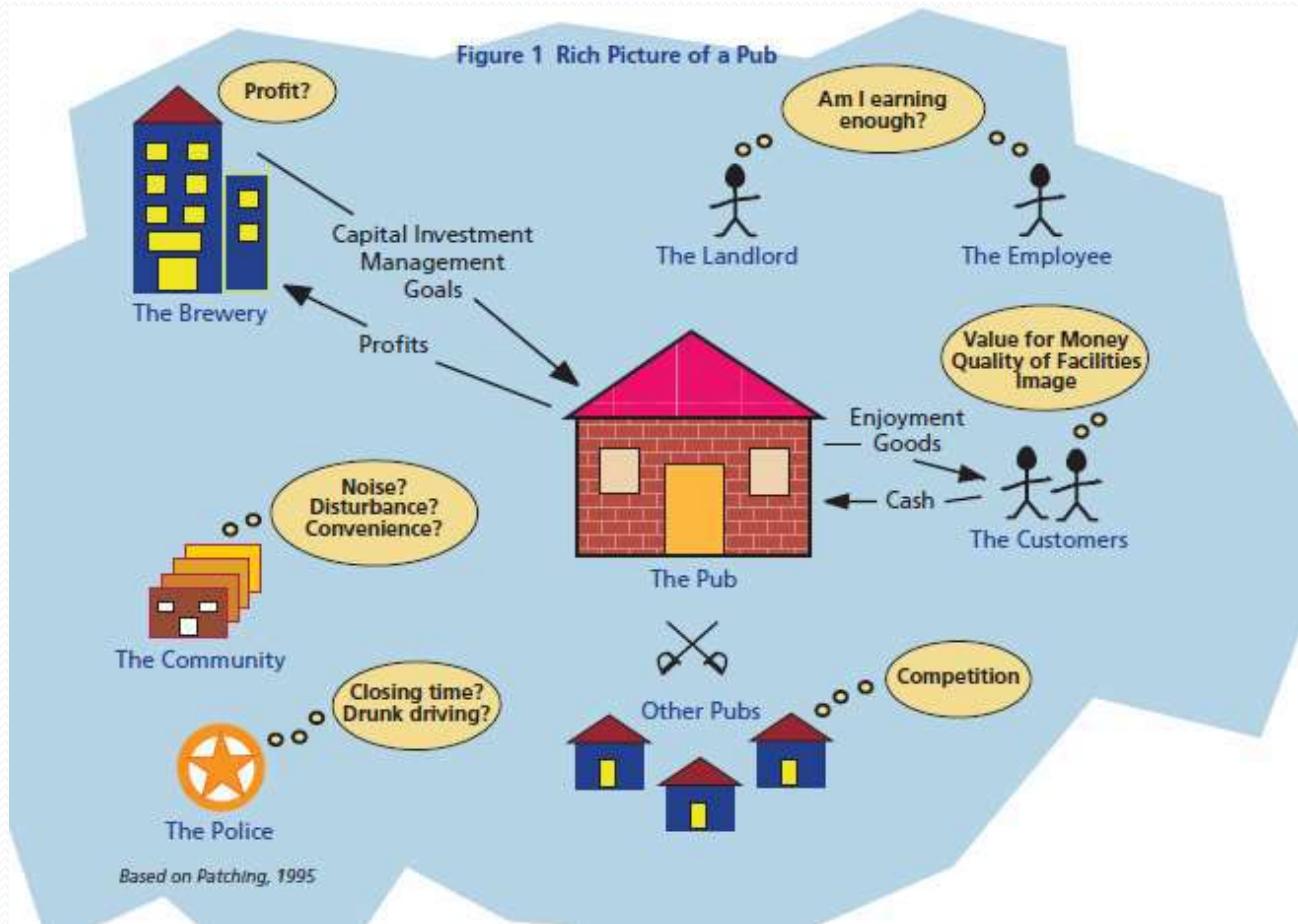
Rich Picture priemonė

- Rich Picture – tai neformalus ir vaizdus (angl. cartoon-like) paveikslas, kurio tikslas sudominti ir parodyti kuriamos sistemos idėją ir naudą investuotojams ir užsakovams.
- RP atsirado kaip Soft Systems Methodology dalis.
- Pagrindinis SSM tikslas yra suprasti sistemas, žiūrint iš sistemos aktorių (naudotojų) pozicijų.
- RP paprastai sudaromas apklausiant ir bendraujant su suinteresuotais asmenimis (tikslinėmis grupėmis).
- RP gali būti sudaromas ir pačių idėjos sumanytojų tam, kad pateikti šią idėją potencialiems investuotojams ir juos sudominti.
 - Pvz. Bakalaurinio darbo vizija skirta darbo vadovams tam, kad jie suprastų, kas norima padaryti, o taip pat patikėtų šia idėja.

Kaip turi atrodyti vizijos paveikslas?

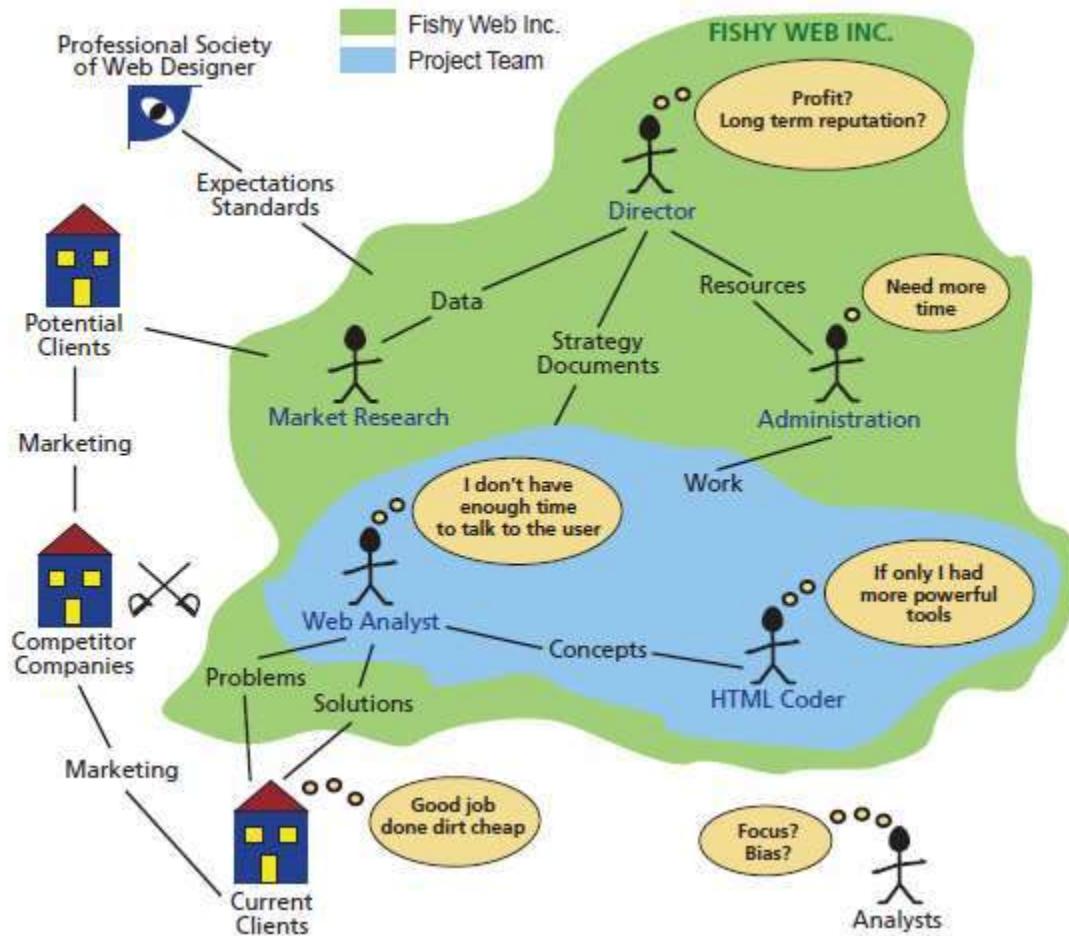
- Turi būti pakankamai detalus – turi matyti visi proceso dalyviai ir pagrindinės kuriamos sistemas dalys.
- Neperkrautas techninėmis detaliemis – turi suprasti ir ne IT specialistas.
- Iš vizijos turi aiškiai matyti norimi kompiuterizuoti procesai ir nauda, kurią gaus užsakovas, vartotojai, ar investuotojas.
- Nėra vienintelio ir teisingo būdo sudaryti vizijos paveikslą – viskas priklauso nuo situacijos ir vizijos paskirties.

Pavyzdys 1: Aludė



Pavyzdys 2: Web dizaino kompanija

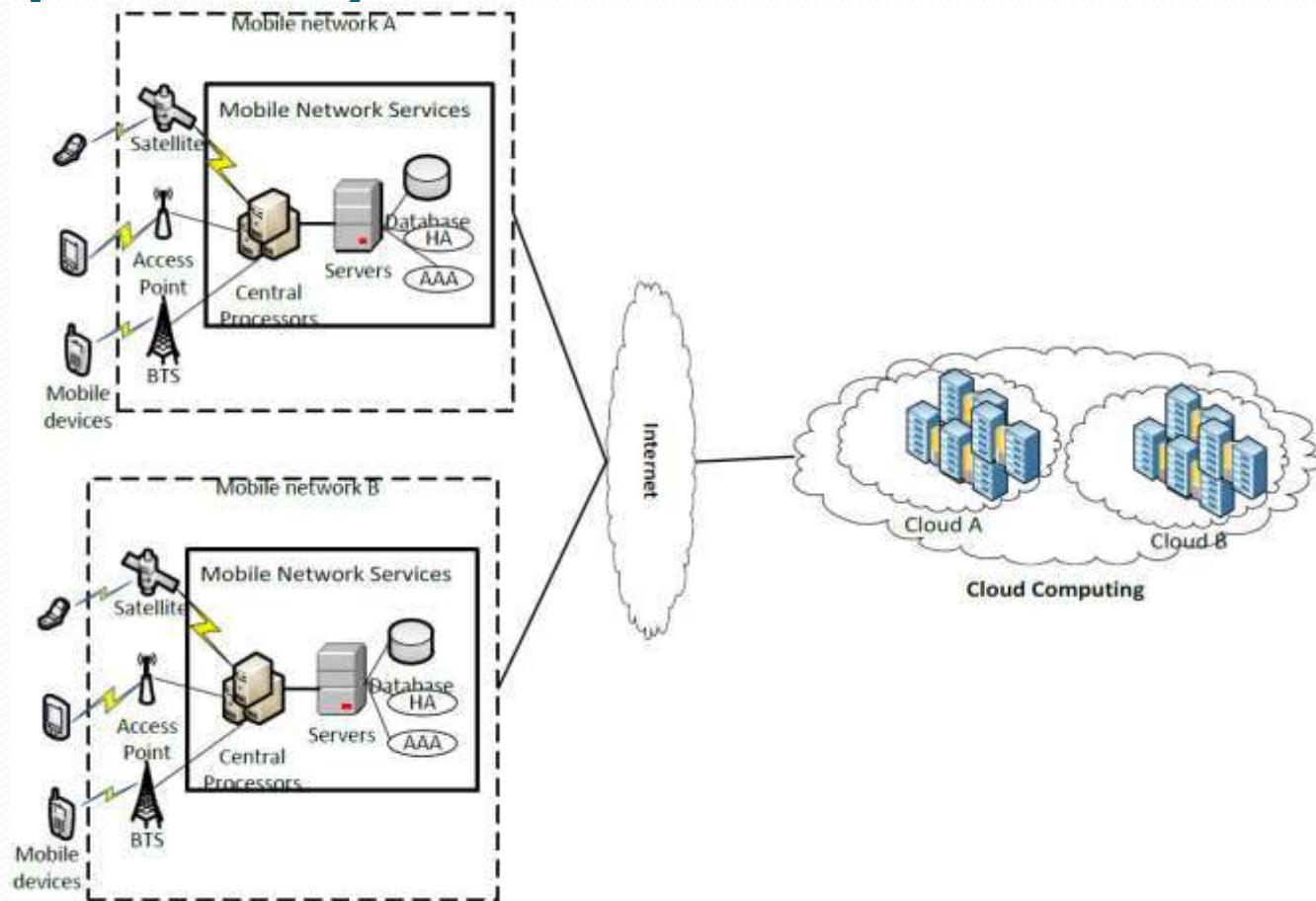
Figure 2 Rich Picture of Web Design Consultancy



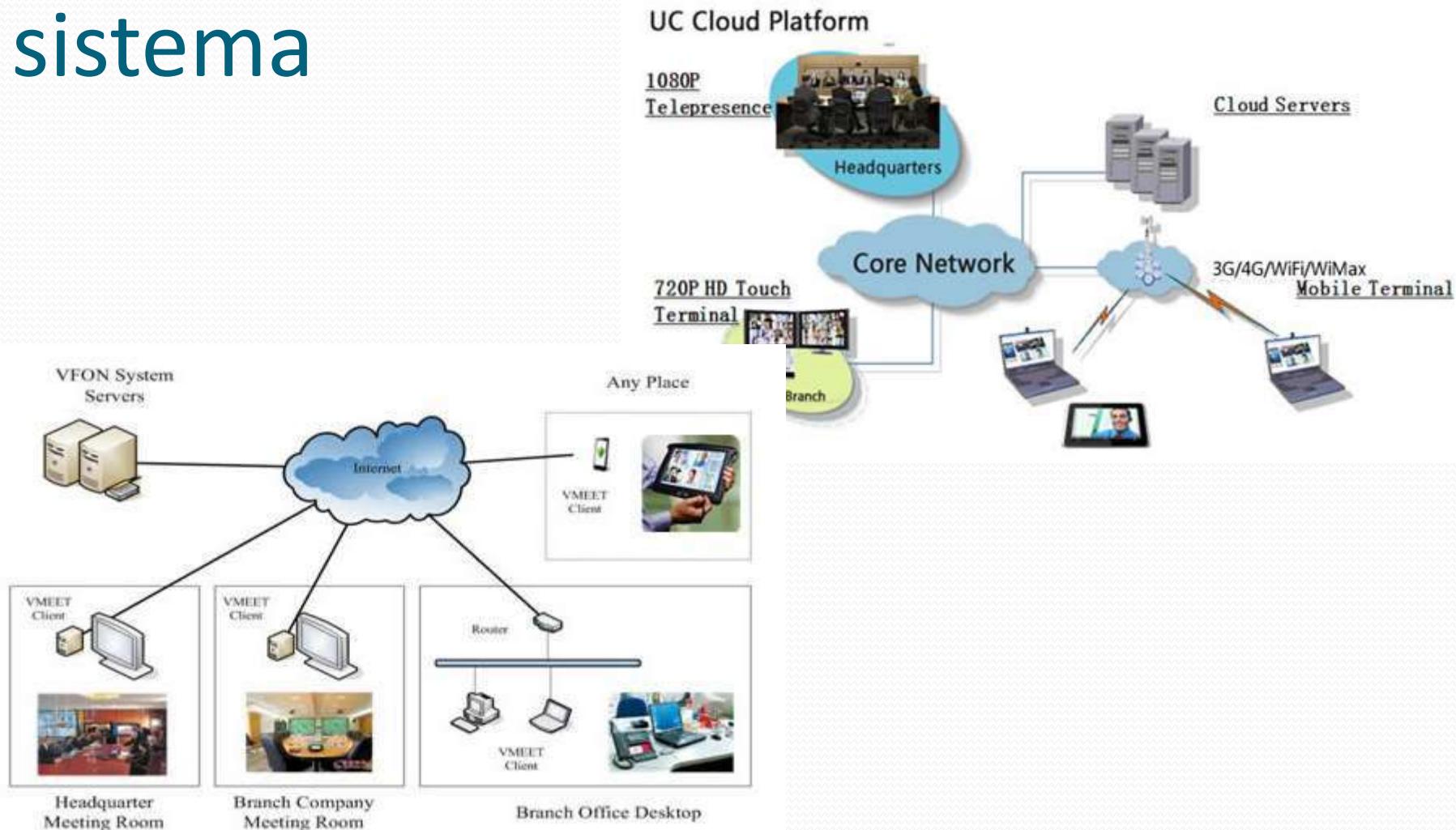
Pavyzdys 3: Automobilio saugaus vairavimo sistema



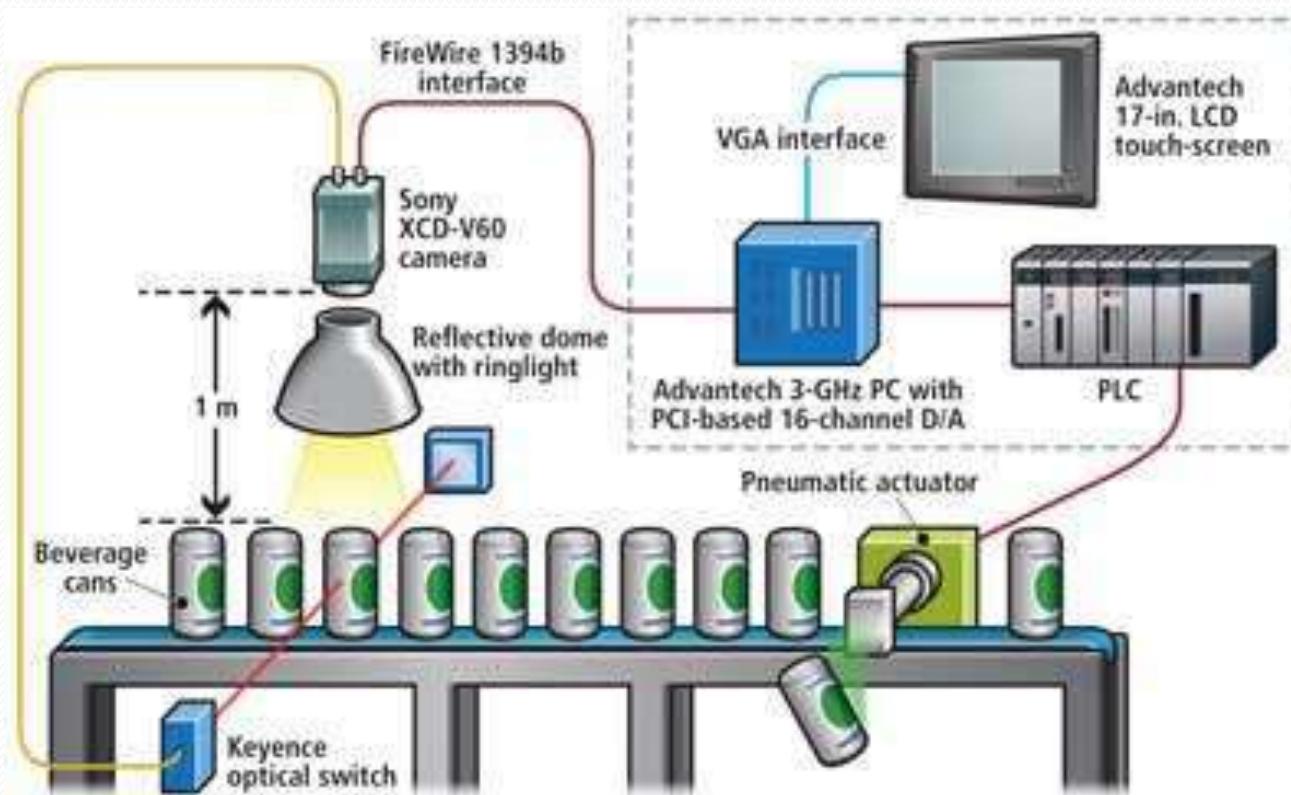
Pavyzdys 4: Debesų kompiuterijos sistema



Pavyzdys 5: Video konferencijų sistema



Pavyzdys 6: Skardinių rūšiavimo sistema

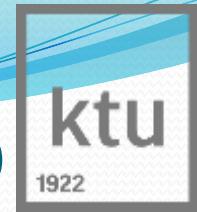


Šaltinis: <http://www.vision-systems.com/articles/print/volume-17/issue-5/features/vision-system-clears-codes-on-cans.html>

Trys pagrindiniai vizijos RP komponentai

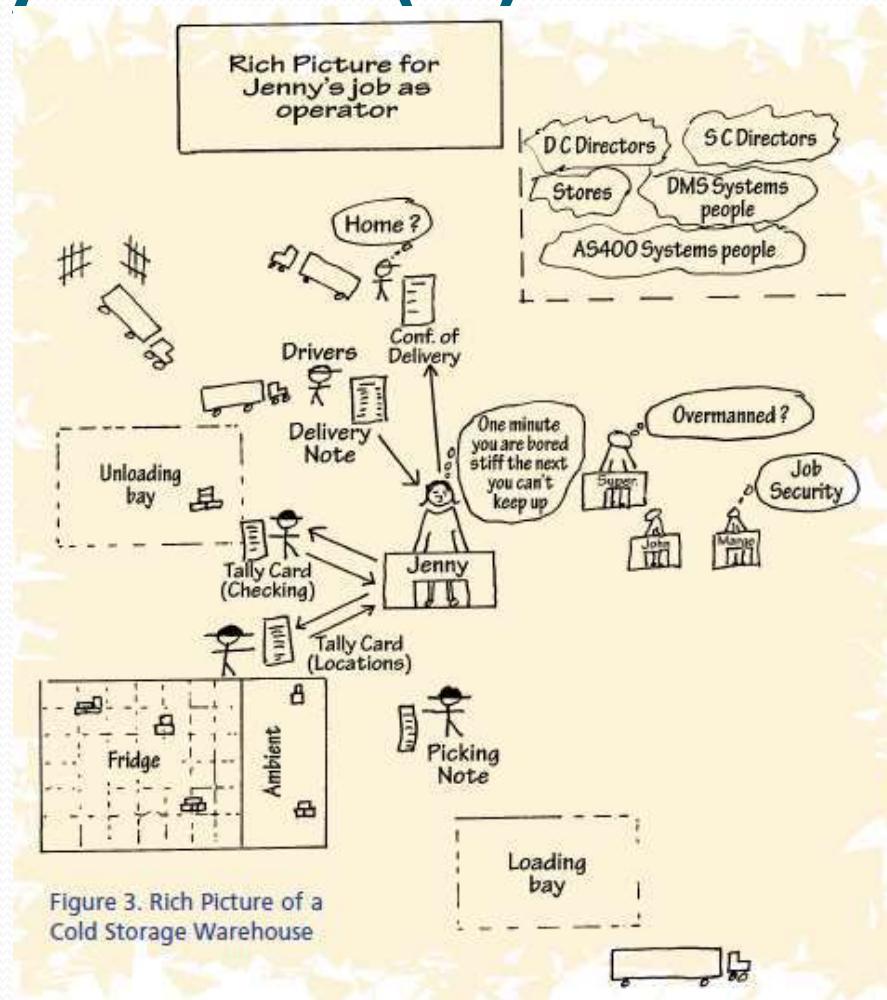
- **Struktūra** (angl. structure):
 - Organizacijos hierarchija, geografinės vietas, įvairi įranga (tame tarpe ir SW ir HW) ir t.t.
- **Procesas** (angl. process):
 - Transformacijos, kurios vyksta atliekant pavaizduotas veiklas. Apima daiktus, dokumentus, duomenis.
- **Interesai** (angl. concern (rūpestis), issue (problema)):
 - Kiekvienas vizijoje pavaizduotas žmogus (aktorius), turi savo interesus (problemas, rūpesčius).
- **Papildomai:**
 - Naudokite kalbą, kurią naudoja žmonės, kuriems skirta vizija.
 - Atsižvelkite į situaciją – vizija techniniams specialistams iš esmės skirsis nuo tos, kuri skirta vadybininkams.

Vizija – pradinis projekto gimimo taškas

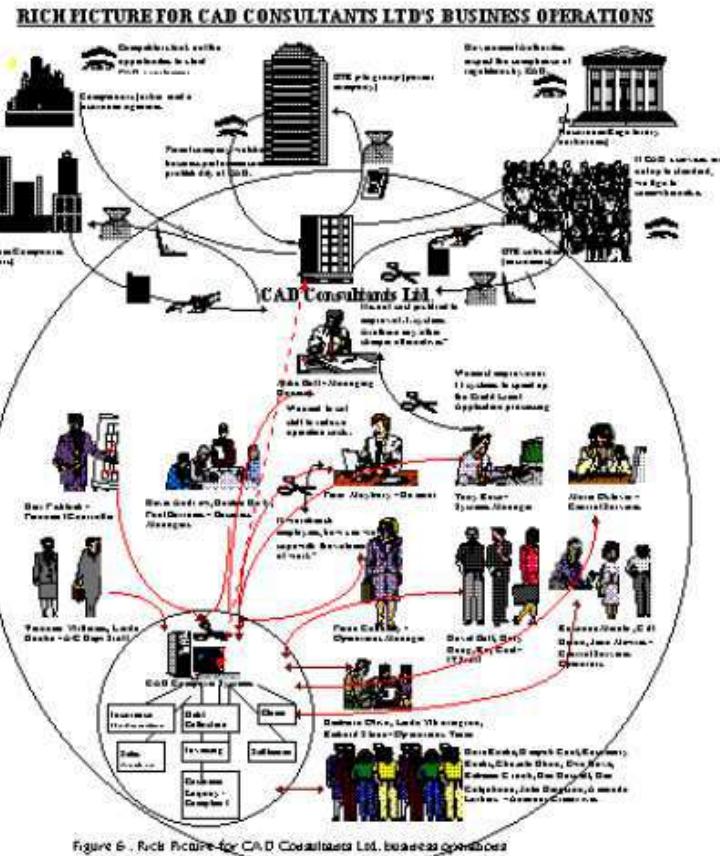
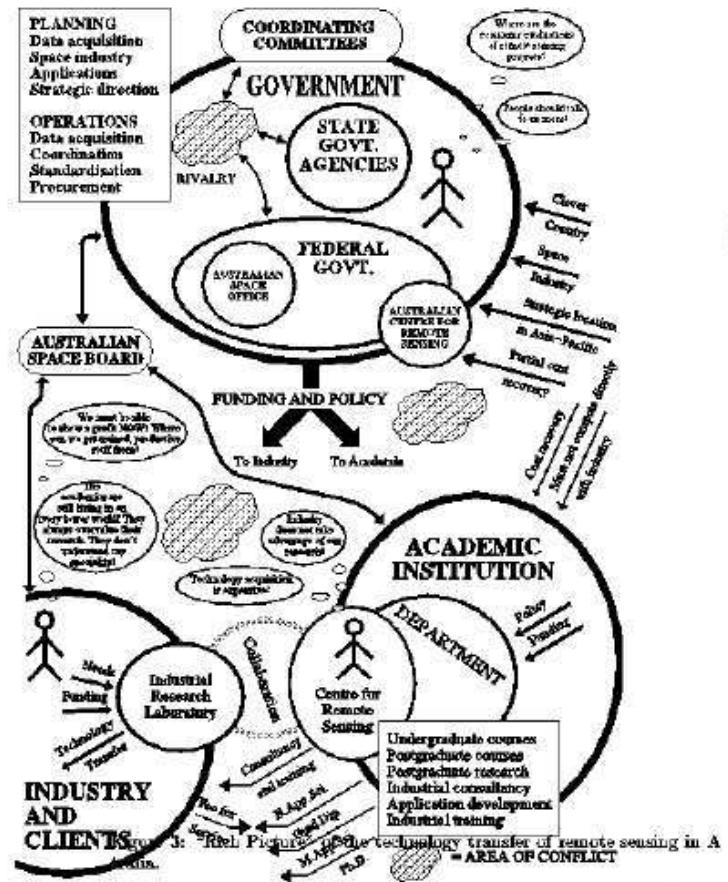


- Vizija dažnai gimsta pradinio kolektyvinio idėjos/problemos svarstymo metu (angl. brainstorming).
- Labai patogu naudoti RP vizijos aprašymui tokių diskusijų metu. Vėliau galima idėją išgryninti.
- Visi diskusijos dalyviai aiškiai mato problemą ir jos prendimo būdus.
- Vienas paveikslas tūkstantį kartų geriau nei šimtai eilučių teksto.
- RP lengvai supranta tiek IT specialistai, tiek vadybininkai, tiek užsakovai, tiek vartotojai – nėra jokios specifinės notacijos ir nereikia papildomų žinių.

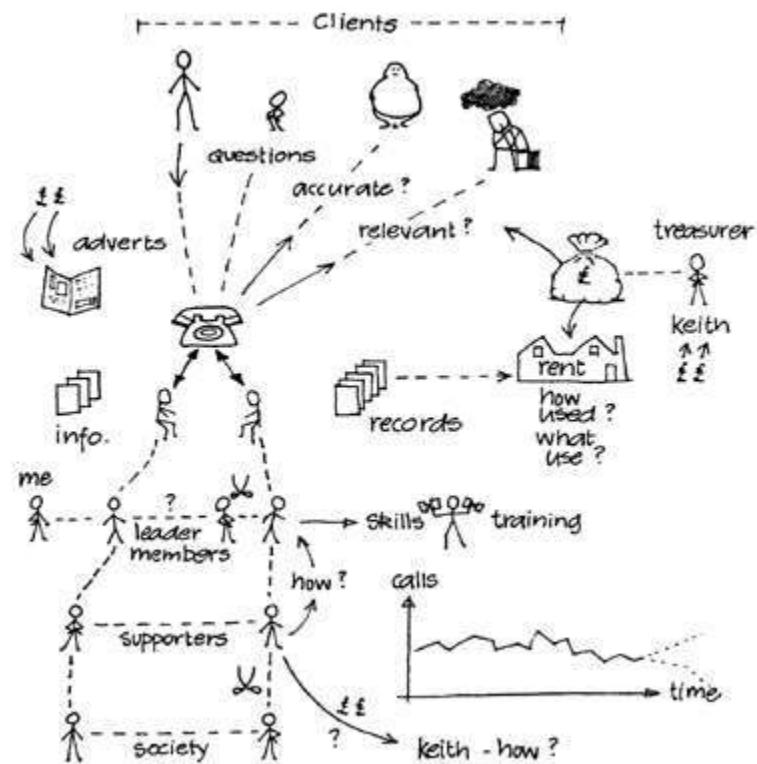
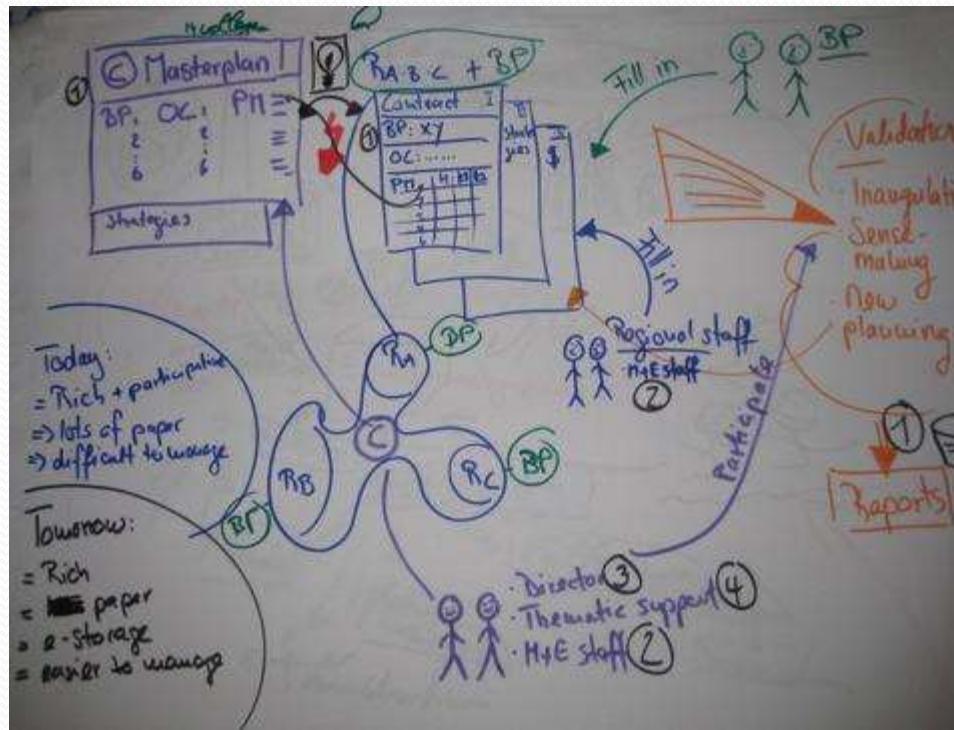
Kiti pavyzdžiai (1)



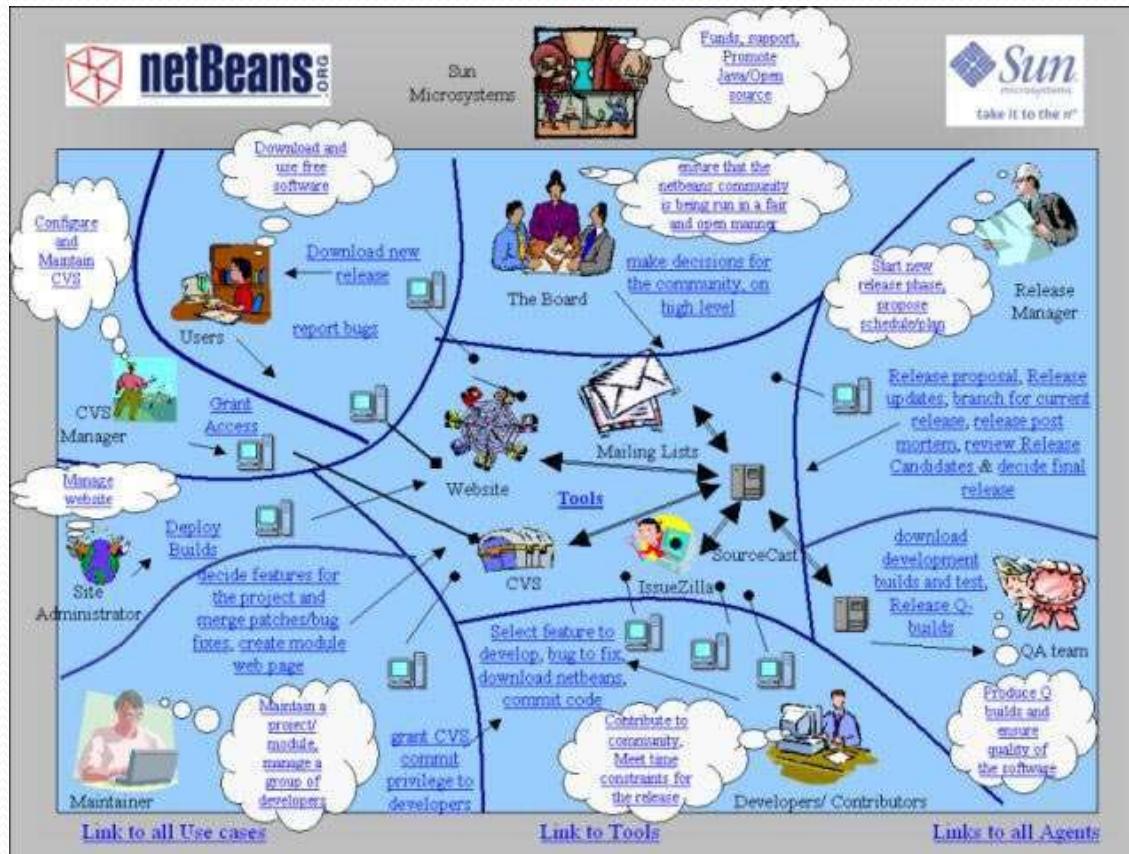
Kiti pavyzdžiai (2)



Kiti pavyzdžiai (3)



Kiti pavyzdžiai (4)



Priemonės RP ir vizijų sudarymui

- MS Visio (labiausiai rekomenduojama)
- MS Word
- ir bet kurios kitos priemonės, kurios leidžia sudaryti paveikslus iš
 - Grafinių elementų (primityvų);
 - Teksto;
 - Nuotraukų ir t.t.

Nepamirškite teksto

- Vizijos RP yra geras dalykas, tačiau niekada nebūna per daug tekste paaiškinti ir pakomentuoti pateiktą idėją.
- Ne visiems pavyksta nupiešti geras vizijas (pirmą kartą garantuotai nepavyks), todėl tekstas pagelbėja tuo atveju, jeigu kas nors nesuprato idėjos.



Ačiū už dėmesį. Klausimai?

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Rekomendacijos technologijų analyzei

Prof. dr. Agnus Liutkevičius,

Kompiuterių katedra

Studentų g. 50-202, tel. 300394

Blogo informacijos pateikimo ir palyginimo pavyzdys

Jutiklis	Technologija	Veikimo atstumai (m)	Tikslumas	Atkūrimo dažnis (Hz)	Matymo laukas (°)	Įtampa (V)	Didžiausias nepertraukiamas srovės suvartojimas (mA)	Kaina (€)
RFD77402	VCSEL	0.1-2.0	±10%	10	55	3.3	15	13.75
VL53L1X	VCSEL	0.04-4.0	±1%	50	27	2.6-5.5	18	21.09
HC-SR04	Ultragarso	0.02-4.0	±3mm	40	15	5	15	2.60
GP2Y0A02YK0F	LED	0.1-1.5	±1%	26	--	4.5-5.5	33	15,27
JSN-SR04T	Ultragarso	0.02-4.5	±3mm	40	70	5	30	18.03

Tai kuris geriausias? Kurio technologija geriausia? Koks minimalus jums reikalingas atstumas? Koks minimalus reikalingas tikslumas? Kuris dažnis geriausias? Kam išvis tas dažnis? Kas nuo jo priklauso? Koks mažiausias jums reikalingas matymo lauko kampus? Koks skirtumas kokia ta įtampa ir kurių sistemų jos geros, o kurių blogos? Kokia didžiausia kaina jus tenkina?

Gero palyginimo pavyzdys (1)

Įrenginys/ sistemas	Kriterijai					
	Pritaikymas namų ūkiams	Šiukslių lygio stebėjimas	Rūšiuoja ≥ 4 kategorijas	Išmatavimai \leq $50 \times 60 \times 85$ cm	Statistikos rinkimas	Prog. Ir. atnaujinimas nuotoliniu būdu
Buitinė perdirbamų atliekų rūšiavimo sistemas	+	+	+	+	+	+
TrashBot	-	+	-	-	+	+
Bin-e	-	+	+	-	-	-
R3D3	+	+	-	-	+	-

Gero palyginimo pavyzdys (2)

Jutiklis	Atstumas $\geq 30\text{cm}$	Paklaida $\leq 5\text{mm}$	Matavimo kampas ≥ 15 laipsnių	Jungčių kiekis < 4
HCSR04	+	+	+	-
SRF05	+	+	+	+
US-020	+	+	+	-
GP2Y0A21YK0	+	+	+	+
F				
VL53L0X	+	-	+	-

Gero palyginimo pavyzdys (3)

Protokolas	Kriterijai		
	Duomenų paketu perdavimo užtikrinimas	Galimybė serveriui inicijuoti komunikaciją	Išsiunčiamų duomenų šifravimas
HTTPS	+	-	+
WebSockets	+	+	+
MQTT	+	+	+

Reikalavimai palyginimo kriterijams



- Objektyvūs ir pamatuojami: skirtinių žmonės tuos pačius kriterijus turi suprasti vienodai.
- Geriausia, kai jie išreiškiami skaičiais: „ne mažiau nei X vienetų“, ne „daugiau nei X vienetų“ arba įvardina **konkrečią** funkciją ar savybę.
- Jei kriterijus gerai įvardintas, tai kitas žmogus turi sugebėti atlikti be jūsų pagalbos palyginimą ir išrinkti tinkamą sprendimą.

Kaip teisingai aprašyti kriterijus?

- Kriterijus reikia aprašyti **prieš** palyginimus ir paaiškinti kodėl jie svarbūs ir kodėl būtent tokias skaitines vertes turi tenkinti.
- Pvz. liepiu jums nupirkti automobilį ir pateikiu tokius kriterijus:
 - Kaina ne didesnė kaip 2000 EUR, nes tik tiek pinigų turiu;
 - Variklis turi būti benzininis, nes mažiau teršia aplinką;
 - CO₂ išmetimai ne didesni nei 130g, nes nereikia mokėti mokesčių;
 - Pavarų dėžė automatinė, nes vairuos nemokantis naudotis mechanine dėže vairuotojas;
 - Spalva raudona, nes paskui lengviau parduoti;
 - Vietų ne mažiau 4, nes turiu 2 vaikus.
- T. y. kriterijus reikia **pagristi**.
 - Jei pažiūrėsite pateiktą pavyzdį su automobiliu, visur yra ",nes" dalis su pagrindimu.

Kokie kriterijai blogi?

- Patogus, geras, greitas, populiarus (na ir kas iš to?😊), didelė bendruomenė, saugus, didelis, mažas, patikimas ir t. t.

Klausimai?

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

**Modeliaus pagrįstas kūrimas (Model-
Driven Development - MDD)**

Prof. dr. Agnus Liutkevičius,
Kompiuterių katedra
Studentų g. 50-202, tel. 300394

Kas yra MDD?

- MDD – model-driven development, tai sistemų kūrimas, panaudojant kūrimo procese įvairaus tipo modelius.
- Dažniausiai naudojama UML notacija ir modeliai.
- Tikslas:
 - Atskirti sistemos funkcionalumo specifikaciją nuo realizacijos.
 - Užtikrinamas pernešamumas ir nepriklausomumas nuo vykdymo aplinkos bei realizacijos technologijų.

Modelio savoka

- Modelis – tai supaprastintas sistemos ar reiškinio aprašas.
- UML modeliai – tai **formalūs** įvairių sistemos aspektų aprašai.
 - Formalus reiškia sudarytas pagal griežtas taisykles ir specifikacijas.
 - Formalius modelius galima paversti kodu (angl. **code generation**)
 - Iš kodo galima generuoti modelius (angl. **reverse engineering** – atvirkštinė inžinerija)
 - Formalius modelius galima automatiškai tikrinti ir validuoti.
 - Modelius galima transformuoti į kitus modelius (angl. **model transformation**)

Modelių naudojimo nauda (1)

- Sistemos programinis kodas atspindi tik sistemos struktūrą, tačiau iš jo sunku, o dažnai net neįmanoma suvokti sistemas:
 - Architektūros;
 - Funkcionalumo;
 - Elgsenos;
- Neturint modelių sunku suprasti dideles sistemas vien iš jų kodo.
- Iš kodo nesimato sąryšių su kitomis sistemomis.

Modelių naudojimo nauda (2)

- Modeliai leidžia aprašyti sistemos
 - Funkcionalumą
 - Architektūrą
 - Elgseną
- žymiai aukštesniame lygyje, nei programinis kodas.
- UML modeliai garantuoja:
 - Vizualizaciją;
 - Supratimą;
 - Komunikavimo galimybę;
 - Vientisumą ir korektiškumą;
 - Testuojamumą.

Modelių naudojimo nauda (3)

- UML diagrama yra daug kartų informatyvesnė ir naudingesnė, negu tiesiog gabalas programinio kodo.
- Atspėkite, kam skirtas ir ką valdo žemiau pateiktas kodas?

	LD	IY,	RODPOS
	LD	IX, 5	
	LD	A, 0x19	
LOOP1	STA	(IY)	
	DEC	IX	
	JPC	LOOP1	

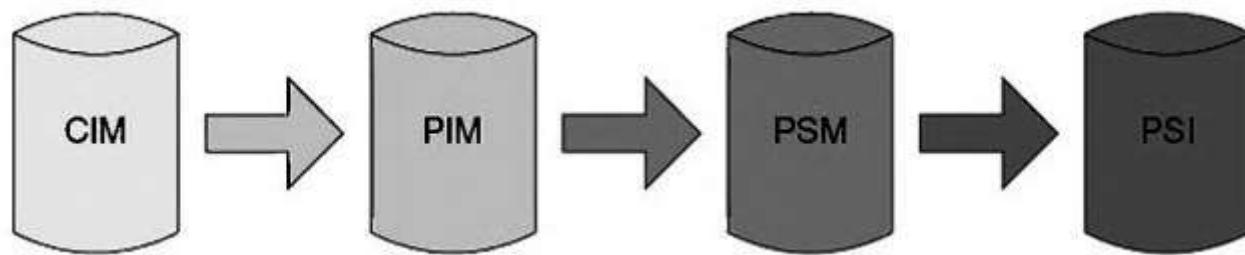
Kada MDD yra tikrai naudingas?

- Savaime modelių naudojimas dar negarantuoja visų minėtų privalumų.
- Modeliai privalo:
 - Tiksliai atitikti modeliavimo kalbą, kuria jie specifikuojami (pvz. UML).
 - Pakankamai išbaigtinėti.
 - Detalūs.
 - Gerai struktūrizuoti (galima dekomponuoti).
 - Vykdomi (angl. executable, simulated).
 - Susieti su kodu.

Modelių tipai (1)

- **CIM** – computation independent model – sistemos funkcionalumas;
- **PIM** – platform independent model – architektūra ir elgsena, bet ne technologijos;
- **PSM** – platform specific model – papildoma informacija apie technologinę platformą ir vykdymo/diegimo aplinką;
 - Parodo, kaip sistema (PIM) „atsigula“ į konkrečią platformą.
- **PSI** – platform specific implementation – tai PSM programinis kodas, kuris gali būti generuojamas iš PSM automatiškai.

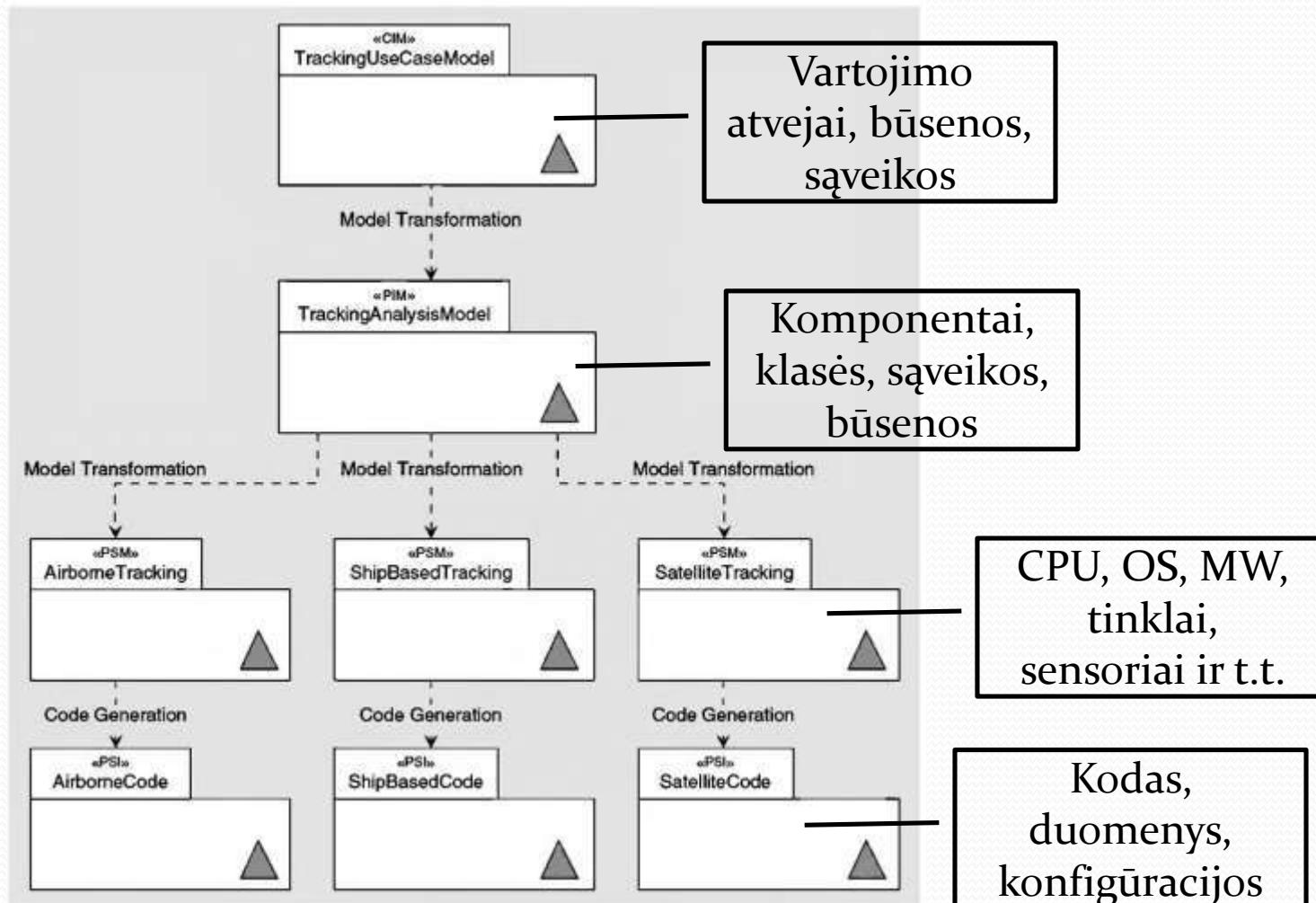
Modelių tipai (2)



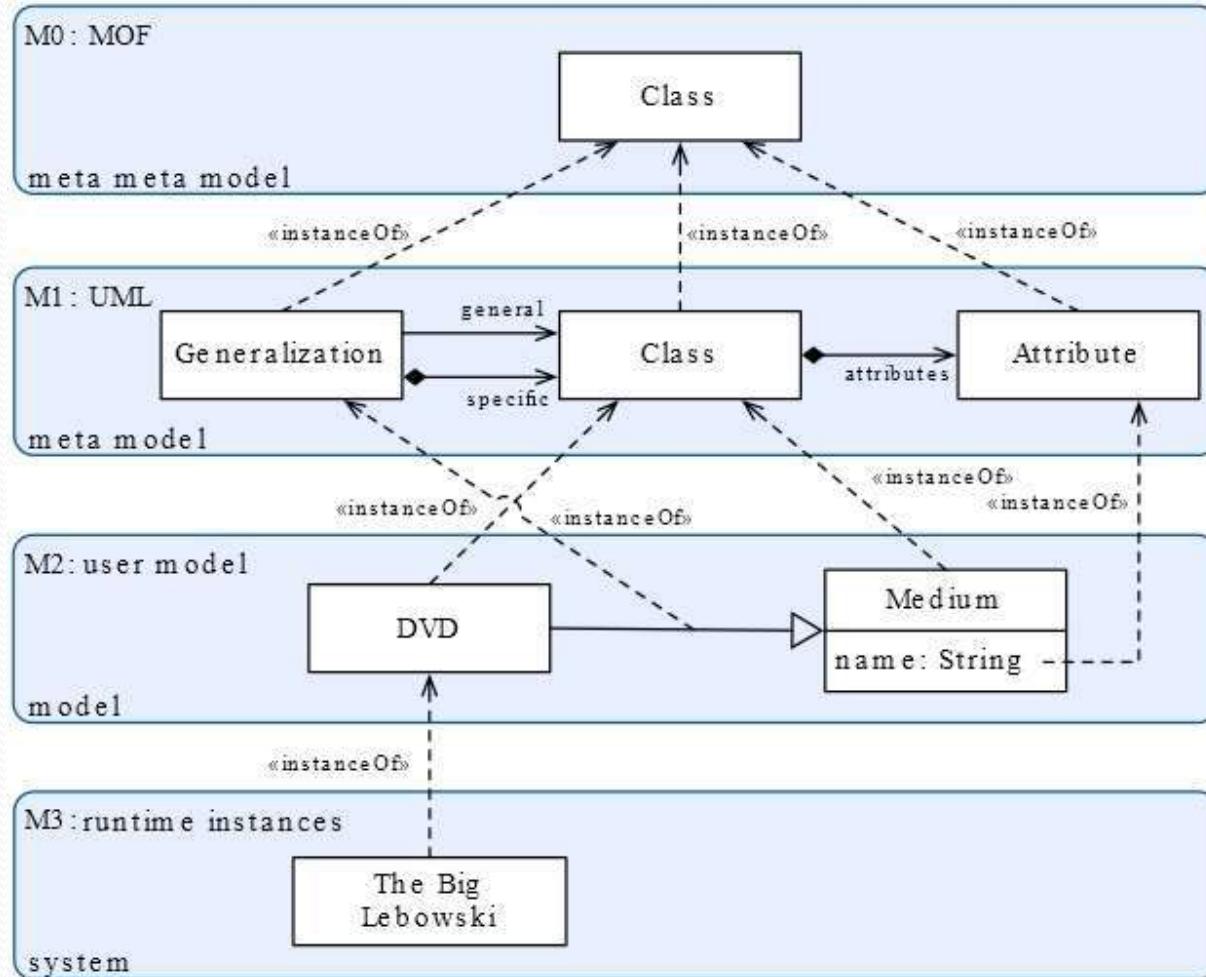
MDD ir UML

- Modeliai dažniausiai sudaromi, naudojant UML notaciją;
- Tačiau egzistuoja ir žymiai daugiau įvairių sistemų modeliavimo kalbų: SysML, MARTES, BPMN ir t.t.
- **SVARBU:** pati UML diagrama nėra modelis, o tik modelio atvaizdas:
 - Modelis dažniausiai saugomas kaip XML ar kitokio formato failas, kurio turinį galima įvairiai transformuoti.

Modelių transformacijos: pavyzdys



Meta-modelio sąvoka



CIM

- Aprašo sistemas funkcionalumą, bet nepasako kokia sistemos architektūra.
- Pagrindinis modelio elementas – vartojimo atvejis (angl. use case);
- **Nepakanka vien nurodyti vartojimo atvejus!** Reikia juos detalizuoti veiklos (angl. activity), sekų (angl. sequence), būsenų (angl. state) ir kt. diagramomis.

PIM

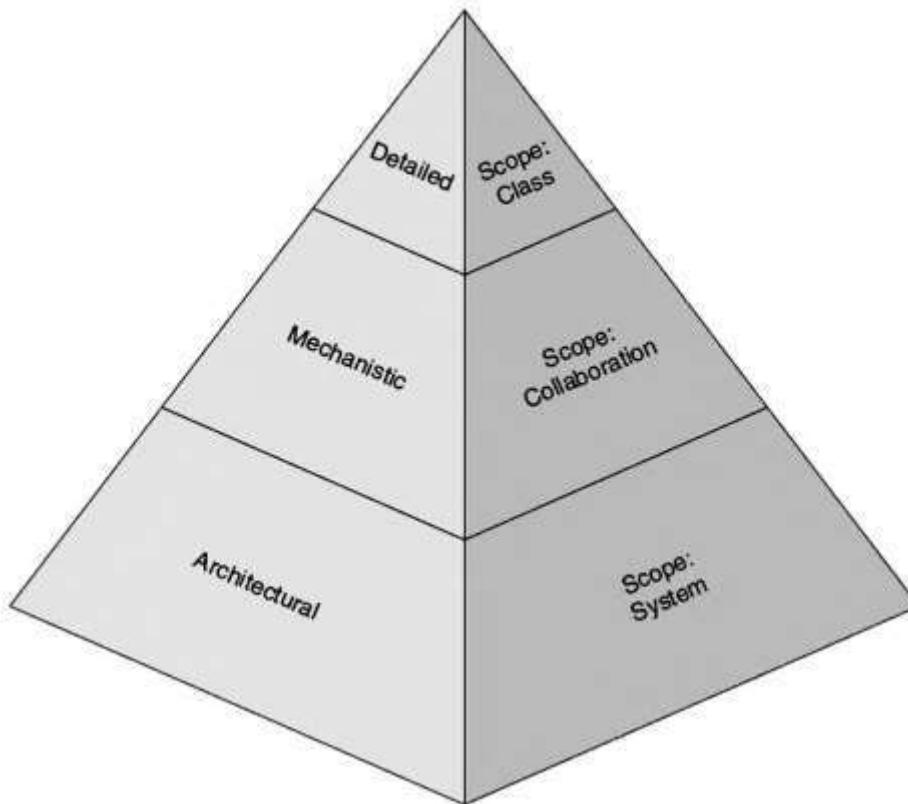
- Aprašo sistemos architektūrą, bet nekreipia dėmesio į vykdymo aplinką ir technologijas.
- Pagrindinis modelio elementas – klasė (angl. class).
- **Nepakanka vien nurodyti klasses!** Reikia jas **detalizuoti** ir aprašyti jų elgseną sekų (angl. sequence), būsenų (angl. state) ir kt. diagramomis.
- PIM apima svarbiausius nekintančius sistemos elementus – tie kurie kinta, priklausomai nuo sistemos realizacijos perkeliами į PSM.
- Geras būdas sudarinti PIM – sudaryti kiekvienam **vartojimo atvejui bent po vieną elgsenos modelį** (sekų, bendradarbiavimo, būsenų ar veiklos diagramų pagalba, kad matytusi, kaip klasės sąveikauja).

PSM

- PSM papildo PIM detalėmis apie diegimo ir vykdymo platformą ir naudojamas technologijas.
- PIM transformuoojamas į PSM:
 - Nustatomi optimizacijos kriterijai, tokia kaip našumas, patikimumas, duomenų pralaidumas, dydis, kaina ir t.t.
 - Kriterijai surikiuojami pagal savo svarbą;
 - Identifikuojamos platformos, projektavimo šablonai ir technologijos, kurios patenkina svarbiausius kriterijus;
 - Konkretūs projektiniai sprendimai pritaikomi PIM;
 - PSM validuoojamas:
 - Įsitikinama, kad PSM negriauna PIM funkcionalumo;
 - Įsitikinama, kad PSM užtikrina optimizacijos kriterijus;

Harmony/ESW proceso PSM

kūrimo lygmenys



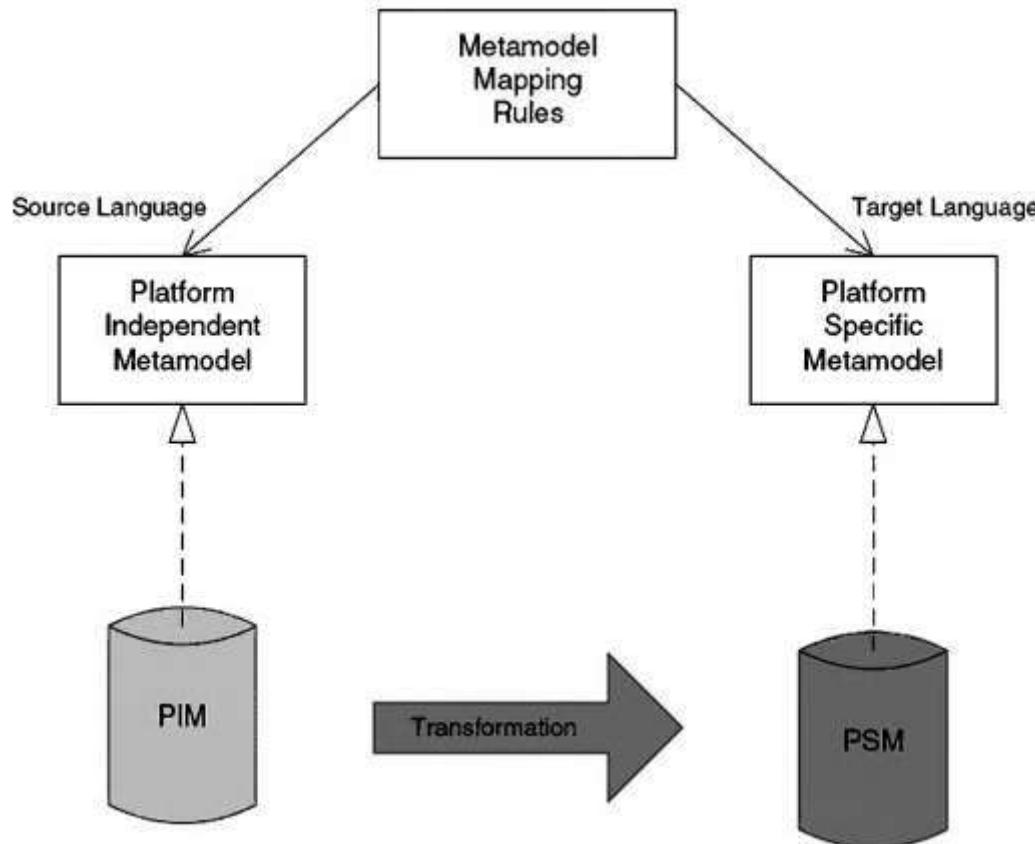
PSI

- Nėra tikras modelis.
- Tai kodas, sugeneruotas iš PSM.
- Šiuolaikinės priemonės leidžia generuoti kodą praktiškai bet kokia programavimo kalba, tačiau
 - Dažnai sugeneruotas kodas yra viso labo griaučiai, kuriuos reikia užpildyti rankiniu būdu.

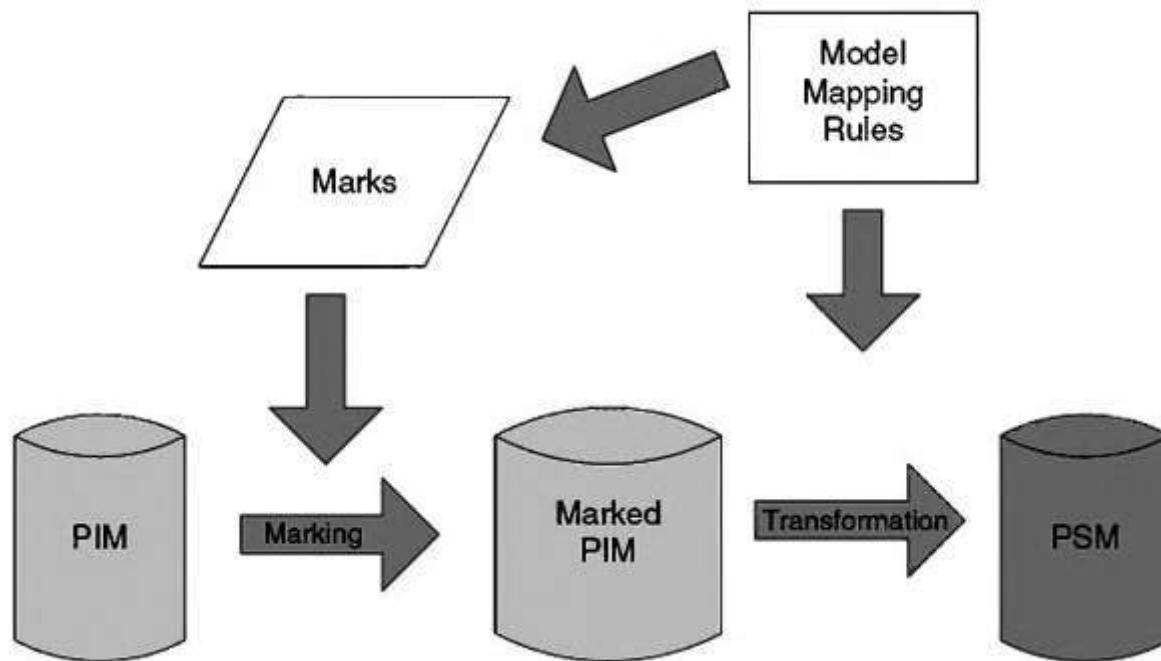
Modelių transformacijos

- CIM->PIM->PSM->PSI (angl. forward engineering)
 - arba
- PSI->PSM->PIM (angl. reverse engineering)
- Dažnai daroma rankiniu būdu, bet galima ir automatizuotai, ypač PSM->PSI.

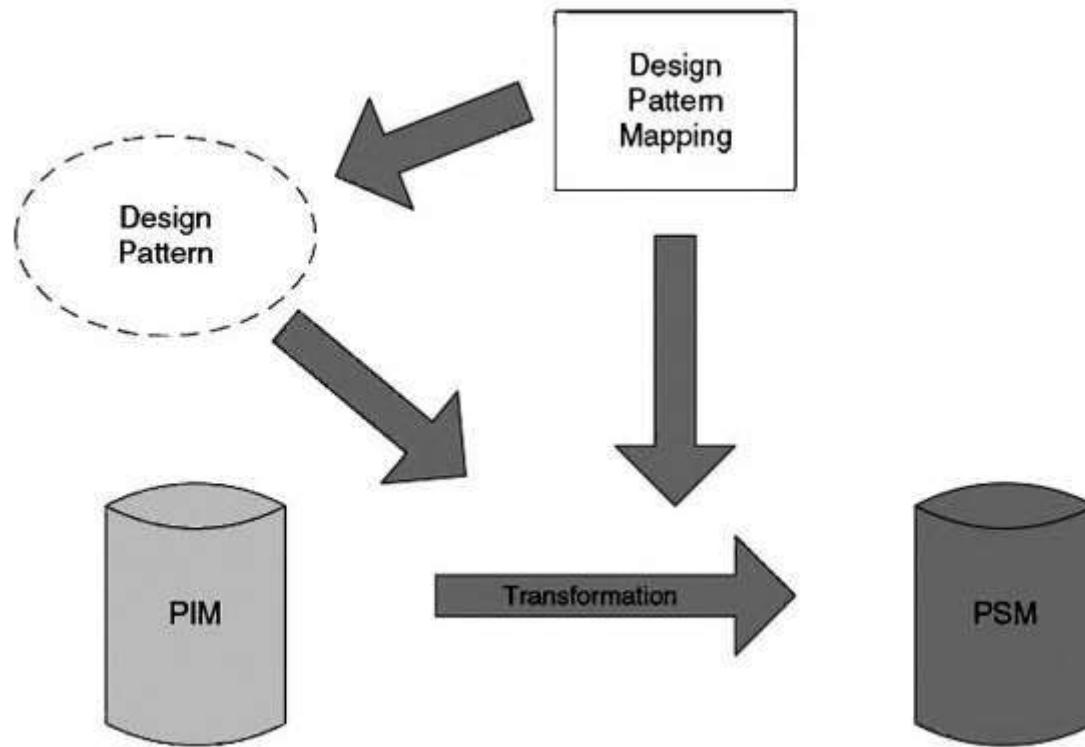
Meta-modelų transformacijos



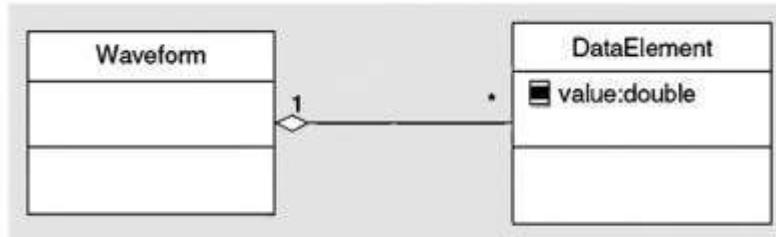
MDA modelių transformacijos



Projektavimo šablonais (angl. design pattern) pagrįstos transformacijos



Pavyzdys: Konteinerio projektavimo šablonas (1)



```

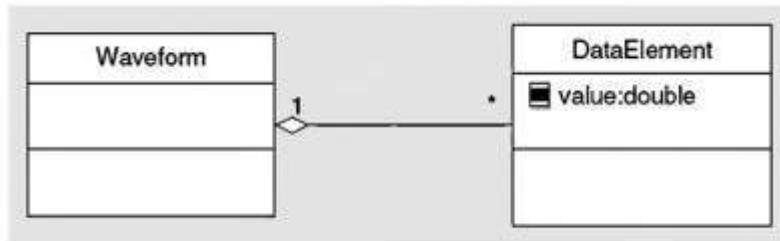
#ifndef Waveform_H
#define Waveform_H

///# auto_generated
#include <oxf\oxf.h>
///# auto_generated
#include <oxf\omcollec.h>
///# link itsDataElement
class DataElement;

///# package pattern

///# class Waveform
class Waveform {
    /// Constructors and destructors
public :
    ///# auto_generated
    Waveform();
    ///# auto_generated
    ~Waveform();
    /// Additional operations
    ///# auto_generated
    OMIterator<DataElement*> getItsDataElement() const;
    ///# auto_generated
    void addItsDataElement(DataElement* p_DataElement);
    ///# auto_generated
}
  
```

Pavyzdys: Konteinerio projektavimo šablonas (2)



```

void removeItsDataElement(DataElement* p_DataElement);
    ///## auto_generated
void clearItsDataElement();

protected :
    ///## auto_generated
void cleanUpRelations();

//// Relations and components /////
OMCollection<DataElement*> itsDataElement; //## link itsDataElement

//// Framework operations /////

public :
    ///## auto_generated
void _addItsDataElement(DataElement* p_DataElement);

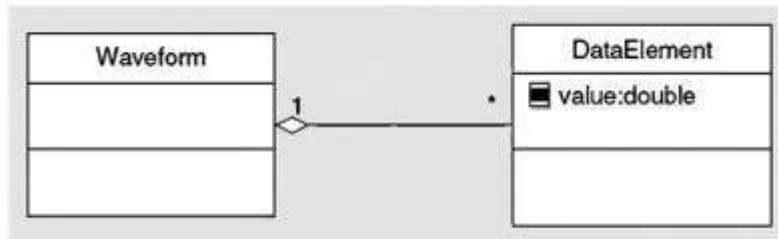
    ///## auto_generated
void _removeItsDataElement(DataElement* p_DataElement);

    ///## auto_generated
void _clearItsDataElement();

};

#endif
  
```

Pavyzdys: Konteinerio projektavimo šablonas (3)



```

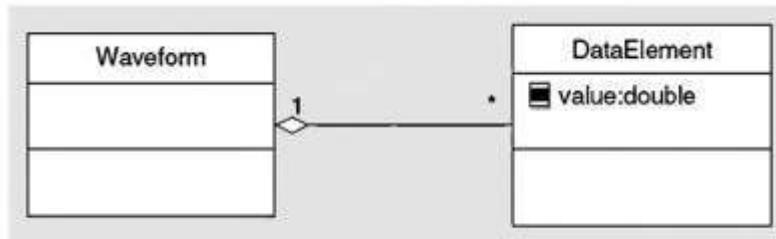
///## auto_generated
#include "Waveform.h"
///## link itsDataElement
#include "DataElement.h"
///## package pattern

///## class Waveform
Waveform::Waveform() {
}

Waveform::~Waveform() {
    cleanUpRelations();
}

OMIterator<DataElement*> Waveform::getItsDataElement() const {
    OMIterator<DataElement*> iter(itsDataElement);
    return iter;
}
  
```

Pavyzdys: Konteinerio projektavimo šablonas (4)



```

void Waveform::addItsDataElement(DataElement* p_DataElement) {
    if(p_DataElement != NULL)
    {
        p_DataElement->_setItsWaveform(this);
    }
    _addItsDataElement(p_DataElement);
}

void Waveform::removeItsDataElement(DataElement* p_DataElement) {
    if(p_DataElement != NULL)
    {
        p_DataElement->_setItsWaveform(NULL);
    }
    _removeItsDataElement(p_DataElement);
}

void Waveform::clearItsDataElement() {
    OMIterator<DataElement*> iter(itsDataElement);
    while (*iter){
        (*iter)->_clearItsWaveform();
        iter++;
    }
    _clearItsDataElement();
}
  
```

CIM->PIM transformacija

- Vartojimo atvejai -> Architektūra +Elgsena
- Sunku automatizuoti, dažnai rankinis ir iteratyvus procesas.

PIM->PIM transformacijos

- Mažiau detalus PIM -> Detalus PIM
- Rankinis procesas

PIM<->PSM

- Transformacijos, kurių dėka modelis tampa pakankamai detaliu kodo generavimui, arba atvirkščiai – iš platformos modelio gaunamas abstraktesnis, visą produktų šeimą aprašantis modelis.
- Rankinis procesas

PSM->PSI

- Kodo generavimas
- Automatizuotas
- Egzistuoja priemonės, leidžiančios generuoti:
 - Sistemos griaučius, struktūrą (iš klasių)
 - Sistemos elgseną (iš būsenų, veiklos ar sekų diagramų)

PSI->PSM

- Reverse engineering procesas, kurio metu iš kodo gaunamas modelis:
 - Struktūros (geba praktiškai visos priemonės)
 - Elgsenos (geba vos kelios priemonės, sudėtingas procesas)



Ačiū už dėmesį. Klausimai?

UML pagrindai

Įvadas,
funkcijų modeliavimas
Prof. dr. Agnus Liutkevičius

UML

- Pli projektavimo metodika ir priemonės
- Sukurta OMG (www.uml.org)
- Leidžia aprašyti kuriamą sistemą (ar jos dalį) norimame abstrakcijos lygyje
- Geras mokymosi resursas:
 - <http://www.uml-diagrams.org/>

UML paskirtis

„**The Unified Modeling Language™ (UML®)** is a standard visual modeling language intended to be used for

- modeling business and similar processes,
- analysis, design, and implementation of software-based systems.“

Kas naudoja UML?

„UML is a common language for

- business analysts,
- software architects and developers
 - used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.“

UML vs. Sistemų kūrimo procesas

„UML is a standard modeling **language**, not a **software development process**:

- provides guidance as to the order of a team's activities,
- specifies what artifacts should be developed,
- directs the tasks of individual developers and the team as a whole, and
- offers criteria for monitoring and measuring a project's products and activities.“

UML savybės

- „**Process independent**“ and could be applied in the context of different processes.
- Most suitable for use case driven, iterative and incremental development processes.
- Is not complete and it is not completely visual.
 - Some information could be intentionally omitted from the diagram,
 - some information represented on the diagram could have different interpretations,
 - and some concepts of UML have no graphical notation at all, so there is no way to depict those on diagrams.“

UML privalumai

- Kuriama sistema yra profesionaliai suprojektuojama ir dokumentuojama dar prieš realizaciją. Tiksliai žinoma ką reikia pasiekti.
- Kadangi sistema pradžioje suprojektuojama, galima nustatyti, kurios vietas bus perpanaudotos – mažėja kūrimo kaina.
- Lengva nustatyti projektavimo klaidas.
- Geras sistemos projektas užtikrina teisingą ir efektyvų realizavimą – mažėja kūrimo kaina.
- UML leidžia aprašyti sistemą norimu detalumu ir norimais pjūviais.
- Atliliki sistemos modifikacijas yra žymiai lengviau turint sistemos UML dokumentaciją – mažėja palaikymo kaina.
- UML leidžia kitiams kūrėjams greitai suvokti jūsų sistemą. Tai visuotinai pripažintas standartas.

Vienos kūrimo iteracijos etapai

- Reikalavimų išgavimas;
- Analizė;
- Projektavimas;
- Kūrimas;
- Diegimas.

Reikalavimų išgavimas

- **Ko iš mūsų nori užsakovas???**
- Verslo procesų identifikavimas:
 - Scenarijų diagramos (angl. activity)
- Dalykinės srities analizė:
 - Aukšto lygio klasių diagrama
- Sąveikaujančių sistemų išskyrimas:
 - Realizacijos (angl. deployment)
- Rezultatų pristatymas užsakovui

Analizė

- Vartojimo atvejų identifikavimas:
 - Vartojimo atvejų diagrama (-os) (angl. use case)
- Vartojimo atvejai detalizuojami:
 - Veiksmų sekų aprašymai
- Detalizuojamos klasių diagramos:
 - Detalizuota klasių diagrama
- Analizuojamos sistemos galimos būsenos:
 - Būsenų diagrama (angl. state)
- Nustatomos objektų tarpusavio sąveikos:
 - Sekų ir bendradarbiavimo diagramos (angl. sequence, collaboration)
- Analizuojama sąveika su kitomis sistemomis:
 - Detali realizacijos diagrama (angl. deployment)

Projektavimas

- Objektų diagramų kūrimas ir detalizavimas:
 - Scenarijų diagramos (angl. activity)
- Komponentų diagramų kūrimas
- Diegimo (angl. deployment) plano sudarymas
- Vartotojo sąsajos projektavimas ir prototipai
- Testų kūrimas
- Dokumentacijos kūrimas:
 - Dokumentacijos struktūros sudarymas

Kūrimas

- Kodo rašymas
- Kodo testavimas
- Vartotojo sąsajų kūrimas ir testavimas
- Pilna sistemos dokumentacija

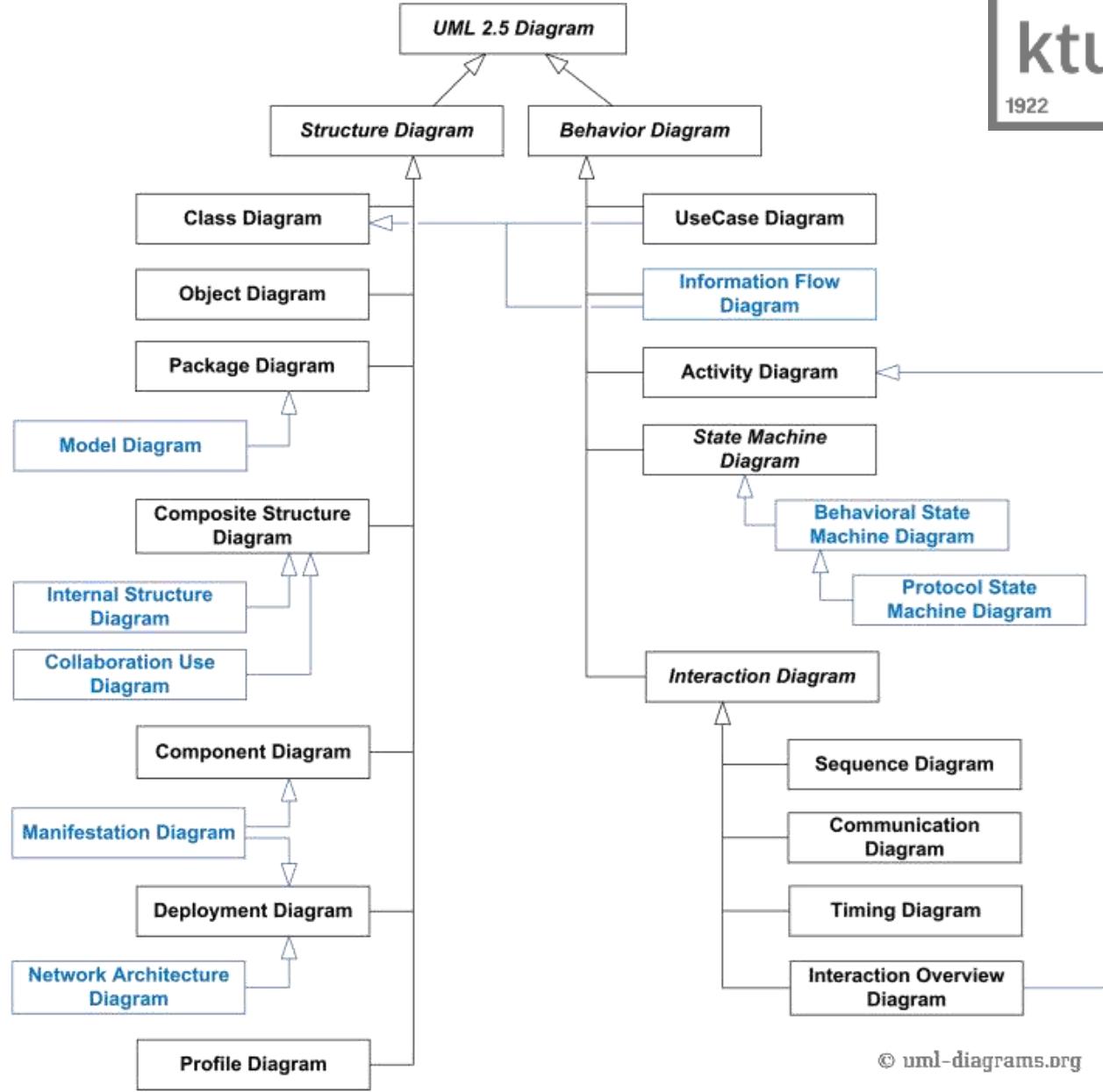
Diegimas

- Sistemos atstatymo plano sudarymas
- Sistemos įdiegimas aparatūrinėje įrangoje
- Integruotos sistemos testavimas
- Šventimas ☺ ... arba ne ☹

UML komponentai

- UML susideda iš įvairių grafinių elementų apjungiamų į diagramas.
- UML yra kalba, todėl egzistuoja griežtos grafinių elementų panaudojimo taisyklės.
- Diagramų tikslas – įvairiais pjūviais aprašyti kuriamą sistemą, t.y. sudaryti sistemos **modelį**.
- UML modelis **aprašo ką sistema turi daryti**, bet jis **nepasako kaip realizuoti** tą sistemą.

UML diagramų tipai (visi)



Šaltinis: <http://www.uml-diagrams.org/uml-25-diagrams.html>

UML diagramų tipai (1)

■ **Klasių diagramos** (angl. class):

- Skirsto daiktus į kategorijas. Klasė aprašo daiktų grupę, kurie turi panašius atributus ir vienodą elgseną.

■ **Sekų (angl. sequence) diagramos:**

- Aprašo objektų sąveiką laike.

■ **Vartojimo (panaudojimo) atvejų (angl. usecase) diagramos:**

- Aprašo sistemos elgseną iš vartotojo pozicijų.

■ **Veiklos arba Scenarijų (angl. activity) diagramos:**

- Aprašo veiksmų sekas.

UML diagramų tipai (2)

- **Būsenų (angl. state) diagramos:**
 - Nusako objektų būsenas ir jų pasikeitimus laike.
- **Bendradarbiavimo (komunikavimo) (angl. collaboration, communication) diagramos:**
 - Aprašo objektų sąveiką.
- **Realizacijos (angl. component, deployment) diagramos:**
 - Nusako sistemos komponentus ir parodo fizinę sistemos struktūrą.

UML modeliavimo ir sistemų kūrimo priemonės

■ Mokamos (turi Trial versijas):

- MagicDraw (<https://www.magicdraw.com/>)
- Enterprise Architect (<http://www.sparxsystems.com/>)
- IBM Rational Software Architect
(<http://www.ibm.com/developerworks/downloads/r/architect/>)
- Microsoft Visio
- ir begalė kitų ☺

■ Nemokamos:

- ArgoUML (<http://argouml.tigris.org/>)
- Visual Paradigm (Community Edition) (<http://www.visual-paradigm.com/>)
- Umbrello UML Modeller (<http://uml.sourceforge.net/>) ir kt.

Vartojimo atvejų diagramos

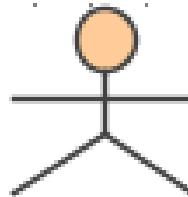
- Aprašo sistemos funkcionalumą
- Esminiai elementai:
 - Aktoriai (angl. Actor)
 - Vartojimo atvejai (angl. Usecase)
 - Ryšiai (angl. Relationships)
 - Sistemos ribos (angl. System boundary)

Aktorai

- Aktoriai aprašo sistemos vartotojus / naudotojus;
- Aktoriai – tai **išorinės esybės**, kurios sąveikauja su sistema (žmonės, kitos sistemos, techninė įranga, laikas ir pan.).

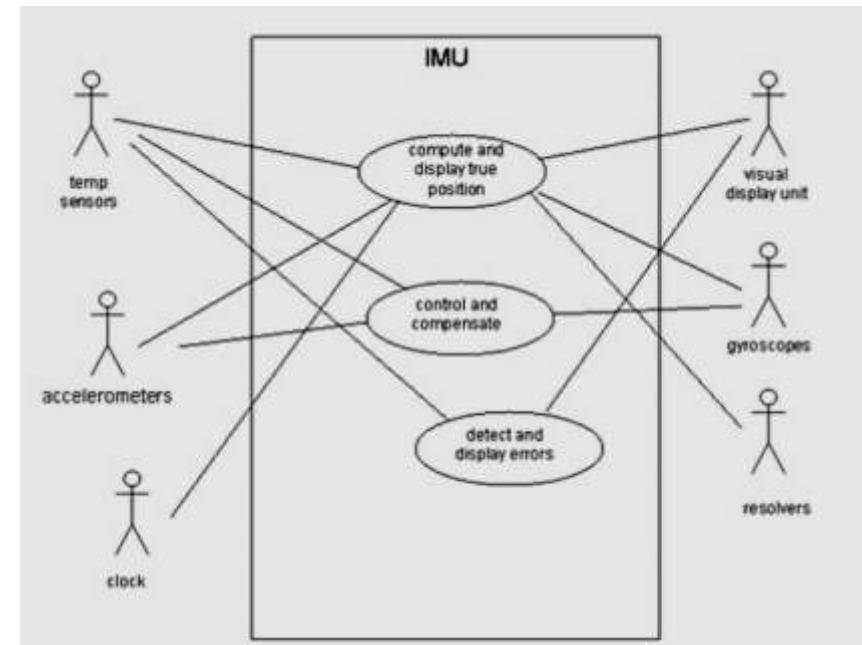
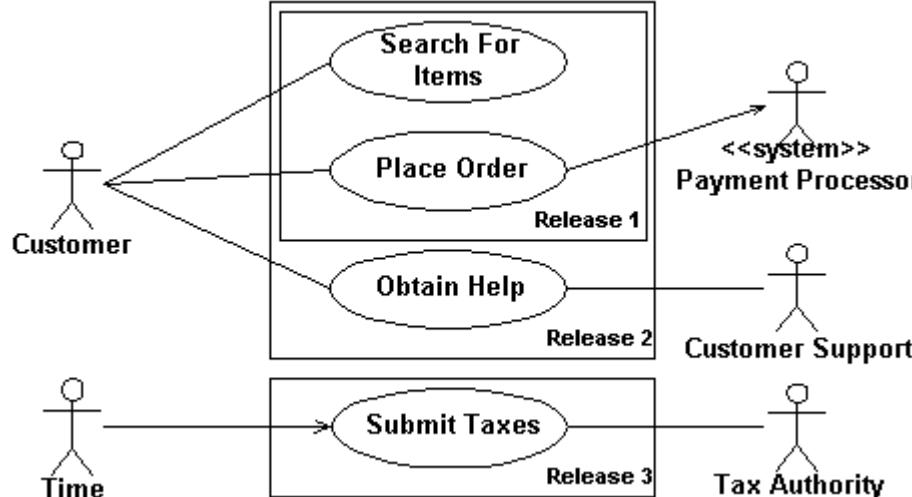
Aktorų rolės

- Kiekvienas aktorius aprašo skirtinę vartotojo **rolę** (o ne konkretų vartotojų!)
 - pvz. administratorius, banko tarnautojas, telefono savininkas ir pan.
- Žymima:



Automatiškai vykdomų procesų aktoriai

- Atvejai, kuriuos inicijuoja pati sistema, irgi turi turėti susietus aktorius: dažniausiai *Laiką* ar *Jutiklius*.
 - Sistema negali būti savo paties aktoriumi!



Vartojimo/panaudiojimo atvejis

- Aprašo vieną aktoriui(-iams) matomą sistemos funkciją;
- Vykdant pasiekiamas konkretus aktoriaus iškeltas tikslas ir gaunamas naudingas (dažniausiai) rezultatas:
 - A **use case** is a kind of behaviored classifier that specifies a [complete] unit of [useful] functionality performed by [one or more] subjects to which the use case applies in collaboration with one or more actors, and which [for complete use cases] yields an observable result that is of some value to those actors [or other stakeholders] of each subject.

Reikalavimai panaudojimo atvejams

■ Atvejai yra **baigtiniai**:

- UML specifications require that "this functionality must always be completed for the UseCase to complete. It is deemed complete if, after its execution, the subject will be in a state in which no further inputs or actions are expected and the UseCase can be initiated again, or in an error state.,,

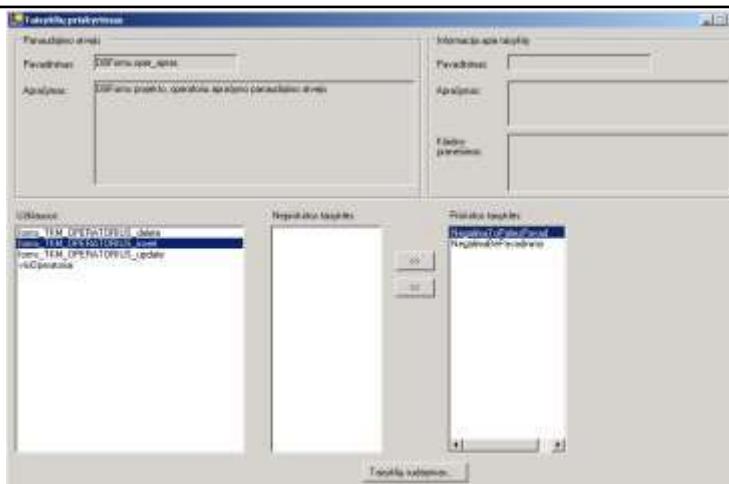
■ Pavadinimas turi atspindėti funkciją (veiksmažodinė forma), suteikiančią **pamatuojamą** ir aktoriui **naudingą** rezultatą:

- We should just follow use case definition to give some name to that unit of functionality performed by a system, which provides some observable and useful result to an actor.

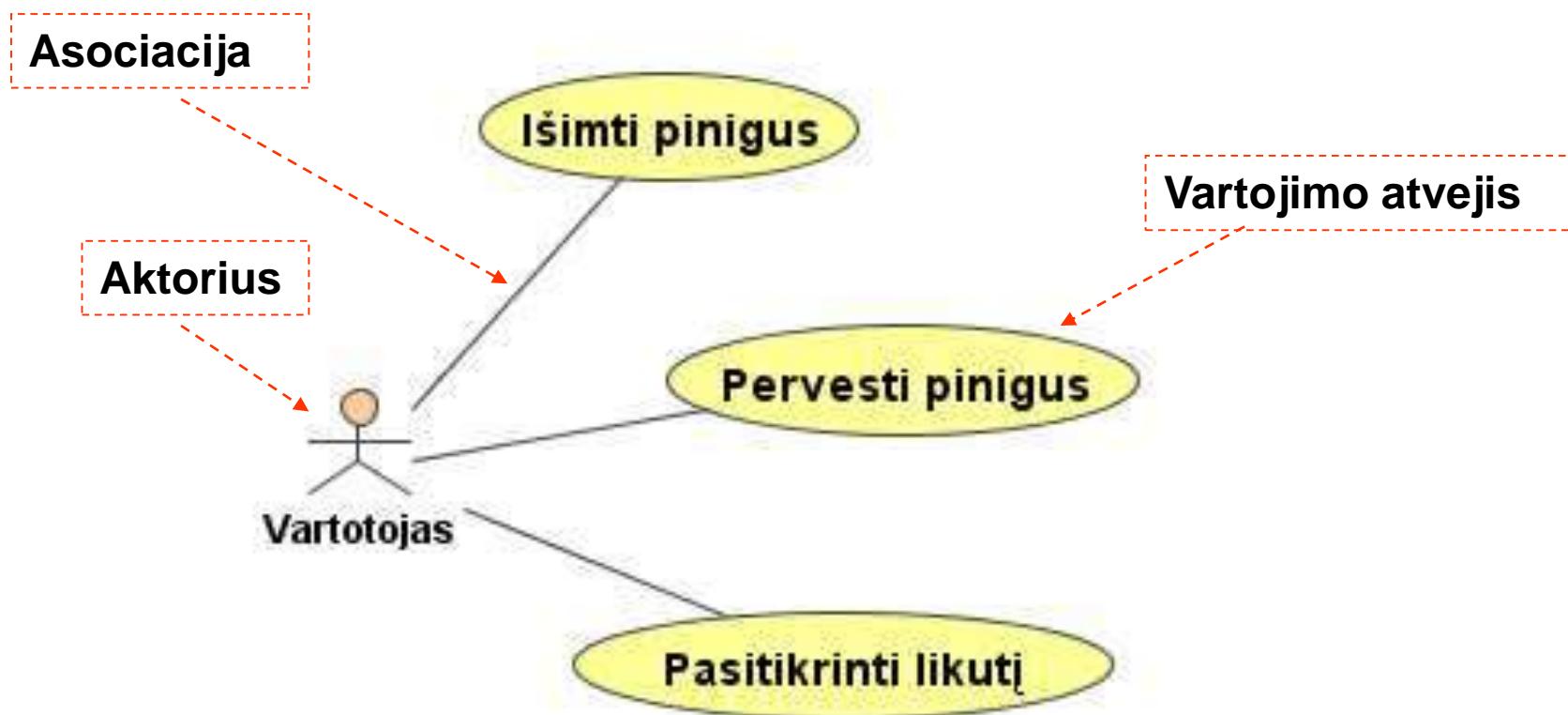
UML kūrėjų patarimas kaip lengviau sugalvoti teisingą atvejo pavadinimą

- One trick to help in the proper naming of a Use Case is to precede the goal with the words: “O System, please help me to . . .” or a similar variant.
 - O System, please help me to **Hail a Ride**.
 - O System, please help **Hail a Ride** for me.
 - O System, please help me to **Establish a Payment Method**.
 - O System, please help to **Establish a Payment Method** for me.

Detalaus panaudojimo atvejo aprašymo pvz.

Panaudojimo atvejis	UC202	<i>Taisyklės pašalinimas iš panaudojimo atvejo</i>
Aprašymas	Tinklo valdytojas iškviečia panaudojimo atvejį atitinkančio grafinės sąsajos komponento kontekstinių meniu, kuriame pasirenka taisyklių pašalinimo punktą. Dialoginiame lange iš panaudojimo atvejo pašalinamos pasirinktos priskirtos taisyklės.	
Aktorai	Administratorius	
Prieš-sąlyga	Panaudojimo atvejui turi būti priskirta nors viena taisyklė.	
Po-sąlyga	Pasirinktos taisyklės pašalinamas iš konkretaus panaudojimo atvejo. Panaudojimo atvejo metu vykdant konkrečias DIM užklausas, pašalintos taisyklės nebebus tikrinamos, nors ir liks saugomos duomenų bazėje.	
Grafinė sąsaja:		

Vartojimo atvejų diagrammos pagrindiniai elementai



Panaudojimo atvejo modelis su keliais aktoriais

- Jeigu su panaudojimo atveju susieti keli aktoriai, tai jie abu vienu metu dalyvauja to atvejo scenarijuje!



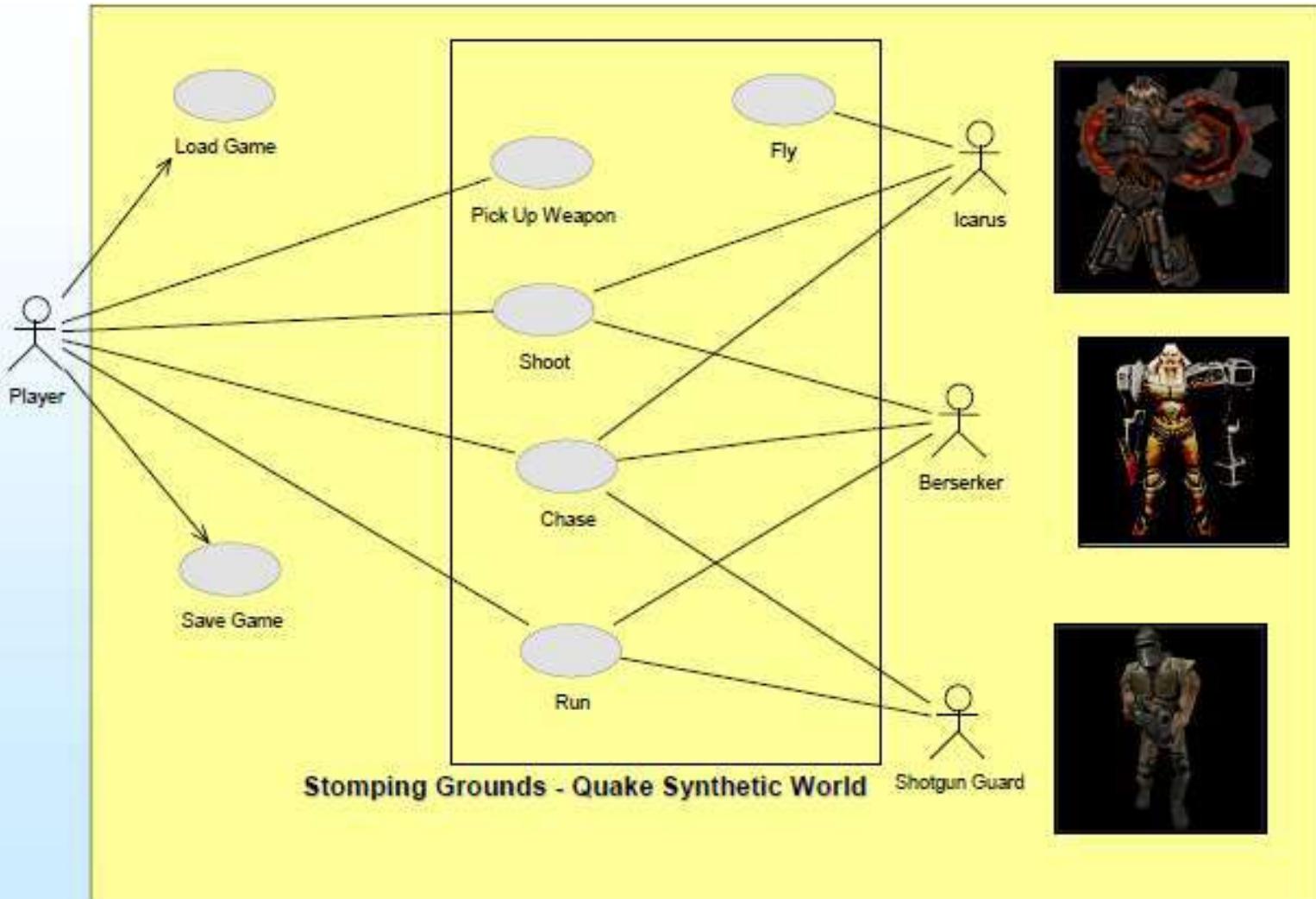
Kardinalumai

■ Kardinalumų skaičiai parodo:

- kiek aktorių vienu metu dalyvauja panaudojimo atvejuje (skaičius prie aktoriaus);
- keliuose atvejuose vienu metu gali dalyvauti aktorius (skaičius prie atvejo).



Quake II vartojimo atvejų diagrama

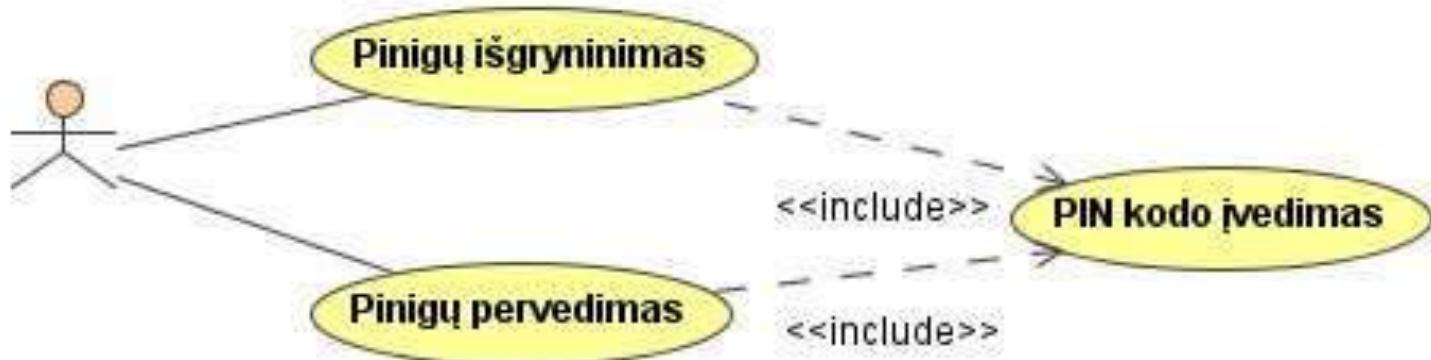


Ryšiai

- <<include>>
- <<extend>>
- Aktorių hierarchija

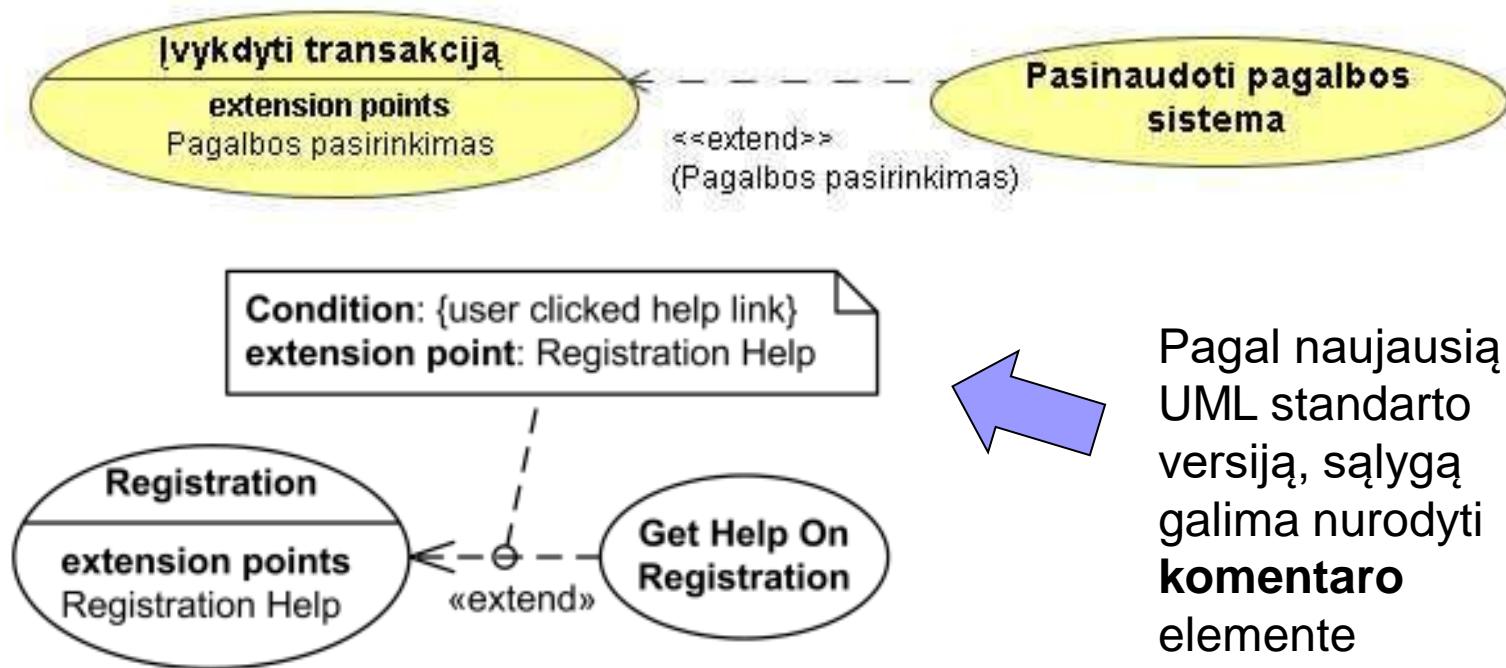
<<include>>

- Naudojamas nurodyti, kad atliekant vieną panaudojimo atvejį, **būtinai** atliekamas ir kitas.

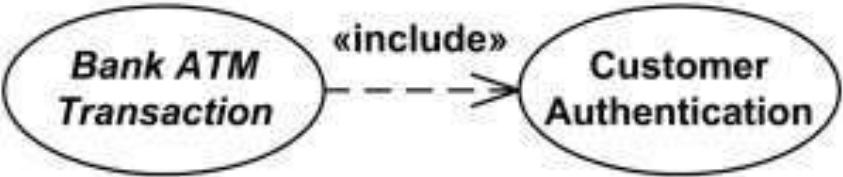


<<extend>>

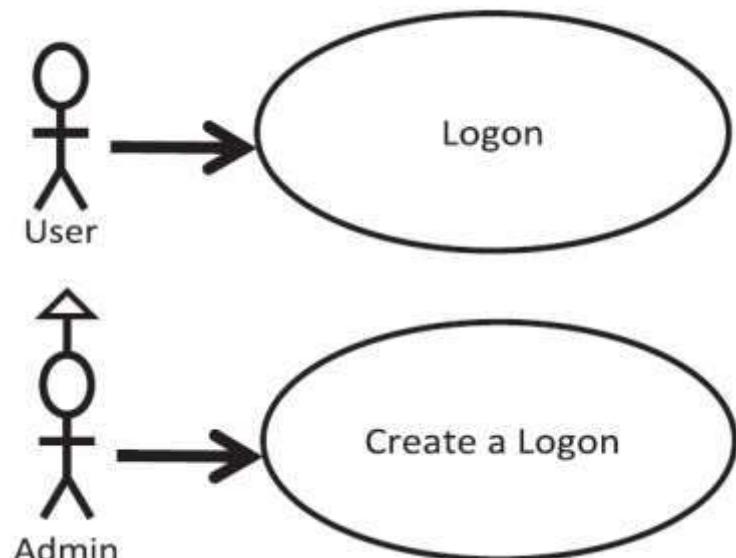
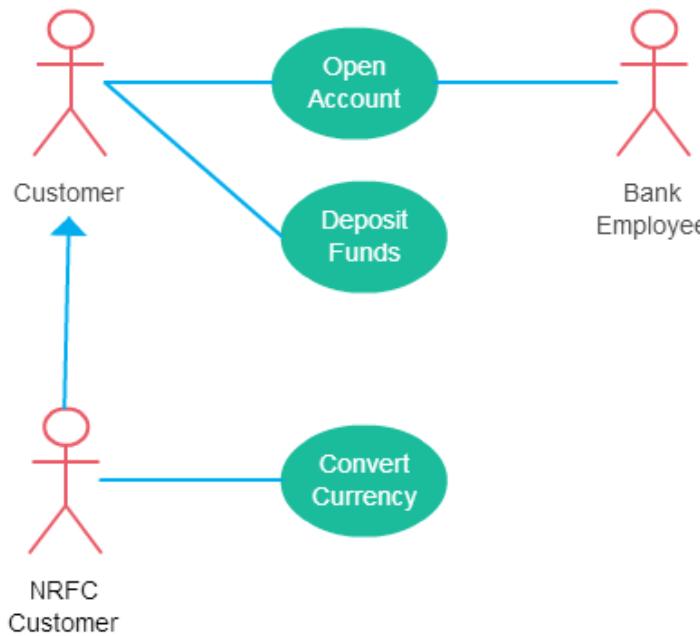
- Naudojamas nurodyti, kad atliekant vieną panaudojimo atvejį, **esant tam tikrai sąlygai**, atliekamas ir kitas.



Extend vs. Include

Extend	Include
	
Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete .
Extending use case is optional , supplementary.	Included use case required , not optional.
Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
Could have optional extension condition.	No explicit inclusion condition.

Aktorų hierarchija



- Parodo, kad aktorius paveldi kito aktoriaus rolę.
- Naudojame tada, kai keli aktoriai gali dalyvauti tame pačiame atvejyje, bet **ne vienu metu (ne kartu)**.

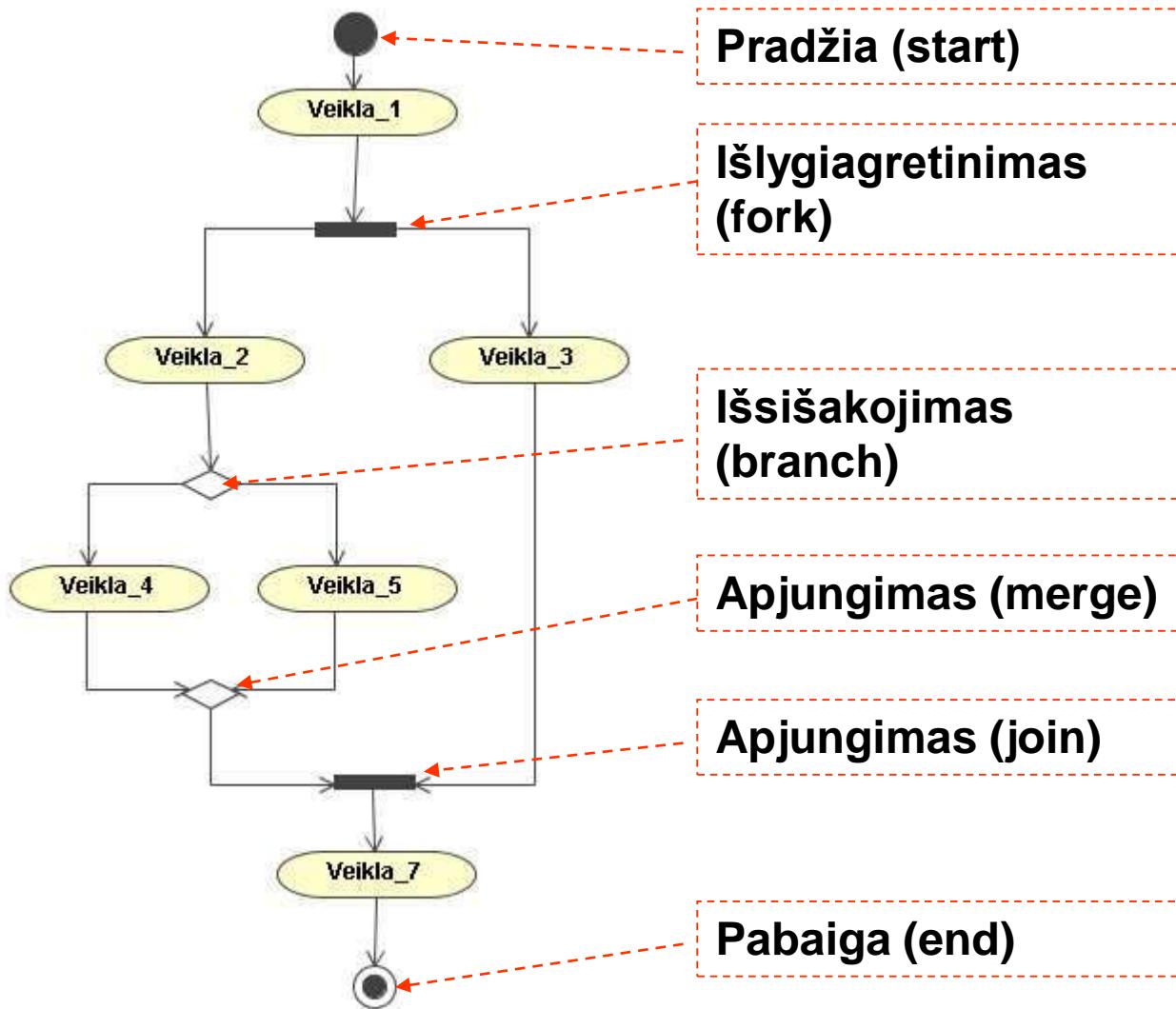
Veiklos (angl. Activity) arba Scenarijų diagramos

- Verslo procesų aprašymui
- Vartojimo atvejų detalizacijai:
 - Kiekvieną vartojimo atvejį detalizuojant viena scenarijų diagrama.
- Būsenų diagramos plėtinys:
 - Būsenų diagrama išryškina būsenas ir parodo veiklas kaip perėjimus tarp būsenų. Veiklos diagramos labiau koncentruojasi į pačias veiklas.

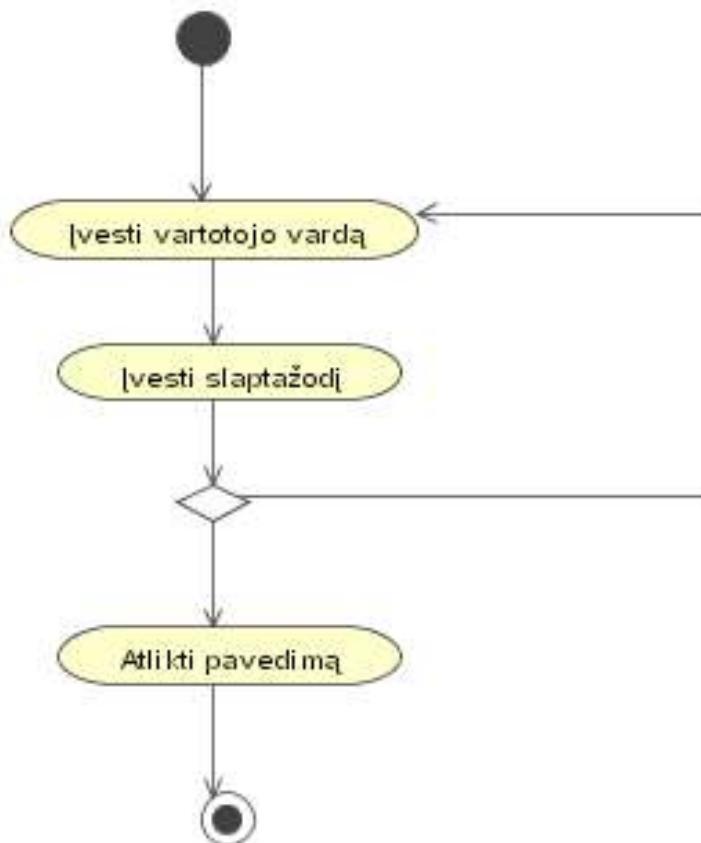
Pagrindiniai elementai

- Veiksmų būsenos – aprašo sistemoje atliekamus veiksmus.
- Perėjimas – jungia dvi veiksmų būsenas, atvaizduoja vykdymo eiga.
- Išsišakojimas (angl. branch)
- Sinchronizavimas:
 - Išlygiagretinimas (angl. fork)
 - Apjungimas (angl. join)
- Objektai
- Signalai
- Laiko įvykiai
- Veiklų suskirstymas (angl. partition)

Veiklos diagrammos modelis



Veiklos diagramos pavyzdys: kur klaida?



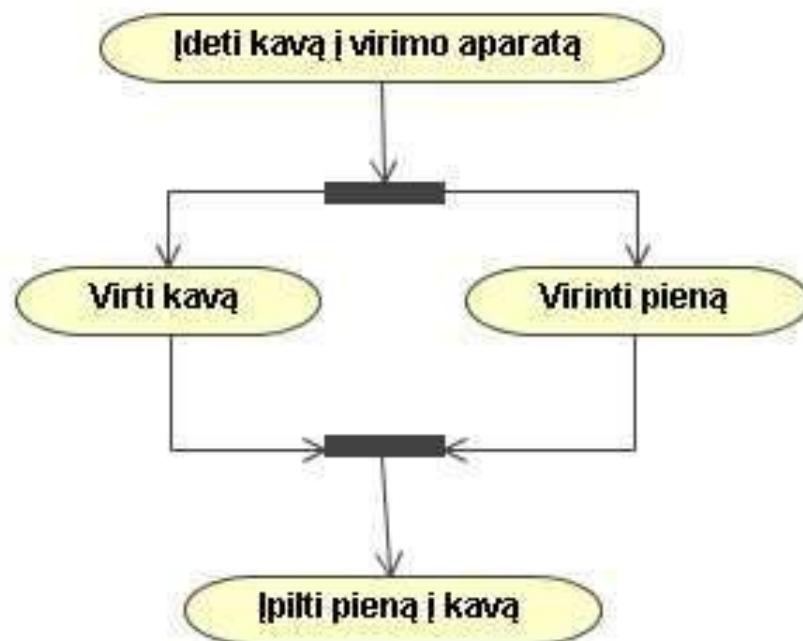
Išsišakojimas ir apjungimas

- Skirtas alternatyvioms veiksmų šakoms aprašyti.
- Nurodomos sąlygos, kurioms esant atliekama viena arba kita veiksmų seka.



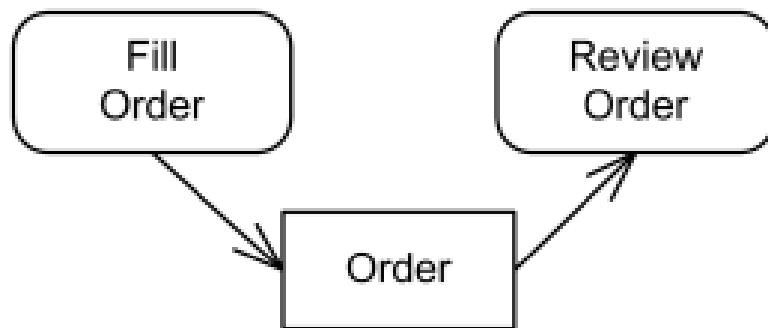
Išlygiagretinimas ir synchronizavimas

- Skirtas lygiagrečioms veiksmų šakoms aprašyti (lygiagretiems procesams).



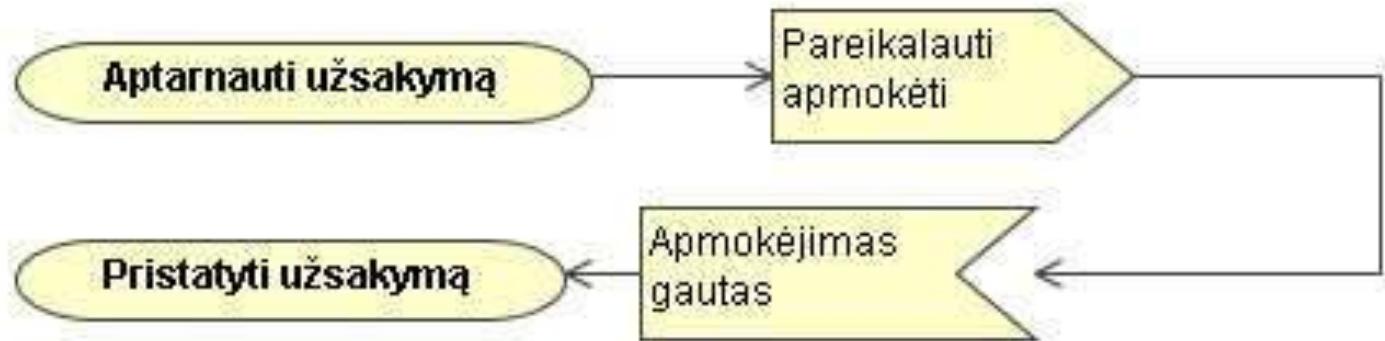
Objektai

- Veiklos metu gali būti sukurti objektai (veiklos produktais), kurie naudojami kitose veiklose.
- Duomenys, kurie “keliauja” per veiklas.



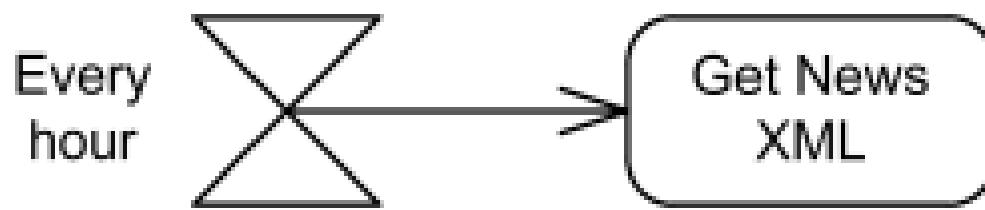
Signalai

- Atliekant veiksmų sekas galima išsiųsti signalą išoriniams procesams (objektams).
- Gautas signalas naudojamas veiksmo inicializacijai.



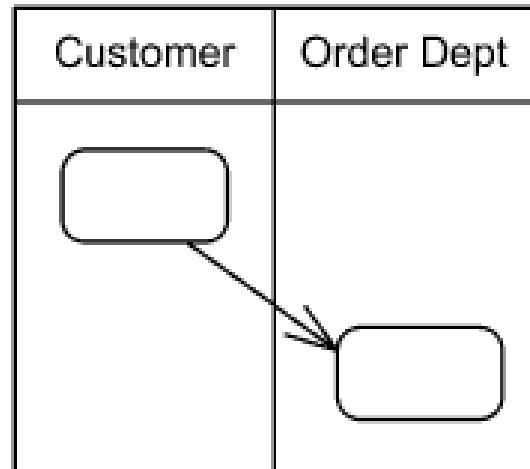
Laiko įvykiai

- Leidžia modeliuoti tam tikru metu ar tam tikru periodiškumu vykstančias veiklas:

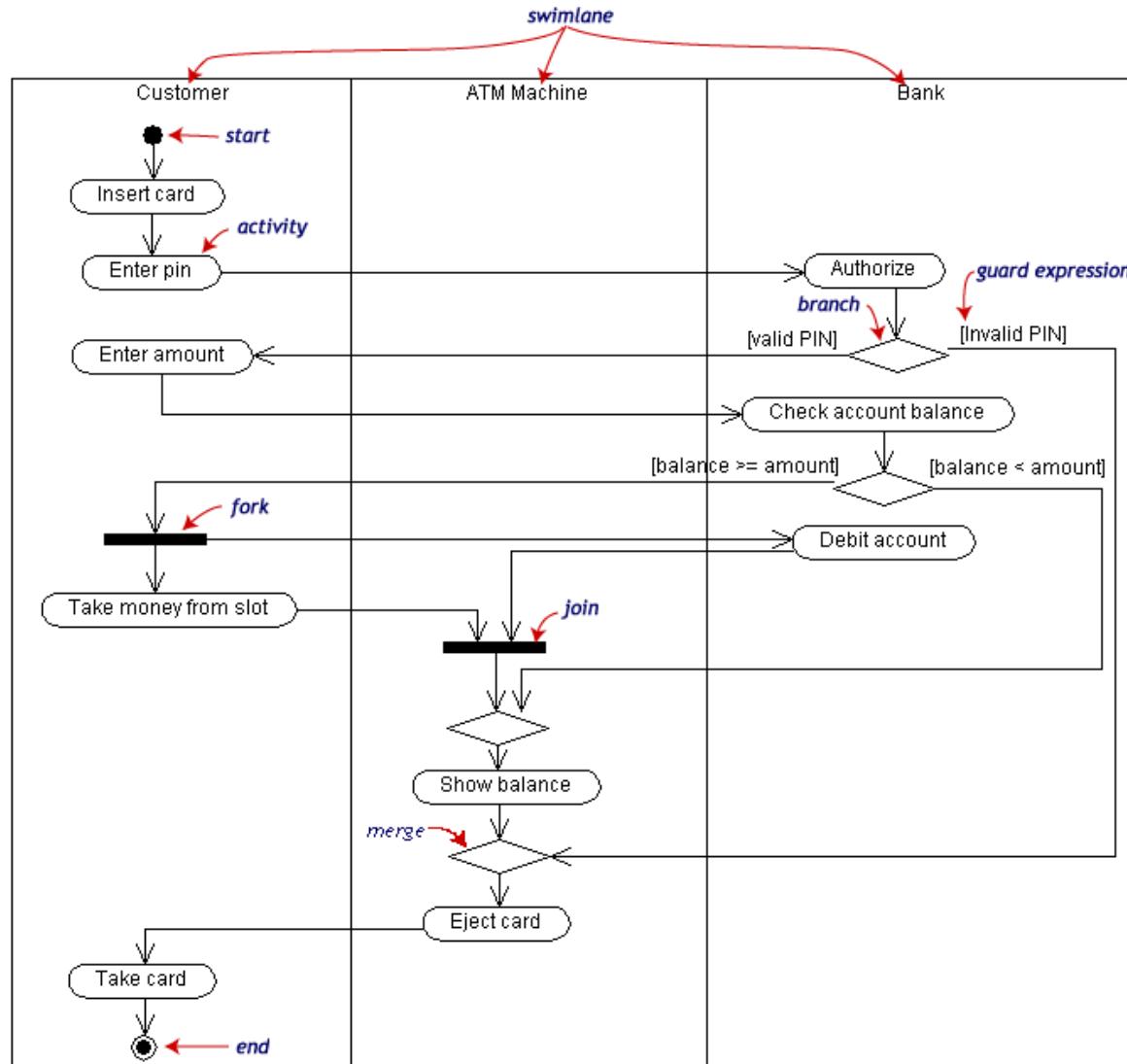


Veiklų suskirstymas

- Leidžia nurodyti, kurie veiklos diagramos veikėjai (aktoriai, sistemos) atlieka konkrečius scenarijaus veiksmus:



Detali veiklos diagramma



Ačiū už dēmesī

UML pagrindai

Struktūros modeliavimas
Prof. dr. Agnus Liutkevičius

Klasės ir objektai

- Objektas – konkreti sistemoje egzistuojanti esybė. Turi vidinę būseną, nusakomą atributų rinkiniu.
- Klasė – objekto tipo aprašas. Aprašo objekto atributų tipus bei galimas operacijas (metodus).

Klasių diagramos paskirtis

- Nusako statinę sistemos struktūrą.
- Aprašo klasses ir sąryšius tarp jų.
- Neaprašo dinaminių sistemos aspektų:
 - Kaip sistemos struktūra kinta jos veikimo metu
 - Kaip keičiasi sąryšiai tarp objektų
 - Dinaminiam aprašymui skirti kiti diagramų tipai

Klasės vaizdavimas

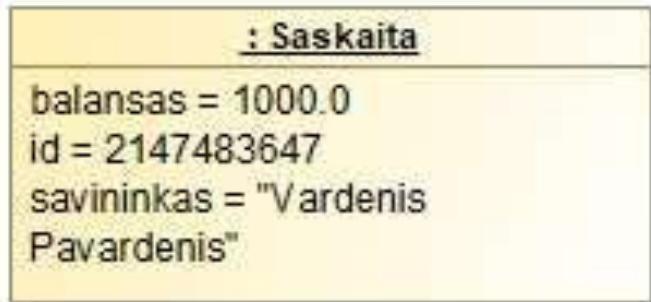
■ Klasė:



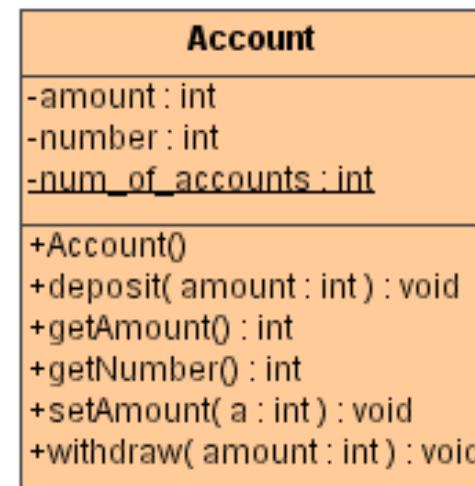
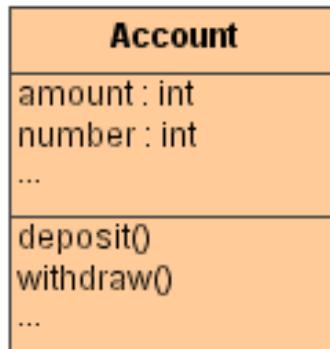
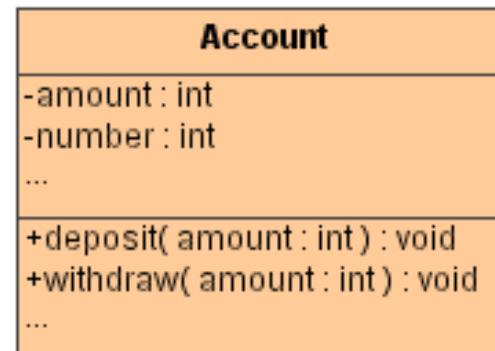
Atributų sritis

Operacijų (metodų) sritis

■ Klasės objektas (realizacija):



Skirtingi vaizdavimo abstrakcijos lygiai



Atributai

- Nusako klasės vidinius kintamuosius
- Gali būti arba primityvus tipas arba kita klasė
- Formalus aprašas:

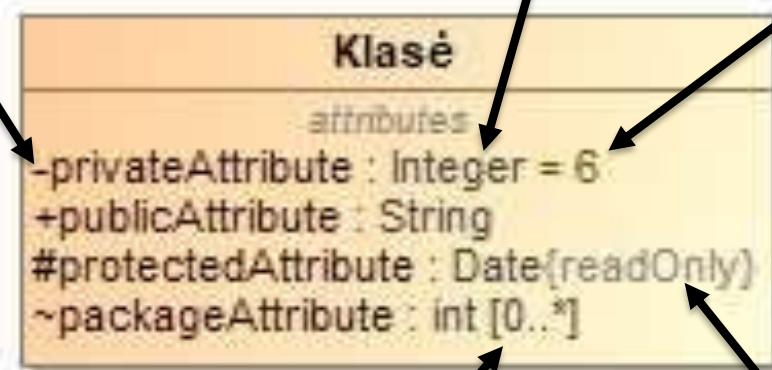
```
visibility name : type_expression
[ = initial_value ]
[ {property_string} ]
```

Atributų vaizdavimas

Atributo matomumas (visibility):

- private (-)
- public (+)
- protected (#)
- package (~)

Atributo tipas



Reikšmė pagal
nutylėjimą (default
value)

Atributo reikšmių
masyvas ir jo dydis
(multiplicity)

Modifikatorius
(modifier): id,
readOnly, ordered,
sequence, unique,
nonunique, union,
redefines,
subsets, *property-
constraint*

Matomumas (prieinamumas)

■ **Private (-)**

- Gali naudoti tik tos klasės objektai.

■ **Protected (#)**

- Gali naudoti ir paveldinčių klasių objektai.

■ **Public (+)**

- Gali naudoti visi objektai.

■ **Package (~)**

- Gali naudoti tik tame pačiame pakete esantys objektai.

Klase
+a : double
#b : int
-c : short
~d : boolean

Operacijos

- Aprašo klasės elgseną, kurią realizuoja metodai
- Metodas – tai detali operacijos elgsenos specifikacija
- Formalus aprašas:

```
visibility name (parameter_list)
[:return_type_expression]
[{property_string}]
```

where:

parameter_list :

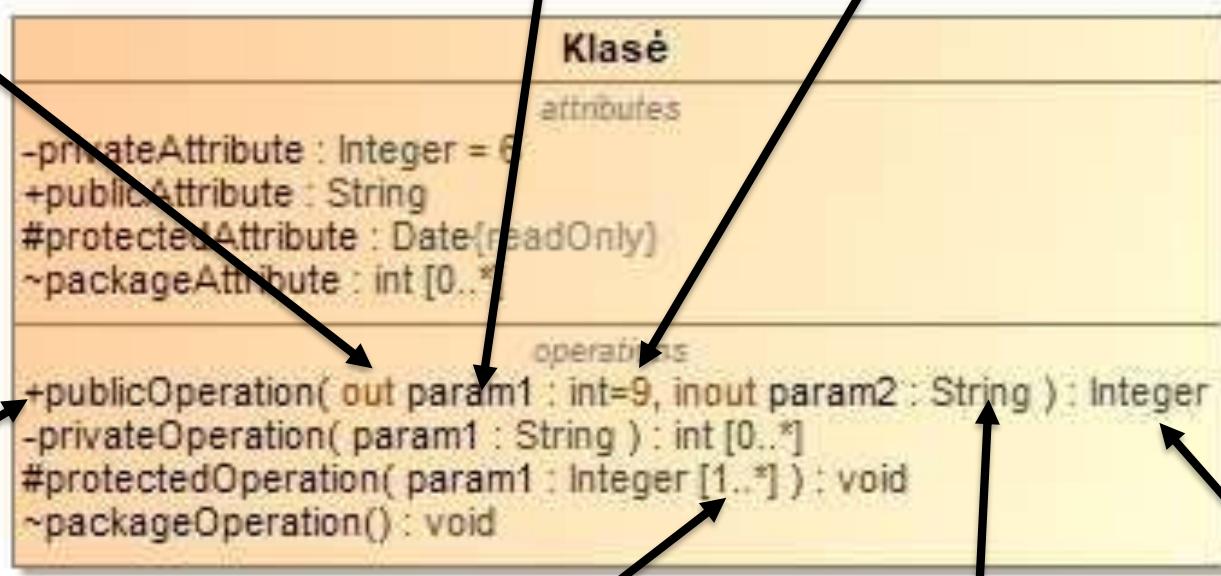
name : type_expression = [default_type]

Operacijų vaizdavimas

Parametru kryptis
(direction): in, out,
inout, return

Operacijos
parametras

Parametru reikšmė
pagal nutylėjimą
(default value)



Operacijos matomumas (visibility):

- private (-)
- public (+)
- protected (#)
- package (~)

Parametru reikšmių
masyvas ir jo dydis
(multiplicity)

Parametru tipas

Grąžinamas
parametras (return)

Statiniai atributai ir metodai

- Bendri visai klasei
- Pasiekiami ne per objekta^ą, o per klasę
 - Pvz. Account.createAccount();
- Naudojama globaliems duomenims bei veiksmams aprašyti

Account
-amount : int
-number : int
-num_of_accounts : int
+createAccount()
+deposit(amount : int) : void
+withdraw(amount : int) : void
...

Realizacija Java kalboje

```
public class Account
{
    // attributai
    private int amount;
    private int number;
    private static int num_of_accounts;

    //konstruktorius
    public Account ()
    {
        this.number = num_of_accounts;
        num_of_accounts++;
    }

    public void deposit(int amount)
    {
        this.amount += amount;
    }

    public void withdraw(int amount)
    {
        this.amount -= amount;
    }

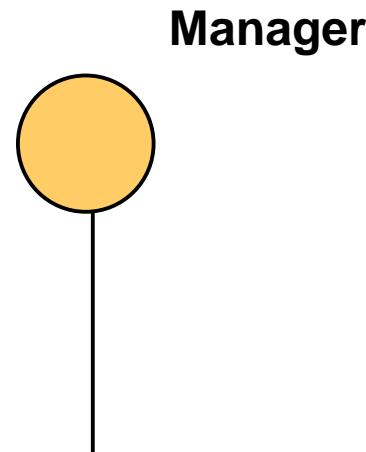
    public int getAmount() {return this.amount; }
    public int getNumber() {return this.amount; }
    public void setAmount(int a) { this.amount = a; }
}
```

Account
-amount : int
-number : int
<u>-num_of_accounts : int</u>
+Account()
+deposit(amount : int) : void
+getAmount() : int
+getNumber() : int
+setAmount(a : int) : void
+withdraw(amount : int) : void

Interfeisai (angl. interface) (ne grafinės sąsajos!)

- Interfeisas aprašo operacijas, kurias privalo realizuoti interfeisą paveldinčios (realizuojančios) klasės
- Nusako **privalomą** realizuoti funkcionalumą ir savybes
- Iš interfeiso neįmanoma sukurti objektų – tai ne klasė, bet operacijų aprašų šablonas paveldinčiai klasei!

Interfeisai



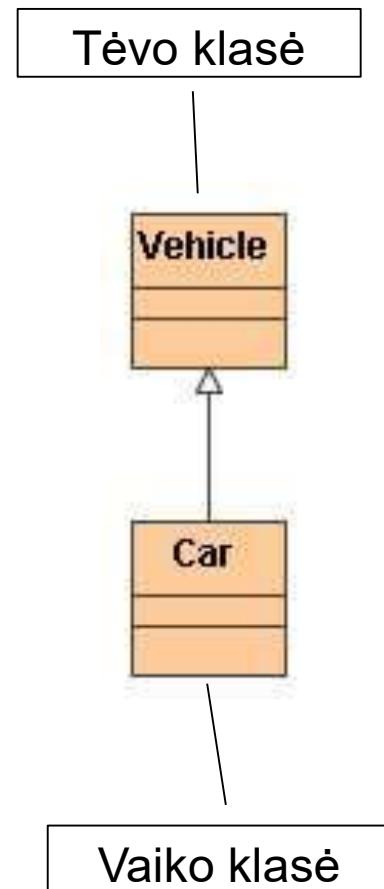
- Interfeisą realizuojanti klasė turės realizuoti metodus deploy() ir manage()
- Žymima kaip klasė su apskritimu (arba tik kaip apskritimas)
- Kartais papildomai žymima stereotipu <<interface>>

Ryšiai tarp klasių

- Paveldėjimas, apibendrinimas (angl. generalization)
- Asociacija (angl. association)
 - Agregacija (angl. aggregation)
 - Kompozicija (angl. composition)
- Realizacija (angl. refinement, realization)
- Priklausomybė (angl. dependency)

Paveldėjimas

- *is-a* sąryšis
- Nusako klasių hierarchinius sąryšius
- Visi atributai ir metodai paveldimi iš tėvinės klasės



Paveldėjimo sąryšio naudojimo dažniausia klaida

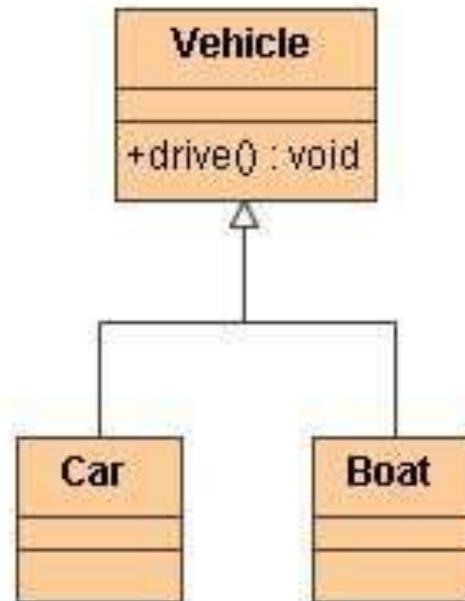
- Paveldėjimas nėra skirtas parodyti sąveikai tarp klasių – tam skirta asociacija arba priklausomybė!
- Iš esmės paveldėjimas pasako, kad viena klasė yra kitos klasės patobulintas variantas (vaiko klasė gali turėti papildomų atributų ir metodų lyginant su tėvo klase)

Polimorfizmas

```
public class Vehicle
{
    // implementation
    public void drive()
    {
        // ...
    }
}

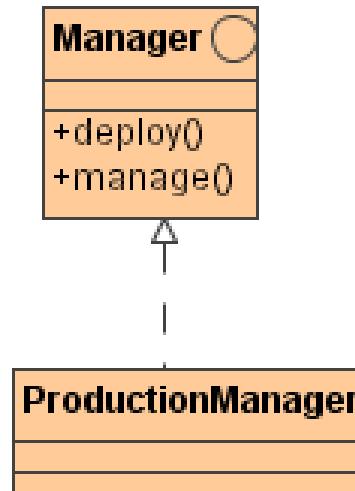
public class Car extends Vehicle
{
    // ....
}

public class Boat extends Vehicle
{
    // ...
}
```



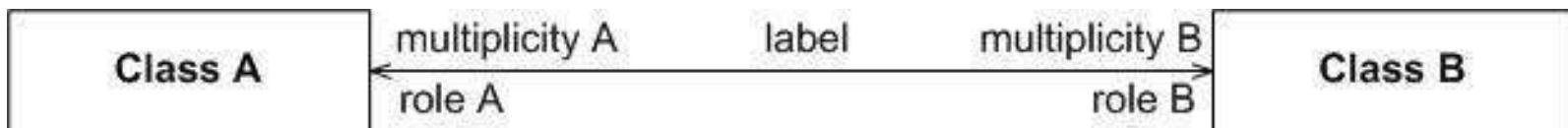
Realizacija

- Žymimas interfeiso realizavimas
- Nuo paveldėjimo skiriiasi tuo, kad aprašyti metodai ne paveldimi, o turi būti realizuoti
- Taikoma ir abstrakčioms klasėms
 - Abstrakti klasė – tai klasė turinti nerealizuotų metodų (neturinčių atitinkamo programinio kodo).



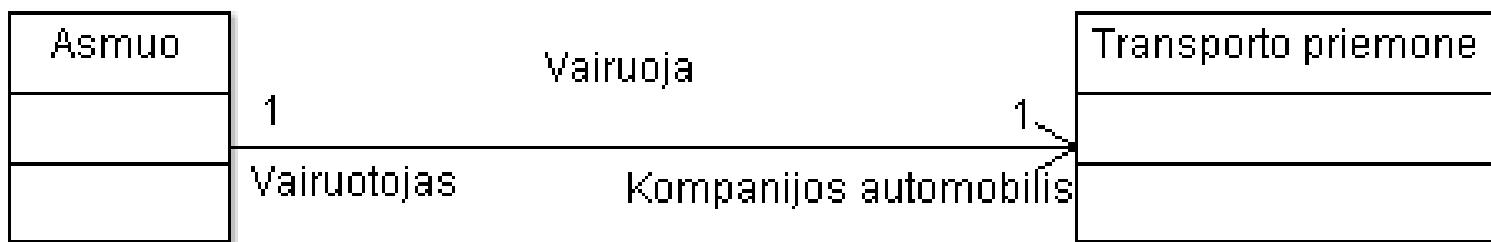
Asociacijos

- Nusako semantinį ryšį tarp klasių
- Dvipusis arba vienpusis ryšys



Asociacijos savybės

- Pavadinimas (pvz. vairuoja)
- Kardinalumai (1, 1..*, 0..*, * ir pan.)
- Rolės / vaidmenys (pvz. vairuotojas, kompanijos automobilis)

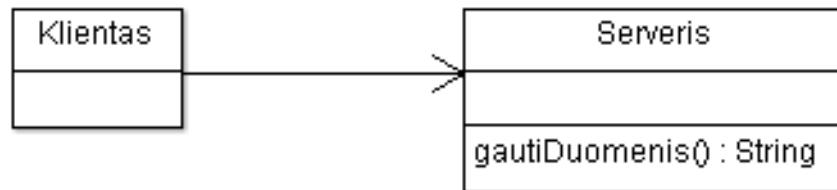


Asociacijos prasmė ir naudojimas (1)

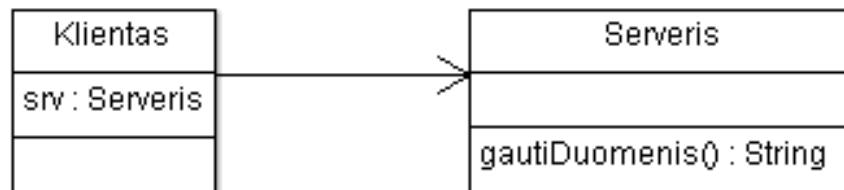
- Asociacija skirta parodyti, kad vienos klasės objektai naudojasi kitos klasės objektais(-u), t.y. naudojasi tų objektų:
 - Operacijomis
 - Atributais
- Tam, kad naudotis kitos klasės objektais, reikia turėti nuorodą į tos klasės objektus, todėl asociacija programiškai realizuojama kaip:
 - Klasės **atributas**, kurio tipas atitinka klasę, kurios metodais ar atributais yra naudojamasi

Asociacijos prasmė ir naudojimas (2)

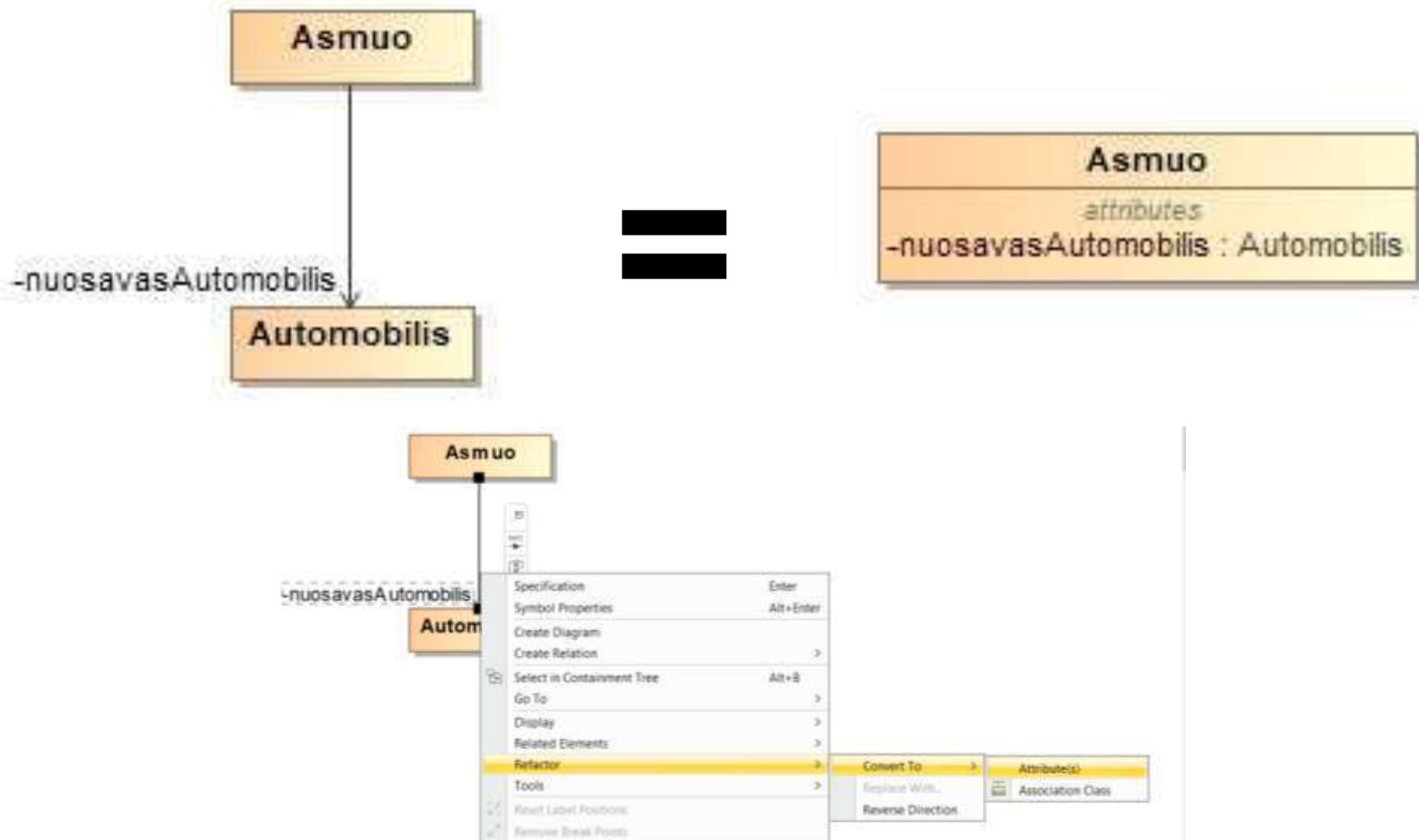
- Asociacijos rodyklė rodo į tą klasę, kurios metodais ar atributais yra naudojamasi:



- Klasės “Klientas” objektai naudojasi “Serveris” klasės objektu (pvz. kviečia metodą “gautiDuomenis”), kas reiškia, jog “Klientas” atributų sąraše yra “Serveris” tipo atributas (šiuo atveju “srv”):

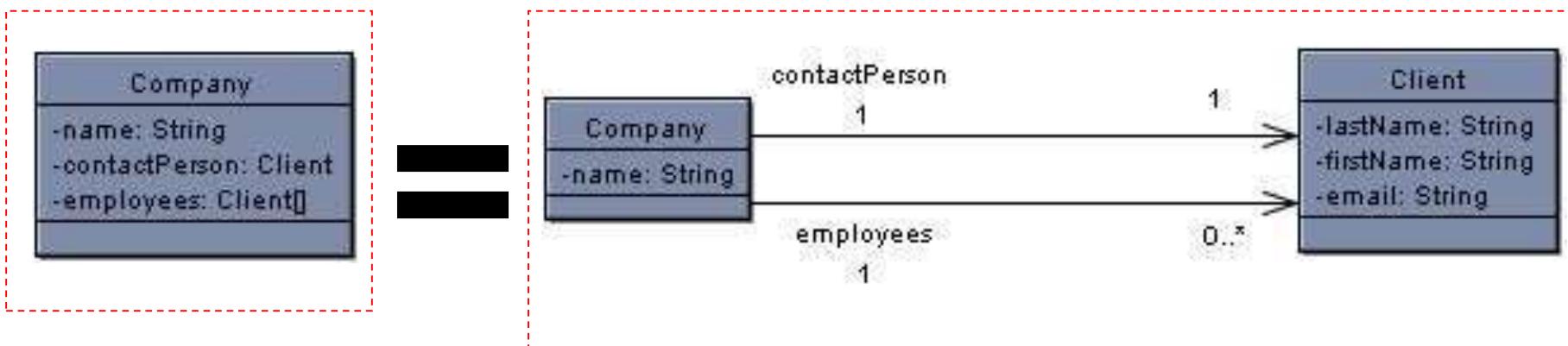


Asociacija = atributas



Asociacijų su kardinalumais realizacija

- Priklausomai nuo asociacijos kardinalumo, asociacija atitinka atributą su viena reikšme arba atributą su reikšmių masyvu
- Pvz. “employees” realizuojamas kaip masyvas, nes atitinkama asociacija turi $0..*$ kardinalumą.



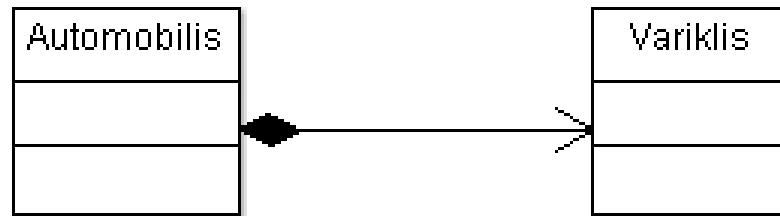
Dažniausia asociacijos naudojimo klaida

- Asociacijos kryptis nebūtinai atitinka duomenų perdavimo kryptį!
- Asociacijos rodyklė parodo kreipimosi į kitos klasės objektą kryptį, bet ne duomenų perdavimą:
 - Pvz. jeigu kreipiamausi į metodą, kuris gražina duomenis, tai duomenys bus perduodami priešinga nei rodyklė kryptimi...

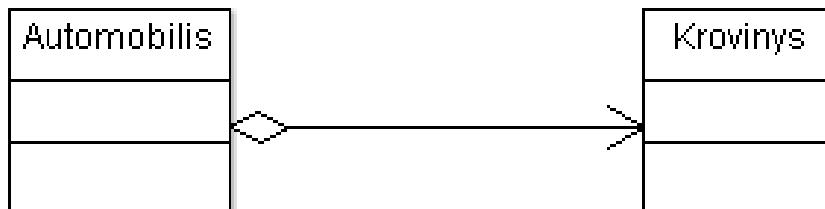
Asociacijų subtipai: agregacija ir kompozicija

- *whole/part* ryšiai
- Pasako, jog viena klasė susideda iš kitų klasių (dalių):

- Kompozicija



- Agregacija



Agregacijos ir kompozicijos skirtumas

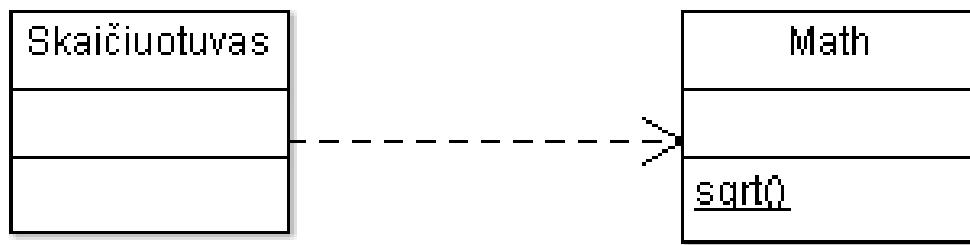
- Kompozicija stipresnis ryšys:
 - Kompozicija parodo, kad klasės dalyvaujančios kompozicijoje, negali viena be kitos funkcionuoti
 - Kompozicijos atveju, vienos klasės objektas visą savo gyvavimo laiką yra kitos klasės dalis
 - Agregacija parodo, kad tam tikrais laiko momentais (bet nevisada!) vienos klasės objektas(-ai) tampa kitos klasės objekto(-ų) dalimi
- Tie k agregacija, tie kompozicija yra ta pati asociacija! (realizuojama kaip klasės atributas)

Priklausomybė (angl. dependency)

- Naudojama netiesioginėms priklausomybėms tarp klasių nurodyti:
 - Vienai klasei perduodamas kitos klasės objektas kaip metodo argumentas
 - Naudojami statiniai klasės atributai ar metodai
- Prasmė tokia pat kaip asociacijos:
 - Parodo, kad vienos klasės objektas kviečia kitos klasės objekto(-ų) metodus ar atributus

Priklausomybės realizacija

- Nuo asociacijos skiriasi programine realizacija:
 - Nėra atributo, kuris atitiktų kviečiamos klasės objekta
 - Nuoroda į kviečiamą objektą gaunama dinamiškai arba kviečiami statiniai metodai, kas nereikalauja objektų kūrimo, pvz.:

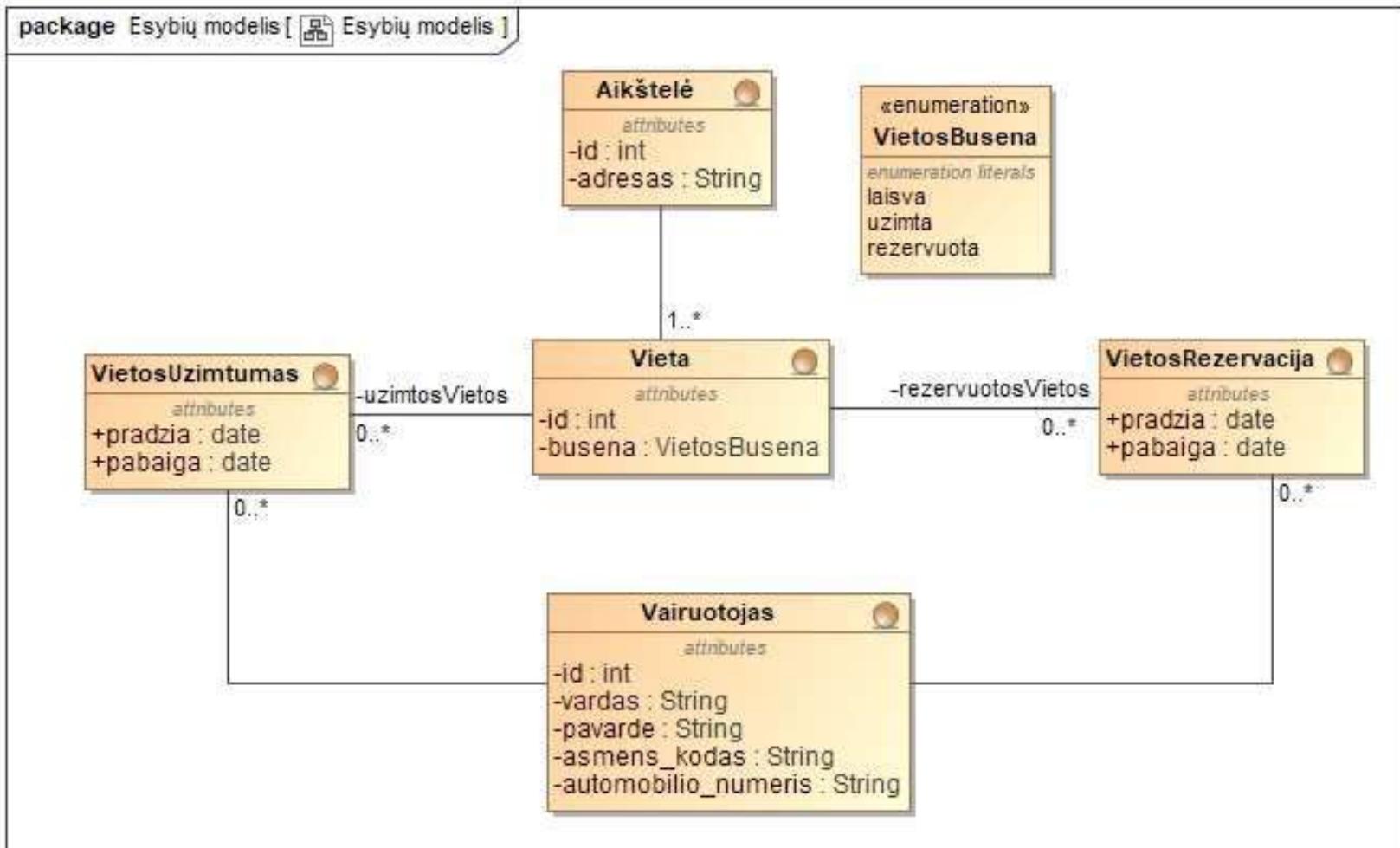


- “Skaičiuotuvas” neturi atributo, kurio tipas “Math”, nes metodas `sqrt()` yra statinis

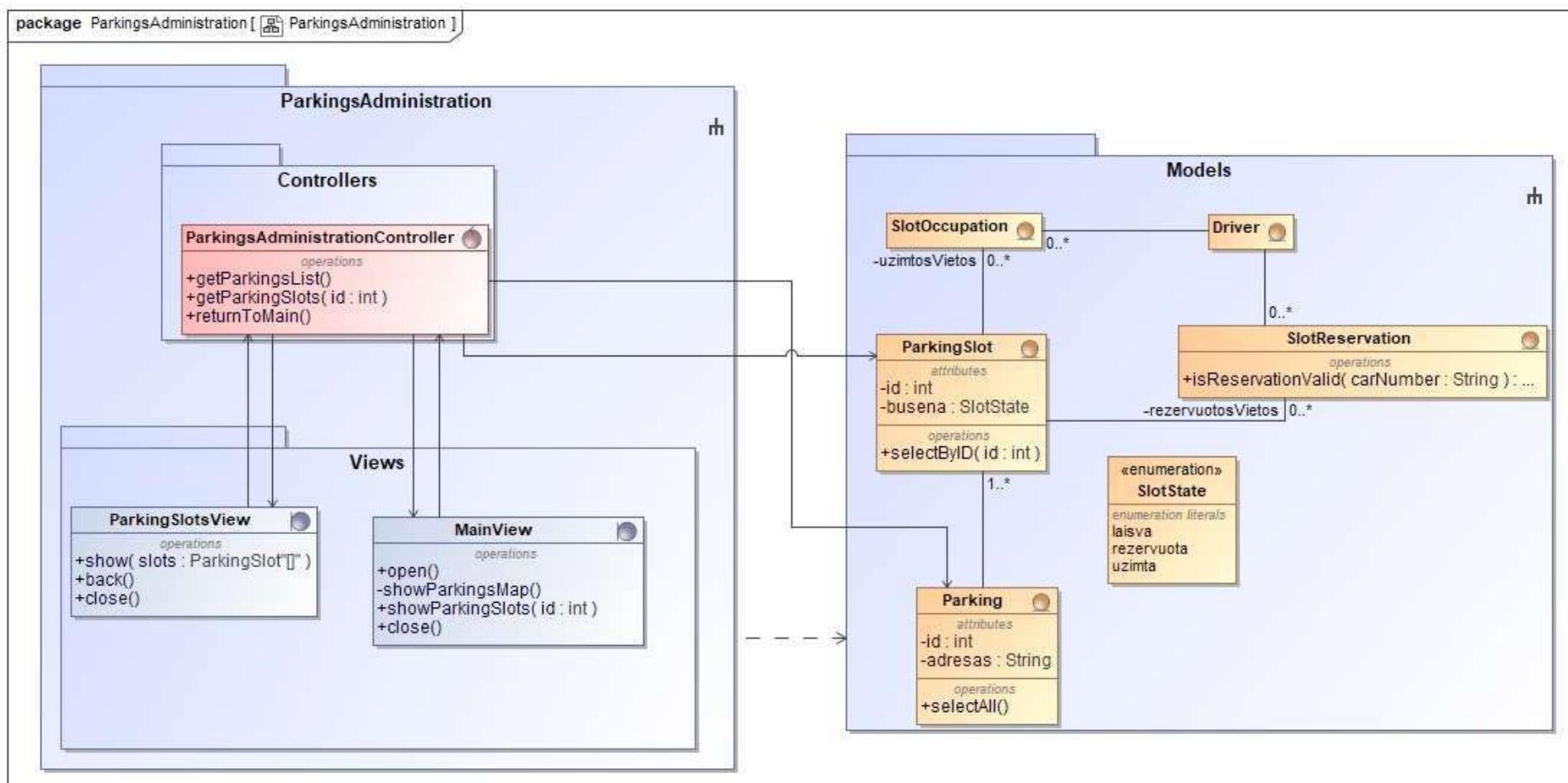
Dalykinės srities modelis

- Dalykinės srities modeliui (duomenų modeliui) naudojame esybių klasses su <<entity>> stereotipu.
- Esybės neturi metodų ir pačios neinicijuoja jokių veiksmų.
- Alternatyva ER modeliams, naudojamiems reliacinių duomenų bazių projektavime.

Dalykinės srities modelio p̄vz.

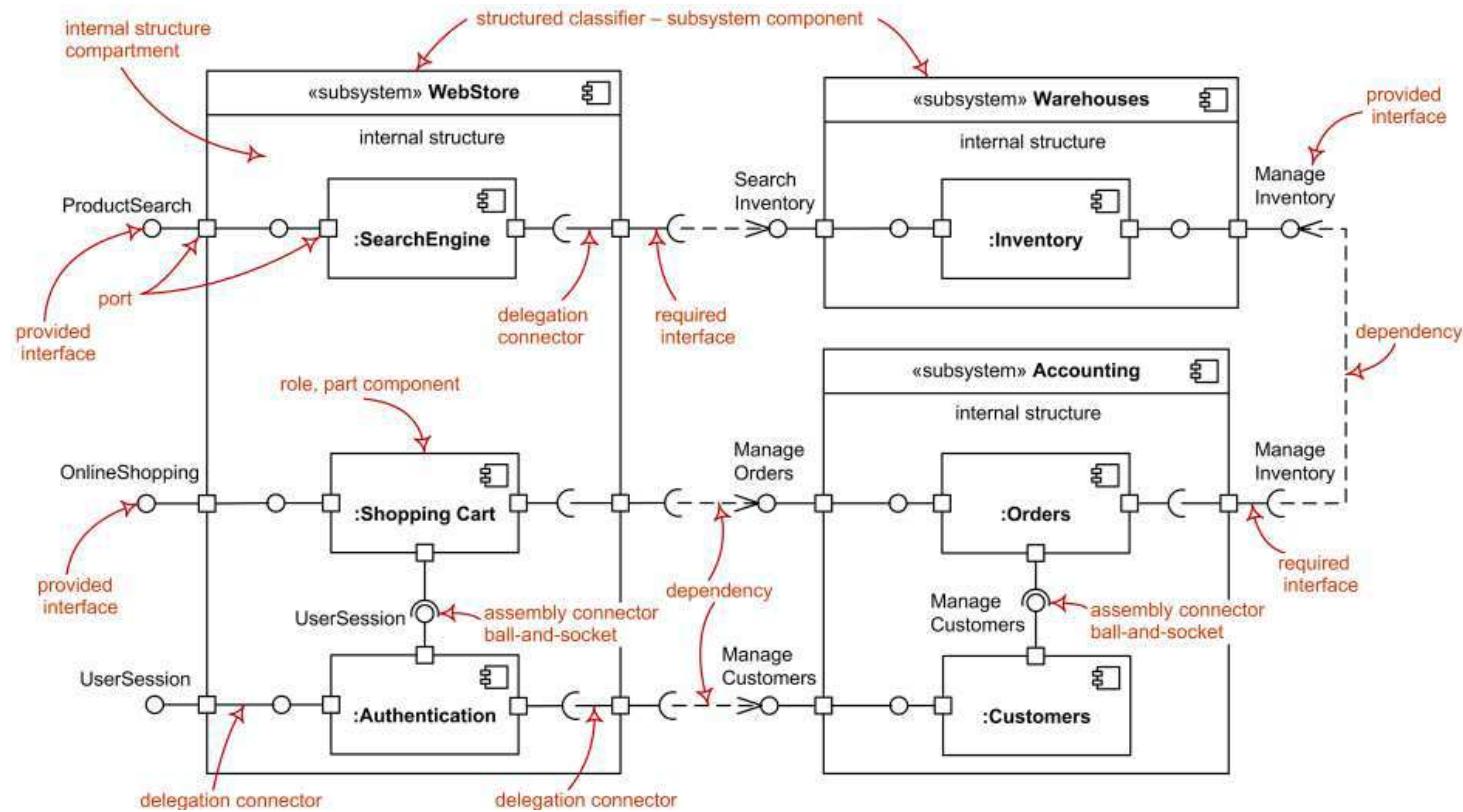


Klasių diagramos pavyzdys (MVC projektavimo šablonas)



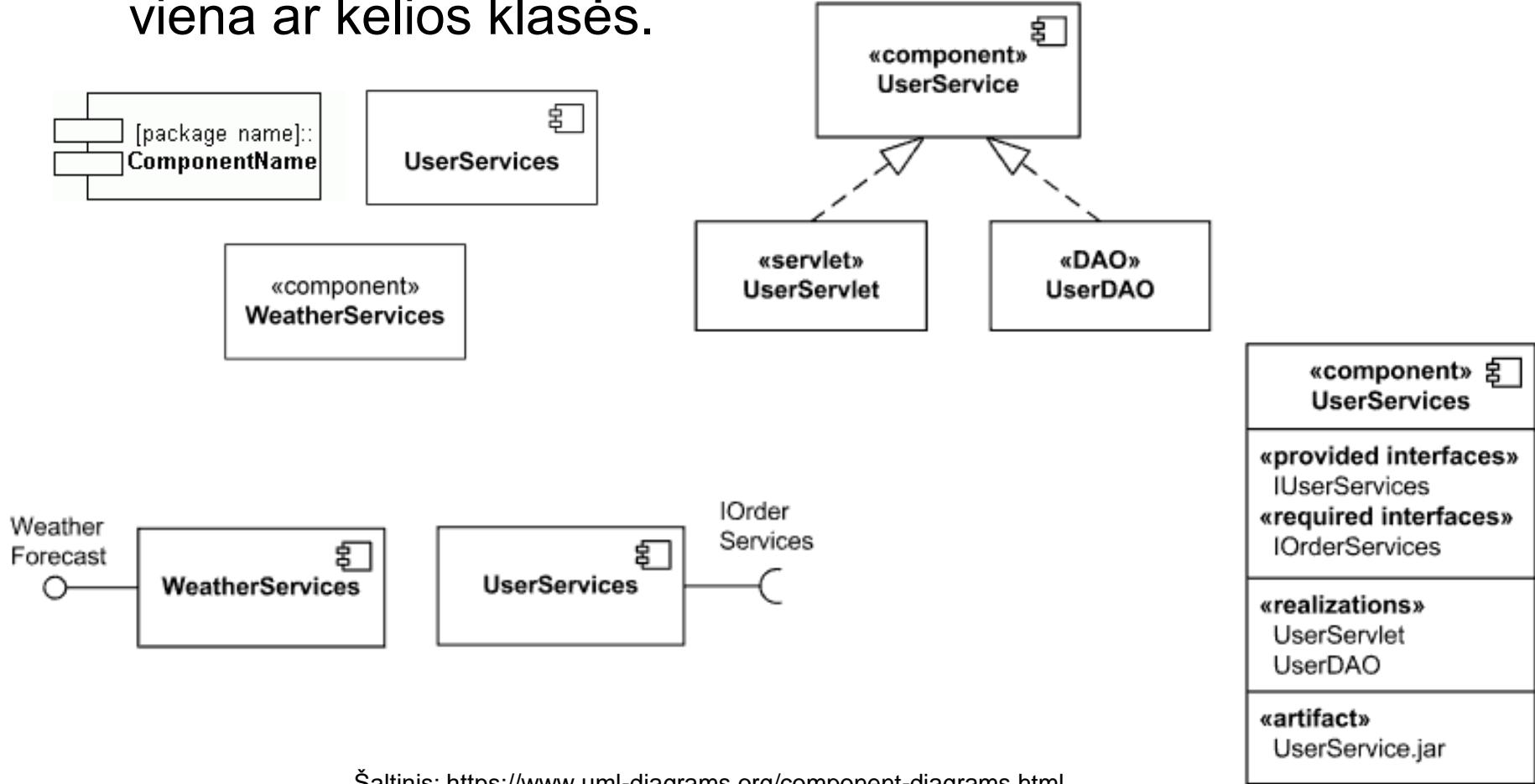
Komponentų diagrama

- Komponentų diagrama parodo sistemos programinės įrangos komponentus ir sąryšius tarp jų.



Komponentas

- Komponentas turi sāsajas komunikavimui, o jī realizuoja viena ar kelios klasēs.



Artefaktai, realizuojantys komponentus

- Artefaktas (angl. artifact) tai fizinė esybė, informacijos dalis, kuri naudojama ar sukuriama sistemos kūrimo ar diegimo metu:
 - tekstiniai dokumentai;
 - programos kodo failai;
 - vykdomieji failai;
 - modelių failai;
 - bibliotekos;
 - archyvai;
 - DB failai (lentelės) ir t. t.

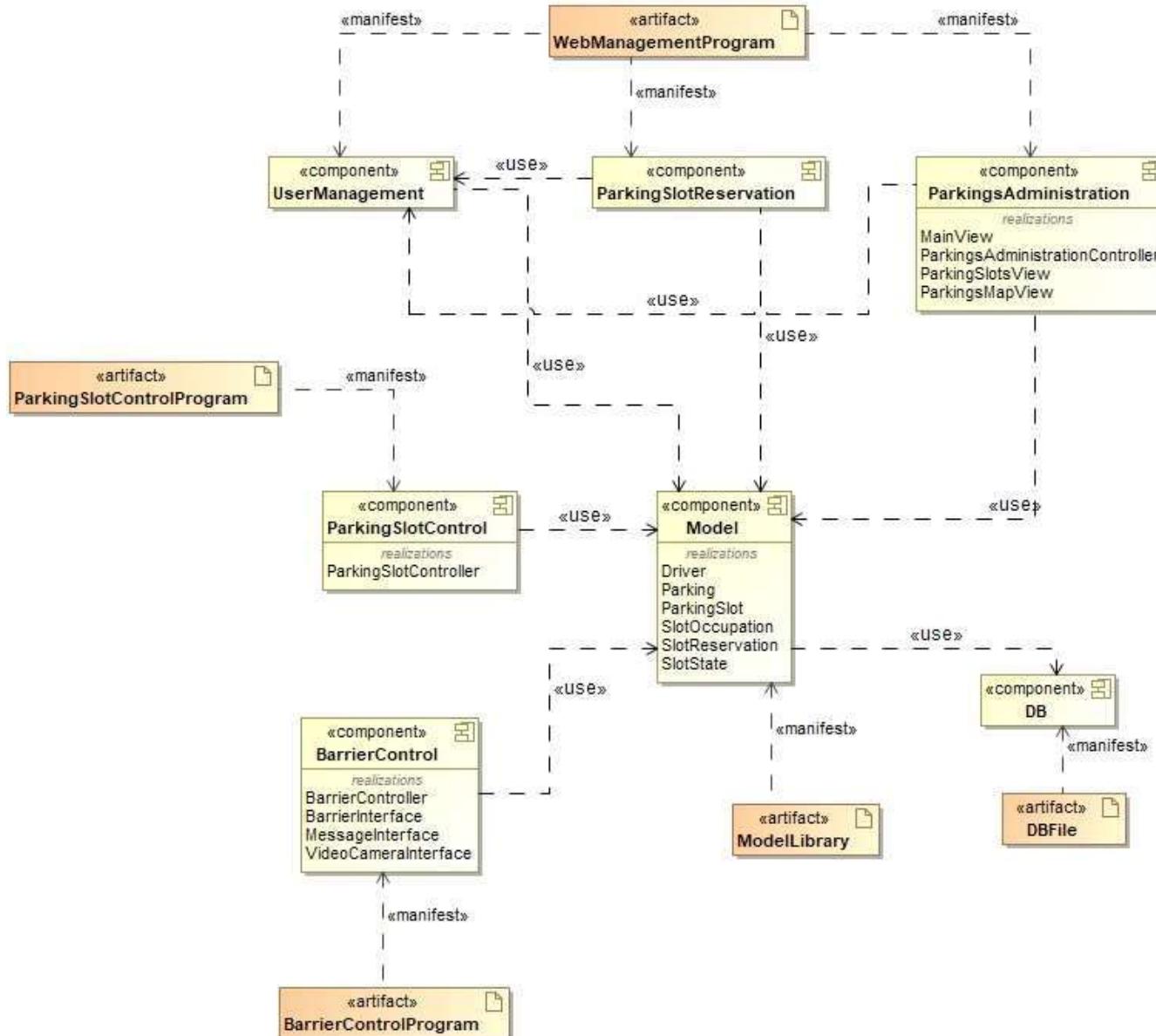


Manifest ryšys

- <<manifest>> ryšys tarp artefakto ir komponento parodo, kad artefaktas realizuoja tą komponentą (komponentas yra artefakto sudėtinė dalis).



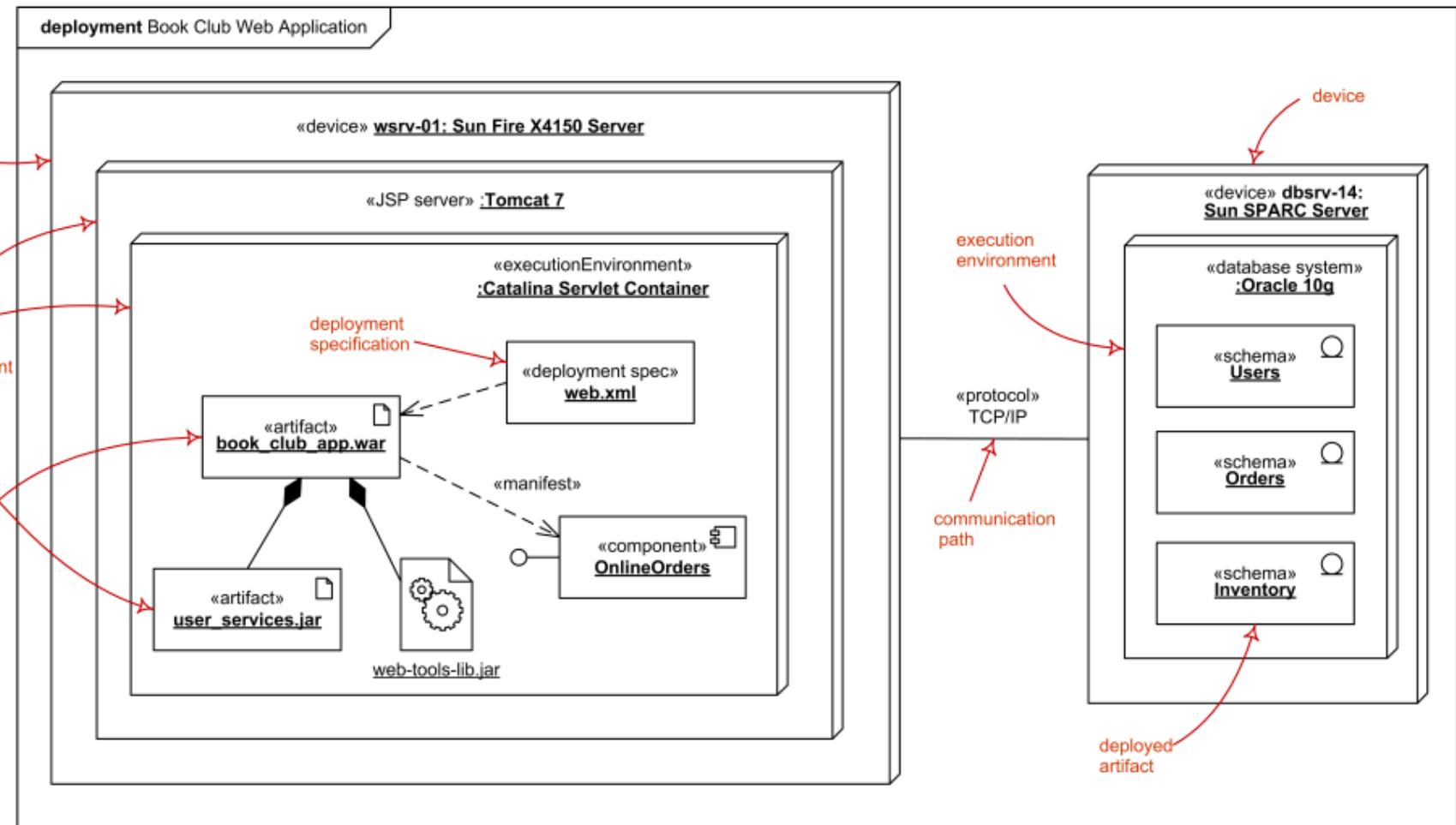
Komponentų diagramos pavyzdys.



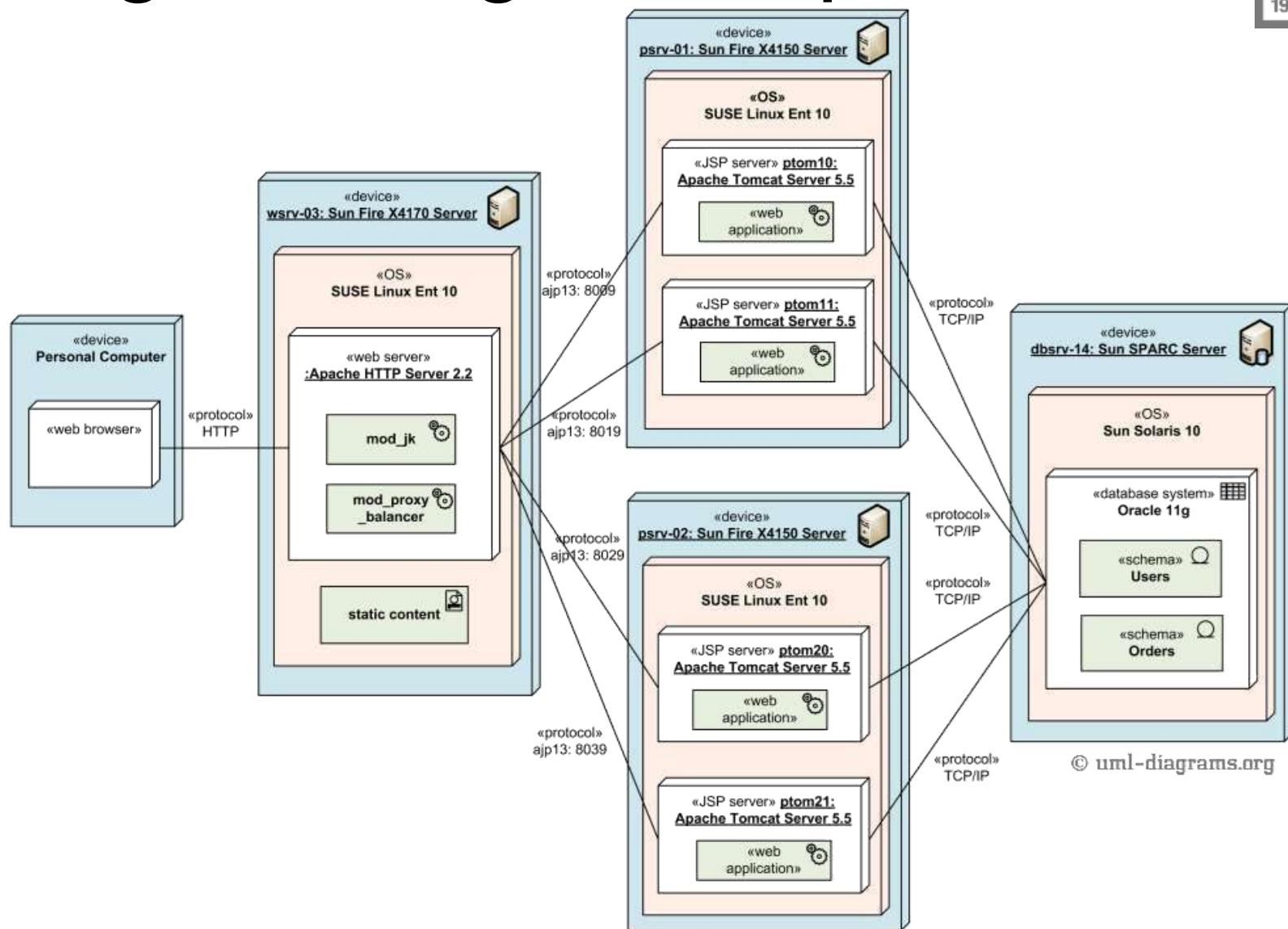
Diegimo (angl. deployment) diagrama

- Deployment diagrama parodo:
 - Fizinę sistemos architektūrą;
 - Sąryšį tarp sistemos programinės įrangos ir aparatūrinės įrangos (kur diegiamą programinę įrangą).
- Diagramoje naudojami artefaktai ir mazgai.
- **Mazgas** yra techninės įrangos komponentas (<<device>>) arba programinės įrangos vykdymo aplinka (<<execution environment>>), pvz. virtuali mašina, web serveris ir pan.
 - UML 1.x komponentai buvo vaizduojami tiesiogiai mazguose, o nuo UML 2.x versijos artefaktai yra diegiami mazguose, tačiau galima nurodyti, kokių komponentus tie artefaktai realizuoja (angl. <<manifest>>).

Diegimo diagramos dalys

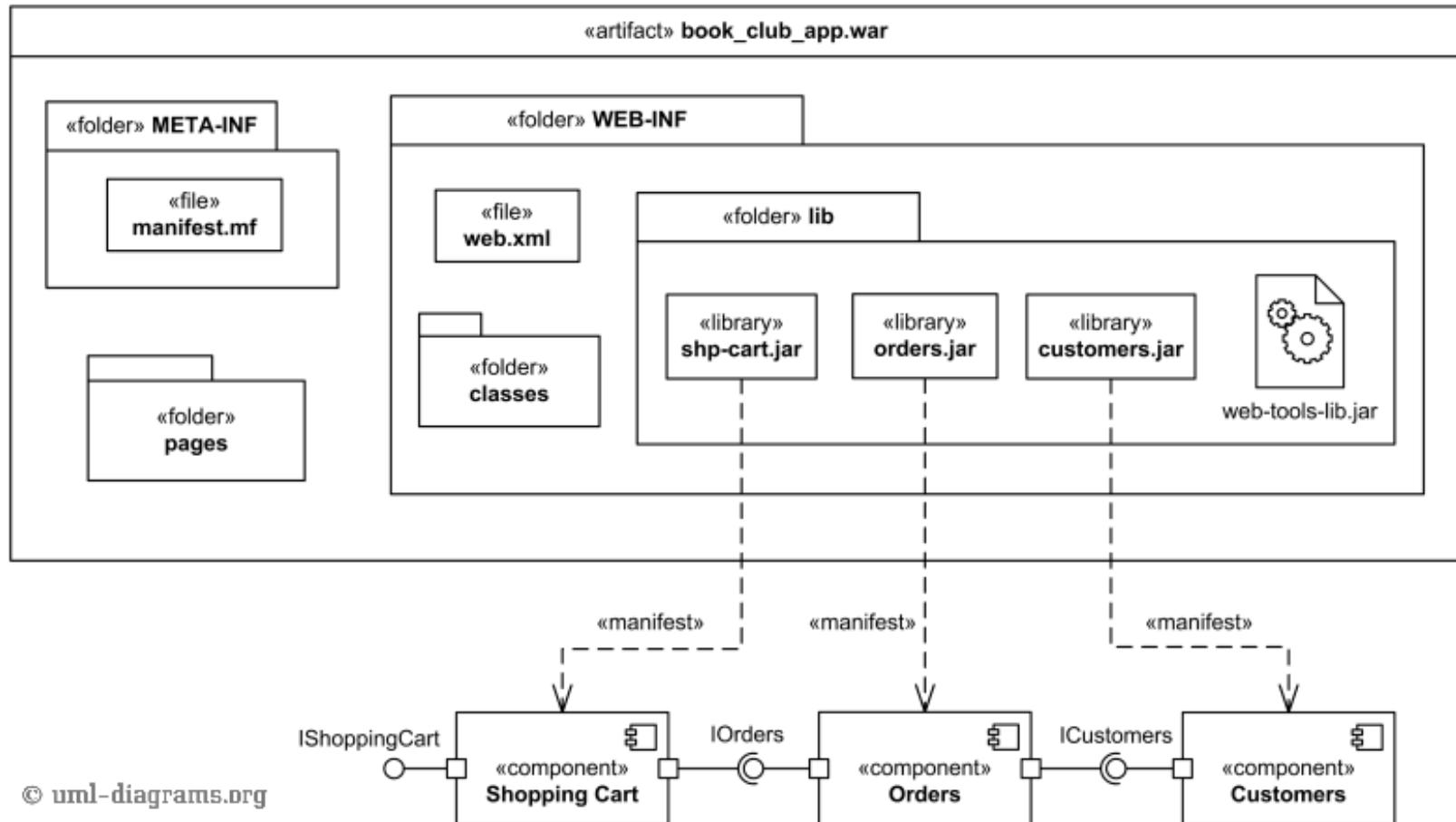


Diegimo diagramos pvz. 1

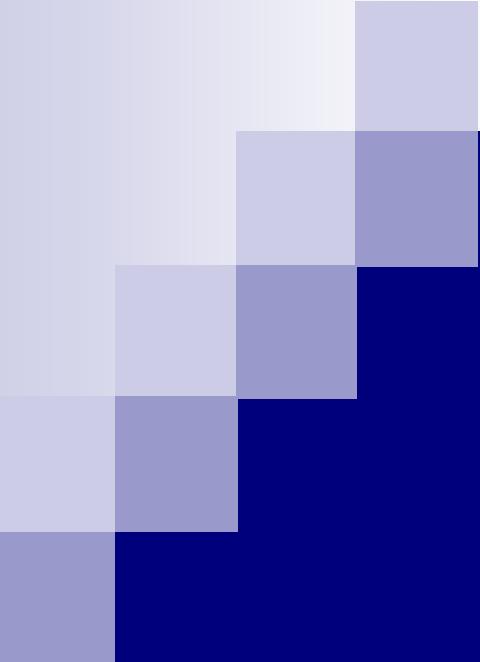


© uml-diagrams.org

Diegimo diagramos pvz. 2



Ačiū už dėmesį



UML pagrindai

Elgsenos modeliavimas
Prof. dr. Agnus Liutkevičius

Elgseną aprašančios diagramos

- Būsenų (angl. state, statechart)
- Sekų (angl. sequence)
- Komunikavimo (angl. communication)
- Apžvelgta anksčiau: scenarijų (activity)

Būsenų diagrama (1)

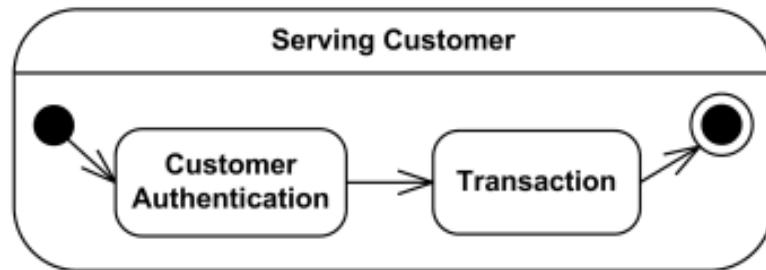
- Būsenų diagrama aprašo visą galimą klasės objekto ar kito modelio elemento (panaudojimo atvejo, interfeiso , protokolo,...) elgseną.
- Formalus modelis – automatas (angl. state machine). Aprašomas orientuotu grafu, kurio viršūnės žymi būsenas, o briaunos – perėjimus.
- Nuo UML 2.4 versijos, apibrėžti du automatų tipai:
 - elgsenos;
 - protokolo.

Būsenų diagramma (2)

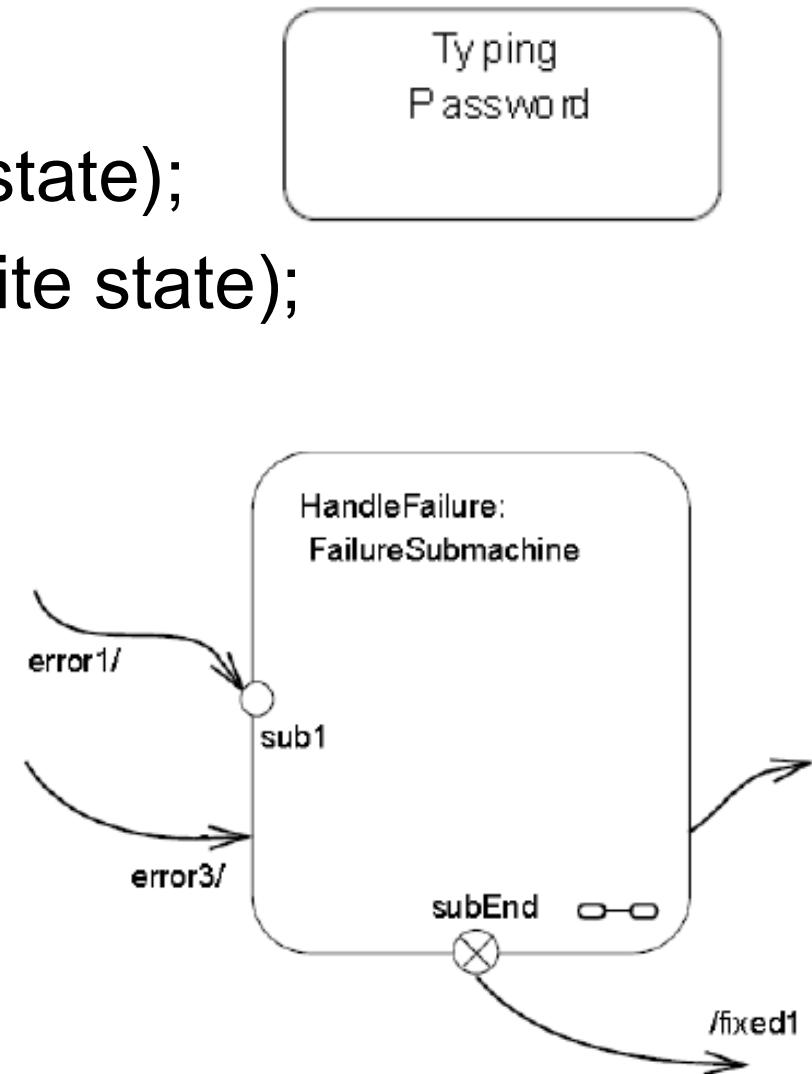
- Specifikuoja objekto gyvavimo ciklą, nusakantį visus įmanomus elgesio variantus.
- Būsena yra situacija objekto gyvavimo metu, kai jis:
 - tenkina tam tikrą sąlygą;
 - laukia tam tikro įvykio;
 - vykdo tam tikrą veiksmą.
- Pasyvių objektų (esybių) būsenas galima apibrėžti jų atributų reikšmių rinkiniu.

Objekto būsena (1)

- Būsena gali būti:
 - paprasta (angl. simple state);
 - sudėtinė (angl. composite state);

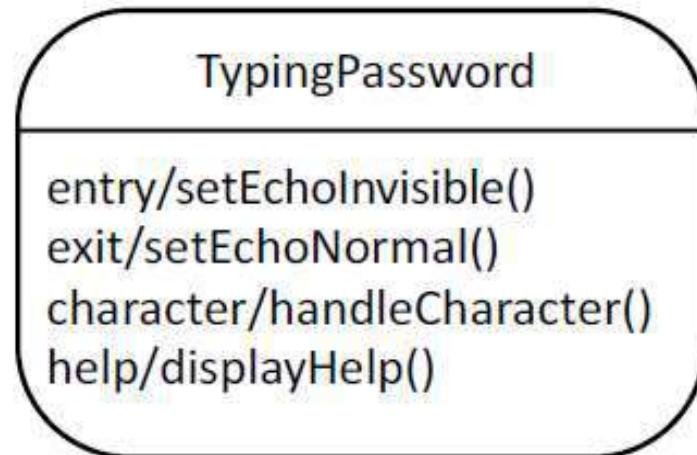


- submašinos
(angl. submachine state).



Objekto būsena (2)

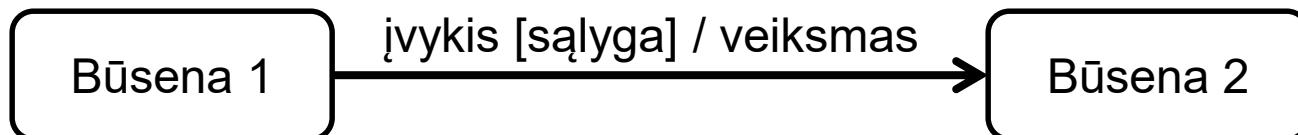
- Būsena **privalo** turėti vardą ir **gali** turėti veiksmus:
 - entry, exit, do;
 - nestandardinius (savo sugalvotus).



Perējimas (angl. transition)

- Perējimas (angl. transition) gali turēti:
 - įvykį, kuris perējimą sukelia (angl. trigger);
 - sąlygą (angl. guard), kuri turi galioti, kad perējimas įvyktų;
 - veiksmą (angl. behavior expression).

[<trigger> [‘,’ <trigger>]* [‘[’ <guard>’]’] [/’ <behavior-expression>]]



Perėjimo prasmė

- Perėjimas tarp būsenų paprastai yra susietas su įvykiu.
- Jei įvykis nenurodytas, perėjimas įvyksta, kai įvykdomi vidiniai būsenos veiksmai (automatinis perėjimas).

Perėjimo sąlyga ir veiksmai

■ **guard:**

- boolean išraiška;
- turi būti tenkinama, kad perėjimas įvyktų.

■ **behavior-expression:**

- veiksmas, kurį atliekame perėjimo metu;
- gali būti ir programinis kodas.

Pseudo būsenos (1)



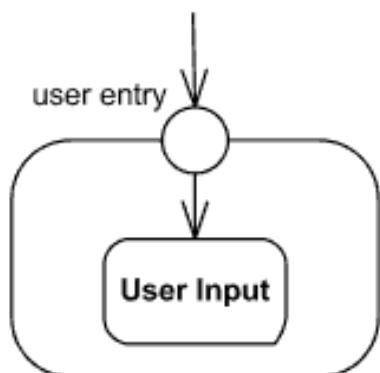
Pradžios (initial state) būsena



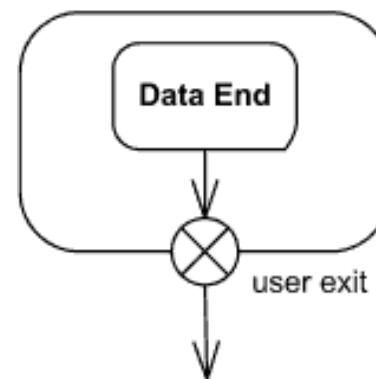
Pabaigos (final state) būsena



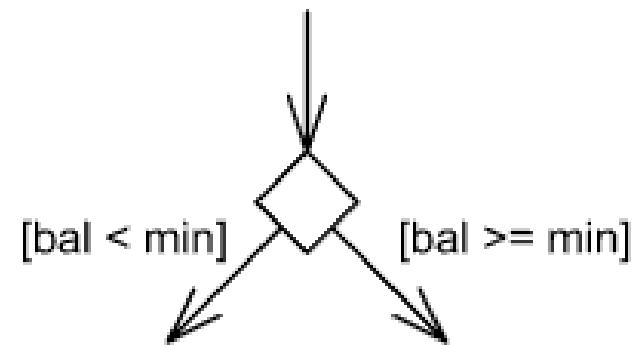
Nutraukimas (terminate)



Iėjimo taškas (entry point) – sudėtinėms būsenoms

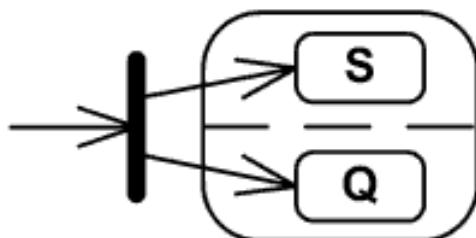


Išėjimo taškas (exit point) – sudėtinėms būsenoms

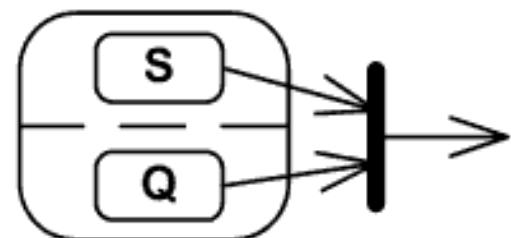


Sprendimas (choice)

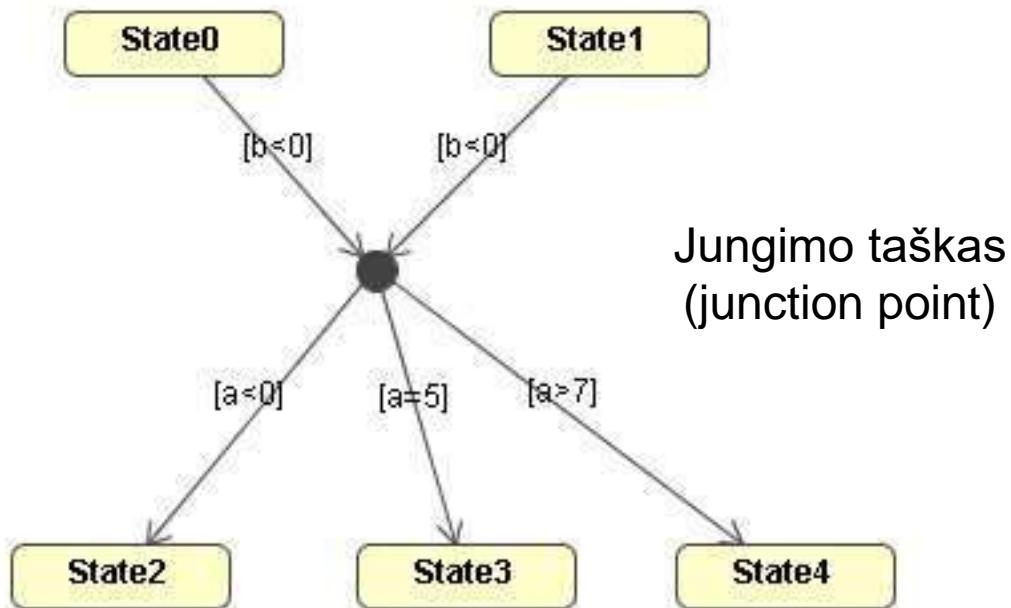
Pseudo būsenos (2)



Išlygiagretinimas (fork)



Suliejimas (join)

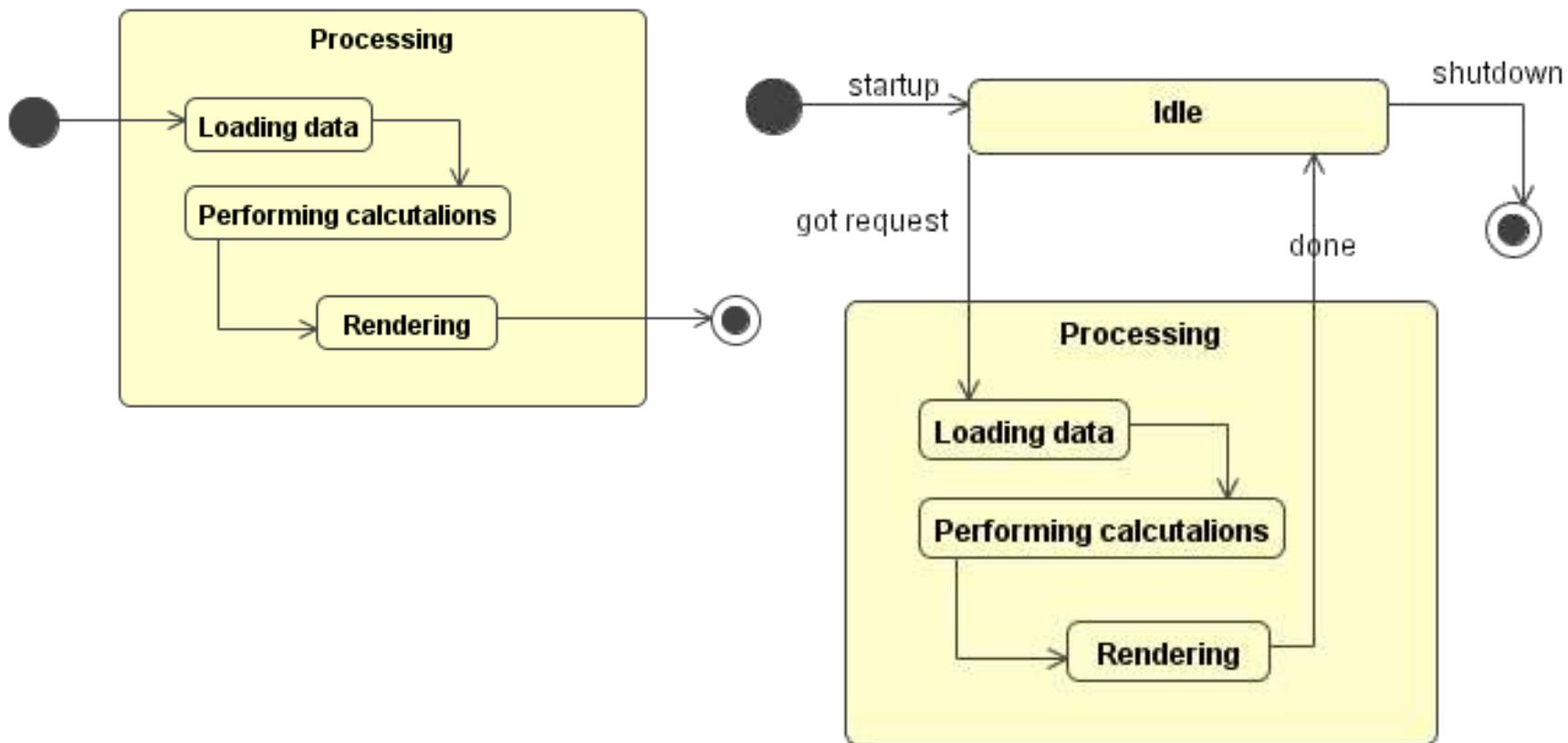


Jungimo taškas
(junction point)

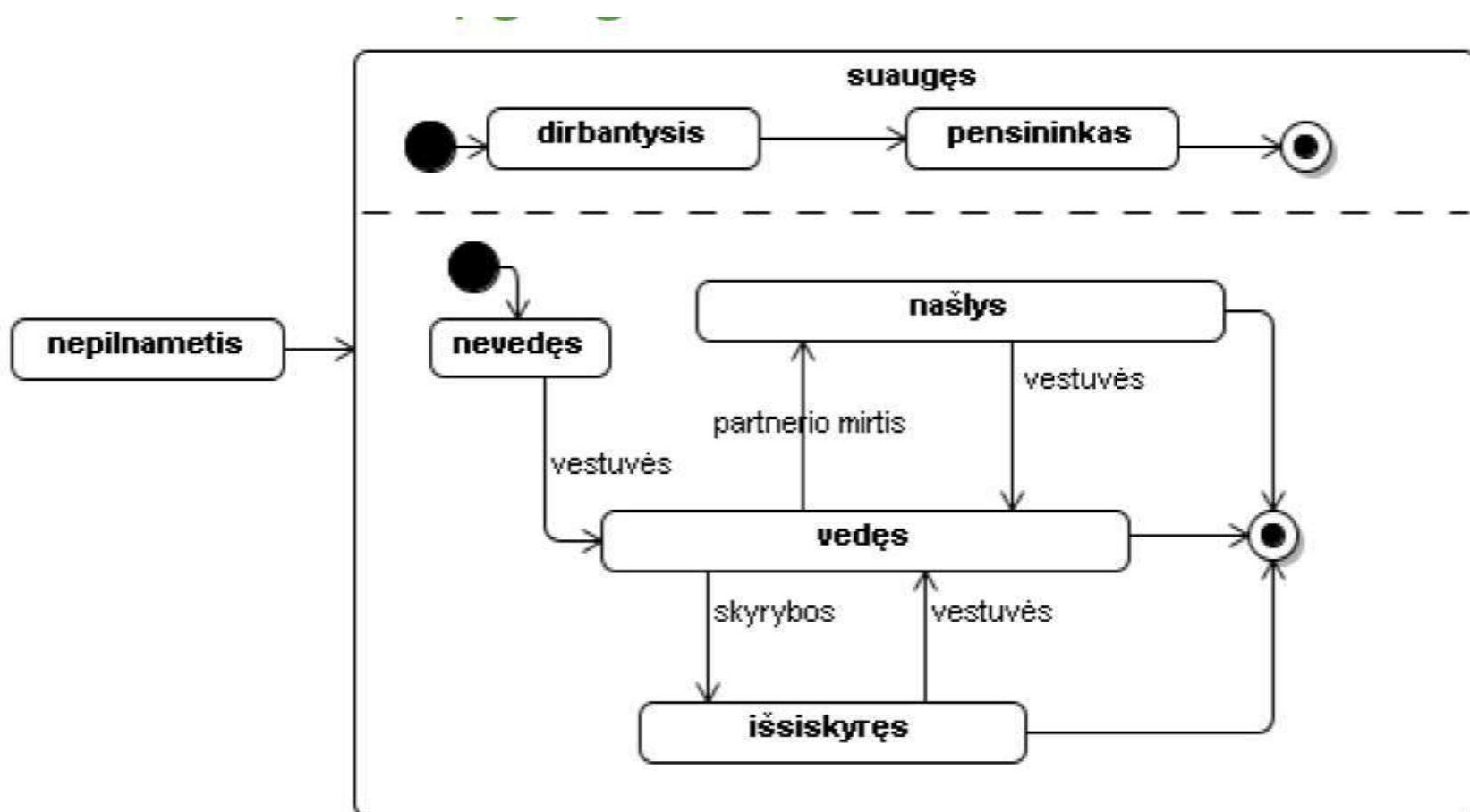
Sudėtinės būsenos (1)

- Būsena gali turėti keletą vidinių būsenų.
- Tokia būsena vadinama sudėtine būsena (angl. superstate), o vidinės būsenos – sub-būsenomis (angl. substates).

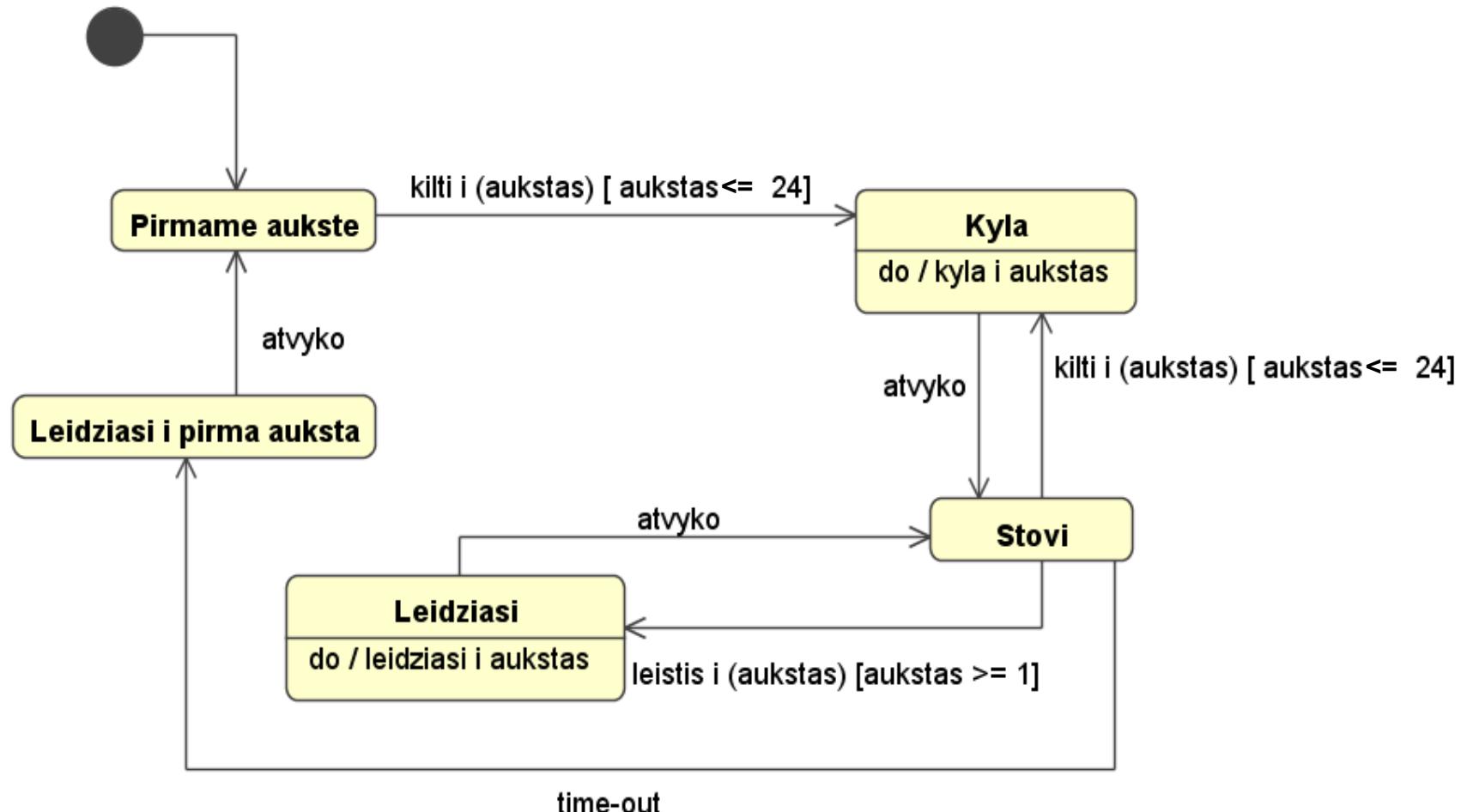
Sudētinės būsenos (2)



Sudėtinės būsenos su išlygiagretinimu

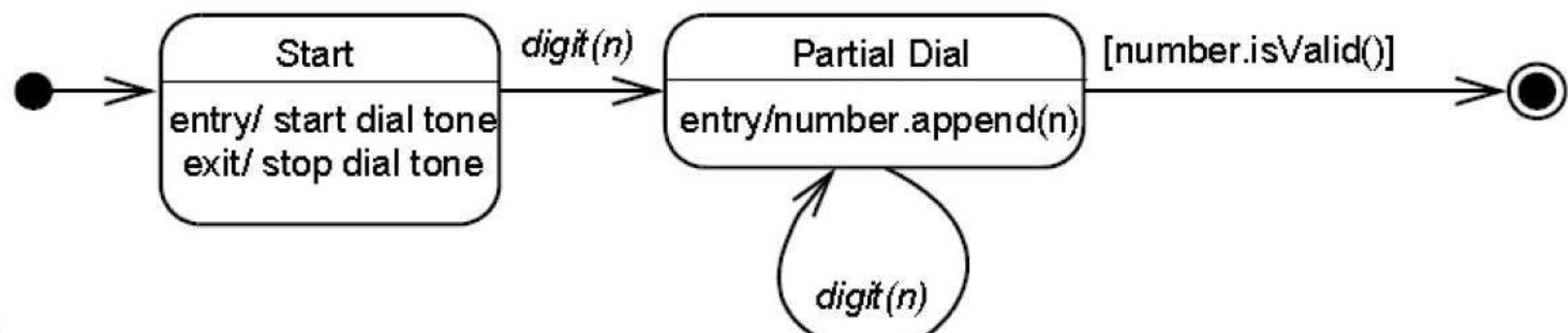


Diagramos pavyzdys (1)

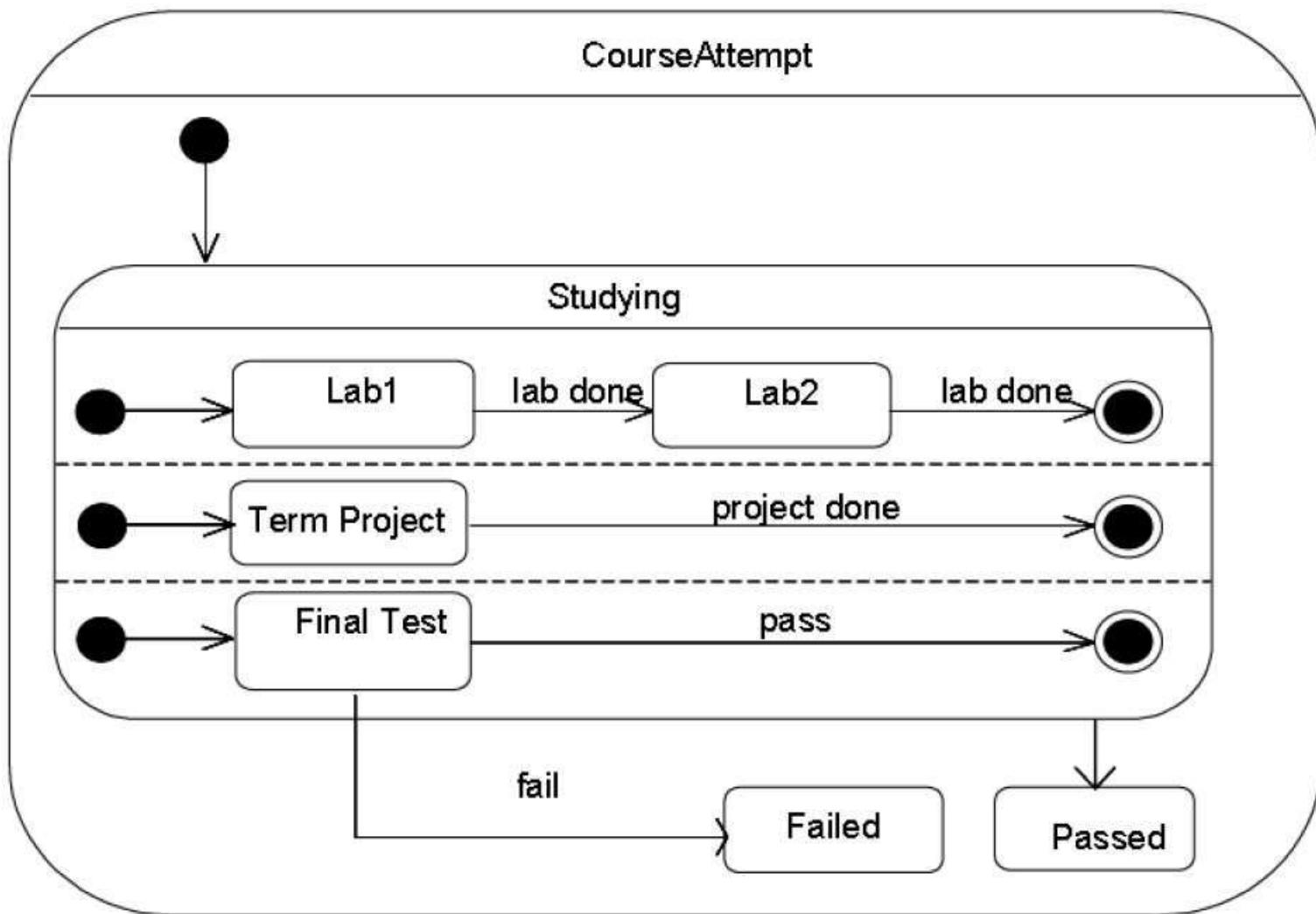


Diagrams pavyzdys (2)

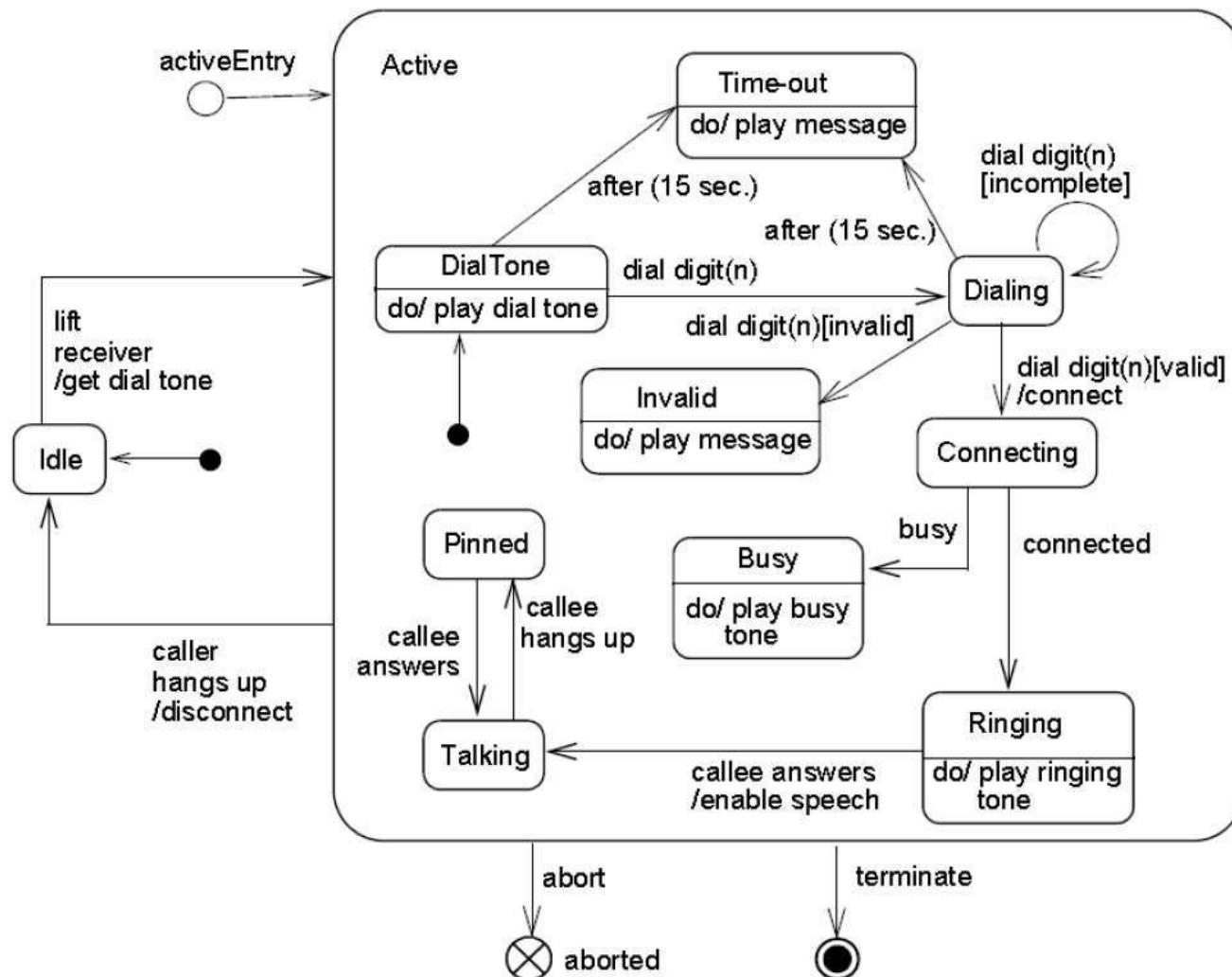
Dialing



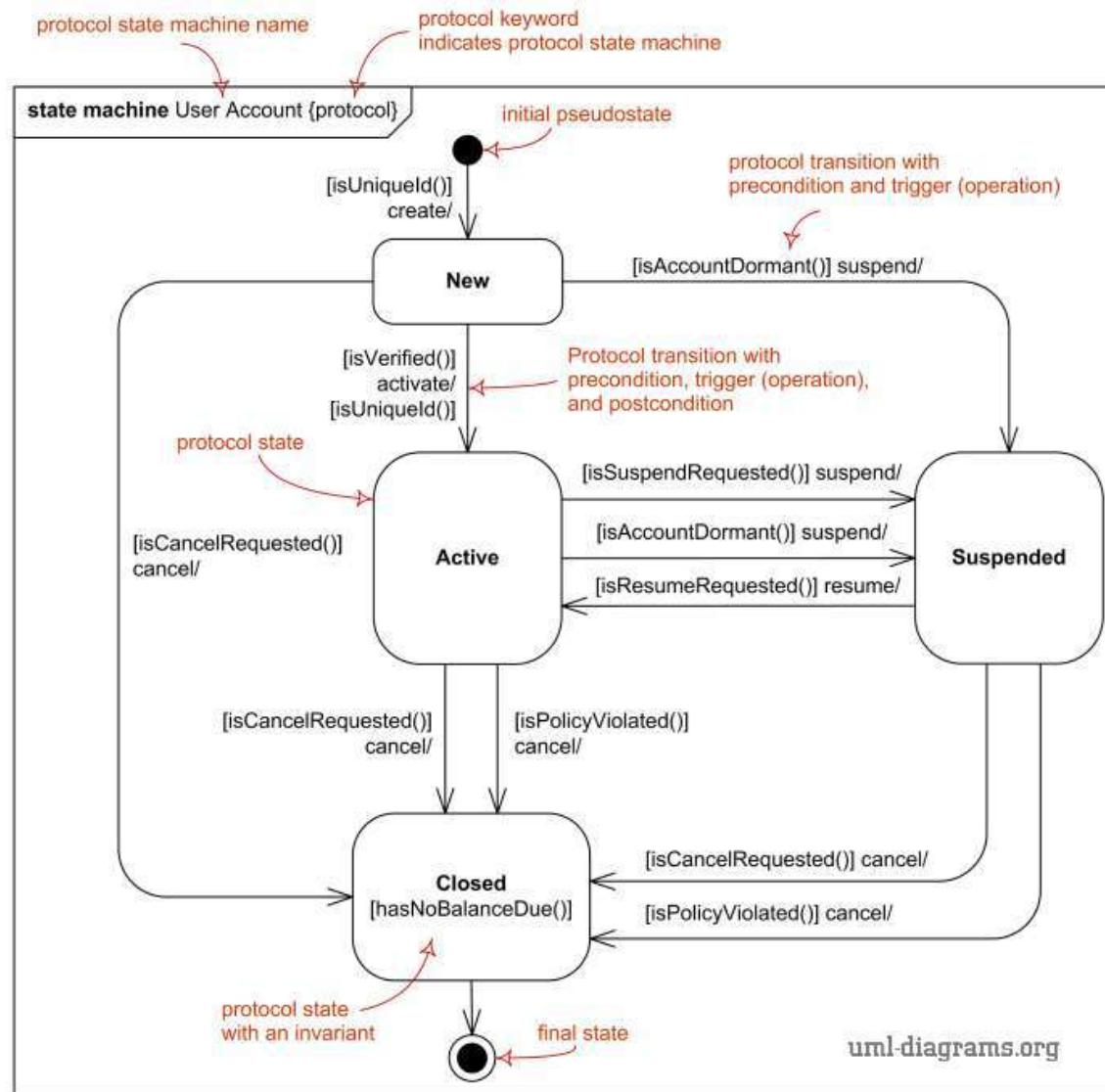
Diagrams pavyzdys (3)



Diagrams pavyzdys (4)



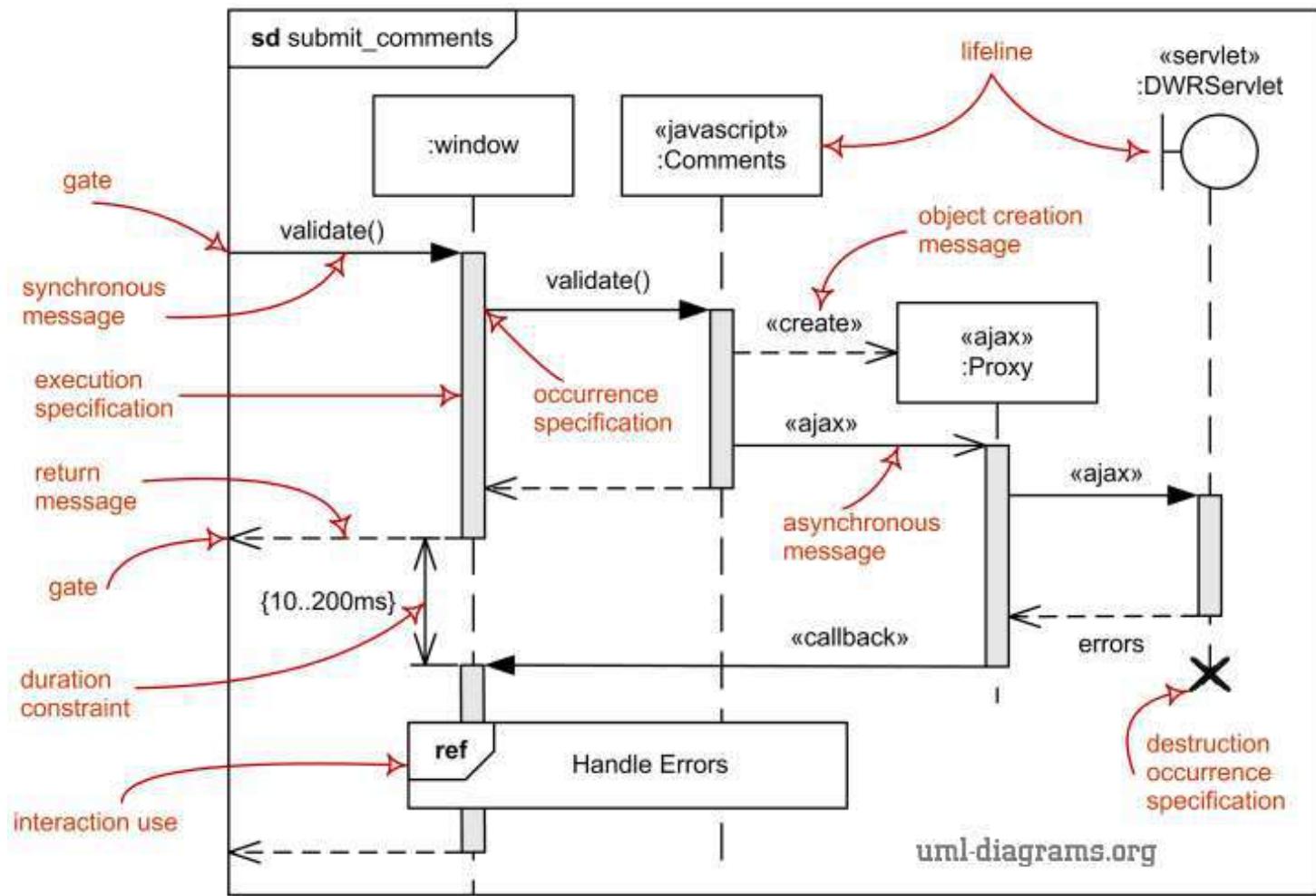
Protokolo būsenų diagramos pvz.



Sekų diagramos

- Sekų diagramos parodo kaip objektai bendrauja tarpusavyje *laike*.
- Naudojama sudėtingų vartojimo atvejų ar veiksmų detalizavimui.
- Vertikali ašis žymi laiką, horizontalioje išdėstomi objektai.
- Skaitoma iš kairės į dešinę iš viršaus į apačią.

Sekų diagramos elementai (1)



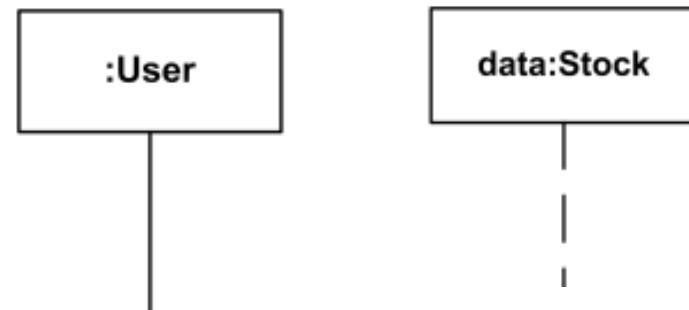
Sekų diagramos elementai (2)

■ Objektai (dalyviai):

- Klasių, aprašytų klasių diagramose egzemplioriai;
- Vartotojai;
- Išoriniai objektai.

■ Aprašoma nurodant:

- Klasę;
- Klasę ir objekto vardą.



Sekų diagramos elementai (3)

■ Pranešimai:

- Žymi informacijos perdavimą iš vieno objekto į kitą.

■ Pranešimų pavyzdžiai:

- Metodo iškvietimas;
- Signalo perdavimas;
- Objekto sukūrimas ar sunaikinimas.

■ Pranešimai numeruojami.

Pranešimų tipai (1)

- Asynchroninis →
- Synchroninis →
- Reikšmės
gražinimas →

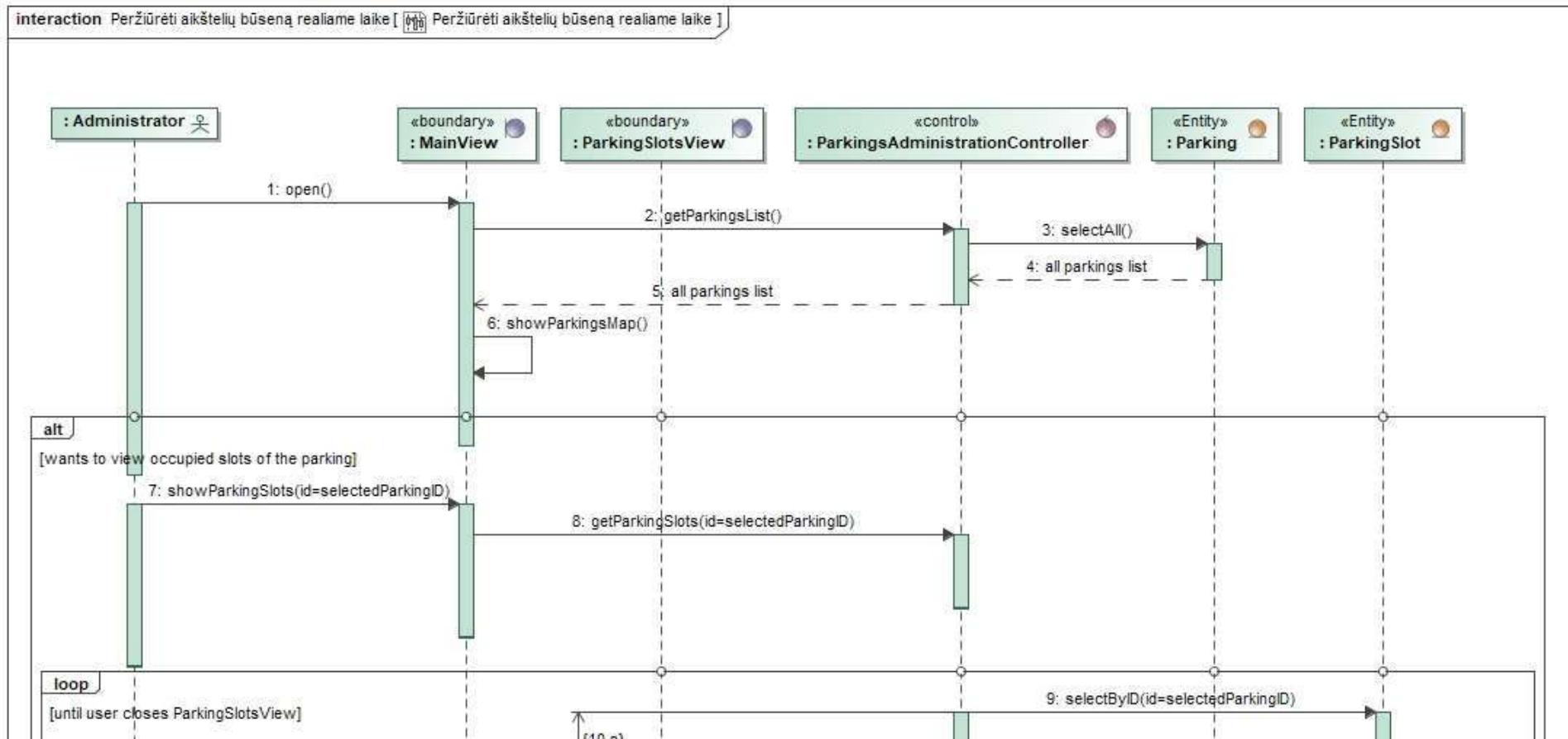
Pranešimų tipai (2)

- **Sinchroninis** – tai pranešimas, kuomet kviečiantysis objektas **laukia**, kol kviečiamas objektas atliks veiksmus, susijusius su pranešimu.
- **Asynchroninis** – tai pranešimas, kuomet kviečiantysis objektas **nelaukia**, kol kviečiamas objektas atliks veiksmus, susijusius su pranešimu.

Dažniausios pranešimų naudojimo klaidos

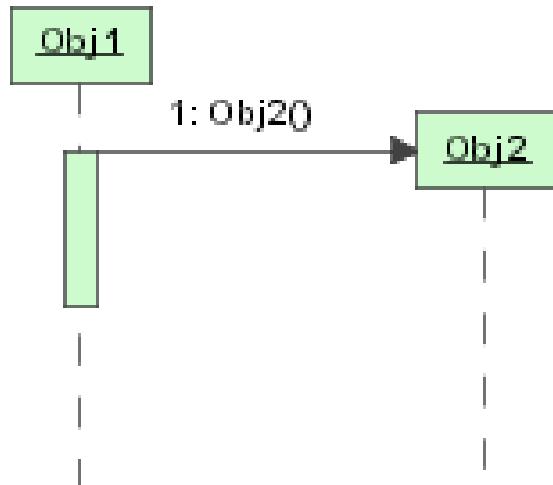
- Ant pranešimo rašomas metodas, kurį kviečiame, t. y. tas **metodas priklauso klasei, iš kuria rodo pranešimo rodyklę!**
- Jeigu metodas priklauso kviečiančiajai klasei, tai rodyklė rodo į save.
- Metodai sekų diagramose turi pilnai atitikti klasių diagramų aprašymus:
 - Pavadinimas, parametrai, gražinamos reikšmės tipas.

Sekų diagramos pavyzdys

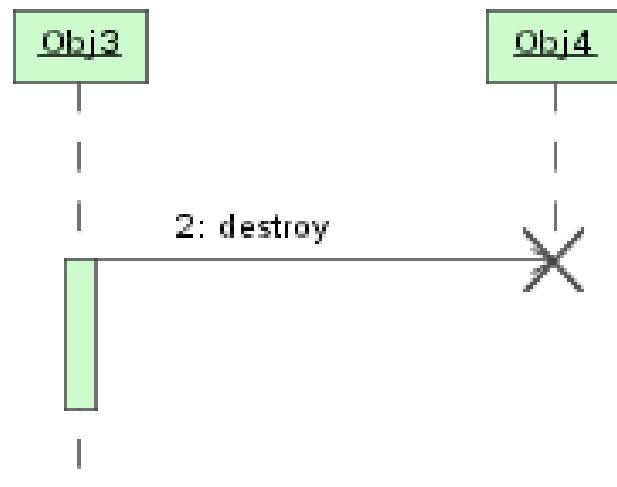


Objekto “gyvavimo linija” (lifeline)

■ Objekto sukūrimas



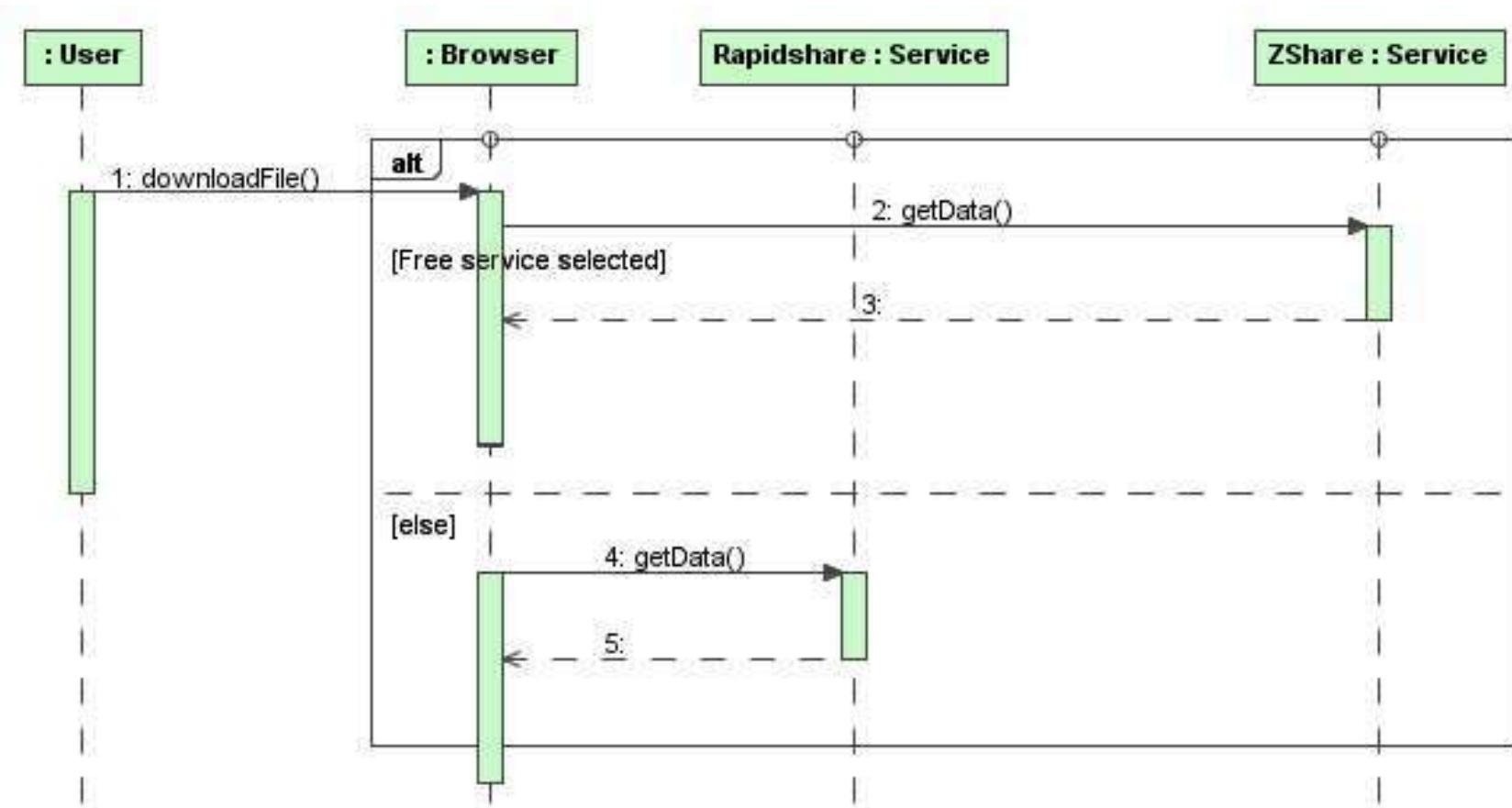
■ Objekto sunaikinimas



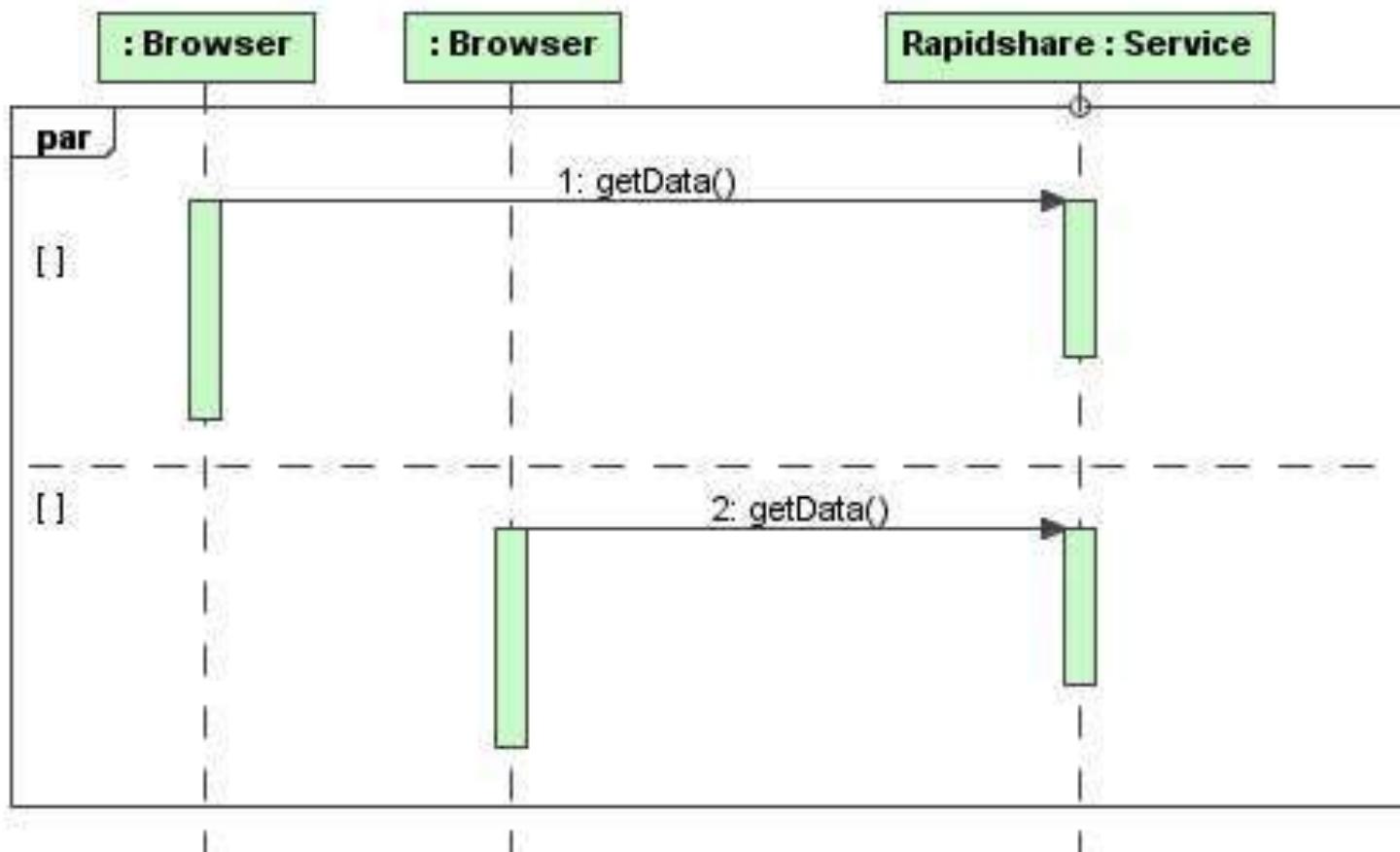
Kombinuotieji fragmentai (angl. combined fragment)

- **alt** - alternatives
- **opt** - option
- **loop** - iteration
- **break** - break
- **par** - parallel
- **strict** - strict sequencing
- **seq** - weak sequencing
- **critical** - critical region
- **ignore** - ignore
- **consider** - consider
- **assert** - assertion
- **neg** - negative

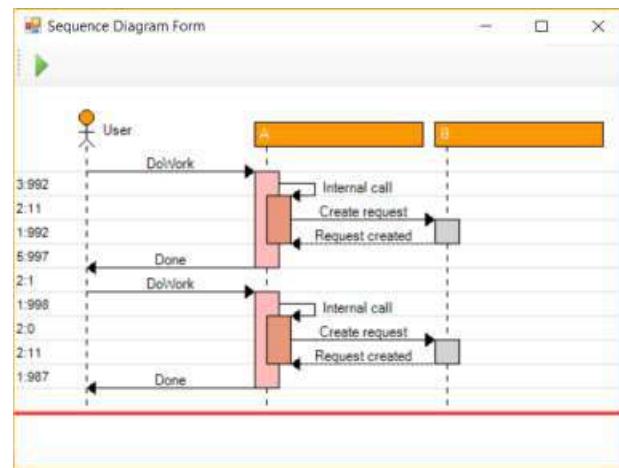
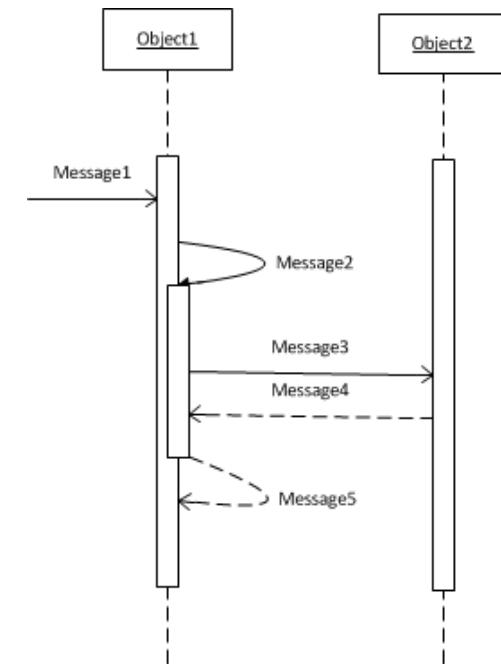
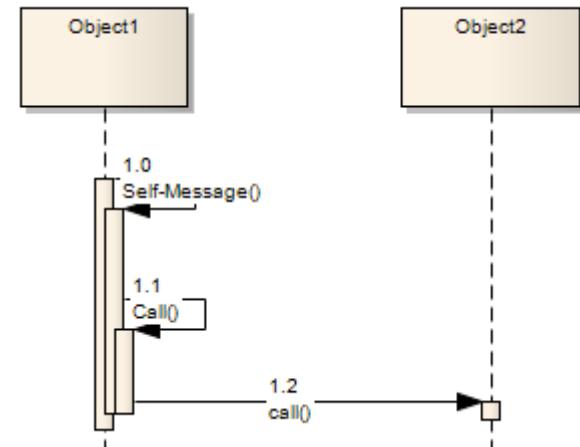
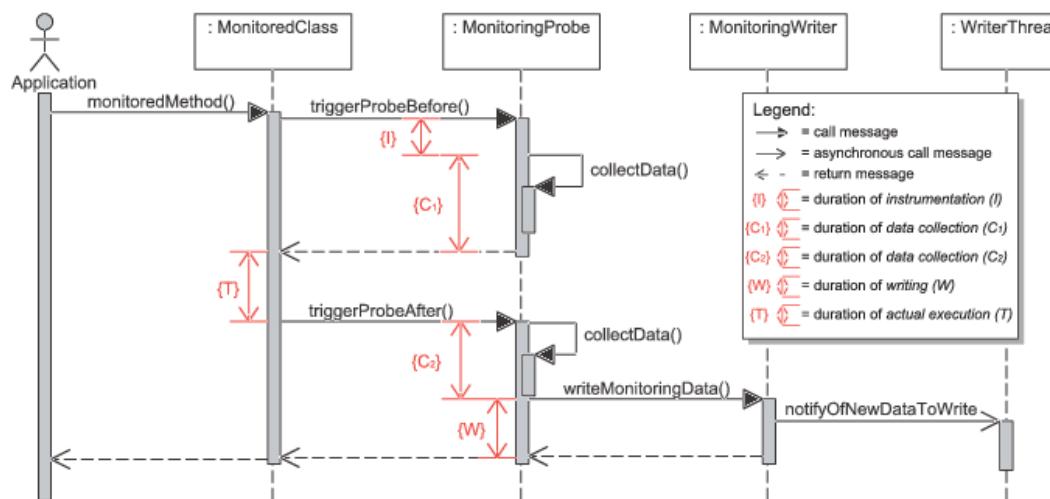
Sąlyginis modeliavimas (išsišakojimai, ciklai ir pan.)



Lygiagretus vykdymas



Metodų kvietimai iš vidinių objekto metodų



Sekų diagramų tipai

- Egzemplioriaus (angl. instance) diagrama – skirta aprašyti vieną galimą vykdymo variantą.
- Bendra (angl. generic) – aprašo visus galimus vykdymo variantus.
- Bendrą diagrama galima gauti apjungiant aibę instance diagramų.
- Du lygiai – egzempliorinis (angl. instance) ir specifikavimo (angl. specification).

Bendradarbiavimo diagramos

- Aprašo kaip elementai sąveikauja laike ir kaip jie yra susieti tarpusavyje.
- Labiau pabrėžiama struktūra ir sąryšiai.
- Pateikiama ta pati informacija kaip ir sekų diagramoje, tik kitu aspektu.

Diagramos elementai

- Tokie patys kaip ir sekū diagramoje:
 - Objektai;
 - Pranešimai.
- Taip pat pateikiama papildoma informacija:
 - Sąryšiai (asociacijų egzempliliai).

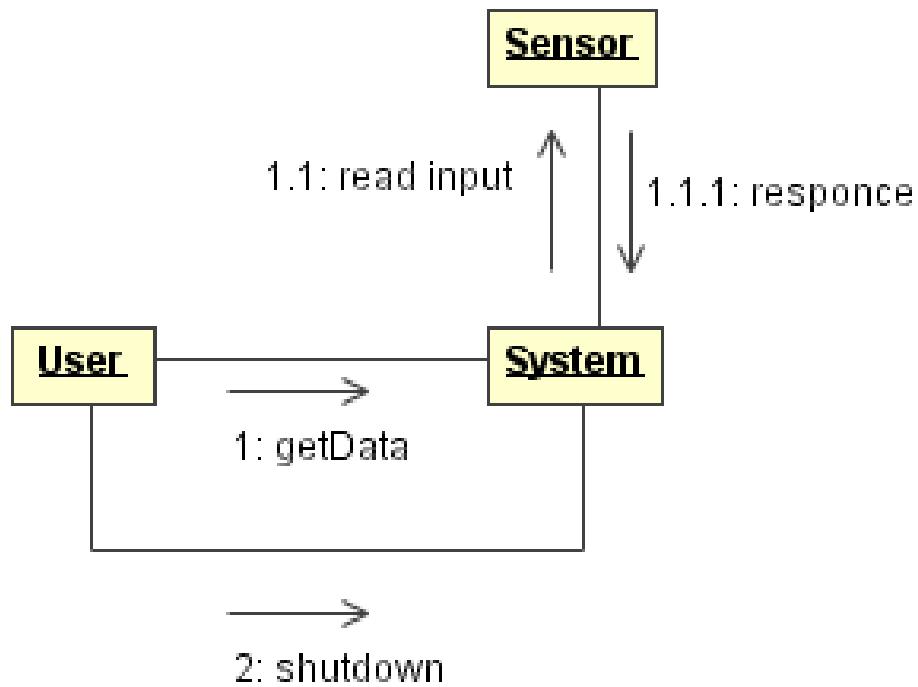
Pranešimo perdavimas

- Žymima rodykle šalia sąryšio:



- Žingsniai numeruojami – taip perteikiama veiksmų vykdymo seka.

Bendradarbiavimo diagramos pavyzdys



Papildomai

- Multiobjektai – atvaizduojama keletas objekto egzempliorių.
- Pranešimas į save
- Asociacijos
- Rolės

Ačiū už dēmesī

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Harmony/ESW kūrimo procesas

Prof. dr. Agnus Liutkevičius,
Kompiuterių katedra
Studentų g. 50-202, tel. 300394

Harmony/ESW kūrimo proceso autoriai

- Sukūrė
 - Dr. Bruce Powel Douglas,
 - Chief Evangelist for IBM Rational® with over 30 years specializing in the development of real-time and embedded systems and software.
 - Dr. Hans-Peter Hoffmann,
 - Chief System Methodologist, IBM Rational Software.



Harmony šaknys

- Anksčiau vadinosi:
 - Rapid Object-Oriented Process for Embedded Systems – ROPES
- Evoliucionavo keletą dešimtmečių.
- Skirtas:
 - Įterptinėms ir realaus laiko sistemoms, kurios turi tenkinti nefunkcinius reikalavimus:
 - Efektyvumas, kaina, saugumas, patikimumas ir t.t.

Sistemų kūrimo proceso elementai

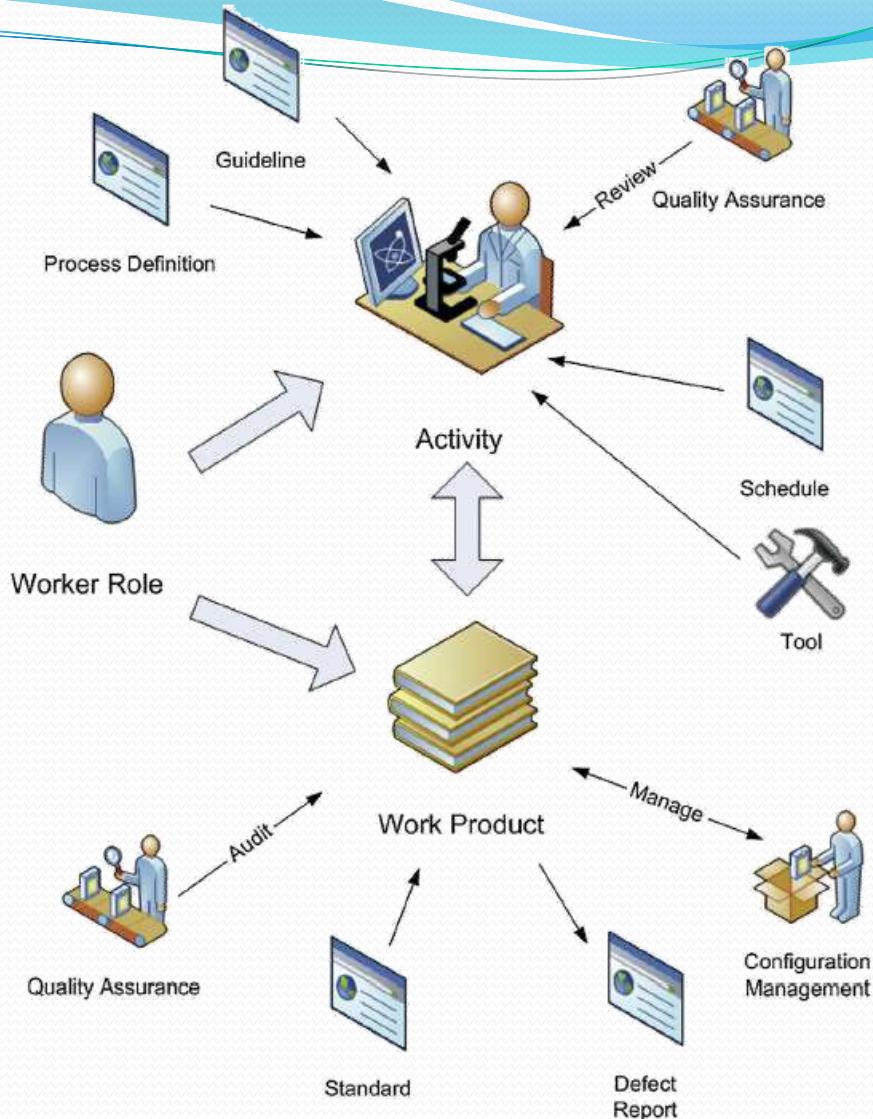
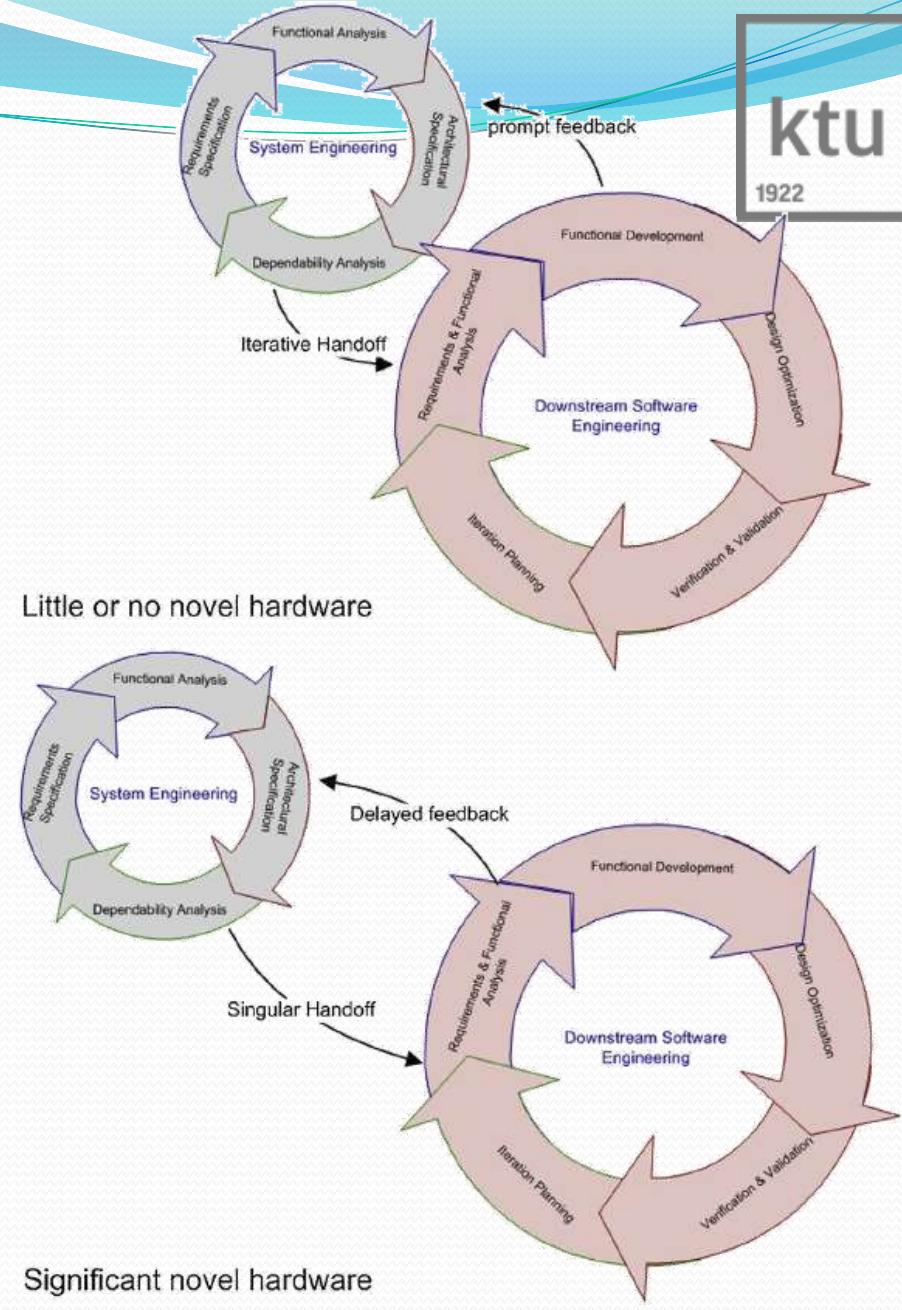


Figure 2.1
Basic elements of a process.

Prisimenam proceso pagrindines savokas

- Rolė – asmuo, kuris atsakingas už tam tikras veiklas.
- Veikla – tam tikra darbo dalis, su savo įvestimis ir išvestimis, kurios gali būti:
 - Darbo produktai;
 - Sąlygos;
 - Rezultatai.
- Darbo produktai – artefaktai (objektai), kuriuos pagamina rolės, atlikdamos tam tikras veiklas.

HW ir SW bendras kūrimas (angl. hardware- software co- development)

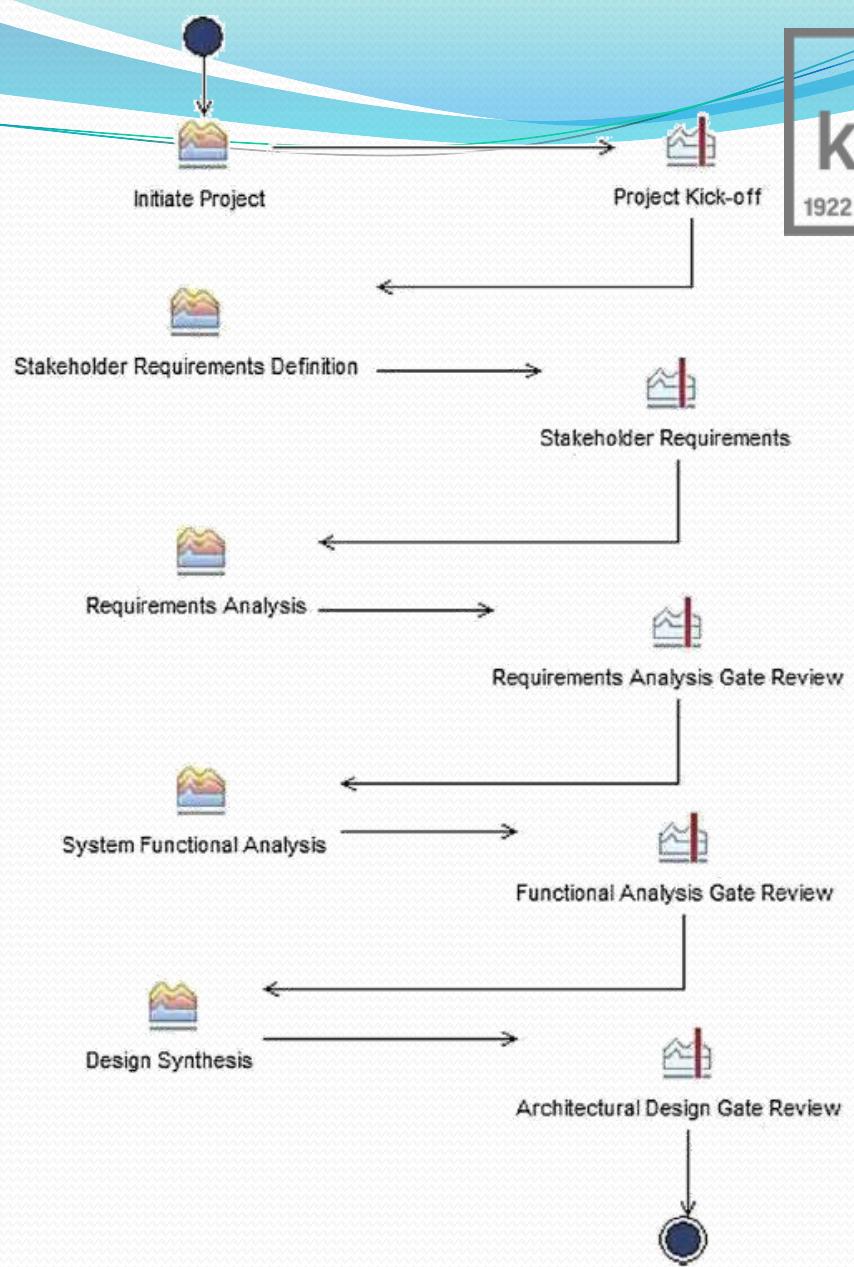


Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p.
ISBN13: 9780124077812

Significant novel hardware

Figure 2.3
Full Harmony (or Harmony Hybrid) workflow.

Sistemų inžinerijos veiklos Harmony kūrimo procese (HW kūrimo veiklos)



Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p. ISBN13: 9780124077812

Figure 2.4
Systems engineering activities of the Full Harmony process.

SW kūrimo (development) proceso veiklos

- Reikalavimų specifikavimas (CIM);
- Architektūros specifikavimas (PIM);
- Optimizacijos kriterijų identifikavimas ir specifikavimas;
- Projektavimo šablonų naudojimas PSM kūrimui;
- Kodo generavimas iš modelio + rankinis programavimas;
- Kodo derinimas;
- Testavimo atvejų kūrimas.

Harmony /ESW proceso veiklos

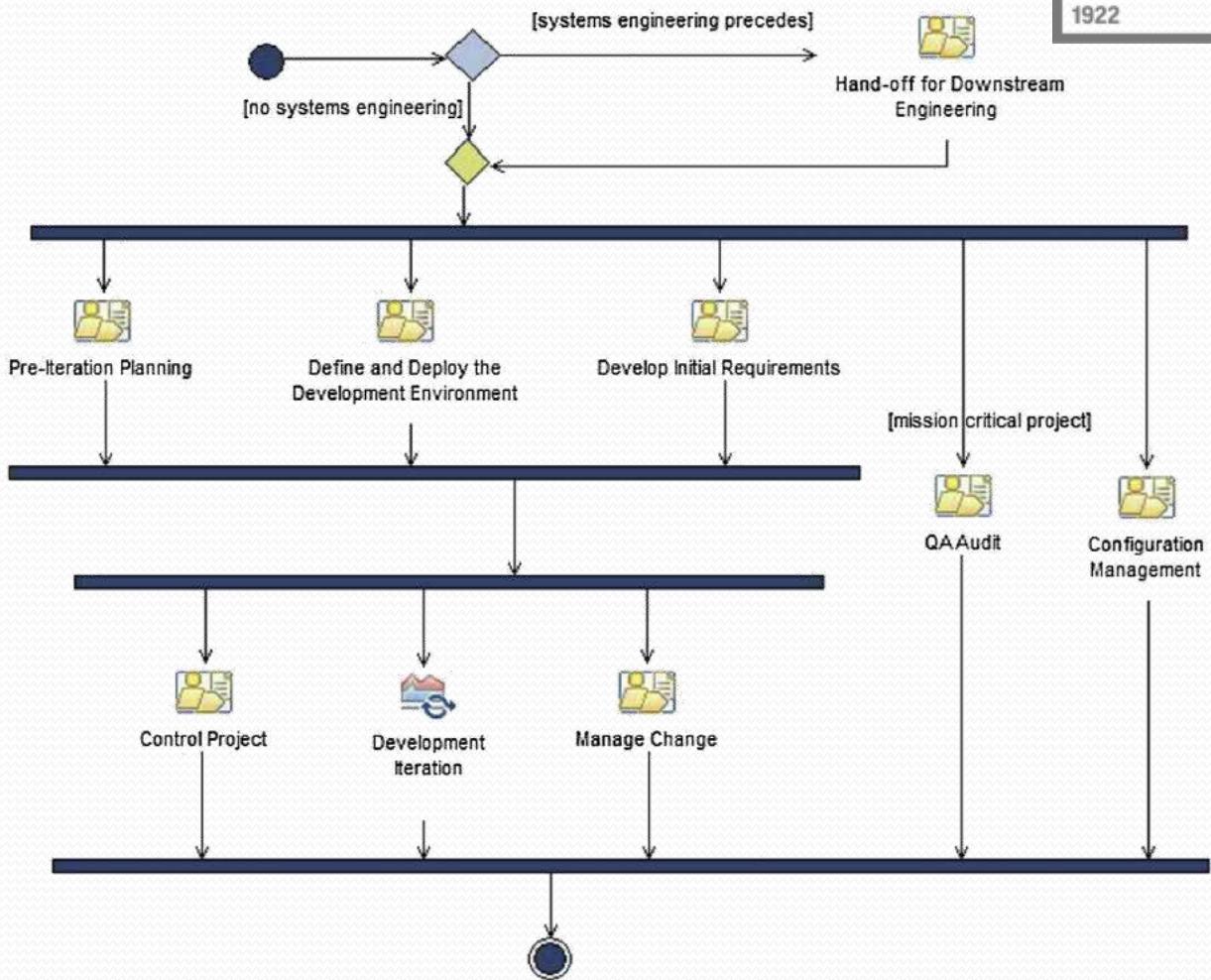


Figure 2.6
Delivery process for Harmony/ESW.

Pre-Iteration Planning

- Create the overall software schedule;
- Create the software team structure;
- Plan to reuse software or create reusable software;
- Identify and plan to reduce project risks;
- Specify the overall model and configuration management (CM) structure;
- If appropriate, perform initial safety and reliability analysis.

Define and Deploy the Development Environment

- Set up project guidance:
 - Standards;
 - Guidelines;
- Setting up the tooling environment.

Develop Stakeholder Requirements

- Identify written textual requirements to meet their needs.
- Stakeholder requirements are statements of stakeholder needs that are converted to system requirements (statements of what the system does) in the **Use Case/User Story Analysis activity**.
 - Note that this usually needn't be done in the Full Harmony process, because that work has already been done by the system engineers

Control Project

- This activity operates in parallel with the iteration and includes tasks such as:
 - updating the risk management plan;
 - monitoring project progress and updating the schedule;
 - performing ongoing safety assessment of the development work and updating the hazard analysis.

Manage Change

- This activity manages changes, such as:
 - requirements changes;
 - problem (defect) reports,
- usually resulting in:
 - updated requirements;
 - designs;
 - code;
 - tests.

Configuration Management

- All work products must be under configuration management (CM), including:
 - specifications,
 - models,
 - code,
 - test cases,
 - test results,
 - quality assurance records, etc.

Quality Assurance (QA) Audit

- For **safety-critical** and **high-reliability** projects, it is common to conduct QA activities (audits and reviews) to **ensure the syntactic correctness of the work**.
- QA reviews look at work products to ensure that they meet the project work product standards, such as:
 - requirements,
 - design,
 - coding standards.

Harmony/ESW procesas Agile požiūriu

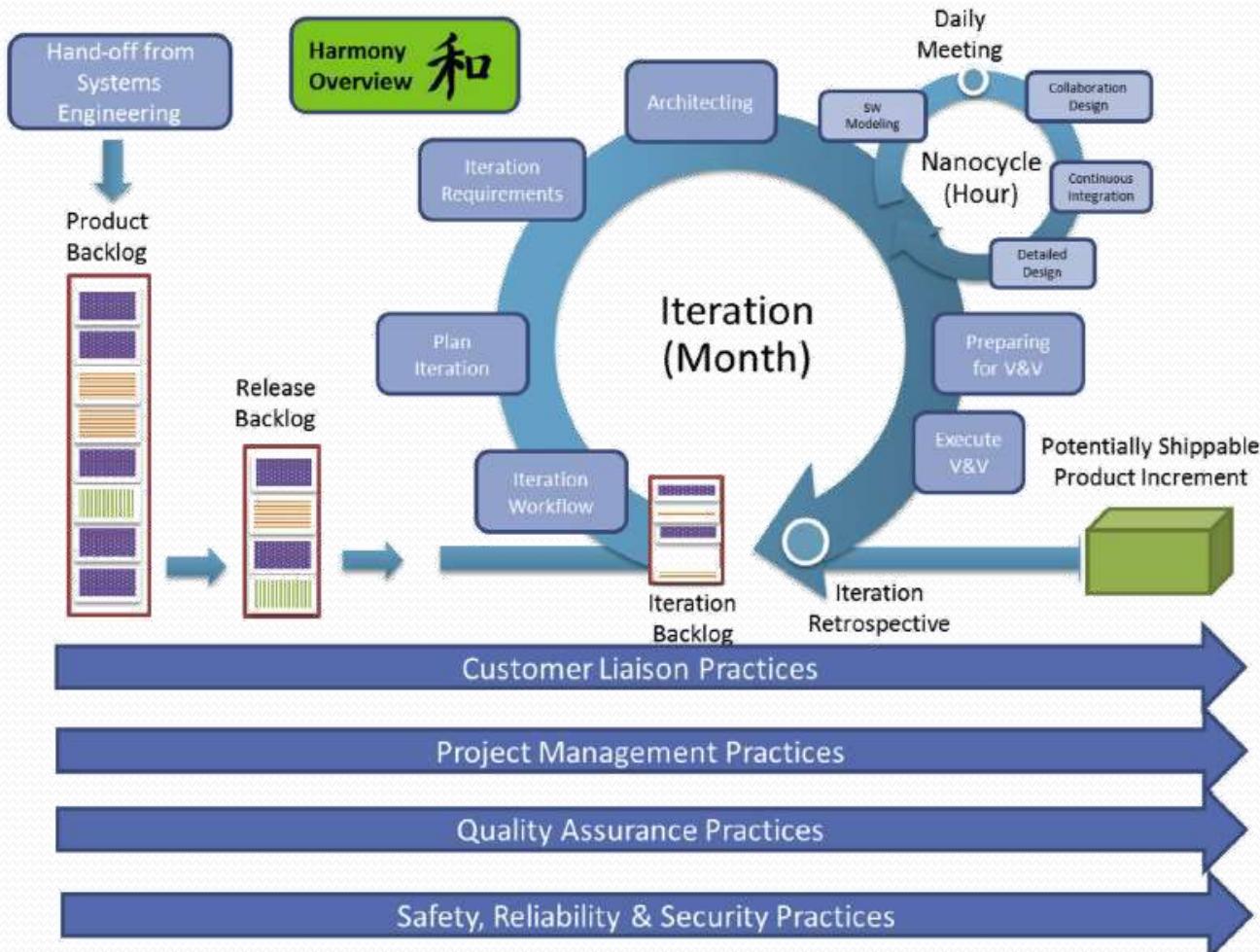


Figure 2.11
Harmony software process: agile view.

Vienos iteracijos veiklos

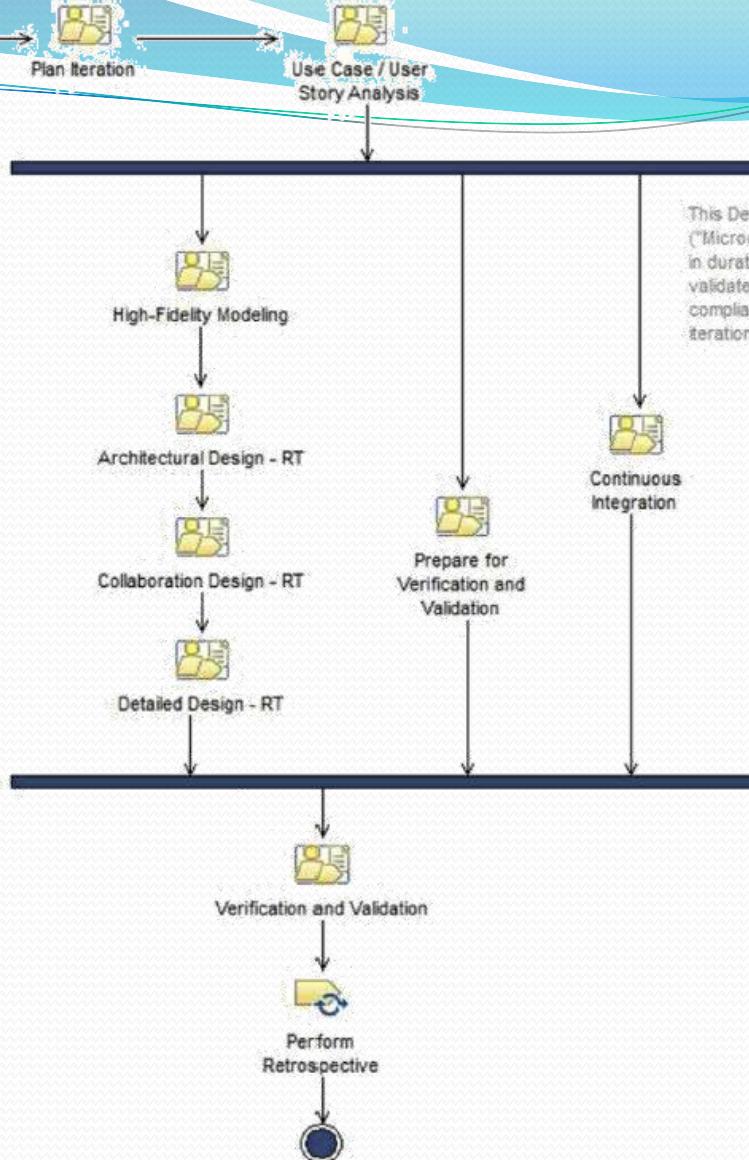


Figure 2.5
Harmony development iteration overview.

Iteration Planning

- Includes specification of the intent of the effort in the development iteration and production of the Iteration Mission Statement.
- The Iteration Mission Statement includes:
 - work items in the release backlog allocated to the iteration backlog, including:
 - Use cases to be realized;
 - Architectural intent to be developed;
 - Existing defects to be repaired;
 - Known risks to be mitigated;
 - Target platforms to be supported;
 - Any additional tasks or efforts to be performed by the team during iteration.

Use Case/User Story Analysis Workflow

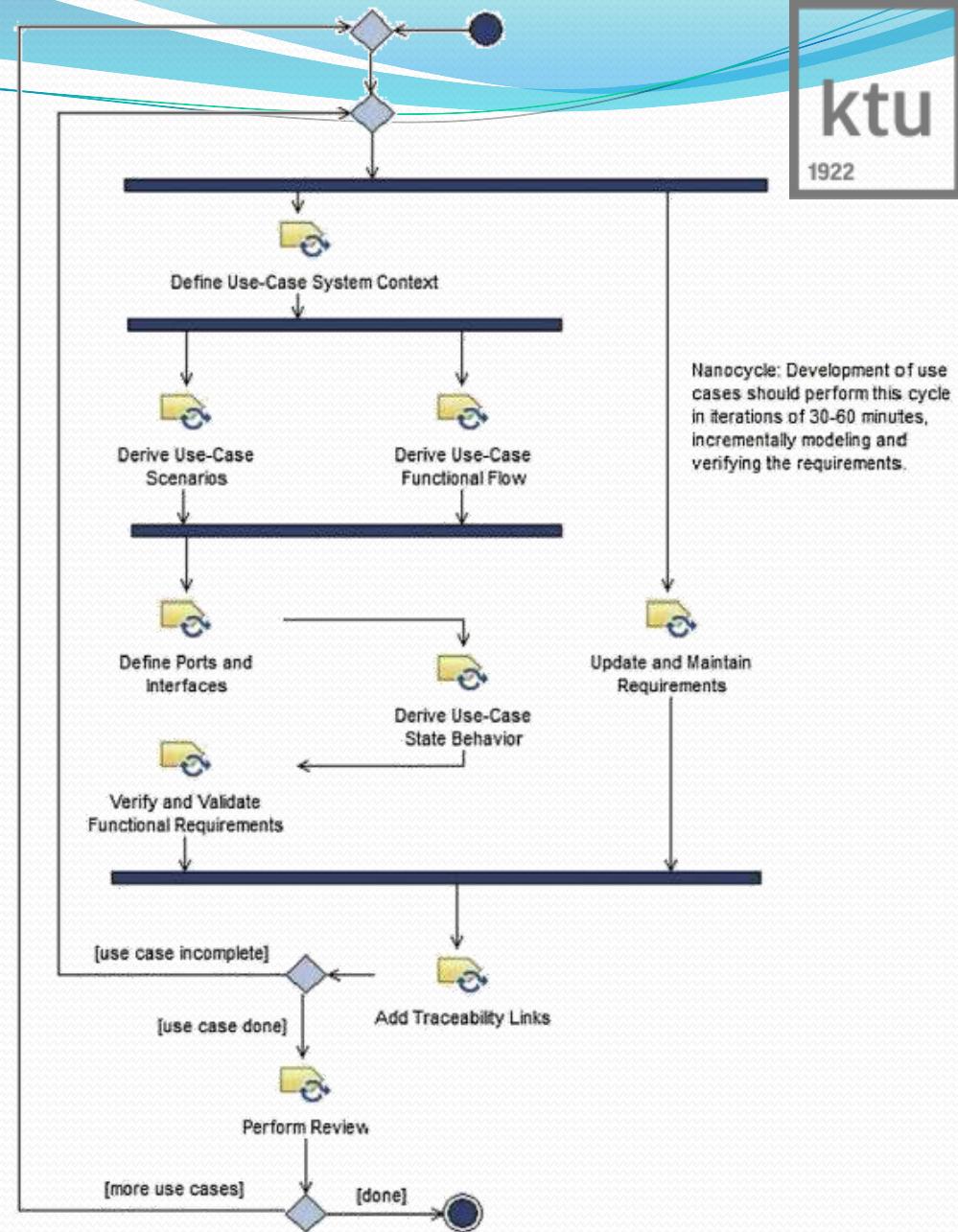


Figure 2.12

Use Case/User Story Analysis in the Harmony/ESW process variant.

Requirements

- Requirements are detailed using a combination of:
 - Sequence diagrams;
 - State machines;
 - Activity diagrams;
 - Control law diagrams (non-UML);
 - Textual descriptions;
 - Quality of service (QoS) constraints;
 - (SysML) requirements diagrams.

High-Fidelity Modeling Phase

- Constructs collaboration of *essential* software elements, and is performed one use case at a time.
- For the current prototype, one collaboration is constructed for each use case implemented by the prototype.
- PIM is constructed in an incremental fashion, one (or a few) use case(s) at time.
- At the very same time, the software baseline from the PIM is created and unit tested so
 - that at the end of the High-Fidelity Modeling activity, both a **PIM and functionally correct** (but not optimized) **source code is available**.

High-Fidelity Modeling Workflow

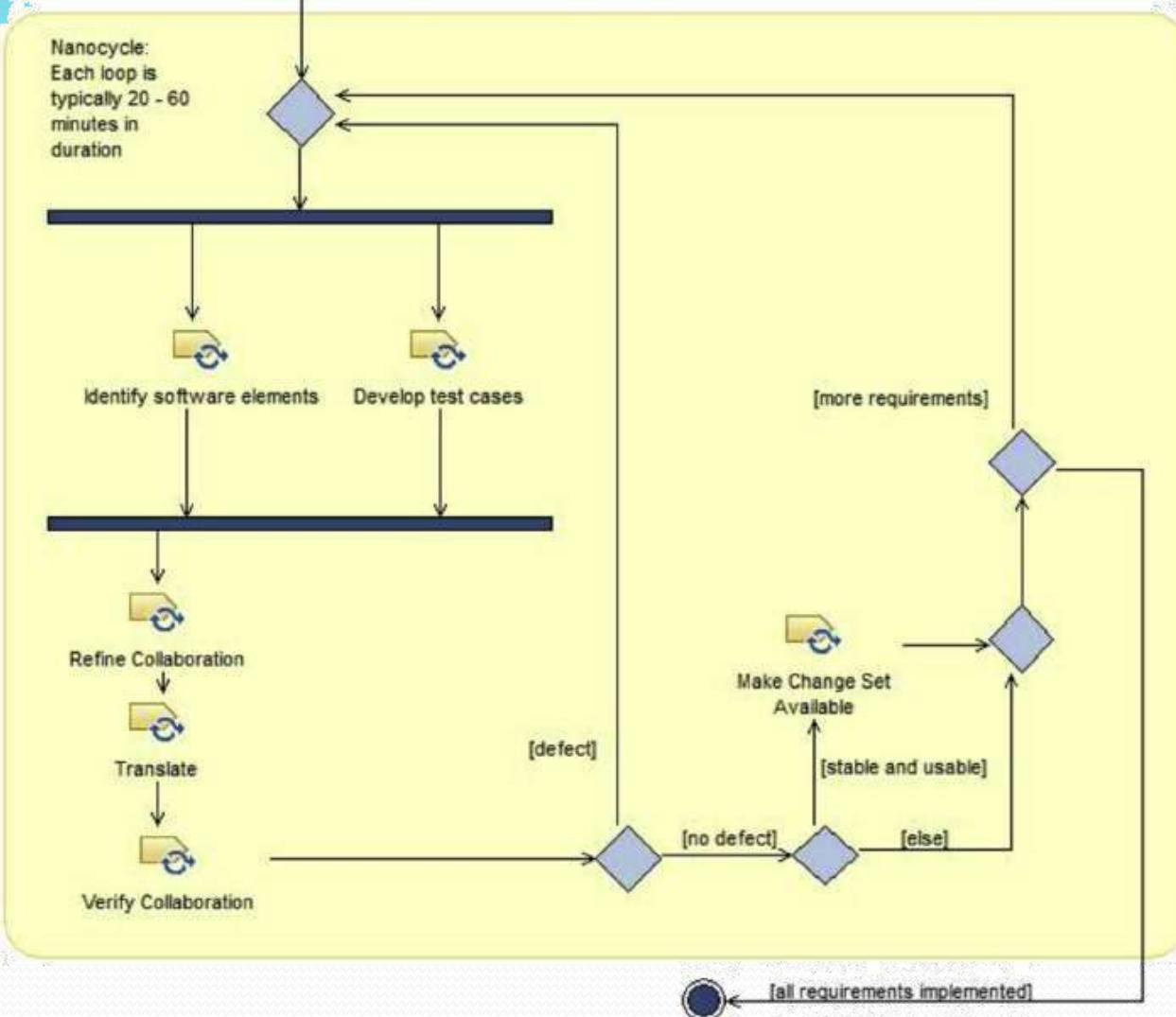
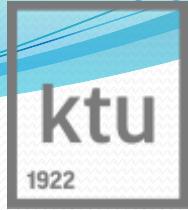


Figure 2.13
High-Fidelity Modeling workflow.

Table 2.1 Essential Software Element Discovery Strategies.

Strategy	Description
Underline the noun	This is used to gain a first-cut object list; the analyst underlines each noun or noun-phrase in the problem statement and evaluates it as a potential object.
Identify causal agents	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify messages and information flow	Messages must have an object that sends them and an object that receives them as well as, possibly, other objects that process the information contained in the messages.
Identify real world items	Real world items are entities that exist in the real world, but are not necessarily electronic devices. Examples include objects such as respiratory gases, air pressures, forces, anatomical organs, chemicals, vats, etc.
Identify physical devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. In the internal architecture, they are processors or ancillary electronic “widgets.” This is a special kind of the previous strategy.
Identify key concepts	Key concepts may be modeled as objects. Bank accounts exist only conceptually, but are important objects in a banking domain. Frequency bins for an online autocorrelator may also be objects.
Identify transactions	Transactions are finite instances of associations between objects that persist for some significant period of time. Examples include bus messages and queued data.
Identify persistent information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify visual elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify control elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects.

Kokie architektūros elementai reikalingi pradinėje modeliavimo fazėje?



- Limit the collaboration at this point to elements that clearly must be present in the high-fidelity model for it to be functionally correct.
- For example,
 - use case “Manage Account” for a banking system need objects such as *Customer*, *Account*, *Debit Transaction*, and *Credit Transaction*.
 - In a navigation system, we need objects or their attributes, such as *Position*, *Direction*, *Thrust*, *Velocity*, *Attitude*, *Waypoint*, and *Trajectory*.
- The goal is to include only the objects, classes, functions, data structures, and relations that are **essential** for functional correctness and **not to include design optimizations**.

Kodo generavimas ir testavimas

- We are not only creating a model – specifically the PIM – but also **producing a source code baseline**.
- You can really only evaluate the correctness of an object model via ***execution and unit test***.
- With executable modeling tools this is very fast and easy automatic process.
 - If you are using less capable tools, the workflow is the same, but you will manually write the code yourself.
- Most developers wait far too long before executing and ensuring the correctness of their models and then pay for it with long, painstaking efforts to get the code finally compiled and correct.

Skirtumai, lyginant su tradiciniais kūrimo procesais

- Analizės fazė vykdoma kiekvienoje iteracijoje (mikrocikle) ir apima:
 - Prototipo apibrėžimas – ką sistema turi daryti:
 - CIM sudarymas (Use Case analizė).
 - Objekto analizė – iš kokių dalijų susideda ši sistema ir kaip tos dalys sąveika:
 - PIM sudarymas (High-Fidelity modeliavimas).
- Kur dingo realizacija / programavimas???
 - Kodas kuriamas kartu su modeliais (automatizuotai arba rankiniu būdu).

Design with the Harmony Process

- Platform-Specific Model (PSM) in MDA, is a concrete blueprint for exactly *how* the essential properties will be realized.
- An analysis model (CIM+PIM) may be implemented by many different designs with different optimization characteristics.
- Design should always be an optimization of an analysis model.

Projektavimas

- Trys projektavimo lygiai:
 - **Architektūrinis** (angl. architectural);
 - **Bendradarbiavimo** (angl. mechanistic);
 - **Detalus** (angl. detailed).
- Tikslas – **Optimizacija**.
- Etapai:
 - Identifikuoti svarbius optimizacijos kriterijus;
 - Surikiuoti kriterijus pagal svarbą;
 - Parinkti projektavimo šablonus ir technologijas, kurie optimizuoją svarbesnius kriterijus mažiau svarbių sąskaitą;
 - Pritaikyti projektinius sprendimus (patobulinti modelius);
 - Verifikasioti ir Validuoti patobulintą sprendimą.

Architektūrinis projektavimas

- Architectural design representation uses the same UML diagrams as in systems architecture and High-Fidelity Modeling:
 - class diagrams to represent the structure;
 - sequence diagrams to represent collaborative behavior;
 - and state machines to model the behavior of individual elements.

Architektūrinio projektavimo veiklos

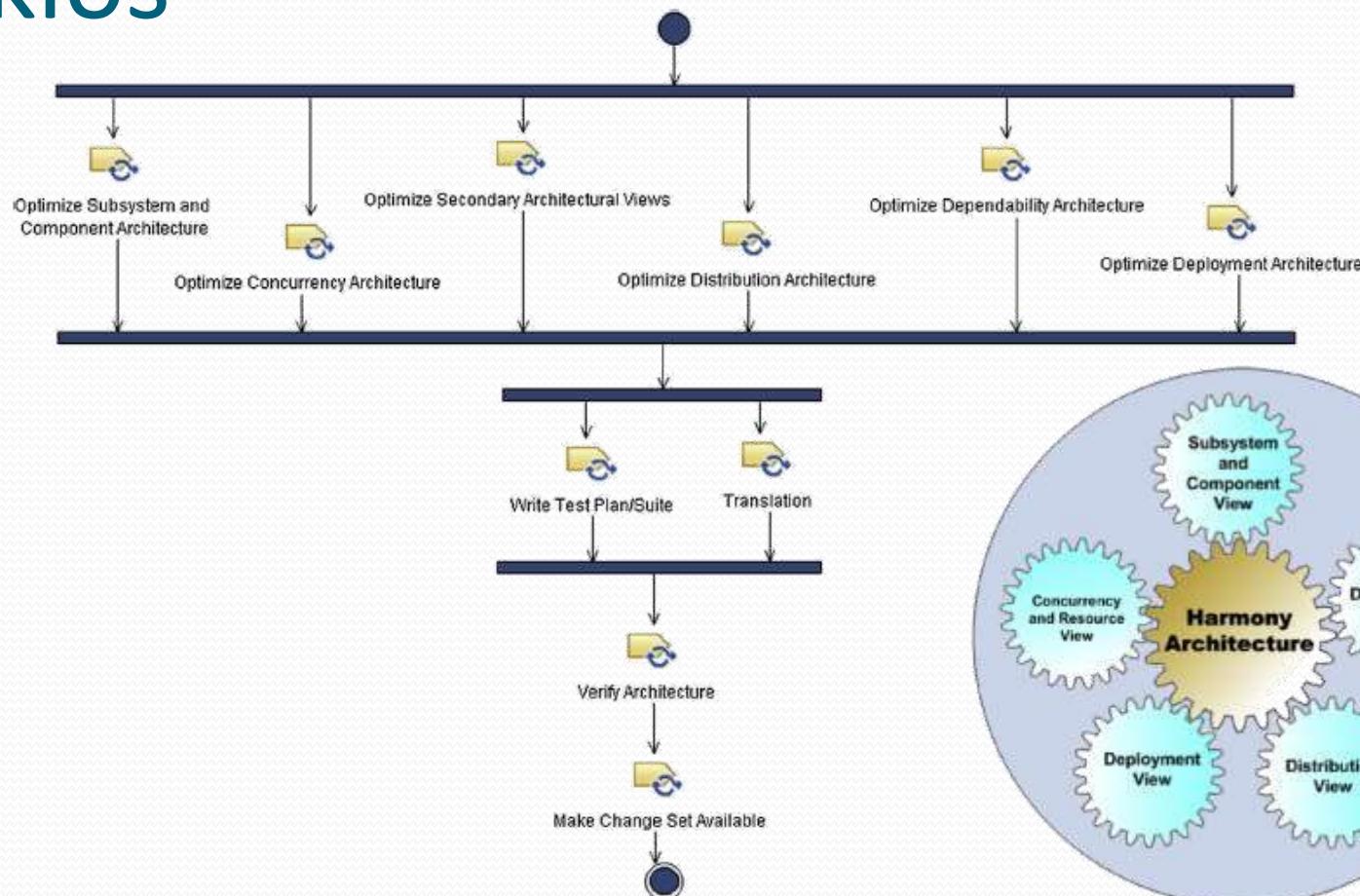


Figure 2.15
Architectural Design workflow.

Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p.
ISBN13: 9780124077812

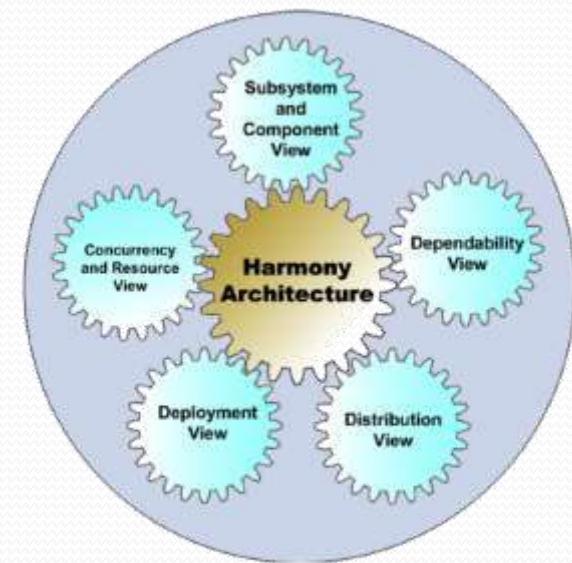


Figure 2.14
Harmony architectural views.

Bendradarbiavimo projektavimas

- This phase is concerned with the optimization of individual collaborations, each realizing a **single use case**.
- Mechanistic design largely proceeds via the application of design patterns, although the scope of the patterns is much smaller than those found in architectural design.
- The Mechanistic Design view is an elaboration of the High-Fidelity Modeling view and uses the same graphical representation:
 - class and sequence diagrams for collaboration structure;
 - sequence, activity, and state diagrams for behavior.

Bendradarbiavimo projektavimo veiklos

- Optimizuojama sąveika kiekvienam vartojimo atvejui.

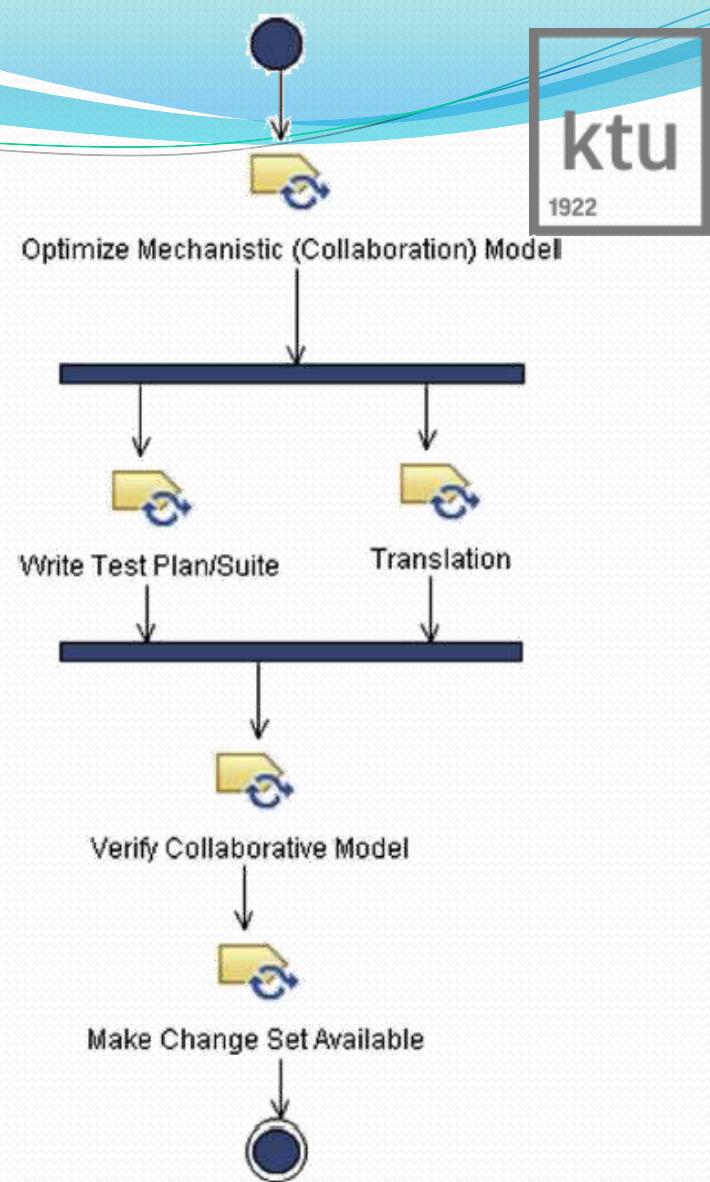


Figure 2.16
Mechanistic (Collaboration) Design workflow.

Detalusis projektavimas

- This phase elaborates the internals of **objects, classes, functions, and data structures**, and has a highly limited scope – the **individual software element**.
- Most of the optimization in detailed design focuses on the issues of:
 - Data structuring (space or time optimization);
 - Algorithmic decomposition;
 - Optimization of an object's state machine;
 - Object implementation strategies;
 - Association implementation;
 - Visibility and encapsulation concerns;
 - Ensuring compliance at runtime with preconditional and postconditional invariants (such as ranges on method parameters).

Detaliojo projektavimo veiklos

- Optimizuojam tik nedideles sistemos dalis, dažnai su kritiniais reikalavimais.

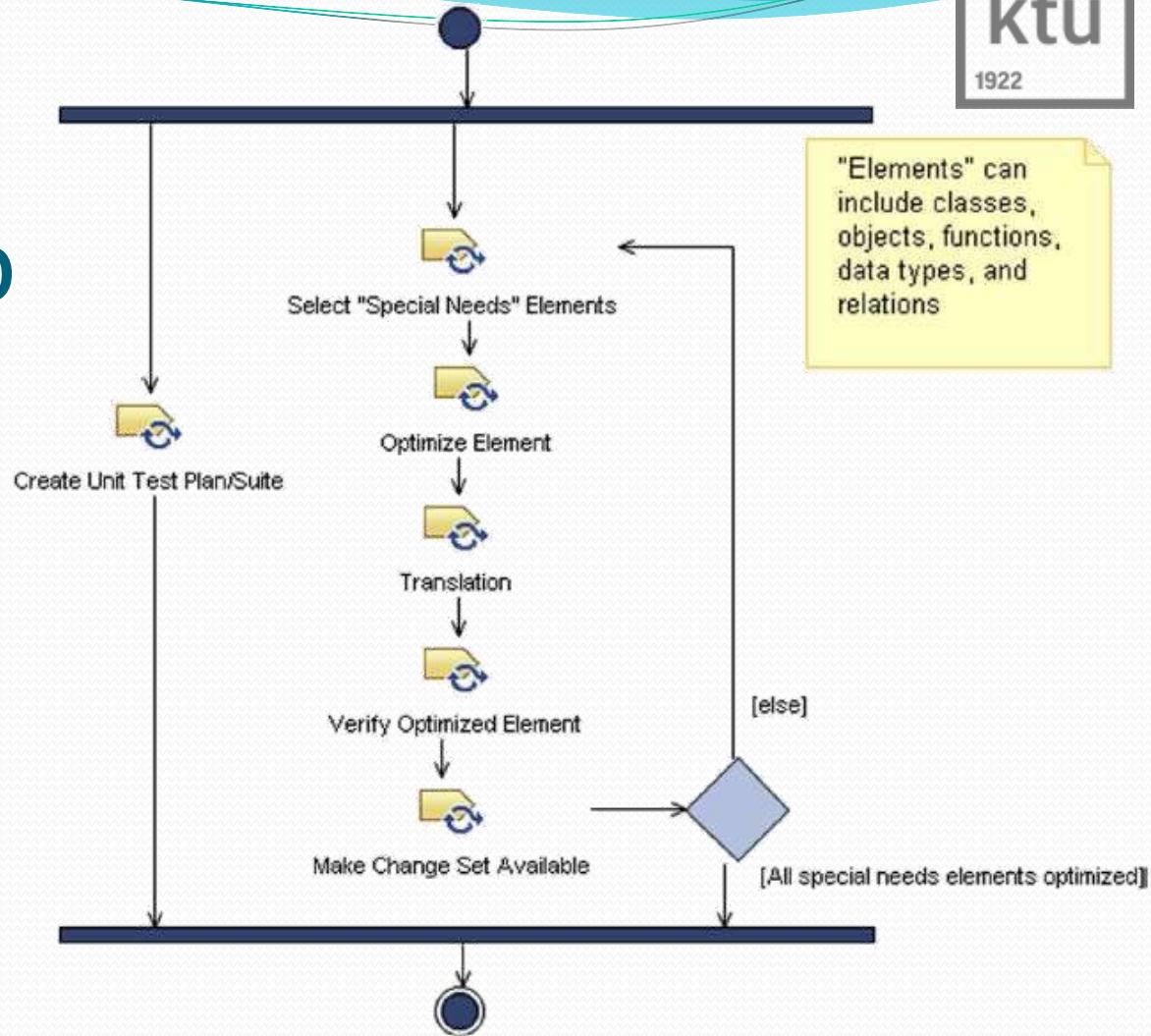


Figure 2.17
Detailed Design workflow.

Verification and Validation (V&V)

- The V&V test phase constructs the prototype from the architectural elements and **ensures that the system produced during the iteration meets its mission statement (validation)**.
- The High-Fidelity Modeling and Design workflows have been performing **incremental integration** to ensure that incremental changes made to the source code base have **resulted in a system that is internally consistent**.
- Demonstrates that the produced **system meets all of the requirements verified from previous iterations** (regression test) and that all of the **new requirements for this iteration have been properly and correctly implemented**.

Verification and Validation (V&V) work products

- The development of the work products to be used in V&V testing – test procedures, test cases, and test harnesses – have been **developed in parallel with the system development work** (*Prepare for V&V Testing* activity in Figure 2.5.)
- Functional and QoS test vectors are primarily derived from the use case sequence diagrams specified in the Analysis phase (Harmony/ESW) or the Systems Engineering phase (Full Harmony).
- The Regression test vectors are a subset of the test vectors from previous development iterations.
- Additional tests, such as stress, volume, coverage, and fault-seeding tests, are added manually.

Verification and Validation (V&V)

veiklos

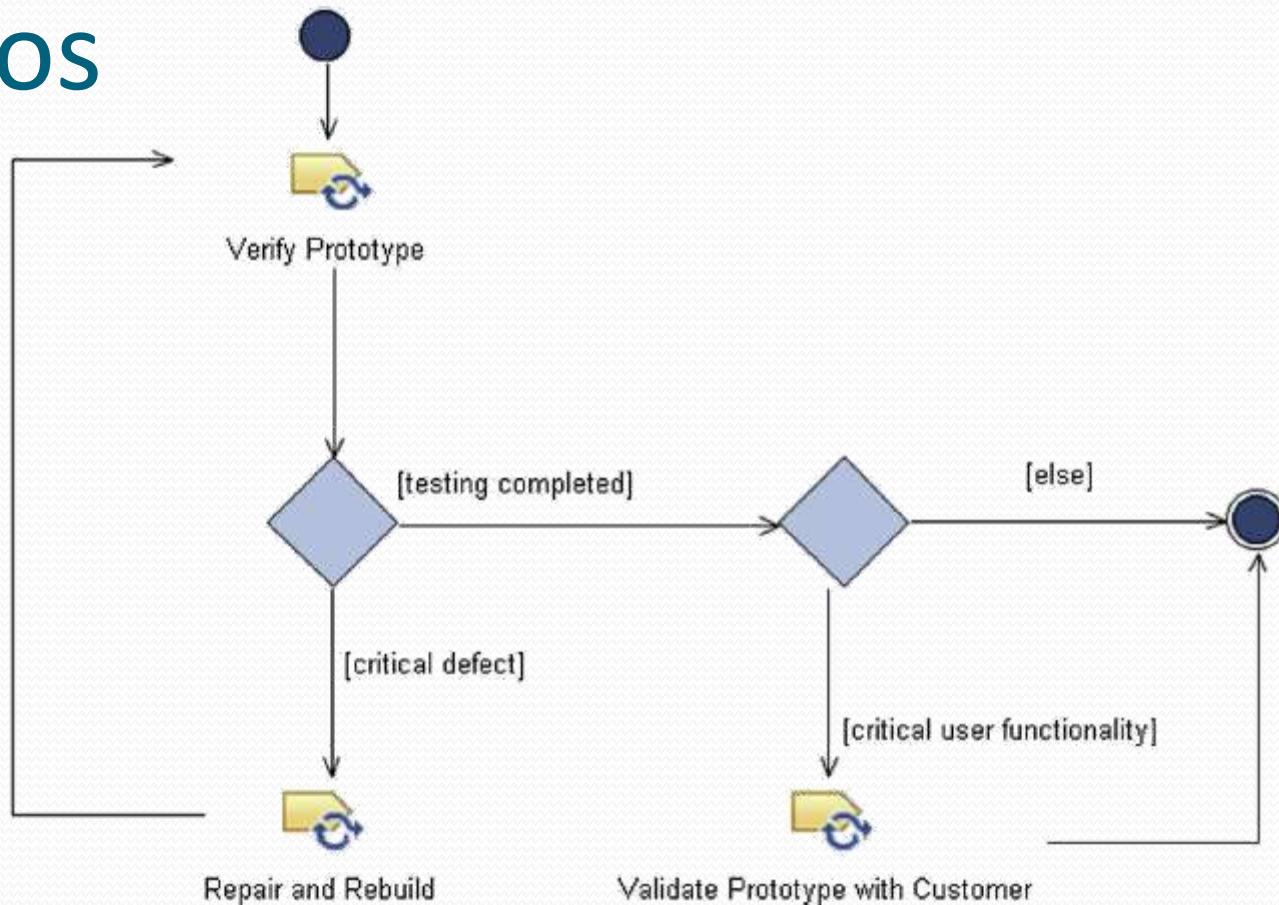


Figure 2.18
Verification and Validation workflow.

Kodėl verta prototipą rodyti užsakovui?

- Following a successful verification, the prototype may be demoed to the customer or released to them for use.
- This can provide valuable feedback, because:
 - The verification of the prototype ensures that it meets the requirements. **However, requirements are often incomplete, inconsistent, inaccurate, or just plain wrong.**
 - Allowing the customer to provide feedback greatly improves the chance that the system will actually meet the needs of the customer.

Increment Review (Party!)

- This activity performs an iteration retrospective to **see how well the project is proceeding**.
- This task assesses both the project against the project and iteration plans and **updates the plans as necessary**.
- The primary work products assessed during the Party phase are:
 - Overall project plan, including the overall schedule;
 - Iteration project plan, including the iteration schedule;
 - Architecture stability, robustness, and optimality;
 - Process efficiency and effectiveness;
 - Risk mitigation activity outcomes, their impact on the project, and looking ahead for upcoming risks;
 - Addressing other project concerns raised by staff or management.

Kas gali būti keičiamą tobulinama?

- In terms of schedule, such adjustments will be things like:
 - reassignment of resources,
 - reordering activities,
 - deletion of activities,
 - reductions (or enhancements) of scope and/or quality,
 - rescheduling subsequent activities, and so on.
- Party phase evaluates architecture on two primary criteria:
 - Is the architecture adequately meeting the needs of the qualities of services that are driving architecture selection?
 - Is that architecture scaling well as the system evolves and grows?
- The process of reorganizing the architecture is called *refactoring* the system.
 - If the project team finds that the architecture must be significantly refactored on each prototype, then this is an indication that the architecture is not scaling well, and some additional effort should be given to the definition of a more scalable architecture.



Ačiū už dėmesį. Klausimai?

UML pagrindai

Projektavimo šablonai
(angl. design patterns)

Objektiškai orientuotas projektavimas

■ Problemos:

- Sunku išskirti objektus ir jų ryšius;
- Lankstumas ir išplečiamumas;
- Pakartotinio panaudojimo galimybė;
- Stati mokymosi kreivė.

Projektavimo šablonai

- Sprendimai standartinėms problemoms.
- Šablonas – aprašo pačią idėją, o ne tikslų sprendimą.
- Orientuoti į projektavimo problemas, o ne į technines ar taikymo sričiai specifines problemas.

Projektavimo šablonai

■ Šablono aprašo elementai:

- Projektavimo šablono pavadinimas;
- Problemos aprašymas;
- Siūlomas sprendimas;
- Šablono taikymo pasekmės.

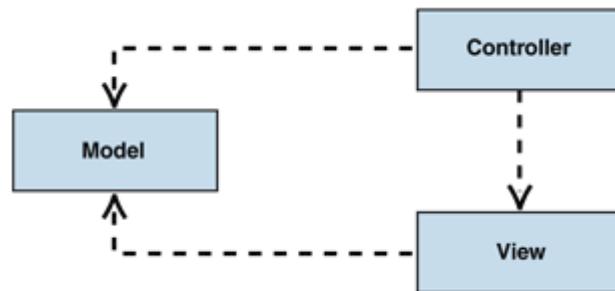
Klasifikacija

- Kūrimo (angl. Creational Patterns)
- Struktūros (angl. Structural Patterns)
- Elgsenos (angl. Behavioral Patterns)

Model – View - Controller

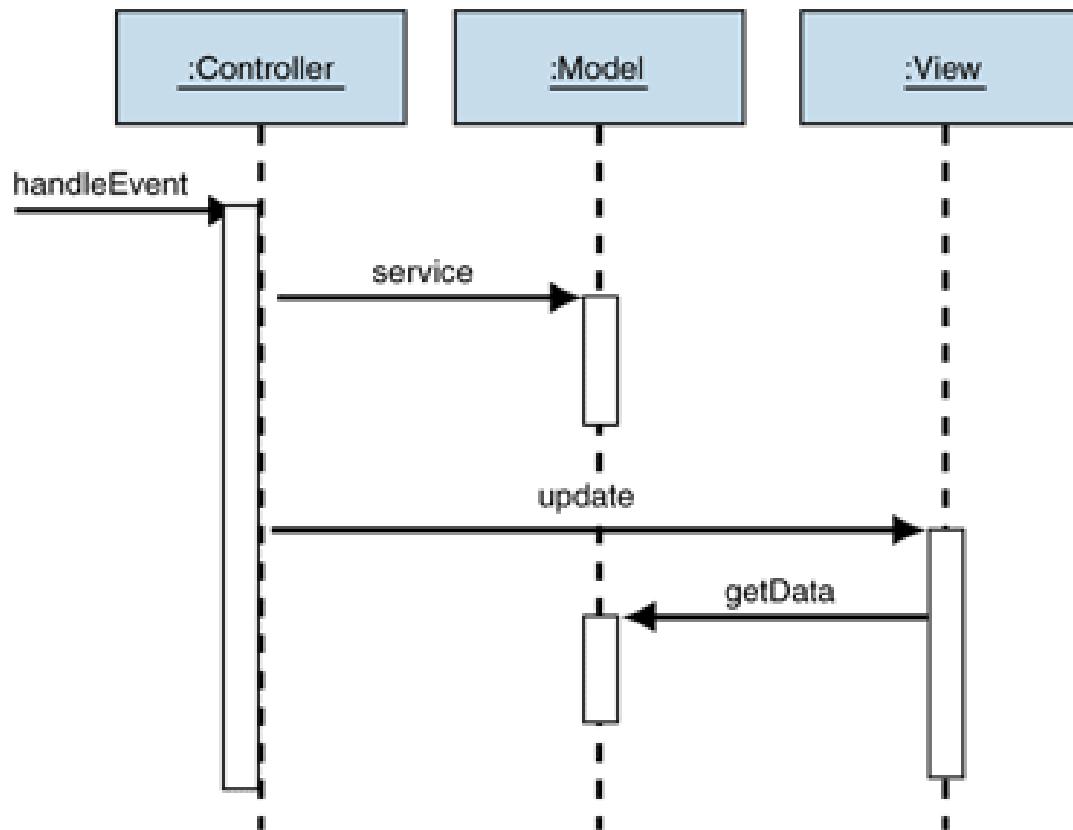
- Aprašo bendrą sistemos architektūrą.
- Trys pagrindiniai komponentai:
 - Model – aprašo sistemos duomenis, jos būseną.
 - Controller – realizuoja sąsają su vartotoju, apdoroja jo įvedamus duomenis.
 - View – realizuoja informacijos apie duomenis atvaizdavimą.

Model – View - Controller



- Controller – atitinkamai įvedamiems duomenims atnaujina modelį ir View.
- View – atsinaujina pagal modelio duomenis.

Model – View - Controller



Model – View - Controller

■ Privalumai:

- Modelis nepriklauso nuo atvaizdavimo ar vartotojo sasajos.
- Keletas atvaizdavimo variantų.
- Atspari pokyčiams.

■ Trūkumai:

- Didėja sudėtingumo lygis.
- Papildomas apdorojimas esant dideliam atnaujinimui kiekiui.

Kompozicija (angl. Composite design pattern)

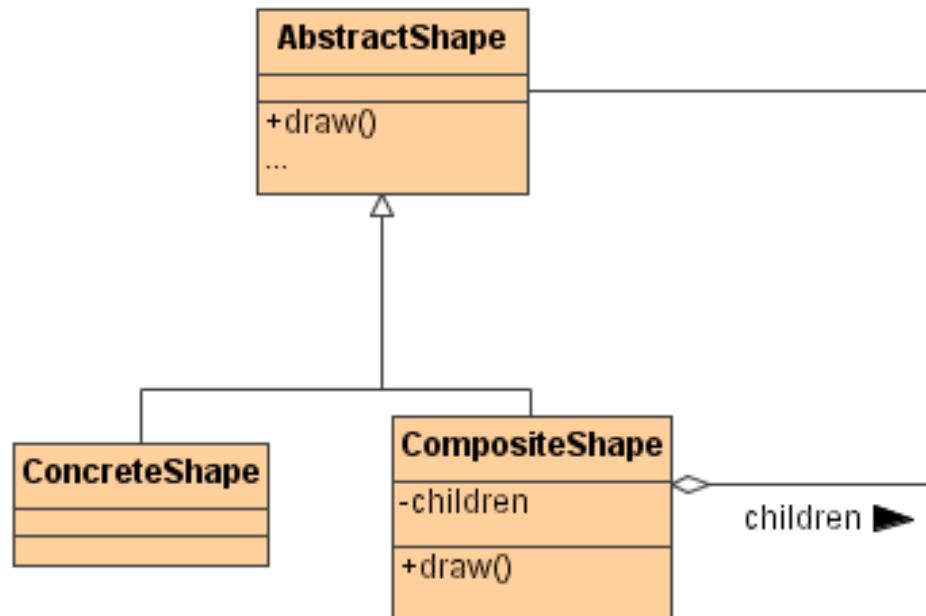
- Apjungia objektus į medžio tipo struktūras, apibrėžiant sudėtinius objektus.
- Leidžia vienodai apdoroti tiek paprastus, tiek sudėtinius objektus (medžio šakas).
- Atitinkamai – Container tipo objektais.

Kompozicija

■ Dalyviai:

- Komponentas – aprašo komponento klasės interfeisą.
- Lapas – apibrėžia paprastus komponentus, jų elgesį.
- Kompozicija – apibrėžia sudėtinį komponentą.

Kompozicija



Kompozicija

■ Realizacija:

- Sudėtinis komponentas realizuojamas kaip klasė realizuojanti komponento interfeisą ir turinti savyje sudėtinių komponentų aibę.
- Sudėtino komponento elgesys realizuojamas aktyvuojant atitinkamus vidinių komponentų metodus.

Kompozicija

■ Privalumai:

- Paprastai aprašomos sudėtingos hierarchinės struktūros.
- Supaprastina kliento realizaciją – visi komponentai pasiekiami per standartinį interfeisą.
- Supaprastina naujų komponentų tipų įvedimą.

■ Trūkumai:

- Sunku apriboti kompozicijos komponentų tipus.

Strategija (angl. Strategy design pattern)

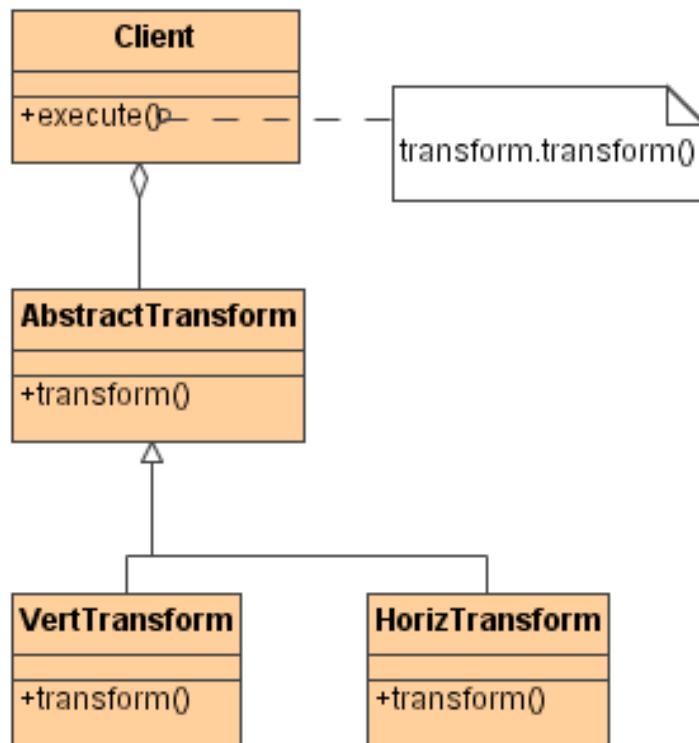
- Taikoma, kai yra daug klasių, kurios skiriasi tik realizuojamu elgesiu.
- Leidžia aprašyti algoritmų šeimas.
- Klientas gali keisti algoritmą vykdymo metu.

Strategija

■ Dalyviai:

- Klientas – objektas, kuris naudojasi strategijos realizuojamai algoritmais.
- Strategy – nusako palaikomų algoritmų interfeisą.
- ConcreteStrategy - konkreti algoritmo realizacija.

Strategija



■ Realizacija:

- Klientas nurodo, kokią konkrečią strategiją naudoti

Strategija

■ Privalumai:

- Alternatyva subklasėm.
- Eliminuoja case operatoriaus naudojimą.
- Leidžia naudoti to pačio algoritmo skirtinges realizacijas.

■ Trūkumai:

- Klientas turi žinoti apie visas esamas strategijas (neišvengiame kodo modifikavimo).
- Sudėtingesnės komunikacijos.
- Padidėjės objektų skaičius.

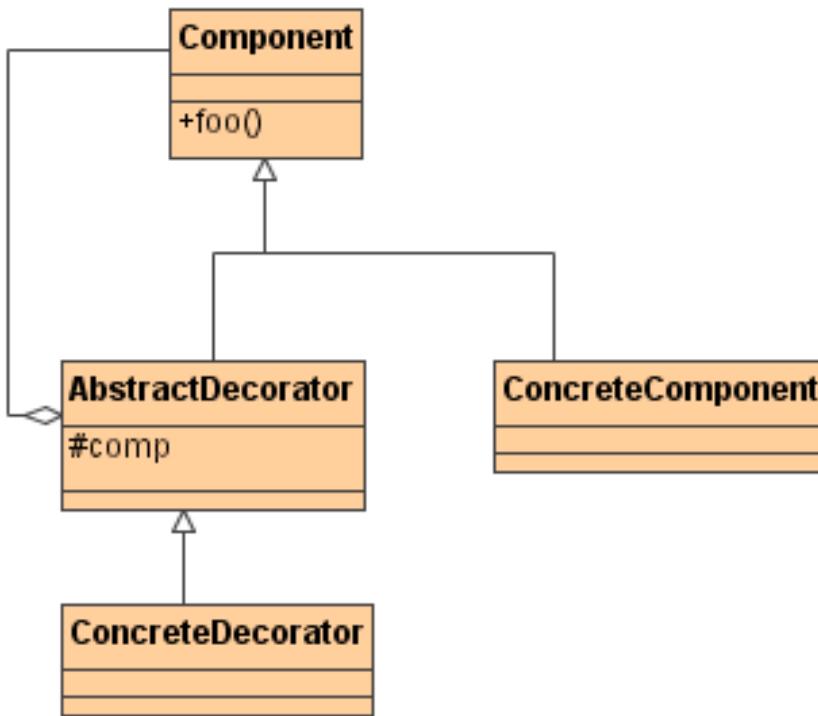
Dekoratorius (angl. Decorator design pattern)

- Alternatyva strategijai
- Leidžia dinamiškai papildyti objekto funkcionalumą
- Dar žinomas kaip *Wrapper*

Dalyviai

- Component – apibrėžia bendrą interfeisą visiems objektams, kurie gali būti dekoruojami.
- ConcreteComponent – konkretus dekoruojamas objektas.
- Decorator – laiko savyje nuorodą į dekoruojamą komponentą ir paveldi Component interfeisą.
- ConcreteDecorator – realizuoja visas interfeiso funkcija, atlieka dekoravimą.

Dekoratorius



■ Realizacija:

- `ConcreteDecorator` iskviecia kiekvieną `comp` metodą.
- Dekoravimo veiksmai atliekami prieš arba po iškvietimo.
- Tokiu būdu modifikuojamas metodo veikimas.

Dekoratorius

■ Privalumai:

- Lankstesnis sprendimas nei statinis paveldėjimas.
- Neperkraunama klasių hierarchija.

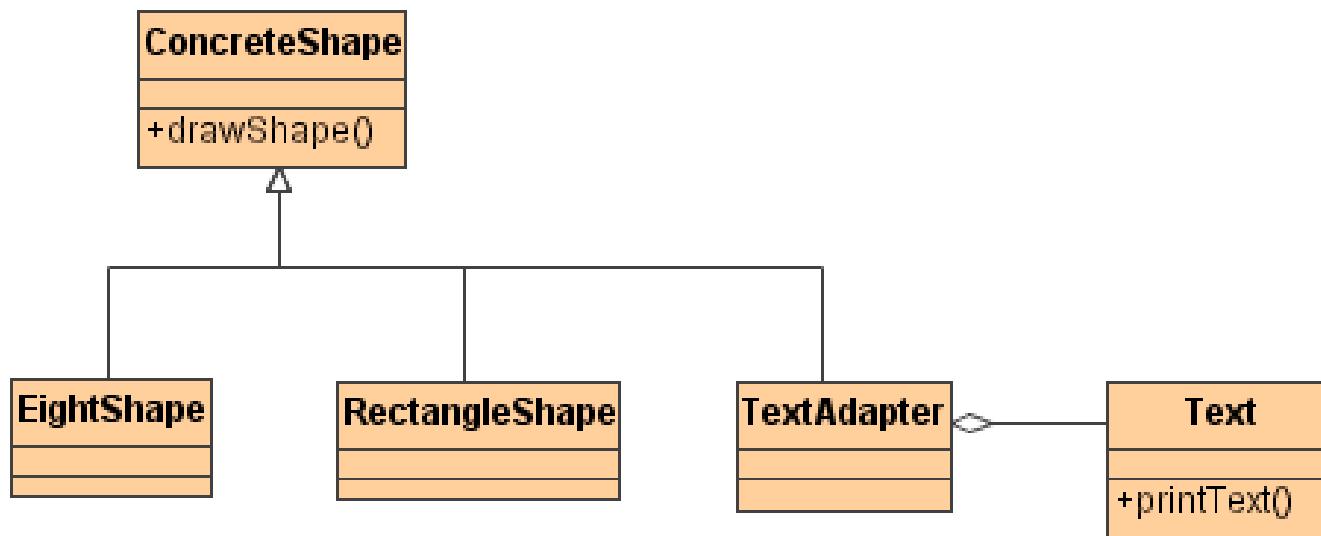
■ Trūkumai:

- Dekoratorius != Pradinis objektas.
- Objektų skaičiaus augimas.

Adapteris

- Naudojamas, kai reikia į klasių hierarchiją įtraukti klasę, kuri nėra suderinama su tévine klase.
- Realizuojama klasė-adapteris, kuri savyje turi nuorodą į įtraukimą klasę ir atlieka visus “pritaikymo” veiksmus.
- Klasė-adapteris realizuoją visus tévinės klasės apibrėžtus metodus.

Adapteris



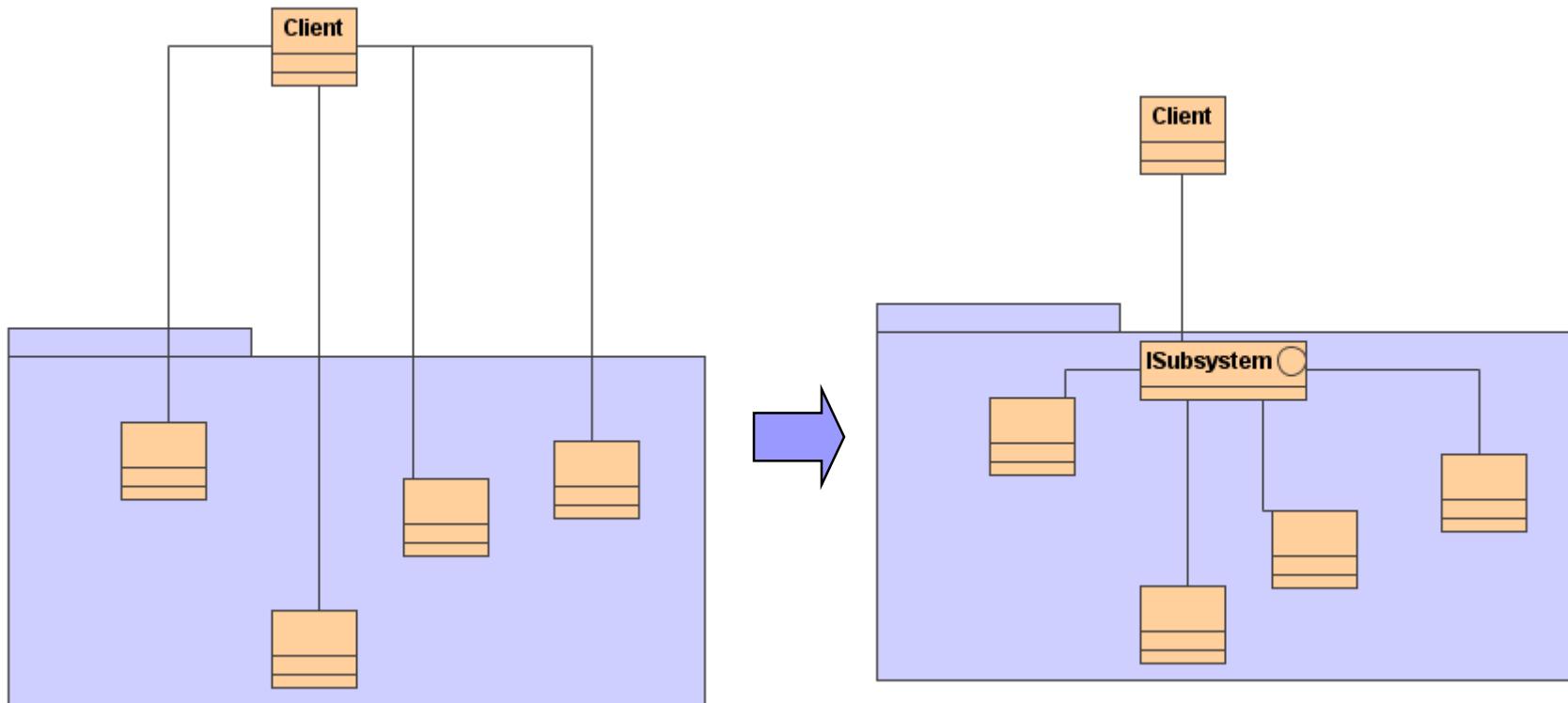
Adapteris

- Pastabos:
 - Objektiniai ir klasiniai (naudojant daugialypį paveldėjimą, angl. multiple inheritance) adapteriai.
 - Kiek darbo atlieka adapteris?
 - Dvigubi adapteriai permatomumo realizacijai.

Fasadas (Facade design pattern)

- Pateikia vientisą interfeisą prie posistemės.
- Posistemės klasės yra “paslepiamos” paketo viduje, klientui suteikiant tik interfeisą.
- Naudojama daugiau kodo struktūrizavimui nei saugumo sumetimais.

Fasadas



Literatūra

- https://www.tutorialspoint.com/design_pattern/index.htm
- <https://refactoring.guru/design-patterns>
- <https://www.dofactory.com/net/design-patterns>

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Harmony/ESW kūrimo proceso principai

Prof. dr. Agnus Liutkevičius,

Kompiuterių katedra

Studentų g. 50-202, tel. 300394

Kas yra sistemų kūrimo procesas?

- Procesas nusako žingsnius, kuriuos reikia atlikti, norint pasiekti tam tikrą tikslą, mūsų atveju – sukurti įterptinę sistemą.
- Procesas apibrėžia:
 - Roles – kas atlieka veiklas.
 - Užduotis – kas bus atliekama.
 - Užduočių vykdymo sekas.
 - Sukuriamus produktus (tame tarpe ir tarpinius).

Sistemų kūrimo proceso elementai

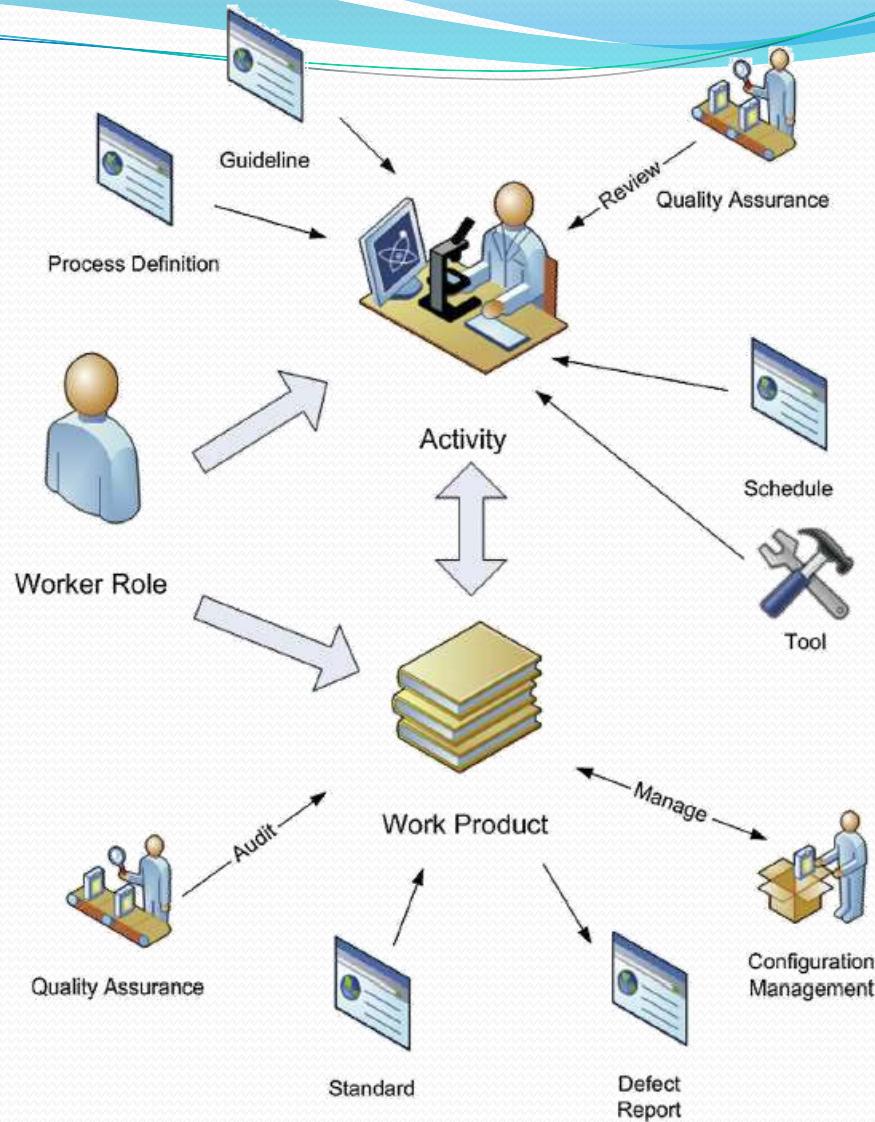


Figure 2.1
Basic elements of a process.

Kam išvis reikalingas procesas ir visos tos taisyklės (1)

- Nedidelę sistemą gali sukurti ir vienas talentingas inžinierius.
 - Kokybė, korektiškumas ar patikimumas pilnai priklauso nuo vieno žmogaus įgūdžių ir patirties.
 - Procesas nereikalingas, nes patyręs inžinierius nesunkiai pasiekia užsibrėžtą tikslą, sugeba su juo susidoroti.
 - Iš tiesų, inžinierius naudoja procesą, tik jis yra individualus, būdingas tik šiam specialistui.

Kam išvis reikalingas procesas ir visos tos taisyklės (2)

- Deja, dauguma sistemų per sudėtingos, kad jas galėtų sukurti 1 žmogus...
- Inžinieriu komanda privalo:
 - Suprasti, kaip pasidalinti darbus;
 - Siekti bendro tikslų;
 - Suintegravoti savo darbus į vieningą sistemą.
- Procesas apibrėžia atskirų rolių veiksmus ir sąveiką, siekiant sukurti kompiuterinę sistemą.

Kuo skiriasi geras procesas nuo blogo?

- Gero proceso požymiai:
 - Daugiau funkcionalumo;
 - Trumpesnis laikas;
 - Mažesnė kaina;
 - Mažiau defektų ir klaidų.

Harmony/ESW principai (1)

- Atitinka Agile kūrimo principus:
- Pagrindinis tikslas – sukurti **veikiančią įrangą**.
- Reikia nuolat sekti šio tikslo įvykdymo progresą, atsižvelgiant būtent iš įrangos veikimo lygi (realizuotą funkcionumą), o ne iš kodo eilučių skaičių.
- Geriausias būdas turėti gerai veikiančią įrangą – nedaryti klaidų ☺, t.y. orientuotis iš kokybiškesnę realizaciją, negalvojant, kad vėliau bus galima tas klaidas aptikti ir pataisyti.

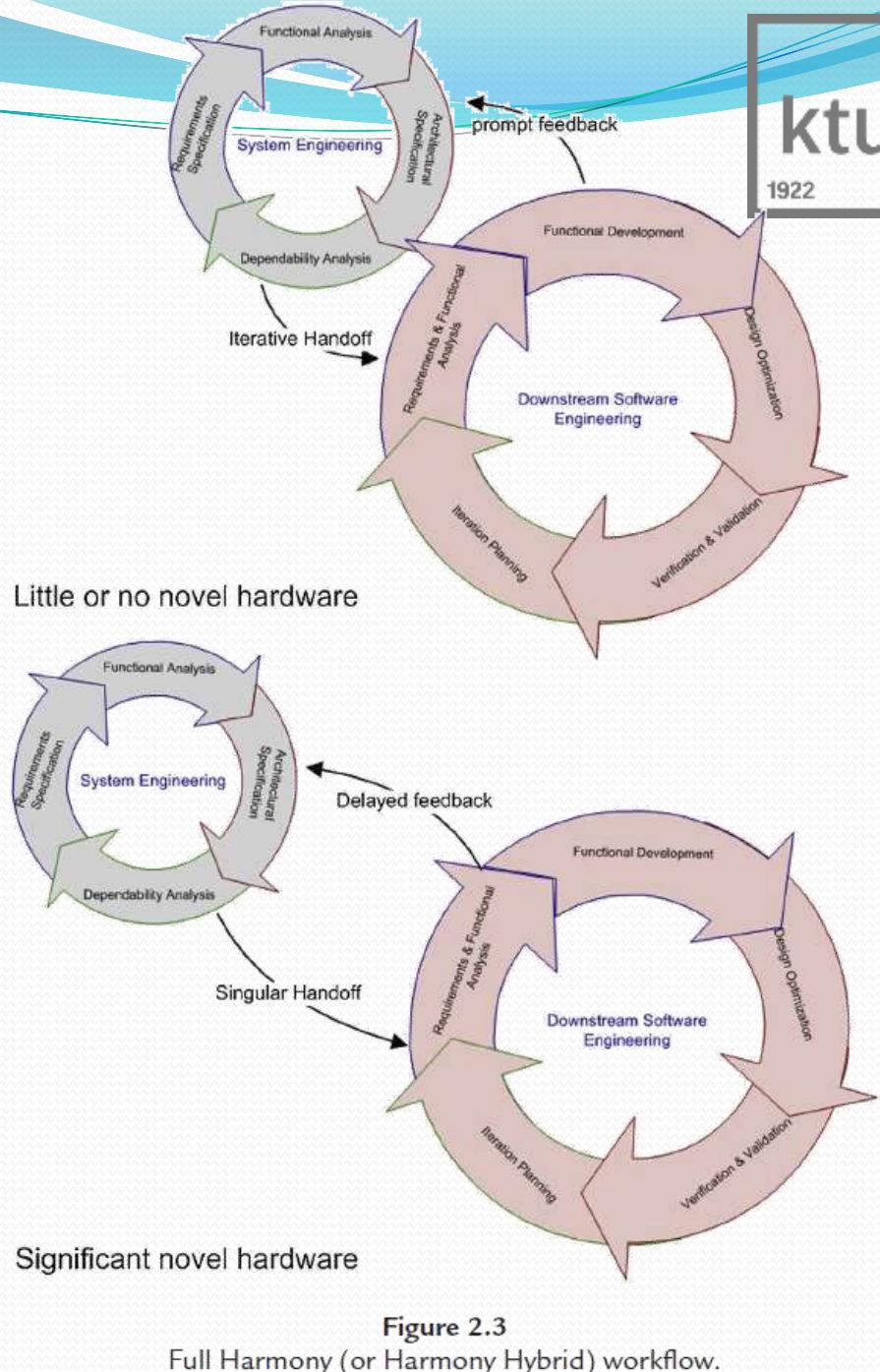
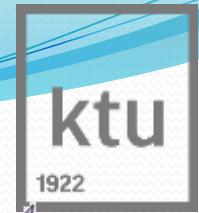
Harmony/ESW principai (2)

- Nuolat bendrauti su užsakovu ir rodyti jam progresą – tai esminis principas.
- Naudoti anksčiau išnagrinėtus 5 lygių architektūros modelius.
- Planuoti, sekti ir adaptuoti.
- Įvertinti rizikas.
- Nuolat vertinti kokybę.
- Naudoti MDA principus.

Pagrindinis principas – veikianti programinė įranga

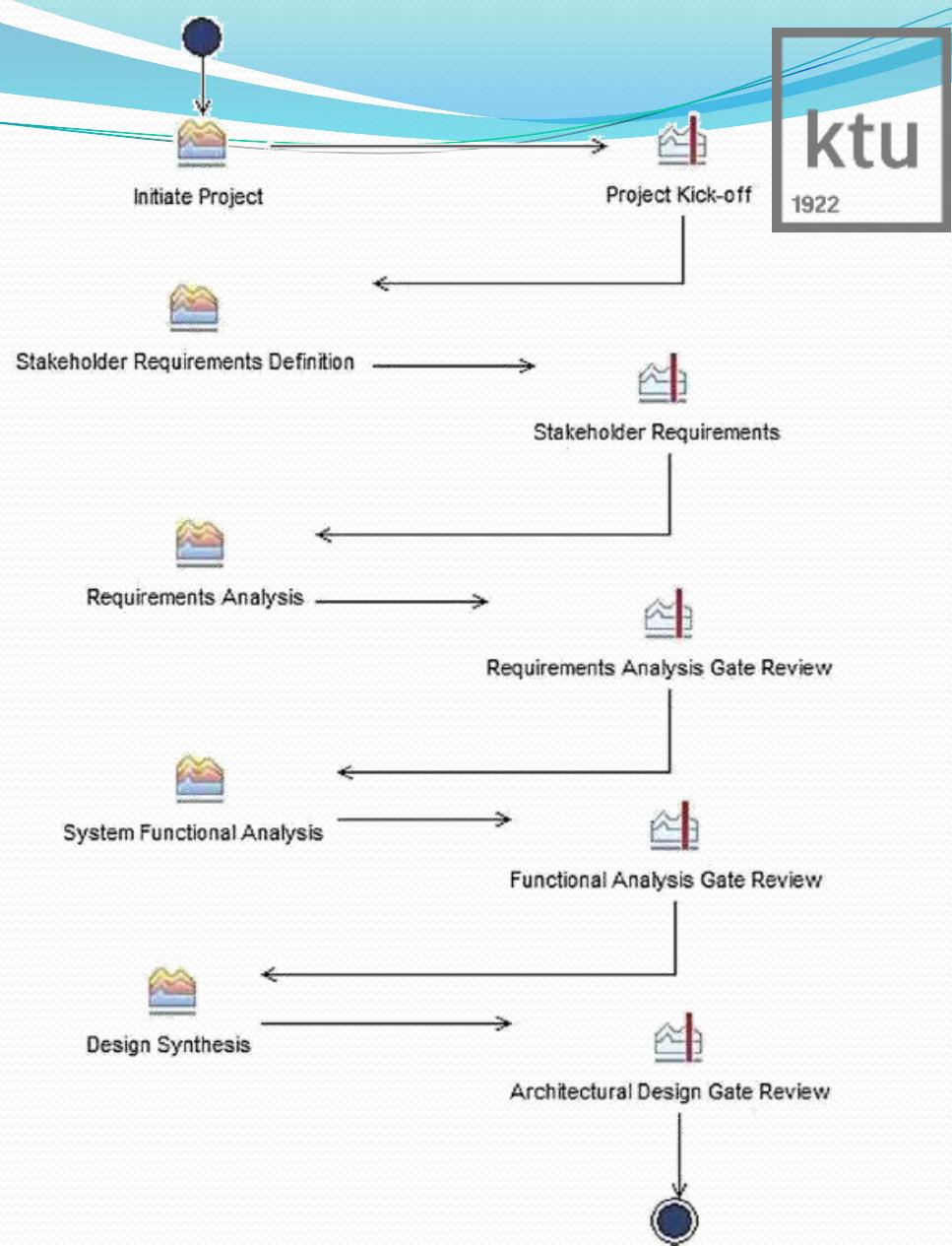
- Įranga turi atitikti vartotojų/užsakovų lūkesčius.
- Kiekviena stambesnė užduotis veda prie naujo funkcionalumo specifikavimo, sukūrimo, ištestavimo ir diegimo įrangos, kurią **galima pademonstruoti!**
- Gerai organizuotame projekte:
 - 80% laiko praleidžiama vykdant veiklas, susijusias su veikiančios įrangos pristatymu užsakovui;
 - Nemažiau nei 50% laiko turi būti skirta tiesioginiams įrangos kūrimui.

HW ir SW bendras kūrimas (angl. hardware- software co- development)



Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p.
ISBN13: 9780124077812

Sistemų inžinerijos veiklos Harmony kūrimo procese (HW kūrimo veiklos)



Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p. ISBN13: 9780124077812

Figure 2.4
Systems engineering activities of the Full Harmony process.

SW kūrimo (development) proceso veiklos

- Reikalavimų specifikavimas (CIM);
- Architektūros specifikavimas (PIM);
- Optimizacijos kriterijų identifikavimas ir specifikavimas;
- Projektavimo šablonų naudojimas PSM kūrimui;
- Kodo generavimas iš modelio + rankinis programavimas;
- Kodo derinimas;
- Testavimo atvejų kūrimas.

Harmony /ESW proceso veiklos

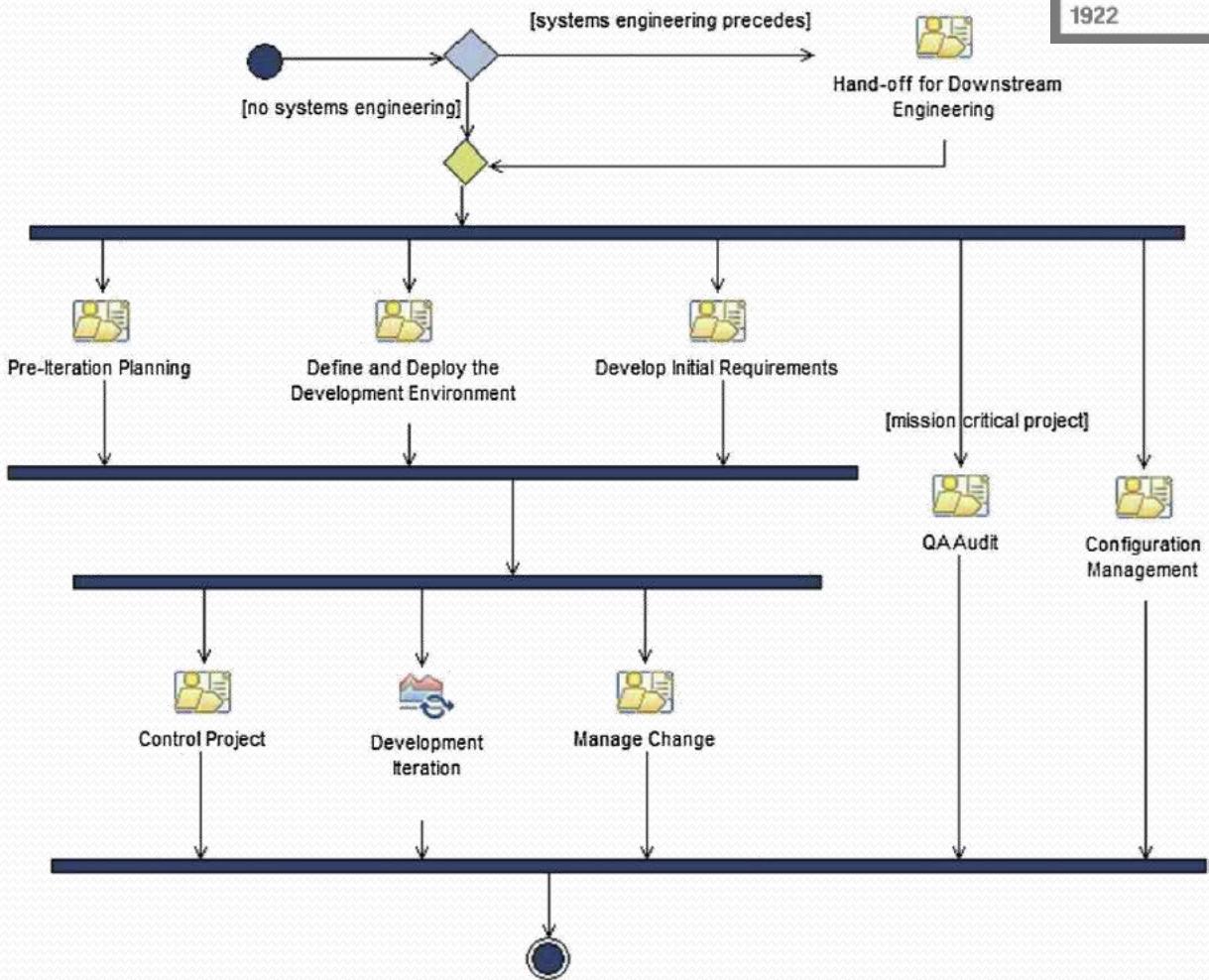


Figure 2.6
Delivery process for Harmony/ESW.

Vienos iteracijos veiklos

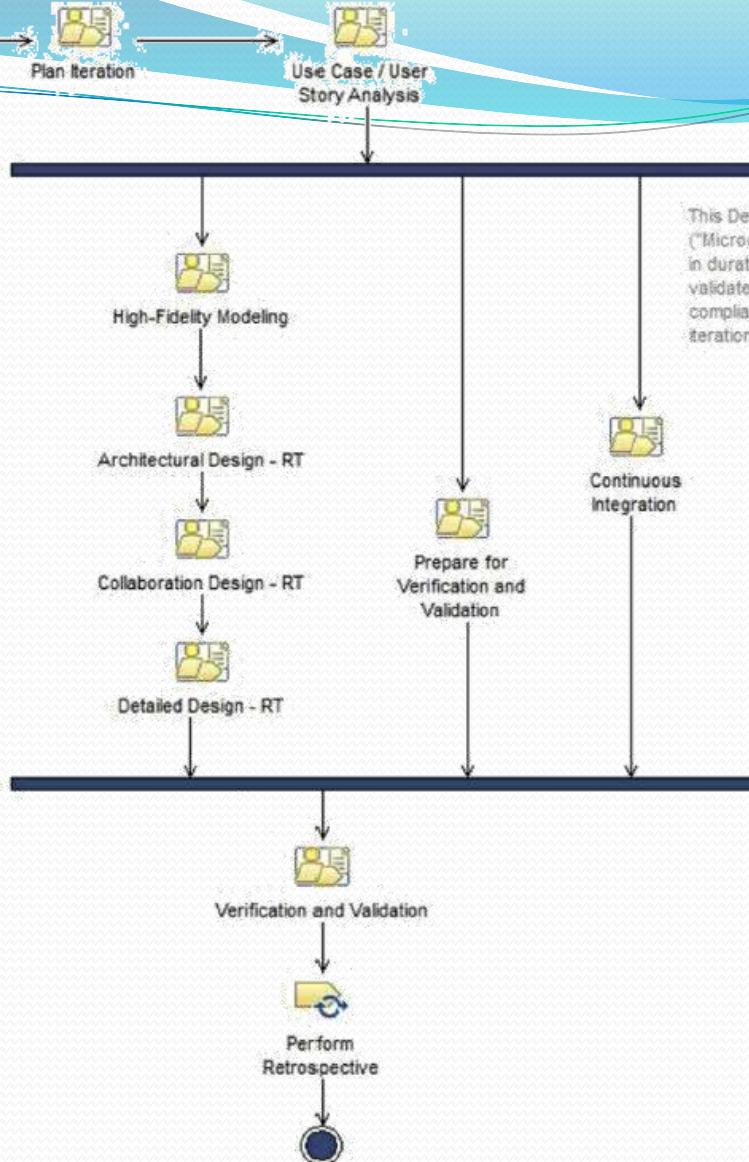


Figure 2.5
Harmony development iteration overview.

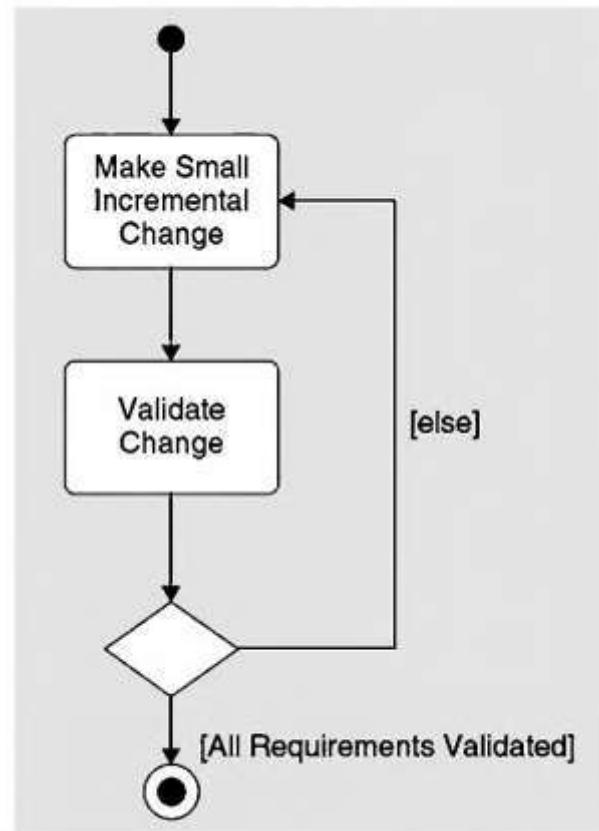
Progreso vertinimas, atsižvelgiant į tikslą (funktionalumą)

- Negalima vertinti projekto progreso pagal vykdomų ar atliktų veiklų progresą.
- Jeigu vertiname pagal įvykdytus reikalavimus, kodo eilutes ar defektų skaičių, tai tik parodo, kaip gerai laikomasi plano, bet ne kaip gerai siekiama tikslas ... nes planai dažniausiai neteisingi ☺
- Geriausias progreso matas – tai sukurtas, veikiantis, pristatytas ir svarbiausia validuotas (vartotojo patikrintas) funkcionalumas, kuris egzistuoja kiekvieno laiko momentu.
 - Pvz. kiek use case atvejų jau realizuota ir validuota.

Pagrindinis įvertis – veikianti programinė įranga (1)

- Ištestuotos ar validuotos įrangos kiekis turi būti įvertinamas **kasdien**.
 - Įranga tobulinama nedideliais žingsniais ir jos modeliai (kodas) vykdomi keletą kartų per dieną.
 - Galima naudoti tiek imitacinių modeliavimų, tiek sugeneruoto kodo testavimą.
- Modelių vykdymas pradedamas kuo anksčiau.
 - Praktiškai realizavus 1 ar 2 vartojimo atvejus, jau galima ir reikia pradėti sistemos vykdymą ir validavimą.

Pagrindinis įvertis – veikianti programinė įranga (2)



Kodo veikimas savaimė dar nieko nereiškia

- Vien tik dažnas kompiliavimas ir tikrinimas ar veikia dar nereiškia, kad **veikia kaip reikia** ☺
- Modifikuojamas komponentas turi:
 - Teisingai sąveikauti su kitomis sistemos dalimis;
 - Turi teisingai elgtis pagal vartojimo atvejų scenarijus.
- 1 vartojimo atvejis = keletas vartojimo scenarijų, kuriuos **visus** reikia ištестuoti.

Gynybinis kūrimo procesas (Defensive development)

- Kurdami SW stengiamės nepasitikėti tuo, ką kuriame

- Kiekviena funkcija, operacija, klasė ar kita įrangos dalis, kuriama numanant, kad bus išpildytos tam tikros **pradinės sąlygos** (angl. conditional invariants).
- Viskas gerai, jeigu tos sąlygos patenkinamos ir teisingos, o jeigu ne?
 - Kuriama įranga veiks nekorektiškai, nebent yra numatomos galimybės savaiminiam tokių situacijų identifikavimui ir pataisymui.

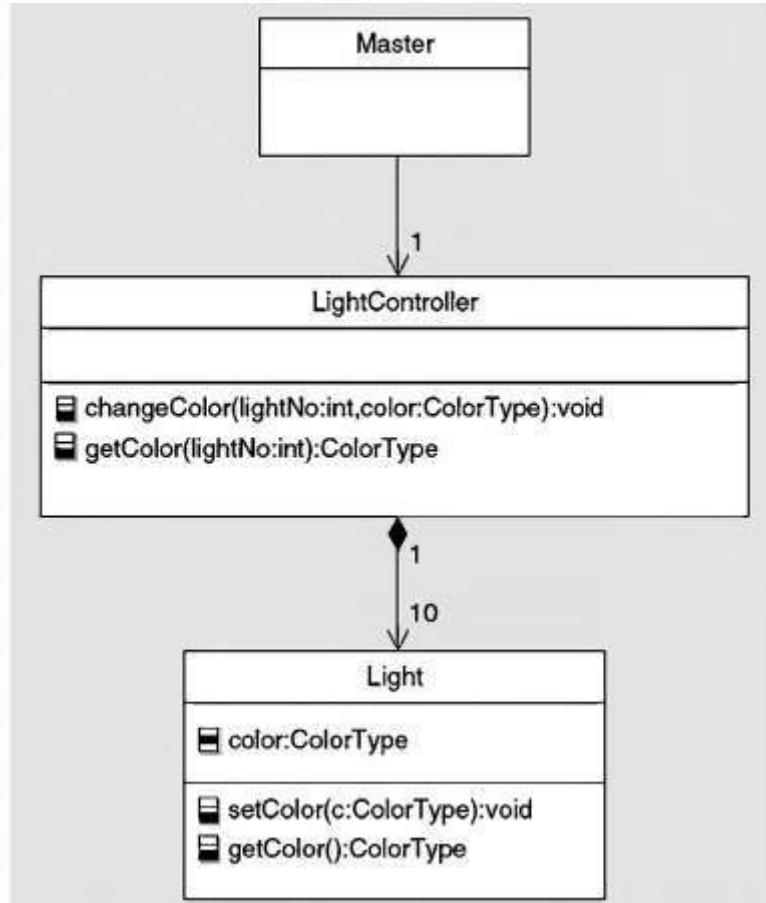
Gynybinio proceso etapai

- Identifikuojamos visos įmanomos pradinės sąlygos – apribojimai.
- Identifikuojama teisinga elgsena ir rezultatai, jeigu pradinės sąlygos bus pažeistos.
- Sukuriama papildoma programinė logika, kuri įvertina, ar sąlygos yra įvykdytos.
- Sukuriama papildoma programinė logika, kuri imasi atitinkamų veiksmų, jeigu sąlygos nėra įvykdytos.
- Validuojama elgsena, kai sąlygos yra įvykdytos.
- Validuojama elgsena, kai sąlygos yra neįvykdytos.

Gynybinio kūrimo proceso idėja

- Pradžioje patikrink sąlygas, o tik tada imkis atitinkamų veiksmų.
- Salygų/apribojimų pavyzdžiai pvz.:
 - Ar pakaks atminties objekto sukūrimui?
 - Ar vartotojas nebandys įvesti visokių nesamoniu?
 - Ar išorinės sistemos visada veiks ir suteiks paslaugas?
 - Ar bus patenkinami QoS reikalavimai?

Gynybinio kūrimo proceso pavyzdys (1)



Kas nutiks, jei kviečiant LightController metodus bus naudojamas $lightNo > 9$?

Kas nutiks, jeigu bus norodomos neteisingas spalvos kodas?

Gynybinio kūrimo proceso pavyzdys (2)

- Reikia patikrinti, ar i metodus paduodami korekтиški parametrai. Galimi tokie veiksmų variantai:
 - Modifikuoti metodus, kad jie nieko nedarytų ir gražintų klaidos kodą.
 - Sugeneruoti išskirtinės situacijos objektą (exception) su detaliu klaidos aprašymu.
 - Priimti neteisingus parametrus, bet pakeisti juos reikšmėmis pagal nutylejimą (defaults).
 - Tiesiog ignoruoti kvietimą.

Gynybinio kūrimo proceso minusai

- Papildomas laikas skaičiavimams ir tikrinimams.
 - Papildomos kodo eilutės.
 - Papildomas programavimo darbas.
-
- Tačiau daugumoje sistemų tiesiog **BŪTINA** visuomet tikrinti pradines sąlygas...

Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (1)

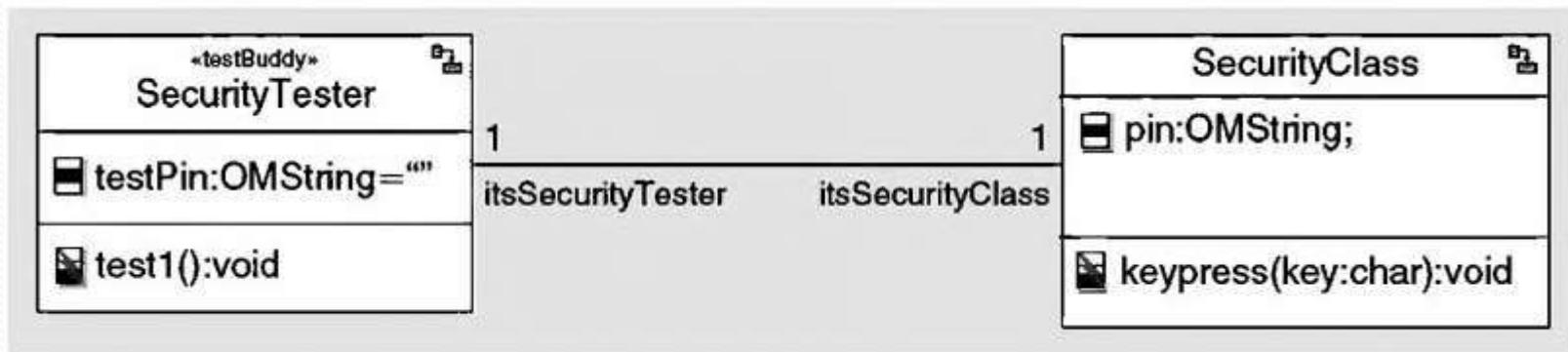
- PIN susideda iš 4 skaitmenų.
- Vartotojas turi turėti galiojančią sąskaitą.
- Įvestas PIN turi sutapti su DB saugomu PIN.
- „Enter“ paspaudimas suaktyvina validavimą.
- „Cancel“ nutraukia operaciją.
- „Backspace“ ištrina vieną simbolį.

Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (2)

- 1 kūrimo etapas:
 - Priimamas PIN iš skaitmenų tol, kol nenuspaudžiamas ENTER.
 - PIN ilgis, jo validavimas ar operacijos nutraukimas nevykdomas.
 - Testavimui naudojama testinė *SecurityTester* klasė.
 - *SecurityClass* priima mygtukų paspaudimus: 0..9, Enter, Cancel.

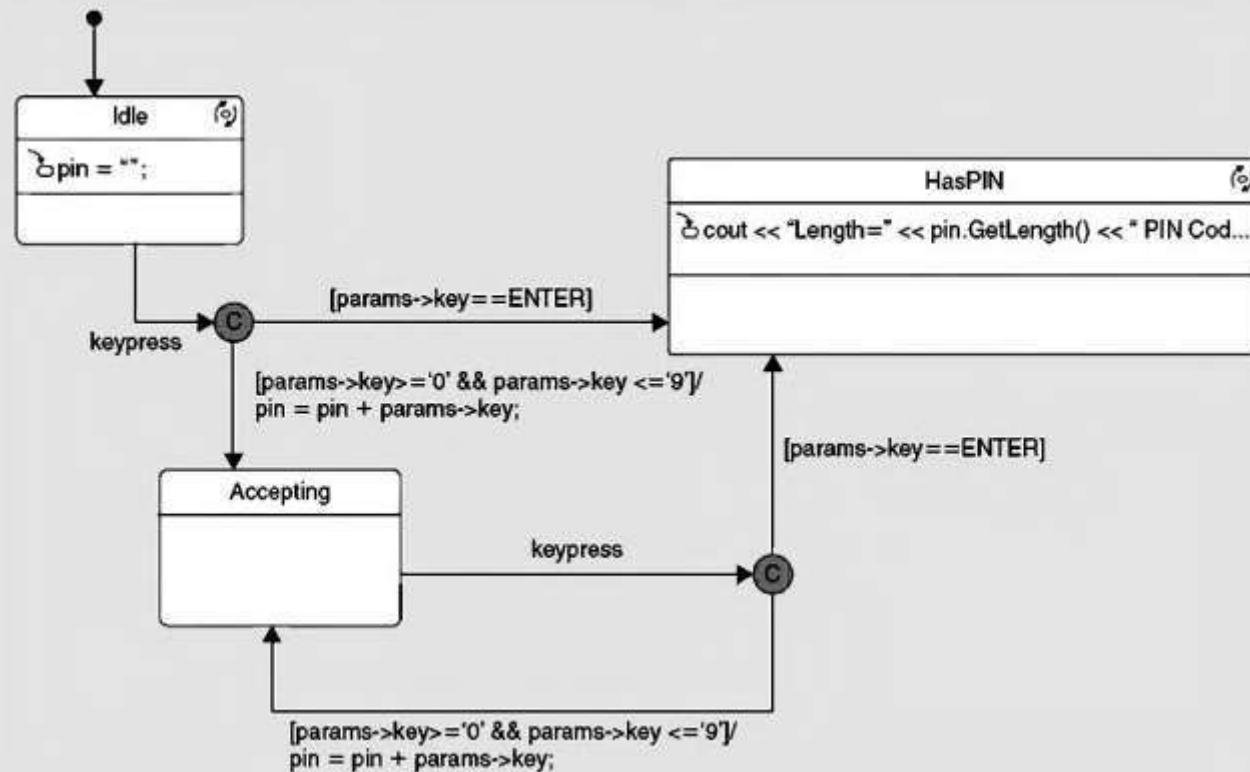
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (3)

- 1 kūrimo etapas:



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (4)

- 1 kūrimo etapas: *SecurityClass* būsenų diagrama

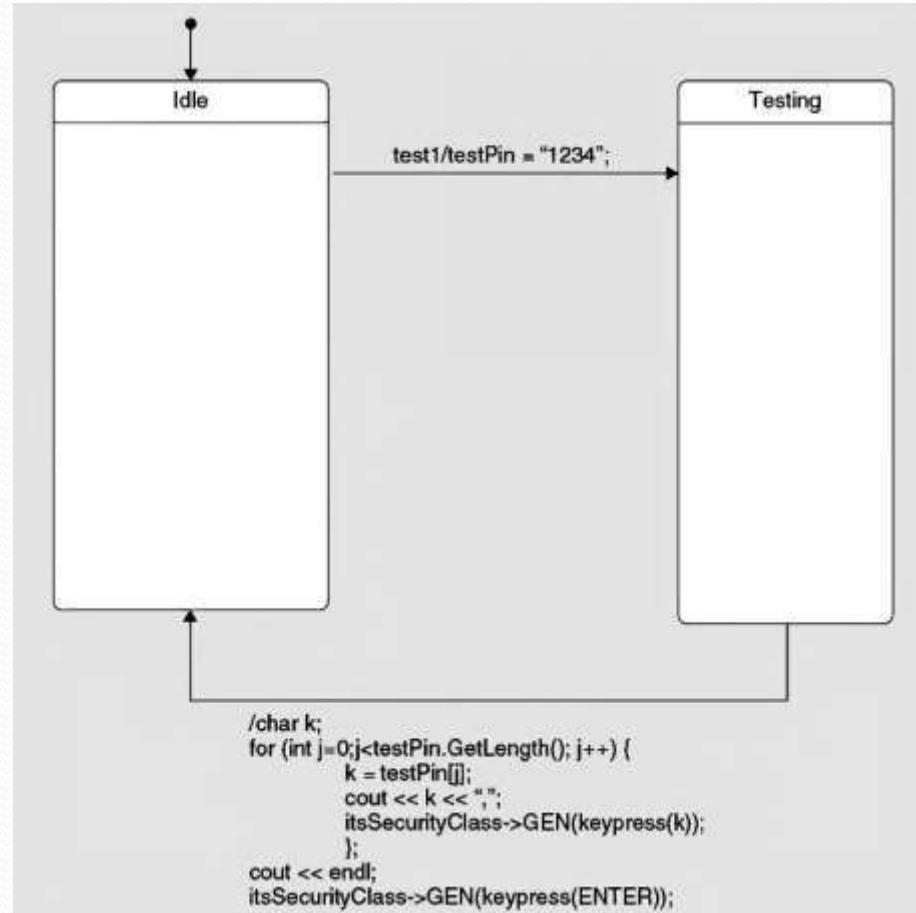


Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (5)

- 1 kūrimo etapas:

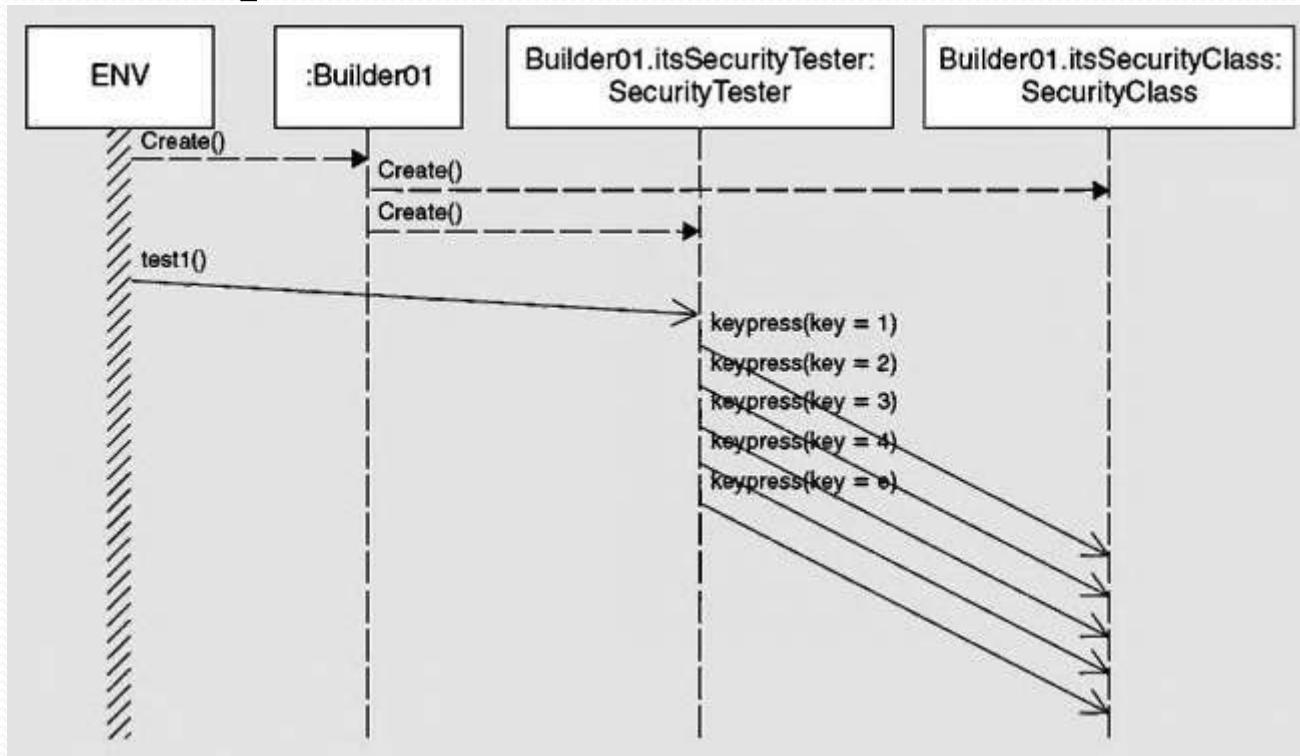
SecurityTester

būsenų diagrama



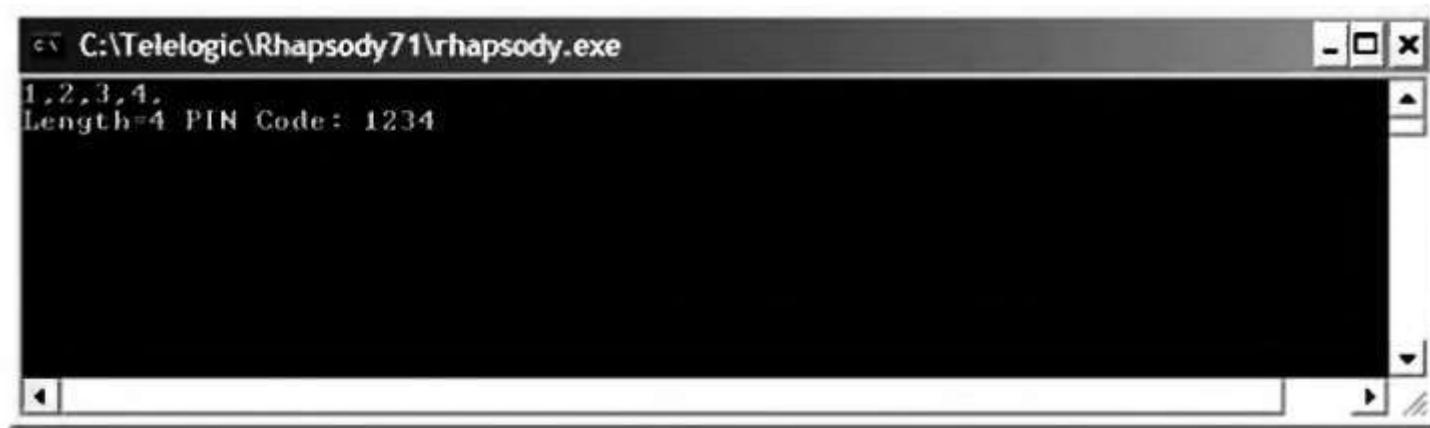
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (6)

- 1 kūrimo etapas:



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (7)

- 1 kūrimo etapas:

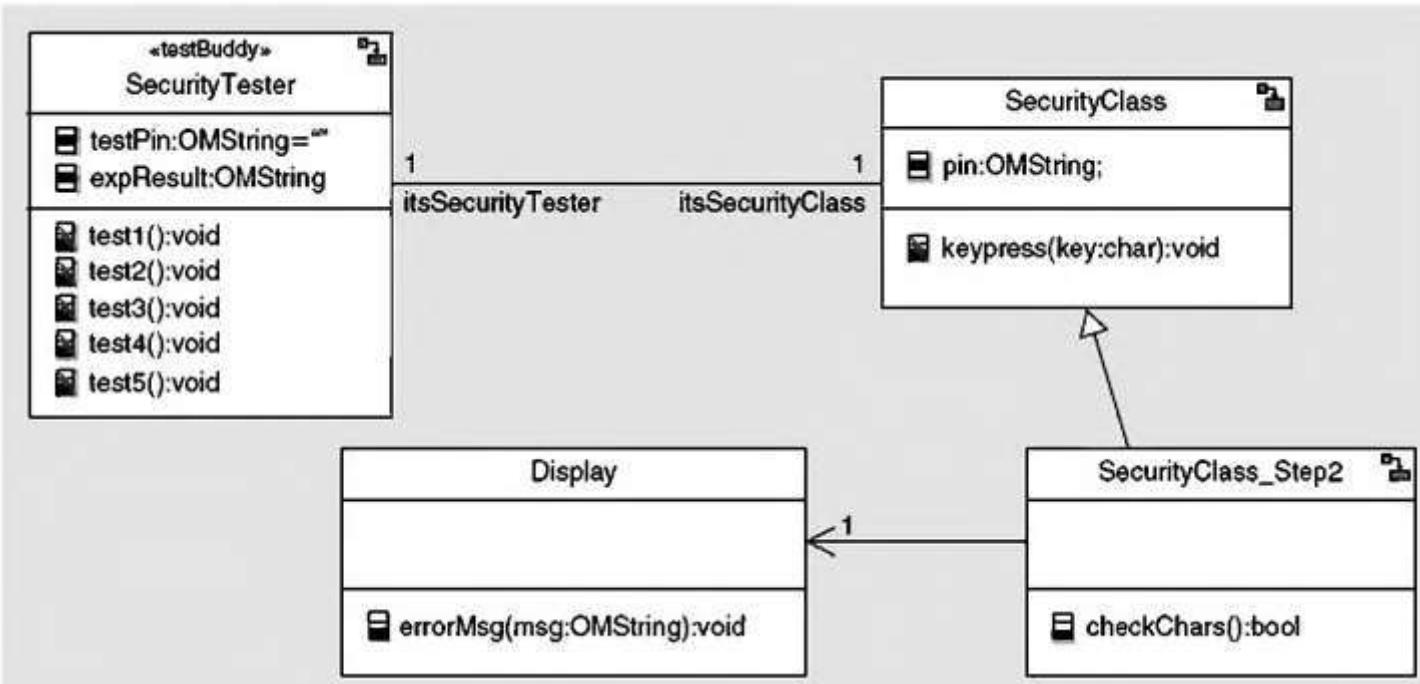


Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (8)

- 2 kūrimo etapas:
 - Tirkriama, ar PIN ilgis = 4.
 - Tirkriama, ar įvedami skaitmenys.

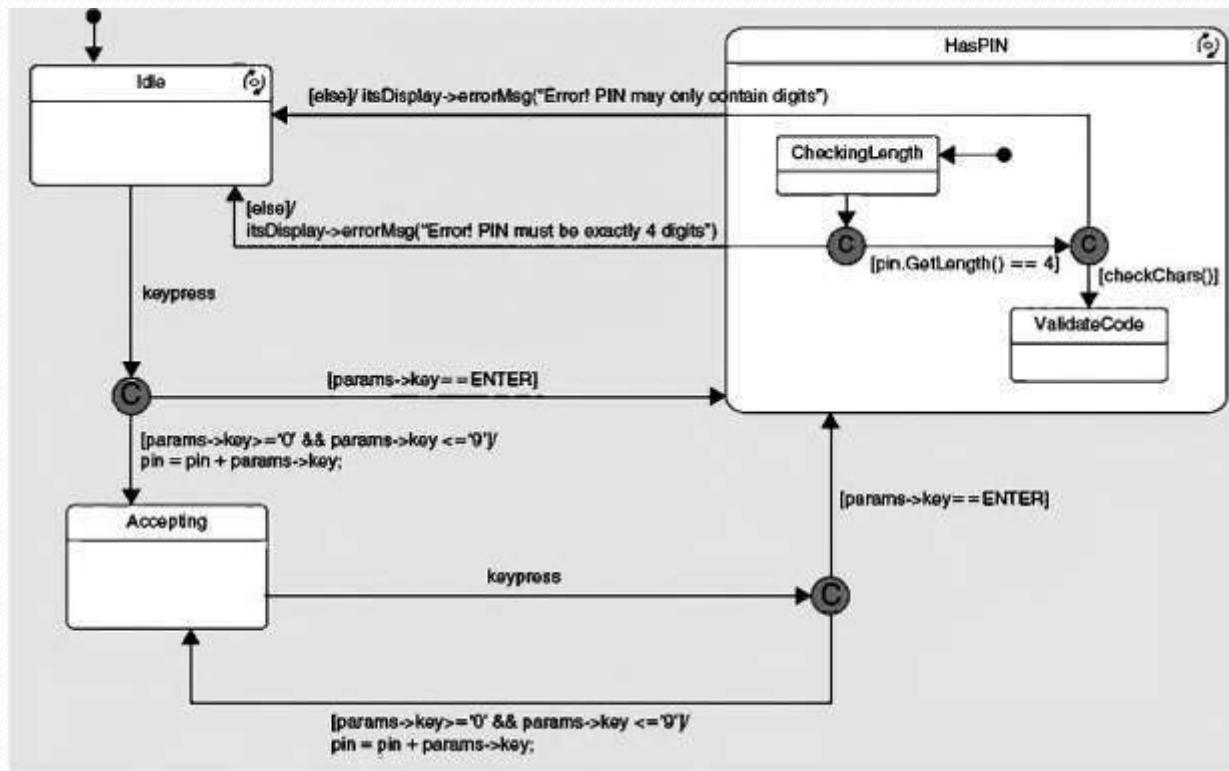
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (9)

- 2 kūrimo etapas:



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (10)

- 2 kūrimo etapas: *SecurityClass* būsenų diagrama

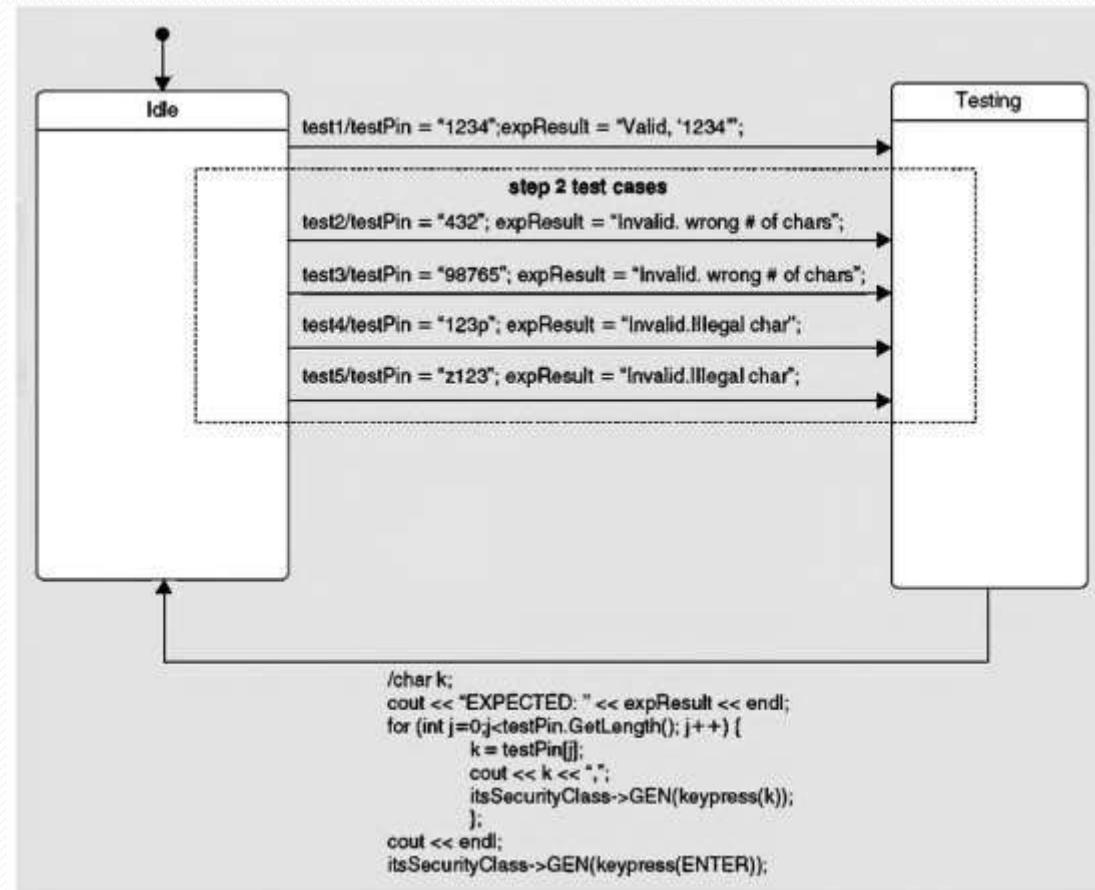


Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (11)

- 2 kūrimo etapas:

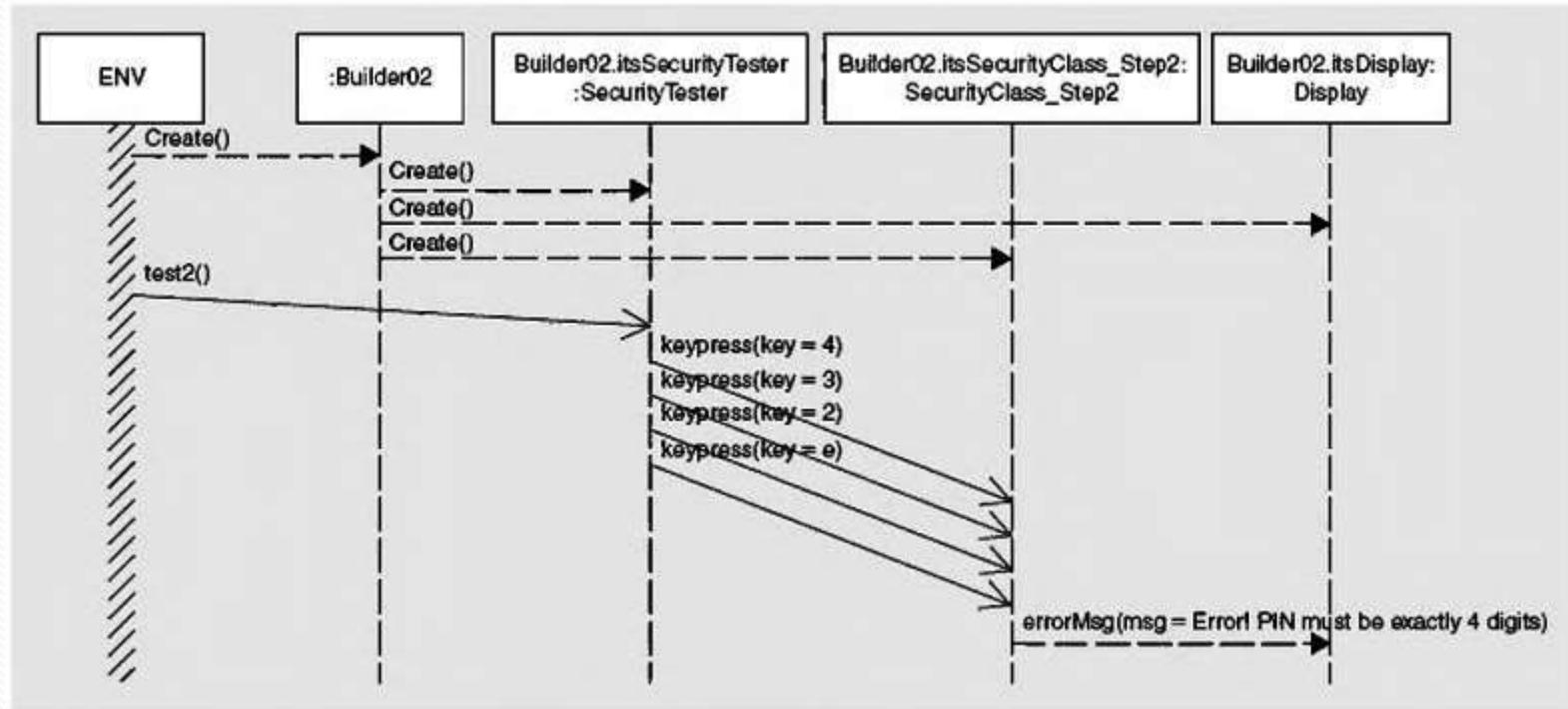
SecurityTester

būsenų diagrama



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (12)

- 2 kūrimo etapas:



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (13)

- 2 kūrimo etapas:



C:\Telelogic\Rhapsody71\rhapsody.exe

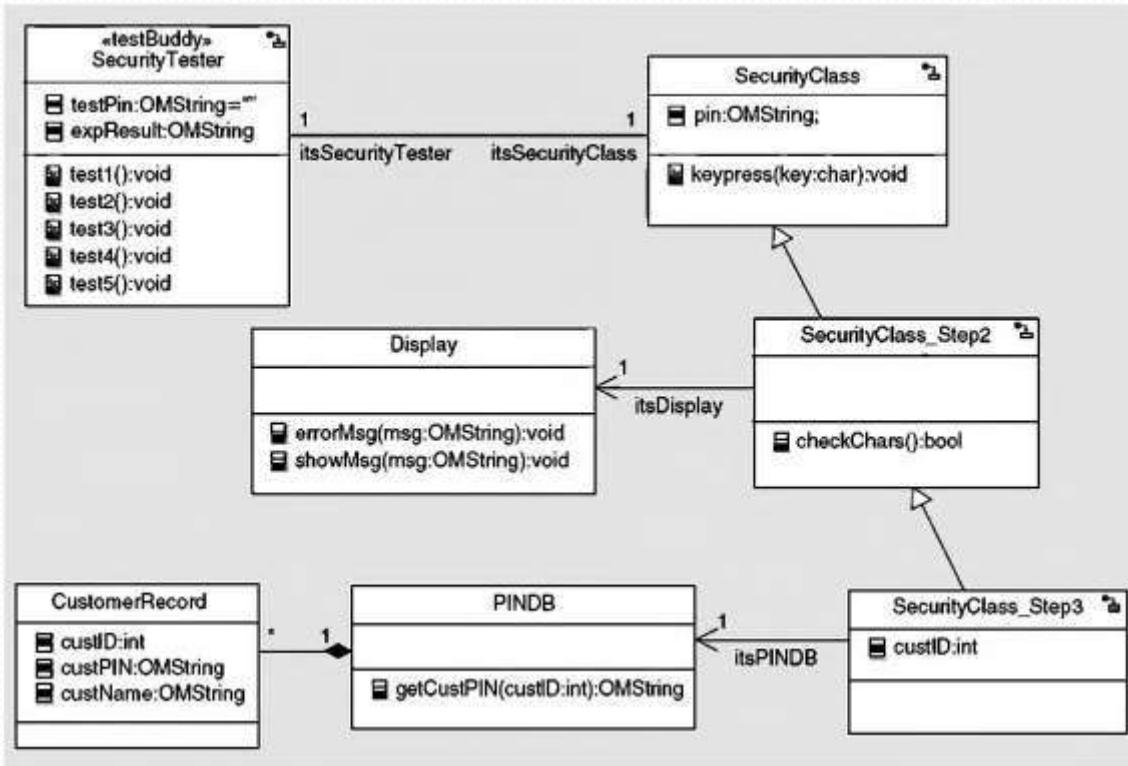
```
EXPECTED: Invalid. wrong # of chars
4,3,2,
Length=3 PIN Code: 432
Error! PIN must be exactly 4 digits
```

Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (14)

- 3 kūrimo etapas:
 - Pridedama CANCEL logika.
 - Jei PIN korektiškas, lyginamas su esančiu DB.

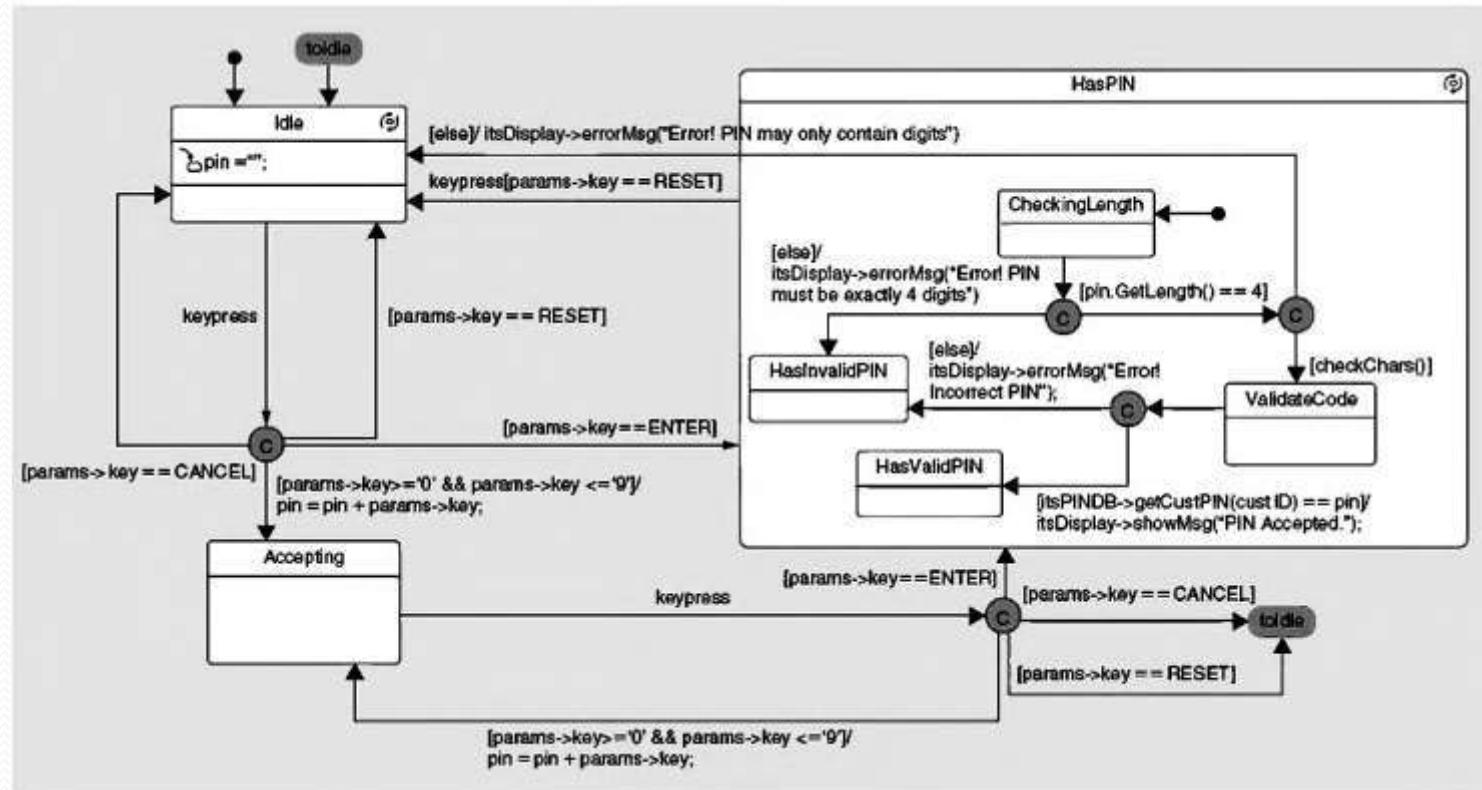
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (15)

- 3 kūrimo etapas:



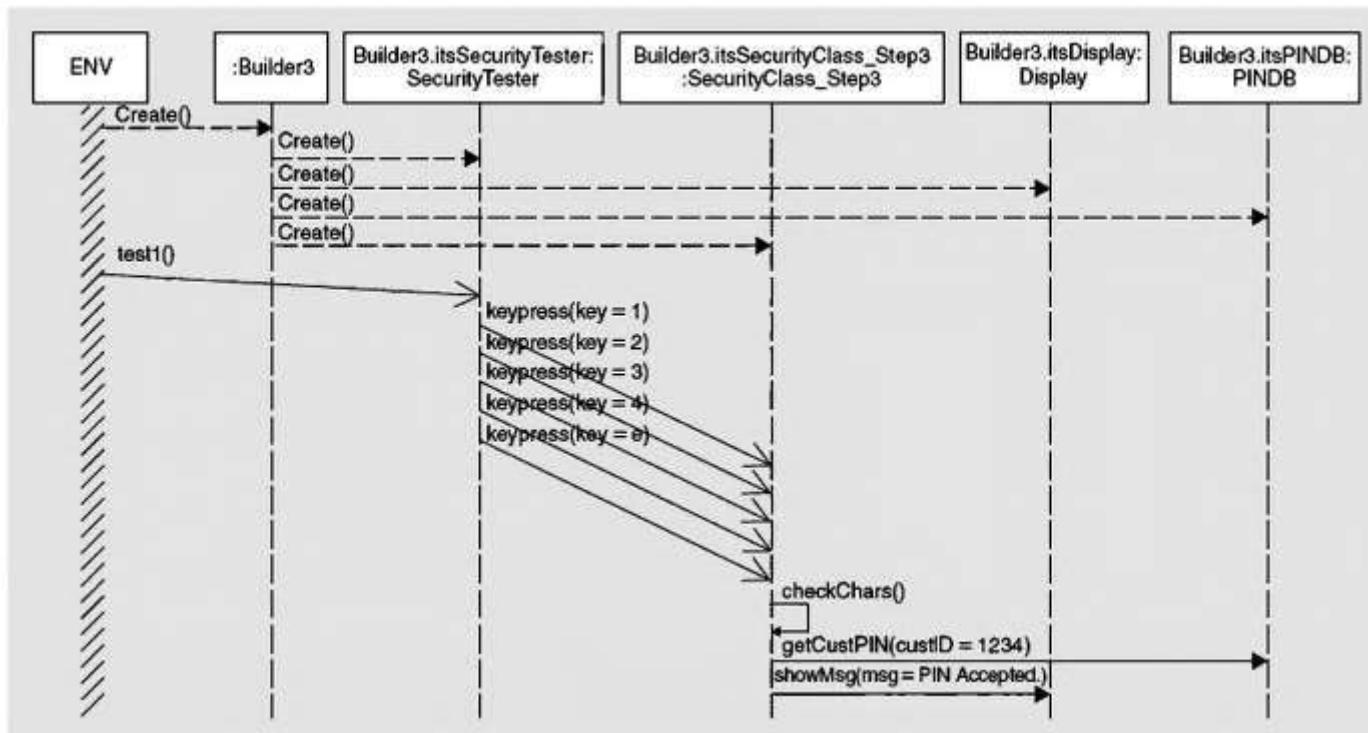
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (16)

- 3 kūrimo etapas: *SecurityClass* būsenų diagrama



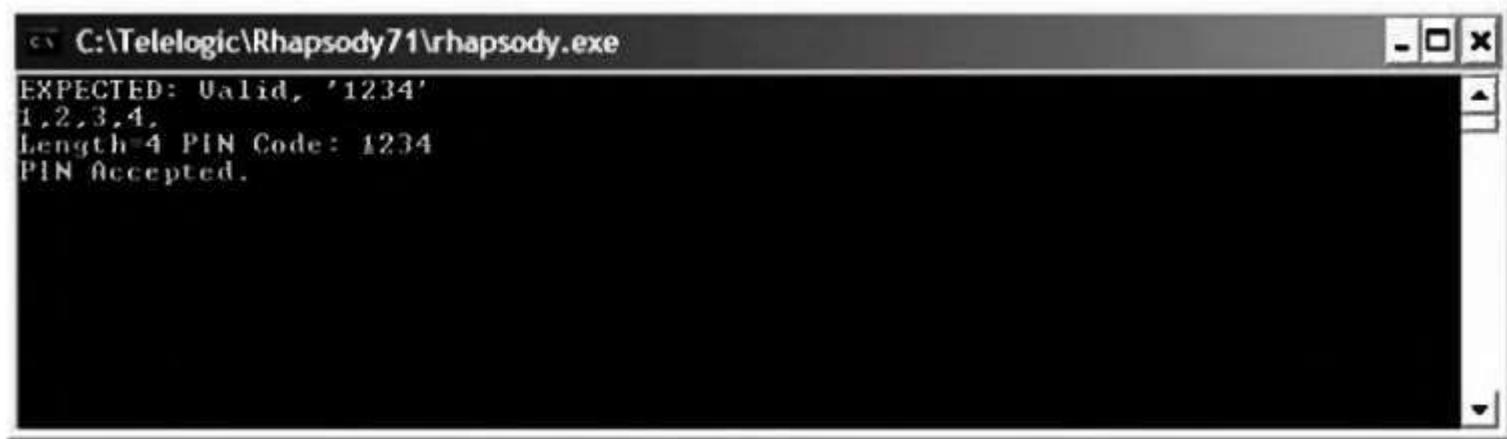
Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (17)

- 3 kūrimo etapas:



Iteratyvus kūrimas: bankomato PIN įvedimo komponento pavyzdys (18)

- 3 kūrimo etapas:



```
C:\Telelogic\Rhapsody71\rhapsody.exe
EXPECTED: Valid, '1234'
1,2,3,4,
Length=4 PIN Code: 1234
PIN Accepted.
```

Planavimas, sekimas ir adaptavimas

- „Tu nepažisti to, ko nepamatuoji“.

Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams* (New York: Dorset House Publishing, 1999).

- Kuo greičiau pamatuojam ir įvertinam (ir kuo dažniau), tuo mažiau mums kainuoja pakeitimai.
- Gerų matavimo metrikų pvz.:
 - Identifikuotų, realizuotų ar validuotų reikalavimų skaičius;
 - Identifikuotų, realizuotų ar validuotų vartojimo atvejų skaičius;
 - Atliktų veiklų skaičius.

Pagrindinė projektų žlugimo priežastis – riziku ignoravimas

- Kaip susidoroti su rizikomis?
 - Identifikavimas;
 - Analizė;
 - Planavimas;
 - Sekimas;
 - Kontrolė;
 - Komunikavimas.
- Riziku valdymo planas būtinas.

Rizikų valdymo planas (risk management plan)

- Planą sudaro pavojų aprašai, su metaduomenimis:
 - Pavojaus aprašas;
 - Pavojingumas;
 - Tikimybė;
 - Rizika;
 - Rizikos įvertinimo metrika;
 - Veiksmai, kilus incidentui:
 - Kokie veiksmai turi būti atlikti;
 - Kada jie turi būti atlikti;
 - Kas turi juos atlikti;
 - Kaip įvertinti veiksmų rezultatus ir efektyvumą;
 - Kas įvertins veiksmus.

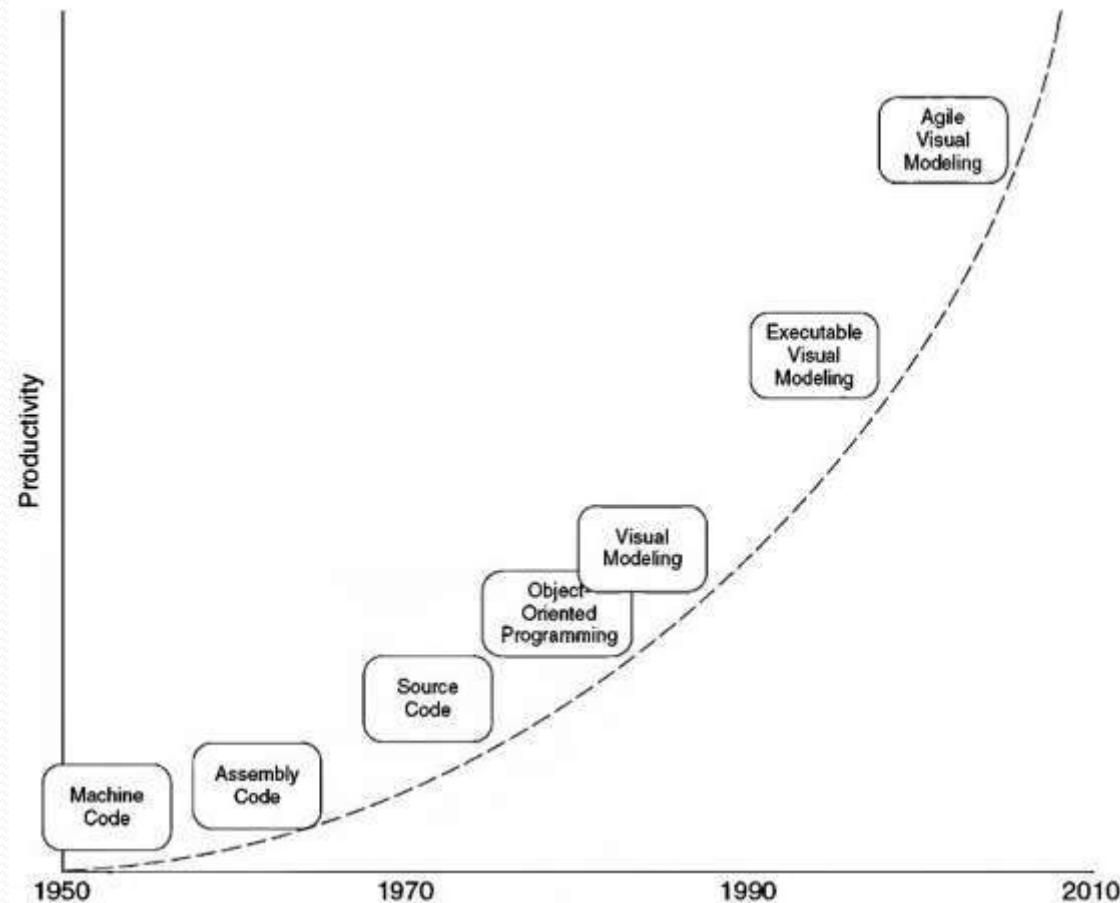
Kokybės užtikrinimas

- Vidinė kokybė – ją matuoja kūrėjai.
- Išorinė kokybė – ją matuoja užsakovai/vartotojai.
- Kūrėjai mąsto:
 - Kokybė = Korektiškumas;
 - Kokybė = Atsparumas;
 - Kokybė = Klaidų nebuvinimas;
- Vartotojai mąsto:
 - Kokybė = Atitikimas reikalavimams;
 - Kokybė = Atitikimas lūkesčiams;
 - Kokybė = Vartojamumas (usability);

Kokybės kriterijų pavyzdžiai

- Suprantamumas (understandability);
- Gera dokumentacija (pagalbos sistema);
- Pernešamumas;
- Palaikomumas;
- Testuojamumas;
- Patikimumas;
- Efektyvumas (resursų naudojimo prasme);
- Našumas;
- Pralaidumas;
- Saugumas.

Modeliavimo svarba



Kodo optimizavimas

- Negalima aukoti funkcionalumo vardan optimalumo .
- Negalima pulsi optimizuoti to, kas ne iki galio pabaigta.
- Lazda visada turi du galus: optimizujant vieną parametrum, gadiname kitą:

$$optimalDesign = \max \left[\sum DegreeOptimized_j \times Weight_j \right]$$



Ačiū už dėmesį. Klausimai?

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

**Penki architektūriniai Harmony
kūrimo proceso atvaizdavimo lygiai**

Prof. dr. Agnus Liutkevičius,

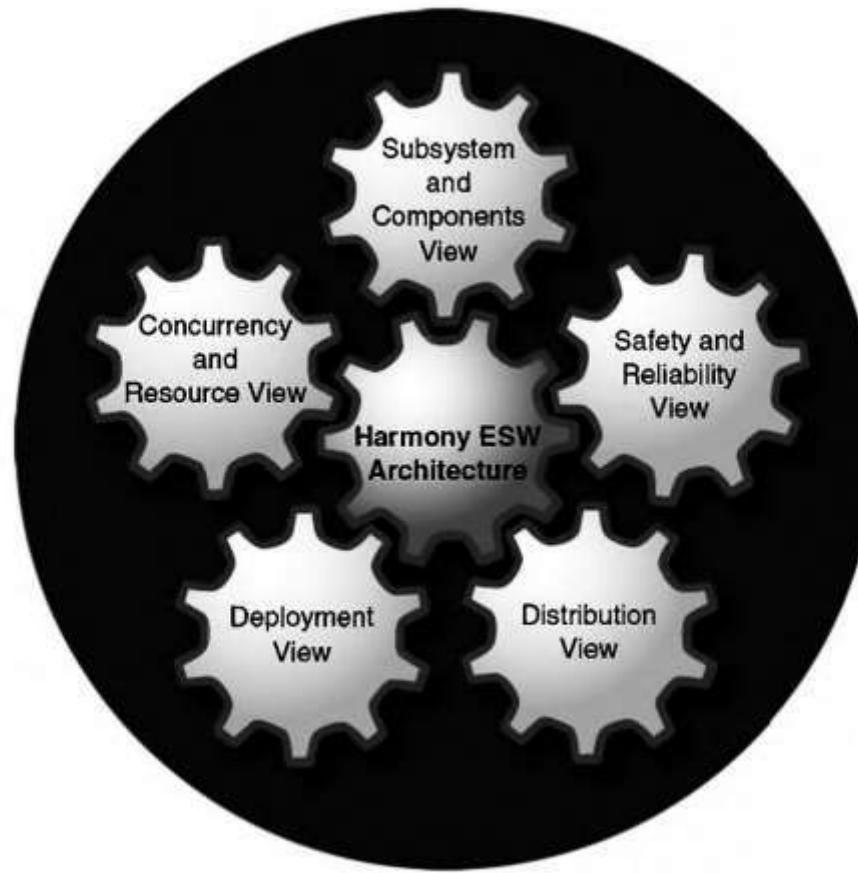
Kompiuterių katedra

Studentų g. 50-202, tel. 300394

Kas yra architektūra?

- Programinės įrangos architektūra apima sistemas **elementus/komponentus, sąryšius** tarp jų, ir tų komponentų ir sąryšių **savybes**.
 - Clements, Paul; Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford (2010). *Documenting Software Architectures: Views and Beyond, Second Edition*. Boston: Addison-Wesley. ISBN 0-321-55268-7

Harmony/ESW architektūros atvaizdavimo lygiai (1)



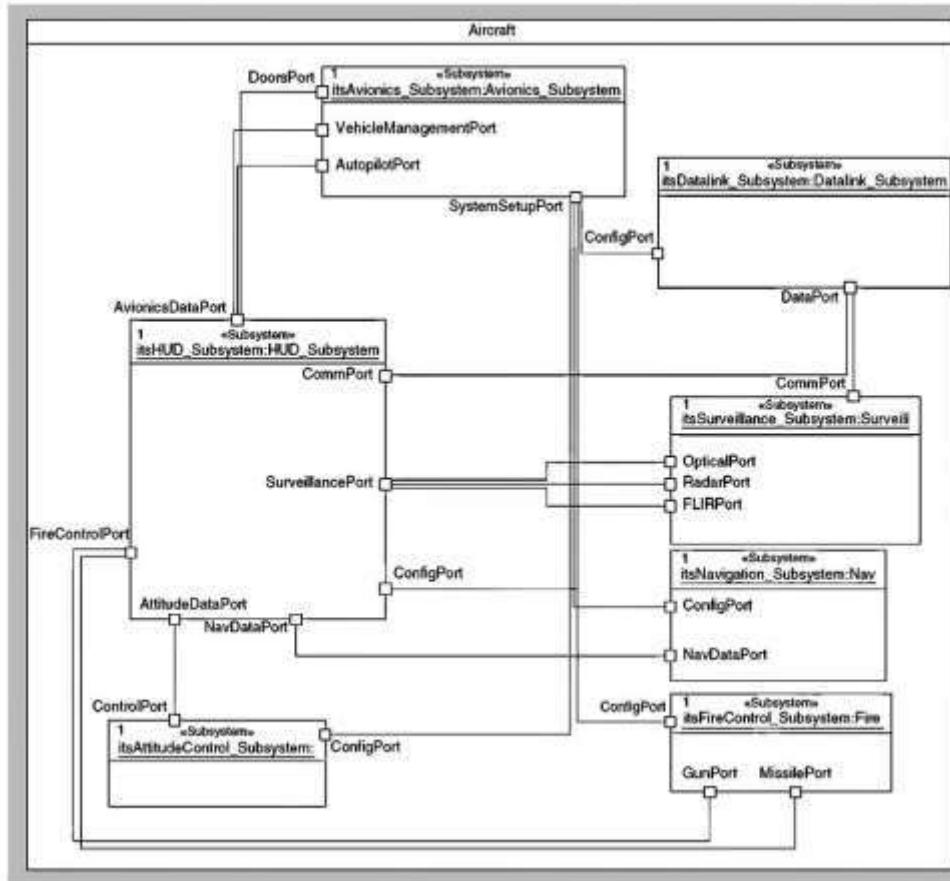
Harmony/ESW architektūros atvaizdavimo lygiai (2)

- Paprastai kiekvienam lygiui sukuriama viena ar kelios diagramos:
 - Posistemių diagrama;
 - Užduočių diagrama;
 - Diegimo ir paskirstymo diagrama;
 - ir t.t.
- Dažniausiai naudojama UML klasių ar diegimo diagramų notacijos.
- Iteratyviai kuriant sistemas, nebūtinai panaudojami visi atvaizdavimo lygiai.

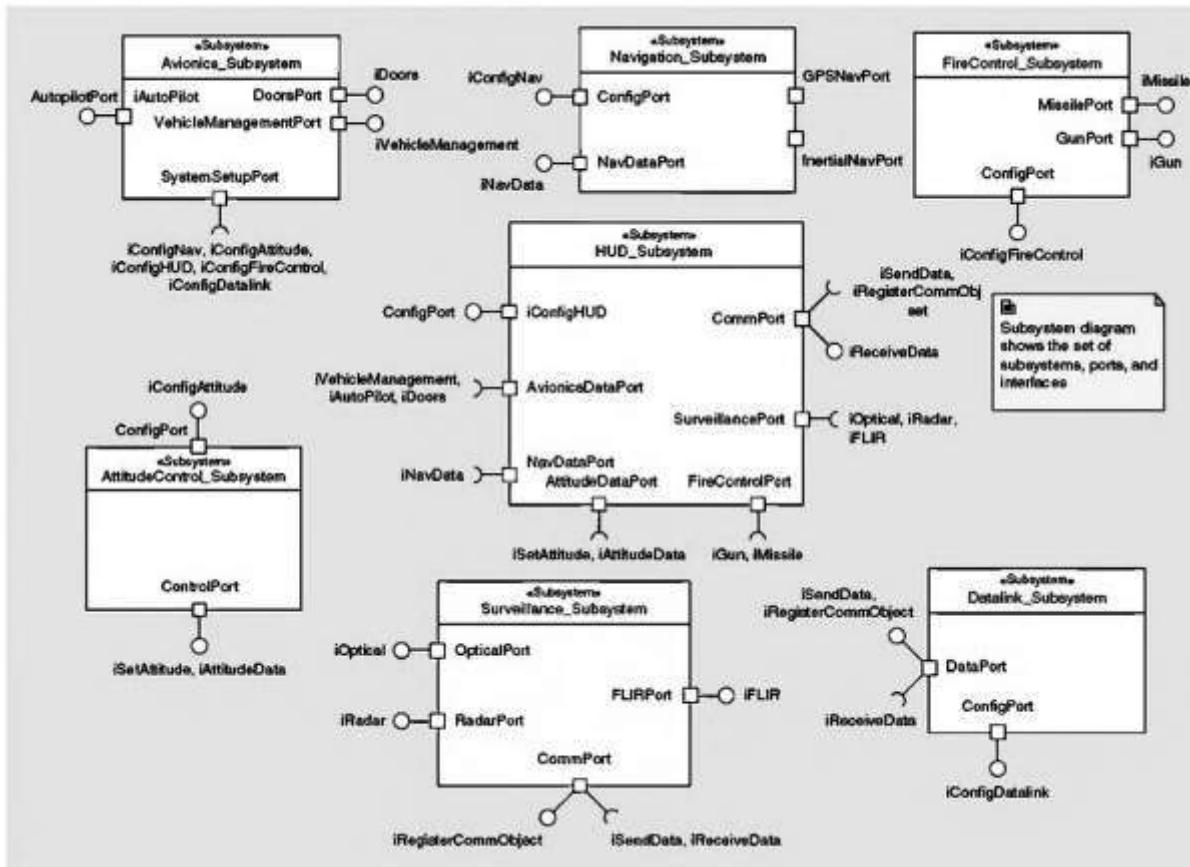
Posistemų ir komponentų architektūra

- Posistemė yra stambiausias architektūros komponentas.
- Posistemų ir komponentų architektūra apima:
 - Komponentų ir posistemų identifikavimą;
 - Atsakomybių šiems komponentams priskyrimą;
 - Programinių sąsajų (angl. interfaces) tarp komponentų specifikavimas:
 - Reikalingos sąsajos (angl. required);
 - Siūlomos sąsajos (angl. offered).

Posistemų diagramos pavyzdys



Posistemų su programinėmis sąsajomis diagramos pavyzdys



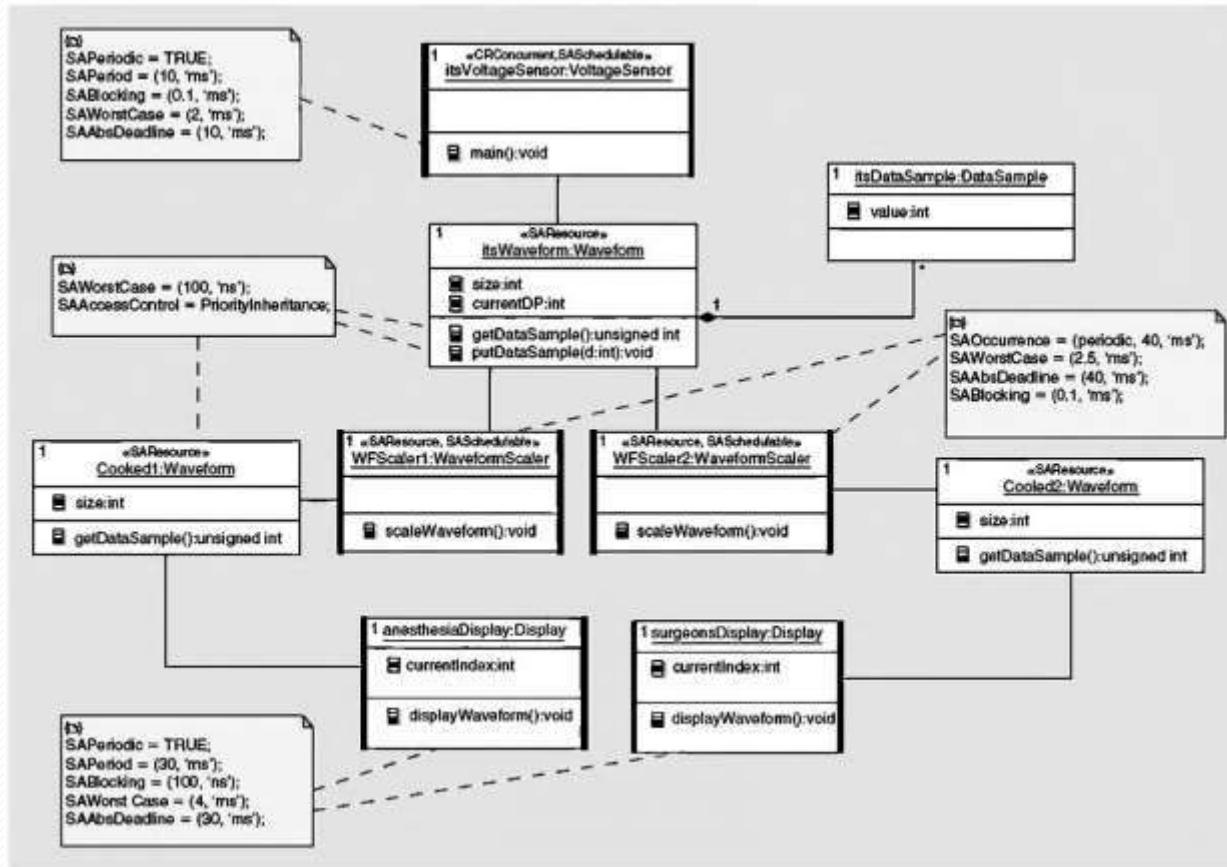
Lygiagrečiai veikiančių komponentų ir resursų valdymo architektūra (1)

- Lygiagretumo ir resursų valdymo architektūra turi milžinišką įtaką sistemos našumui.
- Ši architektūra apima:
 - Lygiagrečiai veikiančius/konkuruojančius komponentus;
 - Šių komponentų susiejimą su OS užduotimis ir gijomis;
 - Konkuruojančių komponentų meta-duomenų specifikavimą:
 - Prioritetas, Periodiškumas, Vykdymo laikas, Blokavimo laikas ir t.t.

Lygiagrečiai veikiančių komponentų ir resursų valdymo architektūra (2)

- Ši architektūra apima:
 - Užduočių planavimo mechanizmus (angl. specification of scheduling algorithms);
 - Resursų identifikavimą;
 - Resursų pasidalinimo principus ir šablonus.
- Architektūra dažnai vaizduojama viena ar keliomis „užduočių“ (klasių) diagramomis, kuriose yra:
 - Aktyvios klasės;
 - Našumo metaduomenys ir apribojimai;
 - Sinchronizacijos mechanizmai: semaforai, monitoriai, eilės ir t.t.

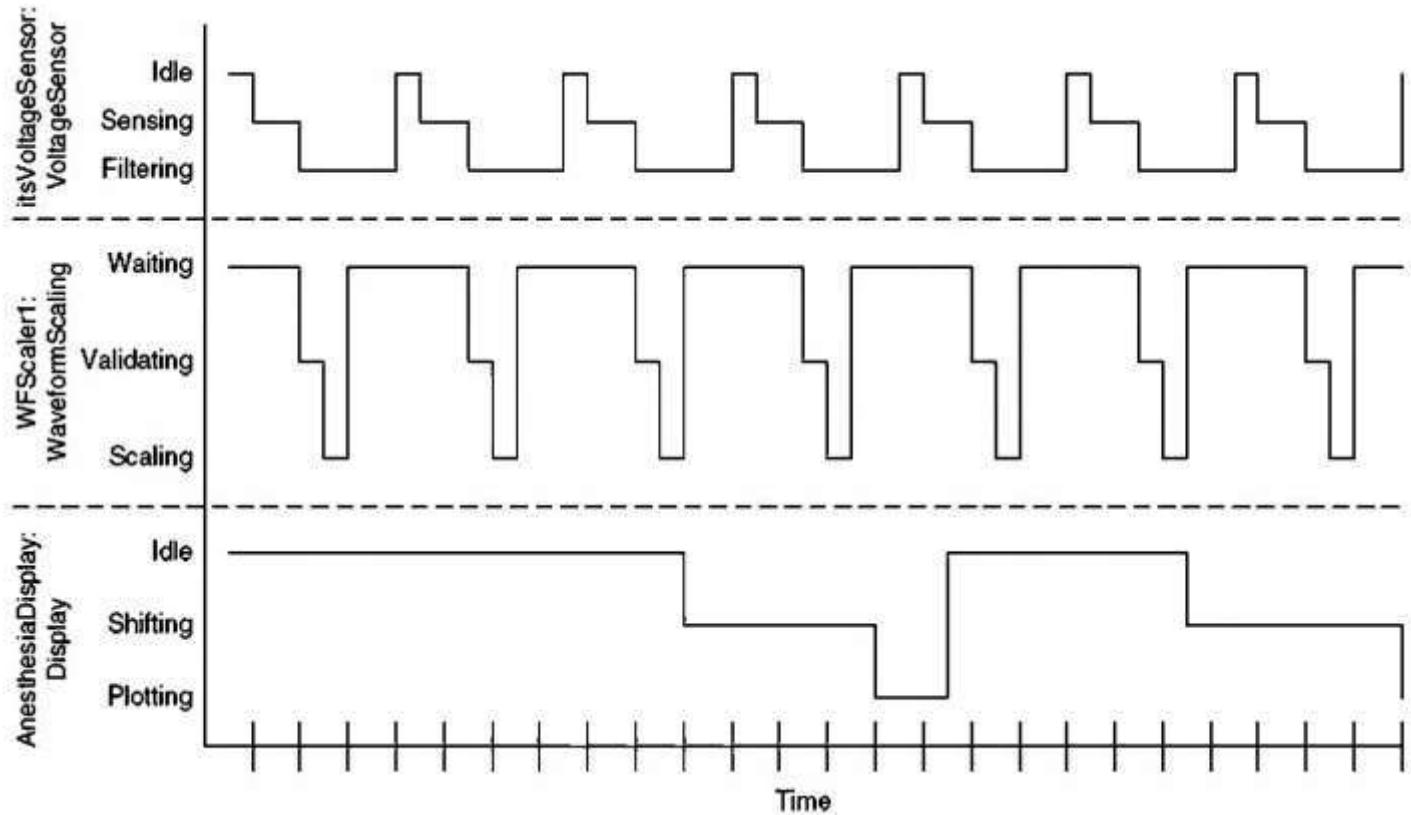
Lygiagrečiai veikiančių komponentų ir resursų valdymo diagramos pavyzdys



UML laiko (timing) diagramos

- Naudingos parodant pasikeitimus laike:
 - Būsenų;
 - Verčių;
- Naudingos palyginant lygiagrečiai veikiančių objektų būsenų pasikeitimus ir tarpusavio sąryšį.
 - Gali palaikyti ne visos UML priemonės, bet
 - Nesunku sukurti su tokiais paplitusiais redaktoriais, kaip MS EXCEL ☺

Laiko diagrammos pavyzdys



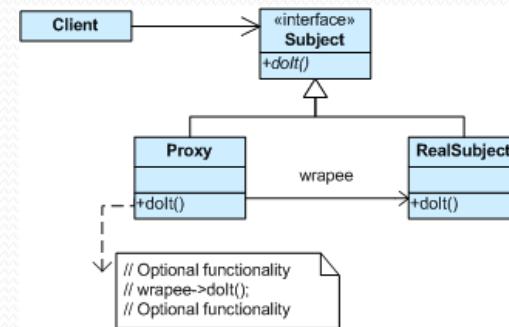
Paskirstymo architektūra (1)

- Paskirstymo architektūra aprašo:
 - Technologijas:
 - Tinklo įranga ir infrastruktūra; protokolai.
 - Metodus;
- kurių pagalba sistemos objektai esantys skirtinguose atminties regionuose ar net skirtinguose kompiuteriuose:
 - Randa vienas kitą;
 - Komunikuoja;
 - Bendradarbiauja.

Paskirstymo architektūra (2)

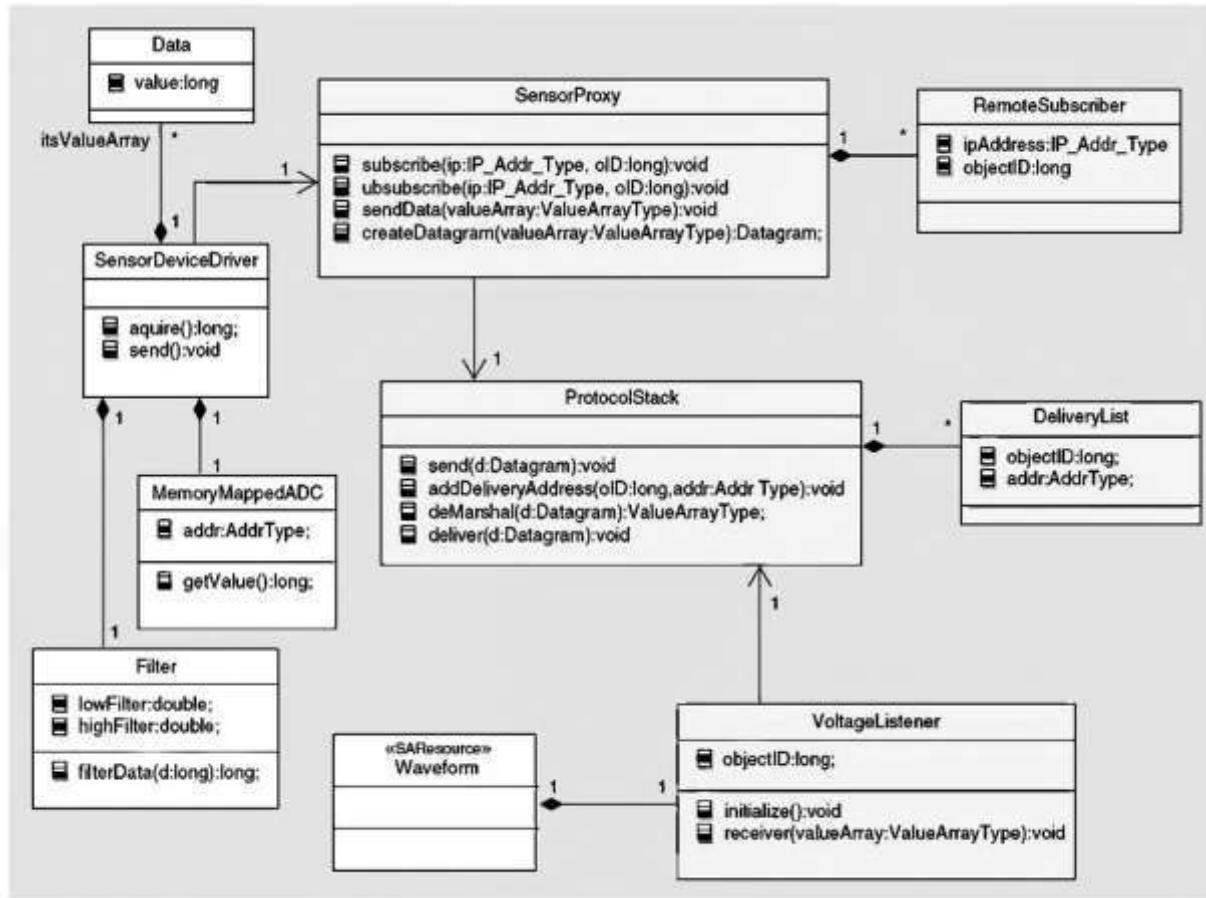
- Dažnai panaudojami įvairūs projektavimo šablonai:

- Proxy;
- Data Bus;
- Broker;
- Port proxy ir t.t.



- Architektūra vaizduojama viena ar keliomis klasių diagramomis, kuriose pateikta informacija apie:
 - Objektų transformacijas (angl. marshalling);
 - Duomenų perdavimą;
 - Įvykių pristatymą ir valdymą.

Paskirstymo diagramos pavyzdys



Saugumo ir patikimumo architektūra

- Saugumo ir patikimumo aspektai nėra aktualūs visoms sistemoms, išskyrus realaus laiko ir įterptines sistemas.
- Susijusi su klaidų ir kritinių situacijų (incidentų), kylančių vykdymo metu:
 - Identifikavimu;
 - Izoliavimu;
 - Korekcija.
- Architektūra dažniausiai vaizduojama:
 - Klasių diagramomis, parodant perteklinius elementus;
 - Elgsenos/sąveikos diagramomis, parodant kaip reaguojama į incidentus.

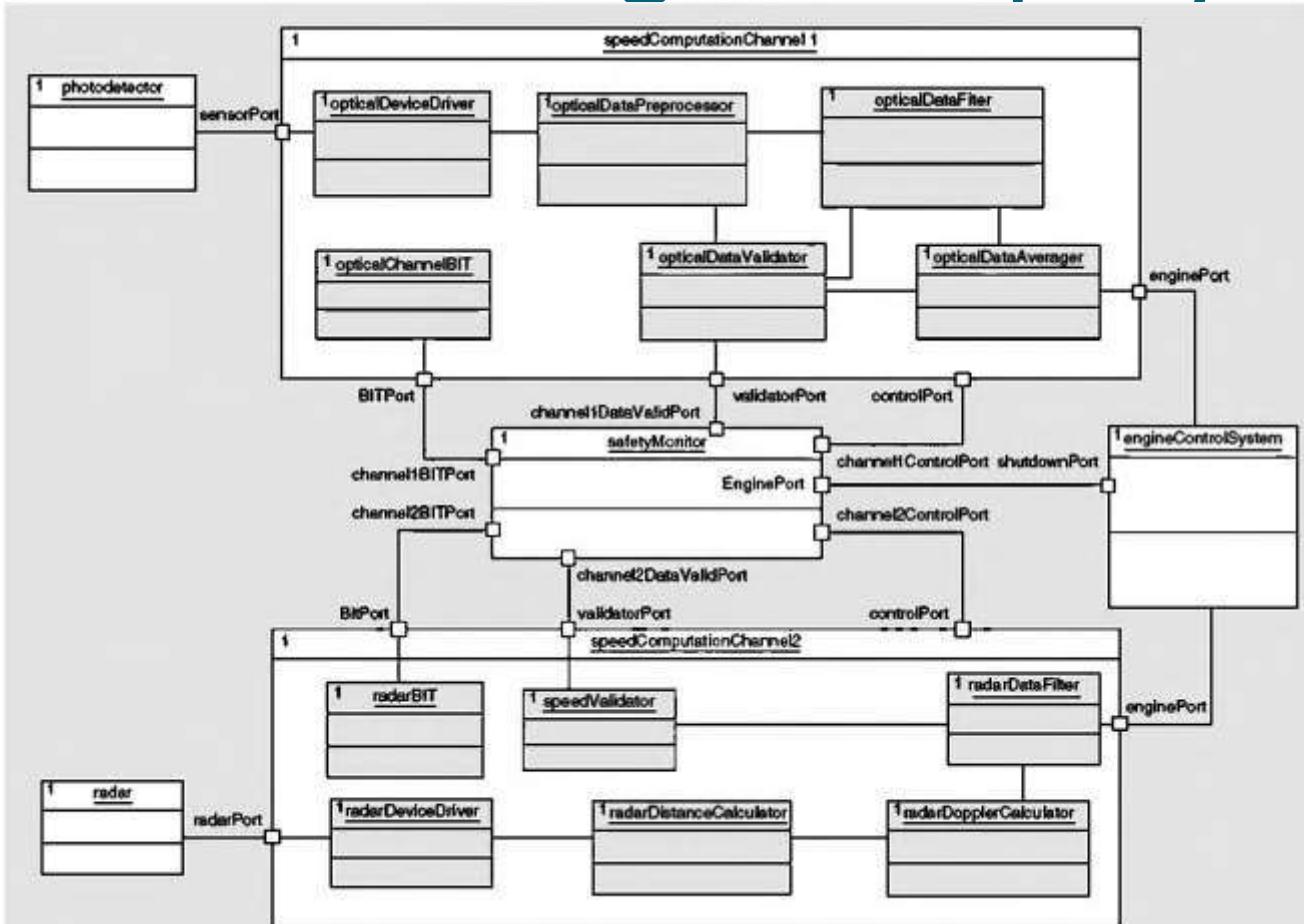
Incidentas

- Incidentas – tai įvykis, susijęs su žala sistemos vartotojams, komponentams ar finansinei būklei.
- Pavojus – tai būsena, kuri gali privedti prie incidento.
- Incidento tikimybė – tai įvertis, nusakantis kaip dažnai incidentas gali kilti, esant tam tikram pavojui.
 - pvz. 0,3% tikimybė reiškia, kad iš 1000 programos vykdymų tikėtina, kad 3 kartus įvyks incidentas.
- Rizika, susijusi su incidentu, tiesiogiai priklauso nuo incidento tikimybės ir jo pavojingumo
 - rizika = tikimybė x pavojingumas

Patikimumas

- Patikimumas susijęs su paslaugų teikimu.
- Dažnai įvertinamas kaip MTBF (angl. mean time between failure) – tai nuspėjamas laikas tarp sistemos sutrikimų.
 - MTBF galima paskaičiuoti išvedus aritmetinį laiko vidurkį tarp sistemos sutrikimų.
- Patikimumas dažnai prieštarauja saugumui, jeigu sistema turi specialią nuo incidentų apsaugančią būseną (angl. fail-safe state).
- Nesant „fail-safe state“, patikumo didinimas salygoja ir saugumo didinimą.

Saugumo ir patikimumo architektūros diagramos pavyzdys

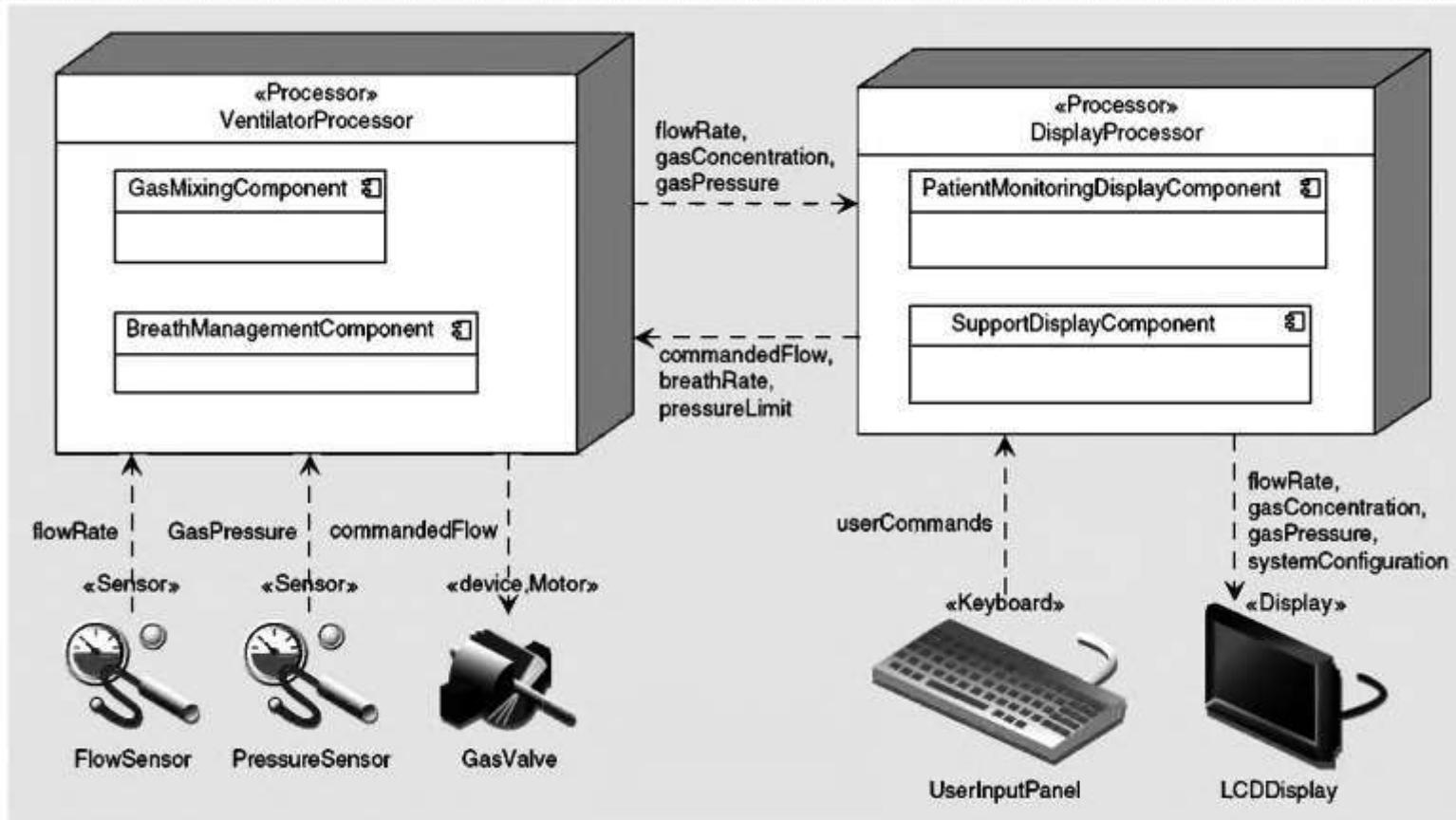


Šaltinis: Bruce Powel Douglass. Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development. Addison Wesley, 2009, 560 p. ISBN-10: 0321545494

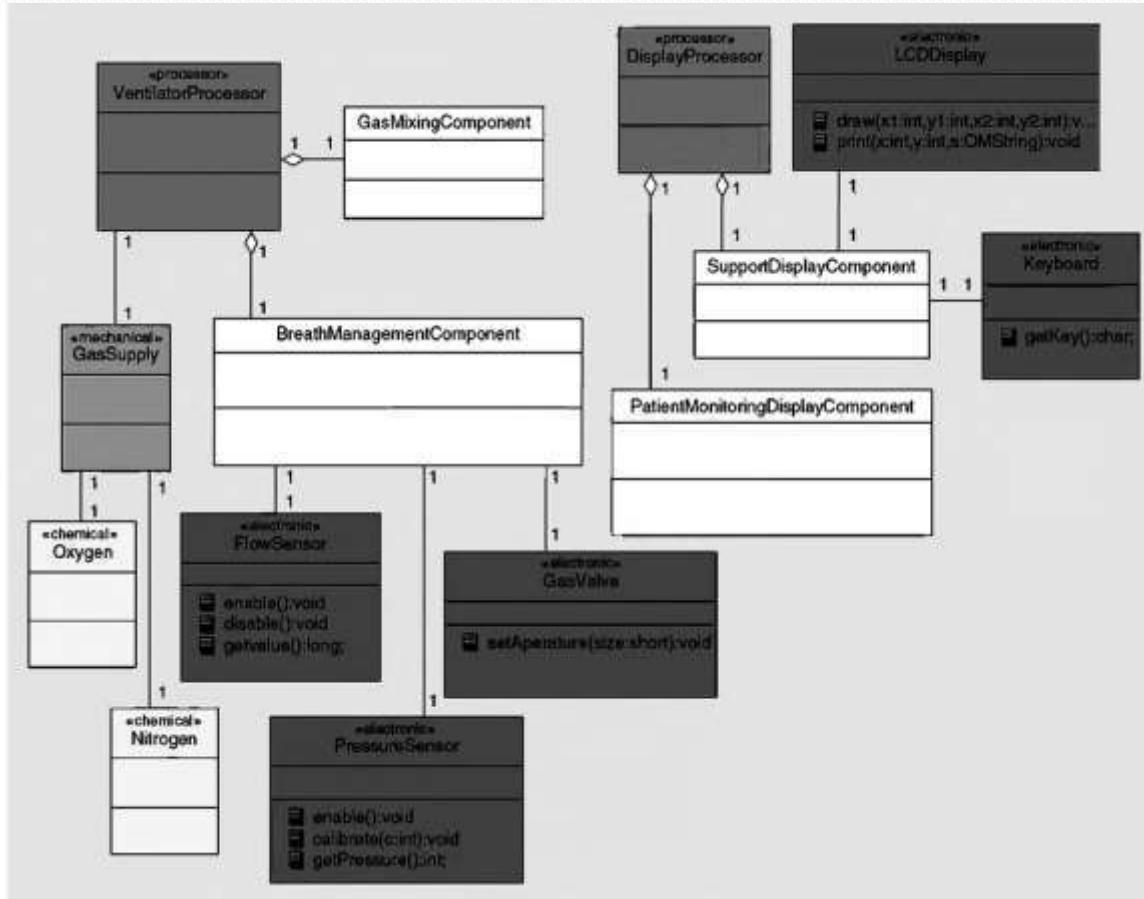
Diegimo architektūra

- Susijusi su paskirstymo architektūra, bet turi skirtingą paskirtį:
 - Paskirstymo architektūra labiau koncentruojasi į komponentų komunikavimo technologijas ir metodus;
 - Diegimo architektūra labiau skirta parodyti, kaip kuriama programinė įranga diegama į techninę įrangą.
- Architektūrai pavaizduoti dažnai naudojama UML diegimo diagrama arba SysML blokų diagrama, tačiau
 - Galima naudoti ir klasių diagramas, kurios leidžia detaliau aprašyti architektūros elemetus.

Diegimo architektūros diagramos pavyzdys: UML diegimo diagrama



Diegimo architektūros diagramos pavyzdys: UML klasų diagrama



Kiti architektūros atvaizdavimo lygiai

- Be aukščiau išvardintų 5 svarbiausių, egzistuoja ir keletas kitų architektūrinių aspektų:
 - Informacijos patikimumas ir apsauga;
 - Duomenų valdymas;
 - QoS valdymas;
 - Klaidų ir išskirtinių situacijų valdymas;
 - SOA principų panaudojimas.



Ačiū už dėmesį. Klausimai?

Kompiuterinių sistemų analizė ir projektavimas (T120B205)

Harmony proceso detalizacija

Prof. dr. Agnus Liutkevičius,

Kompiuterių katedra

Studentų g. 50-202, tel. 300394

Ko reikia, kad būtų galima pradėti iteraciją/mikrociklą?

- Tvarkaraštis;
 - Komanda;
 - Rizikų valdymo planas;
 - Vartotojo reikalavimų panaudos atvejai;
 - Šiam mikrociklui atrinkti panaudos atvejai;
 - Modelio struktūra.
-
- Papildomai gali būti:
 - Pakartotinio panaudojimo planas;
 - Saugumo ir patikimumo analizė.

Mikrociklo misijos aprašymas

- Kokius vartojimo atvejus realizuosim.
- Darbo užduočių sąrašas.
 - Rizikų valdymo veiklos;
 - Defektų šalinimo veiklos;
 - Pakeitimų veiklos;
- Platformos, kurias palaikysim.
- Architektūrinis modelis, kurį naudosim (iš galimų 5: posistemų, lygiagretumo, paskirstymo, saugumo, diegimo).
- Išoriniai elementai, reikalingi mikrociklui.

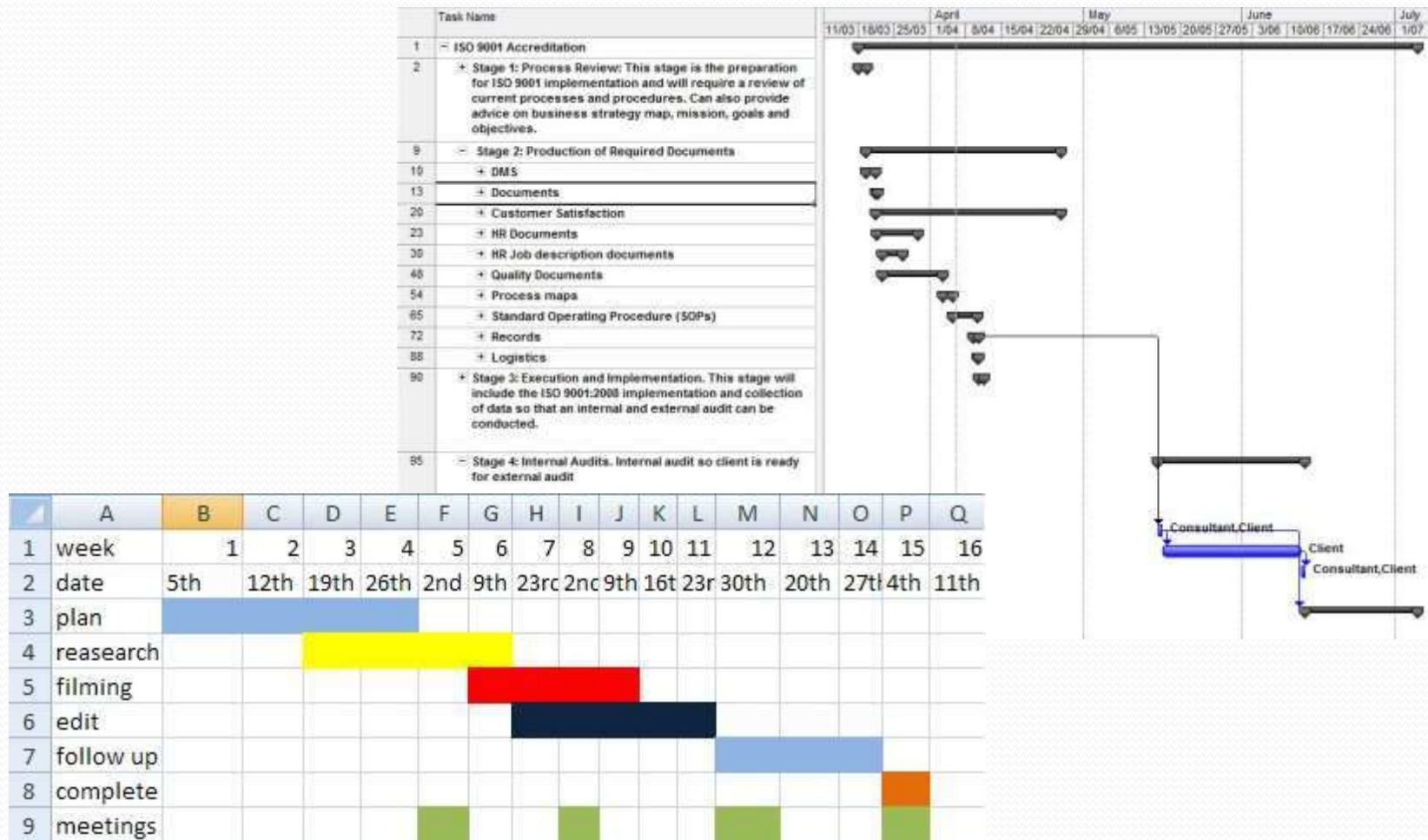
Vartojimo atvejų pasirinkimo kriterijai

- Rizikingumas;
- Infrastruktūra;
- Resursų prieinamumas;
- Kritiškumas (svarba vartotojui);
- Skubumas;
- Paprastumas.

Pradinis planavimas: tvarkaraščio sudarymas

- Tvarkaraštis yra vienas svarbių dokumentų, kuris apibrėžia:
 - Užduočių sekas ir priklausomybes;
 - Užduočių laiką ir trukmę;
 - Resursų (žmonių, įrangos) paskirstymą ir priskyrimą užduotims;
 - Užduočių rezultatus – darbo produktus.
- Priemonės: MS Project ir pan., bet puikiai tinka ir Excel 😊
- Diagramos: Ghant Chart ir pan.

Projekto plano formatų pvz.



Dažnos planavimo klaidos

- Bandymas planuoti minučių tikslumu
 - Tvarkaraščio sudarymo programos gali jums atlikti tokius skaičiavimus, bet ar tai bus tiesa, sakykim 5 metų trukmės projektui? ☺
- Manymas, kad užduočių vykdymo sparta tiesiogiai priklauso nuo vykdytojų skaičiaus
 - Teoriškai, jeigu suplanuojam projektą vienam žmogui, kurio trukmė 1 metai, tai pasamdžius 16000 žmonių ji galima atlikti per 1 minutę. Bet ar tikrai? ☺

Tvarkaraščio sudarymo žingsniai

- Norimo funkcionalumo identifikavimas;
- Rizikų identifikavimas ir analizė;
- Mikrociklų ir jų rezultatų (prototipų) planavimas;
- Sudaryto tvarkaraščio įvertinimas;
- Tvarkaraščio tobulinimas.

Bruce's įvertinimo ir peržiūros metodas (Bruce's Evaluation and Review Technique) – BERT

- Tvarkaraštį sudaro darbo užduočių – darbo vienetų (work item) – sekos.
- Darbo vieneto trukmė – 1-2 dienos.

$$E_{used} = \frac{E_{20\%} + 4E_{50\%} + E_{80\%}}{6} E_c$$

- $E_{20\%}$ - optimistinis vertinimas;
 - $E_{50\%}$ - vidutinis vertinimas;
 - $E_{80\%}$ - pesimistinis vertinimas;
 - E_c – paklaida (iš patirties).
- Idėja remiasi tiesiog Gauso pasiskirstymo dėsniu...

O kam išvis reikia tvarkaraščio?

- Įvertinti:
 - Kainą;
 - Resursus, pastangas;
 - Laiką.
- Parodo ar apskritai tikslinga vykdyti projektą.
- Parodo užsakovui, kada jis gaus naudą.
 - Gera taktika – naudoti pesimistinius planus, nes užsakovas visada gaus tai ko tikisi ar net daugiau.
- Motyvuoja darbuotojus ☺ (turi būti suspaustas, verčiantis pasitempti)

Darbo užduočių sąrašas

Name/ Description	Priority	Size Estimate (points or hours)	State	Target Microcycle	Assignee	Effort Remaining (hours)	Effort Applied (hours)
Background wrong color for all dialog boxes	10	2 hr	Submitted	3	Sam	2	
Bursty interrupts cause system to hang	3	4 hr	In process	2	Bruce	1	3
Change request to improve communication throughput to 20K data/sec	—	—	Rejected as infeasible	—	—	—	—
Evaluate Highlander CORBA ORB on target	4	10 hr	Complete	1	Susan	0	12

Projekto komandos sudarymas

- Dydis – nuo 5 iki 10 žmonių.
- Idealiu atveju viena komanda dirba prie 1 modelio.
- Vadovas privalo žinoti komandos narių savybes ir skirti jiem tokį darbą, kurį jie sugeba geriausiai.
- Realybė:
 - Dažna klaida, kurią daro nepatyrę kūrėjai (ir žinoma studentai), tai komandos sudarymas ne pagal patirtį ar gebėjimus, o pagal „draugystę“.
 - Komandą turėtų sudaryti specialistai, kurie turi skirtingų sričių gebėjimų (SW, HW, programavimo, projektavimo, testavimo), tam, kad galėtų savarankiškai atlikti visas užduotis.
 - Jokios naudos iš komandos, kurią sudaro, pvz. vien SW kūrėjai, ypač, kai reikia sukurti kokią nors įterptinę ar realaus laiko sistemą su HW elementais.

Pakartotinio panaudojimo planavimas

- Komponentų pakartotinio panaudojimo nauda:
 - Mažesnė kaina;
 - Trumpesnis patekimas į rinką (angl. time-to-market);
 - Mažesni resursai;
- Trūkumai:
 - Pakartotinai panaudojamus komponentus visų pirma reikia sukurti, o tai kainuoja papildomą laiką ir pinigus.
 - Neverta naudoti, jeigu nėra ilgalaikės strategijos ir aiškios komponentų panaudojimo vizijos.

Pakartotinio panaudojimo (PP) plano sudarymas

- Identifikuojami PP poreikiai ir tikslai;
- Identifikuojamos PP galimybės;
- Įvertinama PP komponentų kūrimo kaina;
- Nustatoma, kuriuos PP komponentus reikia kurti;
- Įvertinamas poveikis projekto tvarkaraščiui;
- Specifikuojama, kaip PP komponentai bus valdomi ir naudojami;
- Specifikuojama, kaip egzistuojantys PP komponentai bus panaudoti projekte, ir kaip sukurti PP komponentai bus panaudoti ateityje;
- Sudaromas PP planas;
- Atnaujinamas projekto tvarkaraštis.

Ką galima pakartotinai panaudoti?

- Taikomąsias programas;
- Karkasus;
- Profilius (meta-duomenis, konfigūracijas);
- Taikomųjų programų komponentus;
- Posistemės (HW+SW);
- Bibliotekas;
- **Modelius.**

Projekto rizikų mažinimas: pavojų tipų p.vz.

- Pasirinkta tarpinė programinė įranga netenkina nefunkcinių reikalavimų;
- Personalo praradimai ir rotacija;
- Finansavimo praradimas;
- Naudojamos kūrimo priemonės ar bibliotekos turi klaidų (ypač open-source!)
- Nepažįstamos kūrimo/modeliavimo kalbos;
- Neefektyvios kūrimo aplinkos;
- Nepakankama komandos motyvacija;
- Nepakankamas valdymas ir kontrolė;
- Nuolat besikeičiantys užsakovo reikalavimai;
- Neteisingas pradinis planas/tvarkaraštis;
- Problemos su įrangos ir komponentų tiekėjais;
- Partneriai ar subrangovai nesugeba įvykdyti įsipareigojimų;
- Duomenų ir sukurtos įrangos praradimas (gaisras, vanduo, vagystė, ataka ir t.t.)

Kūrimo aplinkos sudarymas ir konfigūravimas

- Priemonės (tools);
- Infrastruktūra (darbo stotys, serveriai);
- Konfigūracijos valdymas (prisimenam versijų kontrolės sistemas – CVS);
- Programinės įrangos kūrimo plano sudarymas (software development plan – SDP) ir patalpinimas viešai visiems prieinamoje vietoje, pvz. į specialų projekto valdymo puslapį.

Analizē viso mikrociklo kontekste

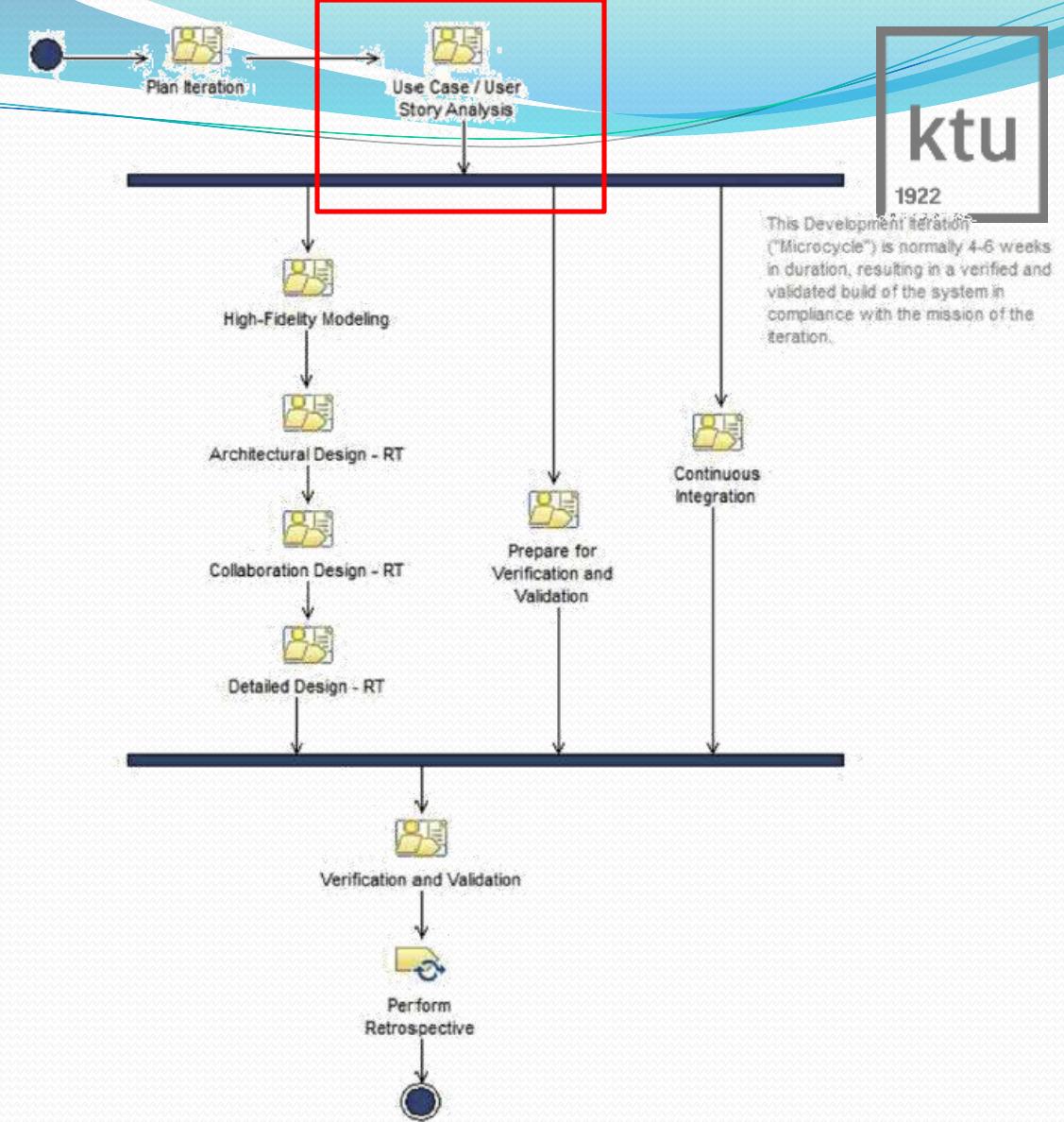
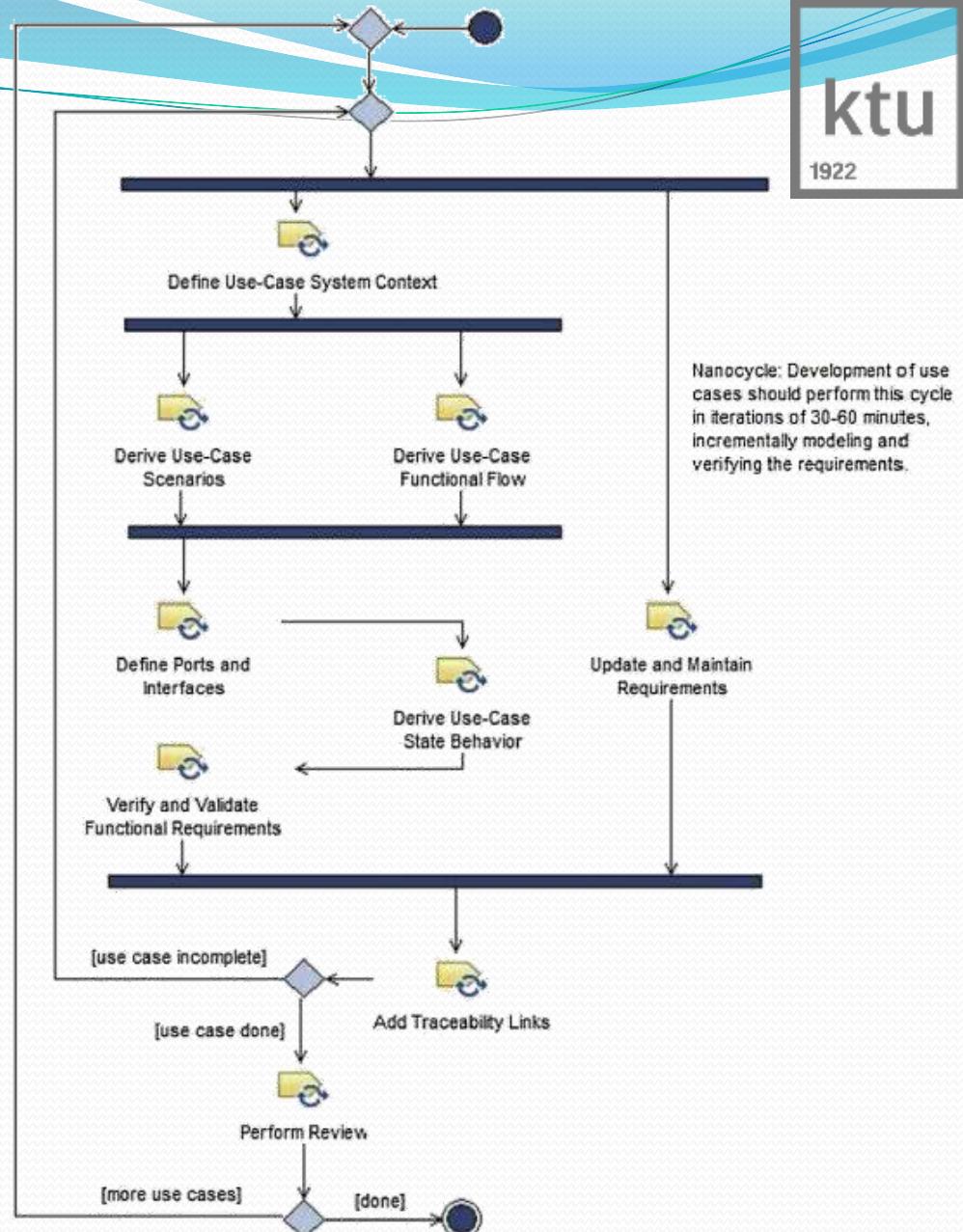


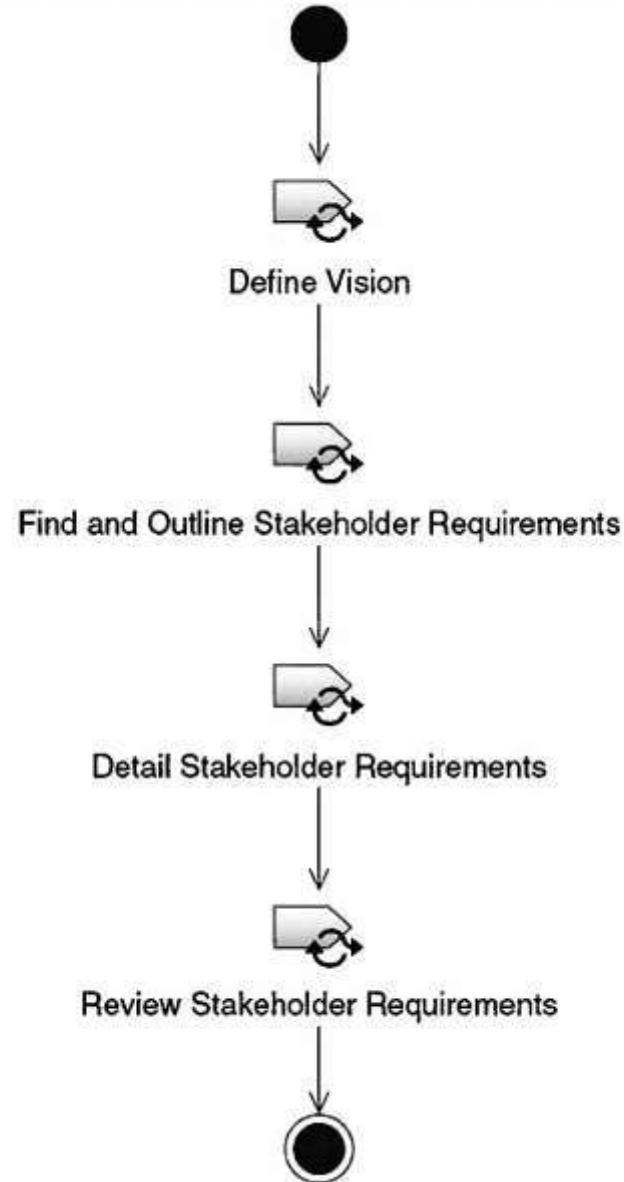
Figure 2.5
Harmony development iteration overview.

Use Case/User Story Analysis Workflow



Užsakovo/vartotojo reikalavimų sudarymas

- Du dokumentai:
 - Vizija;
 - Reikalavimai.



Šaltinis: Bruce Powel Douglass. Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development. Addison Wesley, 2009, 560 p. ISBN-10: 0321545494

Prisimenam, kas yra vizija

- Aprašo **sistemos kontekstą** (todėl kartais vadinama konteksto diagrama, tiesiog kontekstu):
 - Vartotojai;
 - Vartotojų lūkesčiai;
 - Aplinka;
 - Sistemos dalys;
 - Pagrindinės funkcijos (nemaišyti su lūkesčiais – vartotojas tikisi naudos, o ne funkcijų ☺)

Reikalavimai

- IEEE apibrėžia „reikalavimą“ kaip:
 - Sąlyga (condition) ar gebėjimas (capability), kuris reikalingas vartotojui tam, kad išspręsti problemą ar pasiekti tikslą.
 - Sąlyga ar gebėjimas, kurį privalo užtikrinti sistema ar jos dalis taip, kad būtų patenkinami kontrakto, standarto, specifikacijos, ar kitokių formalių dokumentų teiginiai.
 - Dokumentas, apibrėžiantis sąlygas ir gebėjimus.

Reikalavimų apibrėžimas naudojant UML

- UML **does not have** an explicit **concept** called a “**requirement**.”
- Even though
 - use cases,
 - sequence diagrams,
 - state machines,
 - activity diagrams,
 - and constraints of various kinds
- are used to model the properties of requirements, the requirement concept itself is lacking.
- *A requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform (functional) or a performance condition a system must achieve (non-functional).*

Vartojimo atvejai

- Funkciniai reikalavimai yra aprašomi vartojimo atvejais.
- Vartojimo atvejis (use case) – tai sistemos naudojimas su tam tikru tikslu.
 - Vartojimo atvejai turi būti **veiksmažodinės formos**, aiškiai nusakyti, kas bus atliekama.

Vartotojų reikalavimų sudarymas

- Probleminės srities terminų identifikavimas;
- Dokumento struktūros sudarymas (**Software Requirements Specification – SRS**);
- Reikalavimų išgavimas:
 - Anketos;
 - Pokalbiai;
 - kt.
- Kiekvieno vartojimo atvejo detalus aprašymas;
- Liktinių (legacy) sistemų ar aktorių aprašymas;
- Vartotojų rolių (aktorių) aprašymas.

Kada vartojimo atvejis geras?

- Aiškiai suformuluotas – nereikia spėlioti, kas po juo slepiasi.
- Paprastas/nesudėtingas – apima vieną aiškų vartotojo tikslą.
- Testuojamas – galima įvertinti, ar atvejis realizuotas ar ne, ir ar gerai realizuotas.

Reikalavimų susiejimas su vartojimo atvejais

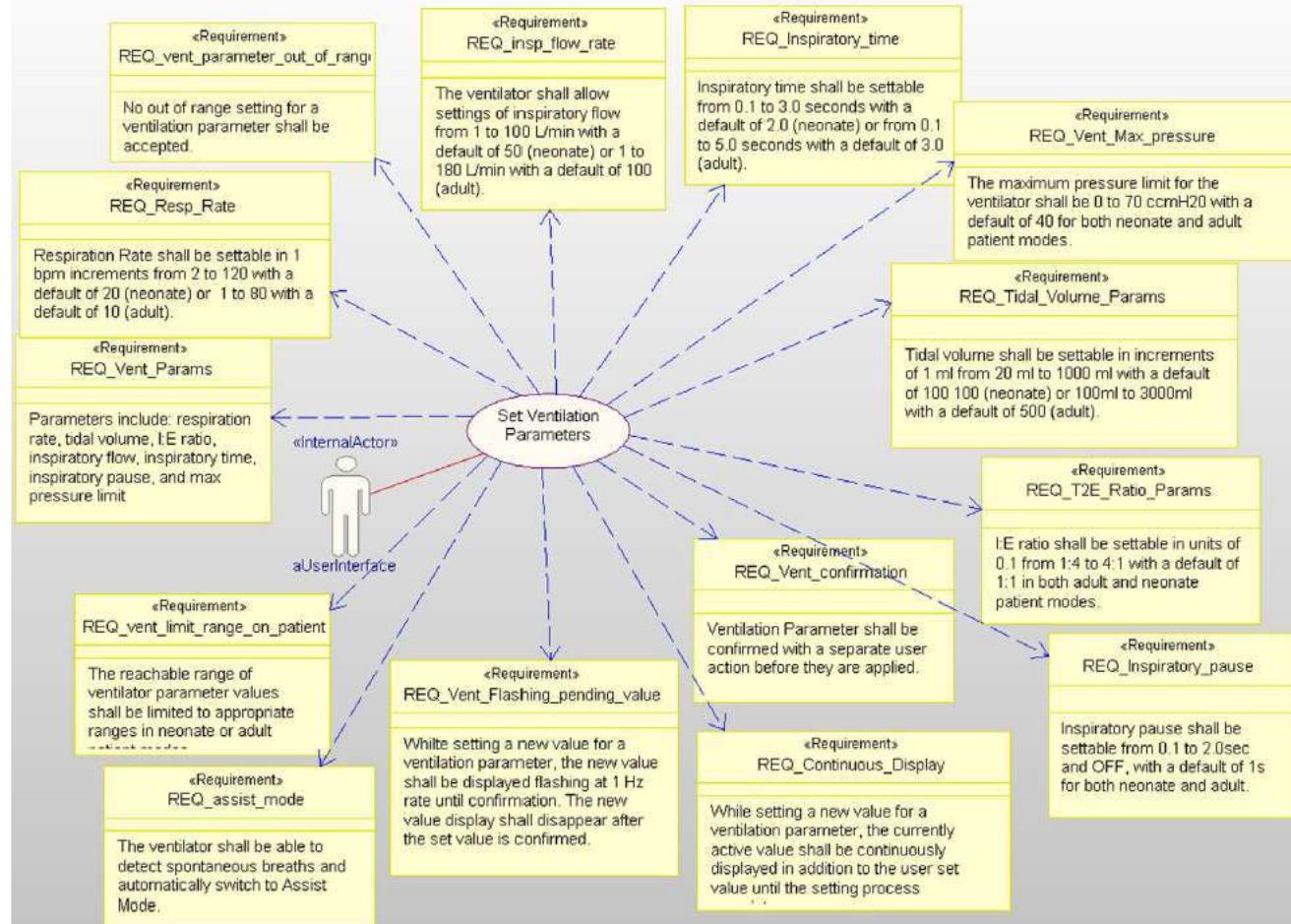


Figure 4.1
Requirements linked to use case.

„Gero“ Use Case požymiai

- It returns a **value** to at least one actor.
- It contains at least three **scenarios**, each scenario consisting of multiple actor-system messages.
- It logically contains at least a few, but up to very many **operational requirements**.
- It does **not reveal** or imply anything about the internal **structure** of the system implementation.
- It is **independent** of other use cases and may (or may not) be concurrent with them.
- There should be no fewer than three and no more than three dozen use cases at the “high level.”
- “Large” use cases **may be decomposed** with «includes» and «extends» relations into smaller “part” use cases.
- Use cases may be specialized with the generalization (“is-a-specialized-kind-of”) relation.

Nefunkciniai reikalavimai

- Use cases are coherent sets of **functional** requirements.
- Good use cases are named with strong verbs.
- If use cases and their contained requirements are verbs, then Quality of Service (QoS) requirements are adverbs;
 - they modify the use case or the specific requirements within those use cases.
 - They specify “how much,” “how fast,” and “to what degree.”
- Such requirements are historically known as **nonfunctional** requirements, but these days they are more commonly known as QoS requirements.

Vartotojo sasajos specifikavimas

- Vartotojo veiksmų analizė.
- Sasajos prototipo sudarymas.
 - Gali būti ir išmetamas.
- Sasajos specifikacijos parašymas:
 - Vartotojo veiksmai, vartojimo scenarijai;
 - Langų maketai su įvairiais elementais;
 - Apribojimai.
- Gerai tinkamai sekantys, veiklos ar būsenų diagramos.

Vartojimo atvejų detalizacija

USE CASE <i><Use case code></i>	<i><Use case name></i>		Date: <i><date of the last change ></i>	<i><use case version></i>
Description:	<i><use case brief description></i>			
User priority:	<i><development priority requested set by users></i>			
Performance:	<i><requested performance></i>			
Primary actor:	<i><primary actor name></i> <i><primary actor interest in this service></i>			
Secondary actor:	<i><secondary actor name></i> <i><secondary actor interest in this service></i>			
Preconditions:	<i><description of the conditions which must be fulfilled before the use case can be executed></i>			
Post-condition	on success:	<i><description of the conditions that describe the state of a system after that this use case is successfully completed></i>		
	on failure:	<i><description of the conditions that describe the state of a system after that this use case is unsuccessfully completed></i>		
Trigger:	<i><description of the event that fires the use case></i>			
MAIN SCENARIO				
1.	<i><actor/ system></i>	<i><action description></i>		
2.	<i><actor/ system></i>	<i><action description></i>		
...	<i><actor/ system></i>	<i><action description></i>		
<i>n.</i>	System:	Terminates use case successfully.		
ALTERNATIVE SCENARIO: <i><condition which leads to this alternative scenario></i>				
<i>h.1.</i>	<i><actor/ system></i>	<i><action description></i>		
		<i><action description></i>		
<i>h.n.</i>	System:	Resumes use case at point x.y.		
EXCEPTION SCENARIO: <i><condition which leads to this exception scenario></i>				
<i>i.1.</i>	<i><actor/ system></i>	<i><action description></i>		
		<i><action description></i>		
<i>i.m.</i>	System:	Terminates use case unsuccessfully.		
ANNOTATION				
		<i><extra information></i>		

Vartojimo atvejų detalizavimas – functional analysis

- Vartotojo/užsakovo pirminiai reikalavimai -> sistemos reikalavimai.
- Dar vis „juodos dėžės“ principas.
- Etapai:
 - Pirminių scenarijų identifikavimas;
 - Veiklos diagramų modeliai;
 - Antriniai scenarijai ir išskirtinės situacijos (exceptions);
 - Validavimas ir peržiūra su vartotojais/užsakovais;
 - Būsenų diagramos sudarymas;
 - Susiejimas tarp vartotojo reikalavimų ir sistemos vartojimo atvejų.

Vartotojo vs Sistemos reikalavimai

- Vartotojo – nusako, kokie vartotojo poreikiai ir tikslai;
- Sistemos – pasako, kaip sistema užtikrins vartotojų poreikius.
- Pvz. medicininis ventiliatorius:
 - [vartotojo ir sistemos]: vartotojas gali nustatyti aparatūros veikimo režimą į „pagal spaudimą“, „pagal tūri“, „ventiliavimas“;
 - [vartotojo]: vartotojas privalo turėti galimybę nustatyti ventiliacijos tūri įvairiomis amžiaus grupėms;
 - [sistemos]: „pagal tūri“ režime, vartotojas gali nustatyti ventiliacijos tūri nuo 100ml iki 1000ml su 1ml žingsniu;
 - [vartotojo]: sistema turi tiekti orą dideliu tikslumu;
 - [sistemos]: „pagal tūri“ režime, sistema turi užtikrinti +/-2ml tikslumą.

Kas yra scenarijus?

- A scenario can be thought of as a specific sequence of inputs and outputs that represents a single path through a use case.
- A use case normally has many different scenarios – several dozen is not uncommon – that show combinations of different inputs and different sequences.
- Scenarios are useful because they relate the system capabilities with the expectations of how the system will interact with other elements in its environment (i.e., actors).

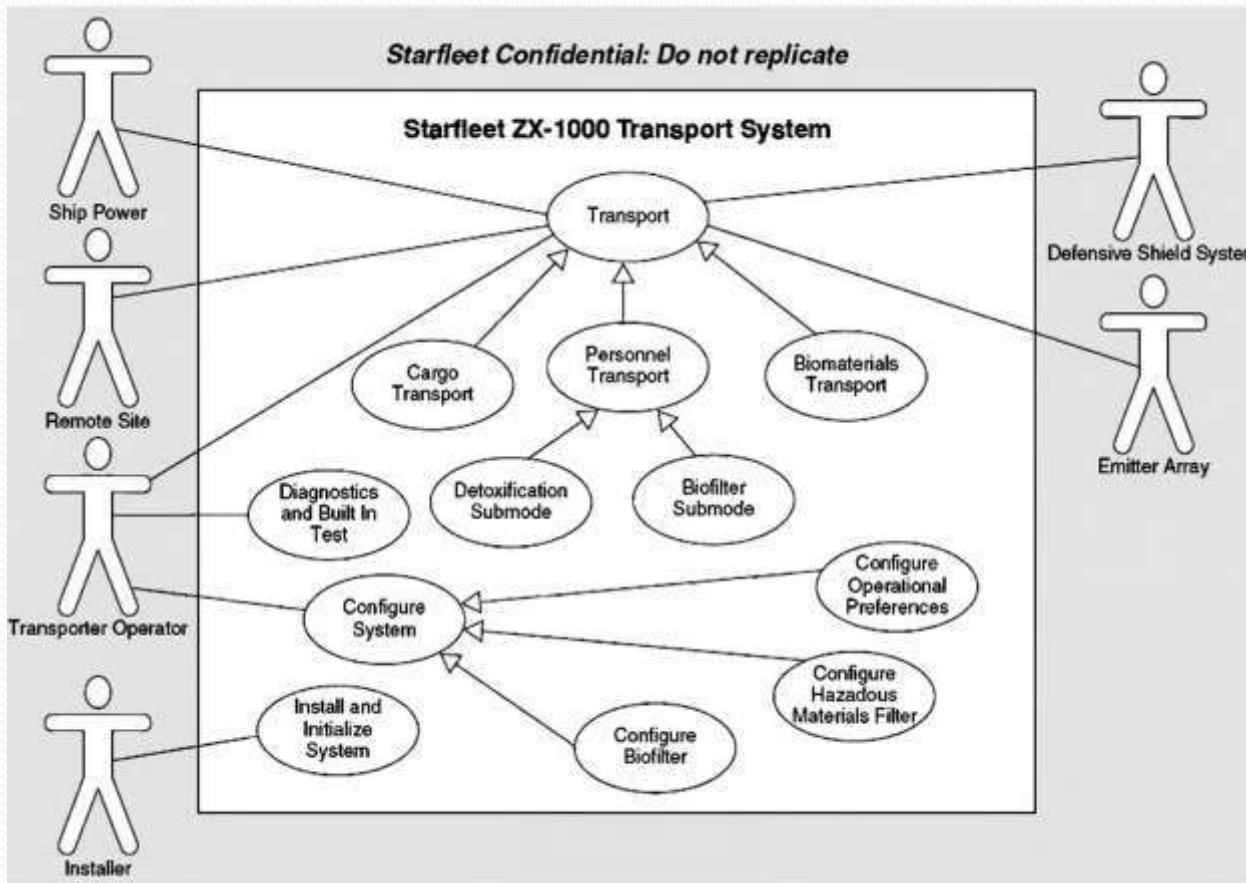
Scenarijaus aprašymas

- Paprastai aprašomi sekų diagramomis **kiekvienam vartojimo atvejui**.
- Kadangi naudojamas „juodos dėžės“ principas, paprastai sąveikauja 2 objekta: vartotojas(-ai) ir sistema.
- Papildomai nurodoma:
 - Vartojimo atvejo pavadinimas;
 - Sekų diagramos pavadinimas;
 - Scenarijaus pavadinimas;
 - Aprašymas;
 - Pradinės sąlygos;
 - Galinės sąlygos;
 - Kiti apribojimai, nematomi diagramoje.

Kiek scenarijų reikia?

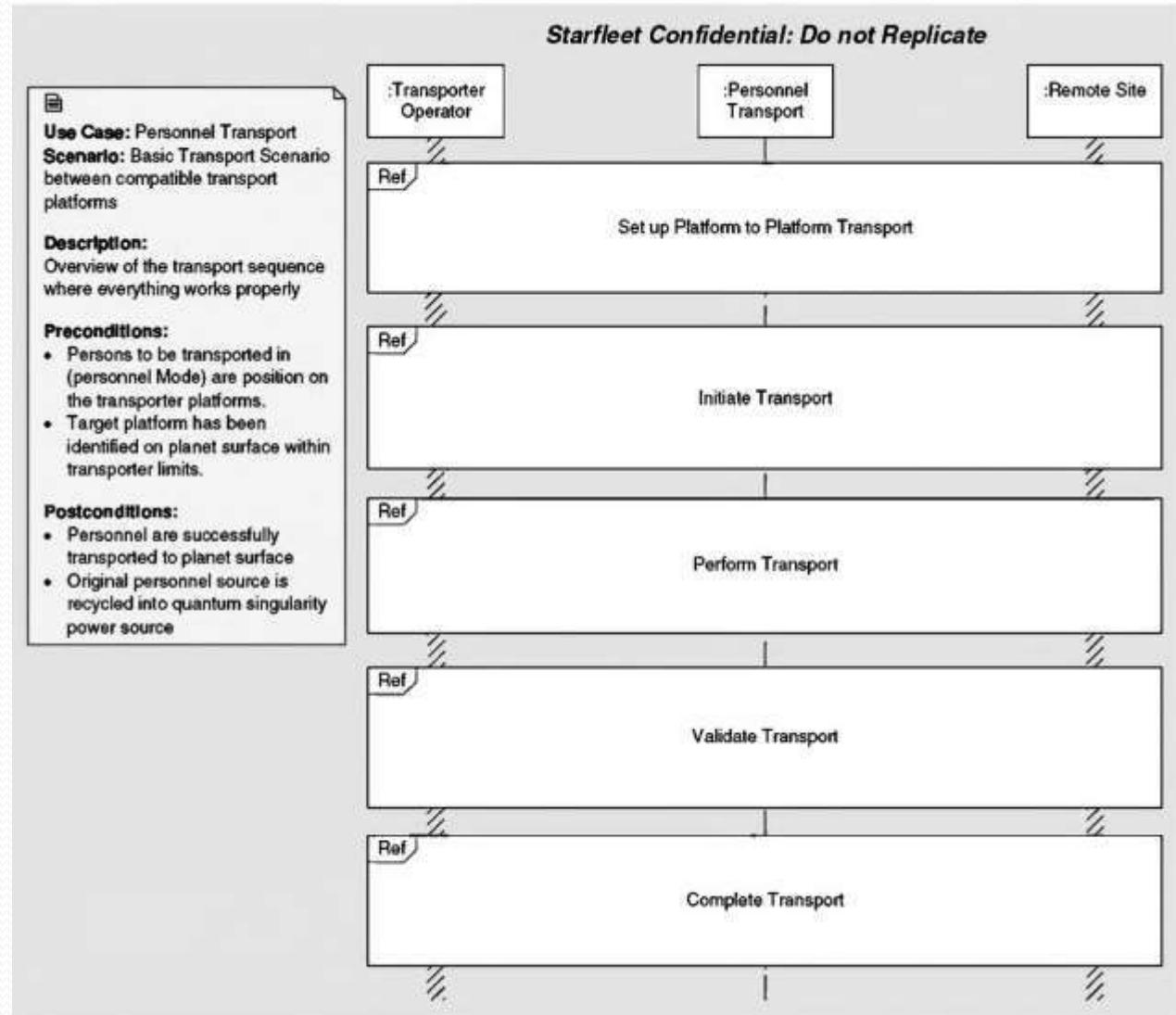
- A *minimal spanning scenario set* for a use case has all the **operational** and **QoS requirements**, contained within that use case, **represented at least once**.
- For faults, it is usually necessary to classify them into *fault equivalence classes*. A fault equivalence class represents a set of faults that are identified and handled using the same means and every fault equivalence class should be represented in at least one scenario.

Teleportacijos sistemos pvz.: vartotojo reikalavimai



Tele- portacijos sistemos

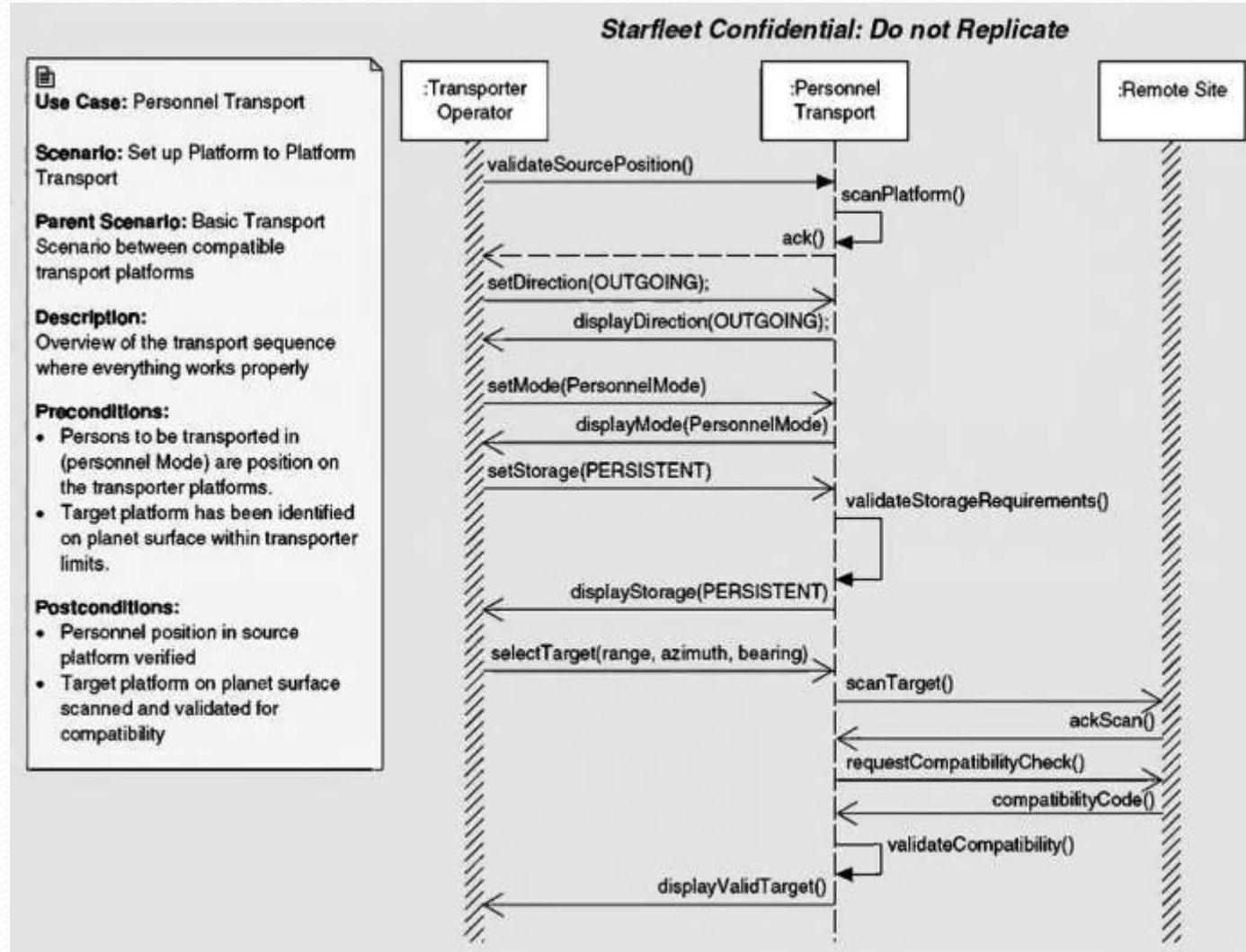
pvz.: mikrociklo pirminis scenarijus



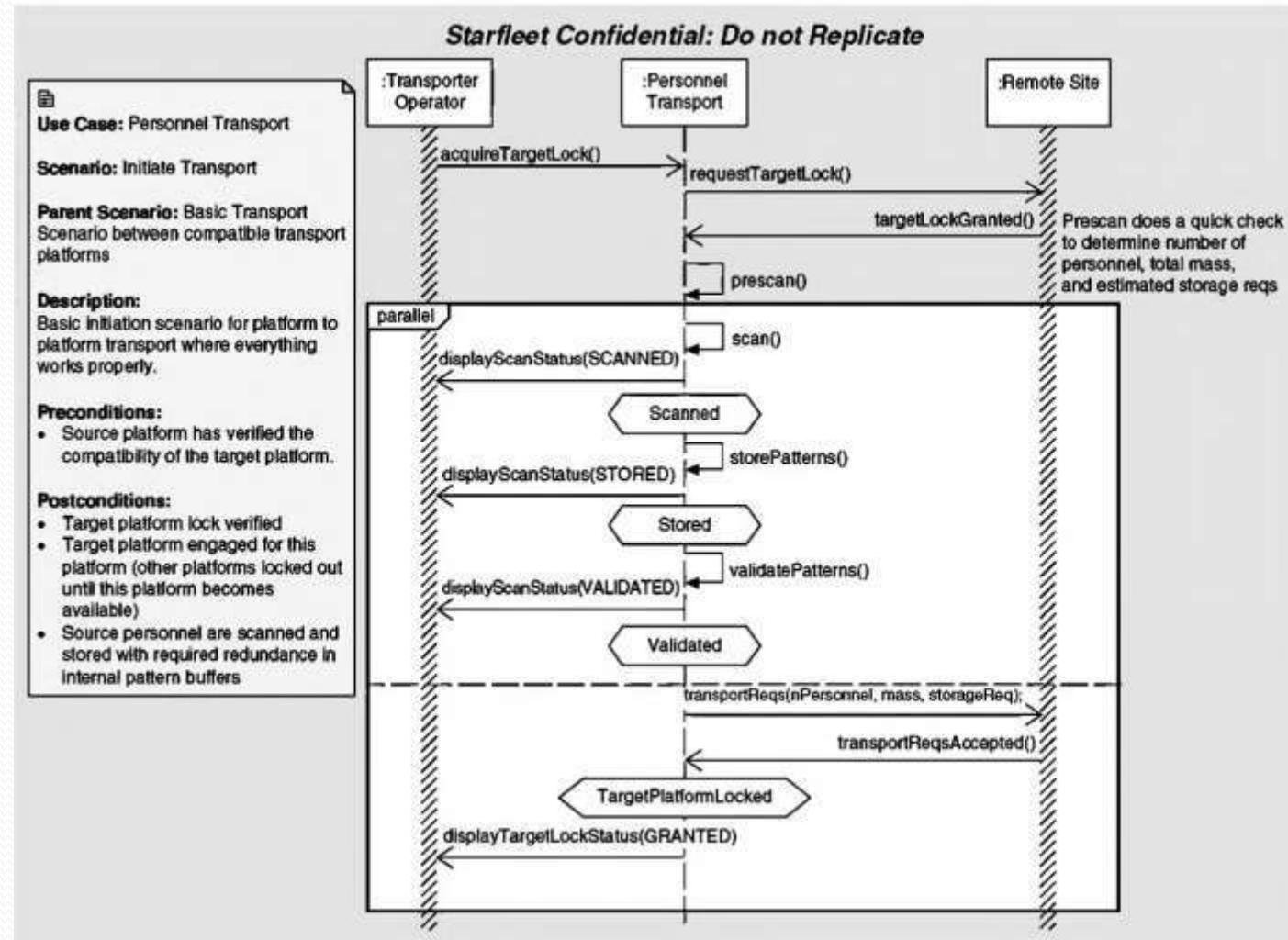
Teleportacijos sistema: kiti scenarijai

- Specifikuojant realią sistemą, reikalingas ne tik idealus pirminis scenarijus (transportavimas iš vienos platformos į kitą), bet ir
 - Iš platformos į kitą vietą;
 - Iš kitos vietas į platformą;
 - Klaidų scenarijai (keleiviai nepilnai įlipo į transportavimo kabinių, per didelis atstumas, dematerializavimo klaidos ir t.t.);
- 1 vartojimo atvejis = 4-20 psl. reikalavimų!!!

Tele- portacijos sistemos pvz.: pirminio scenarijaus detalizacija (1)

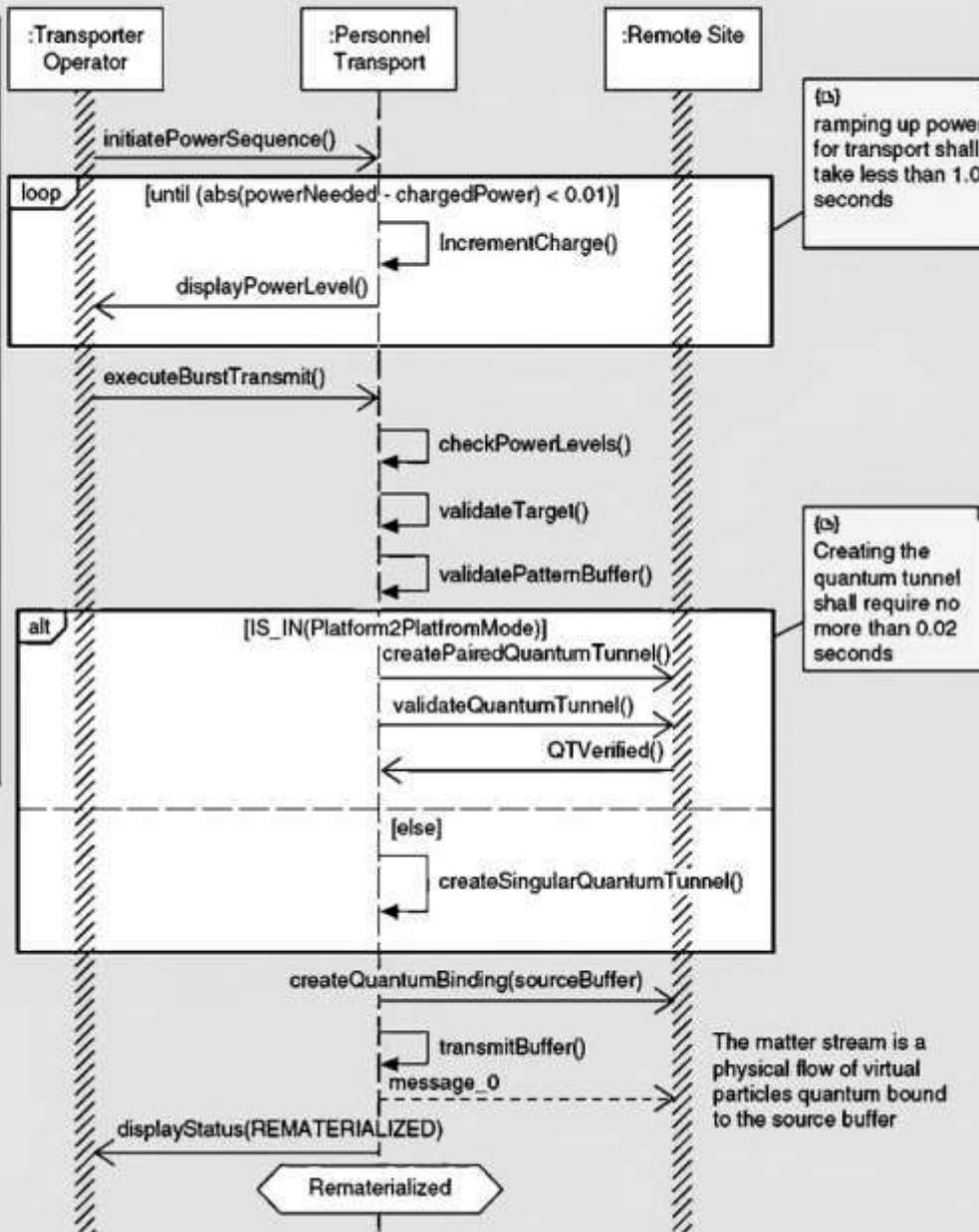


Tele- portacijos sistemos pvz.: pirminio scenarijaus detalizacija (2)



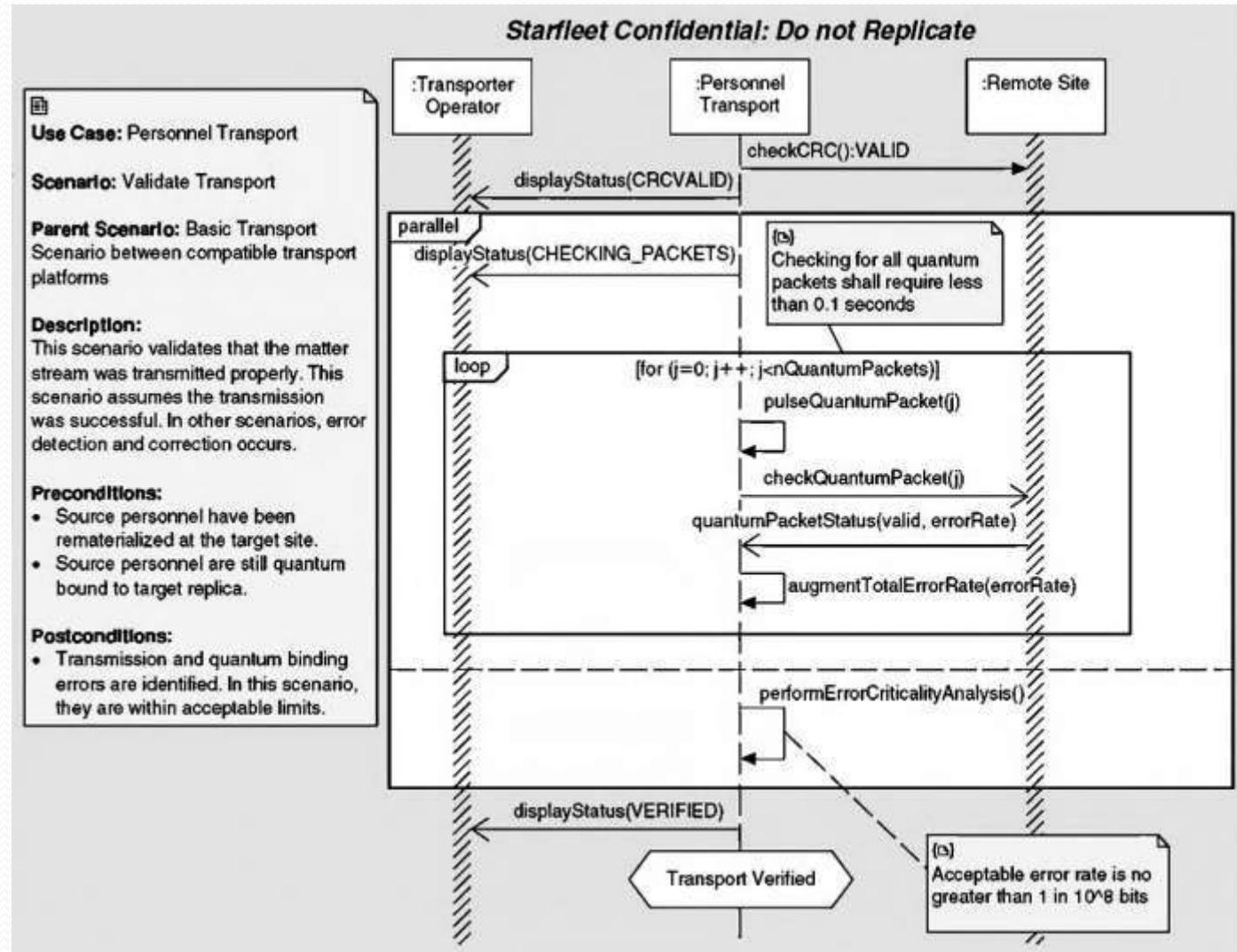
Starfleet Confidential: Do not Replicate

Use Case: Personnel Transport
Scenario: Perform Transport
Parent Scenario: Basic Transport Scenario between compatible transport platforms
Description:
 Basic transport scenario for platform to platform transport where everything works properly. This is done in a burst transmission with 1024-bit CRC check. The target personnel images are rematerialized in the target via quantum binding to original.
Preconditions:
 • Source platform has initiated transport with the target platform and successfully locked the target platform.
Postconditions:
 • Personnel are rematerialized in a quantum-bound state with the original source copies in the source platform

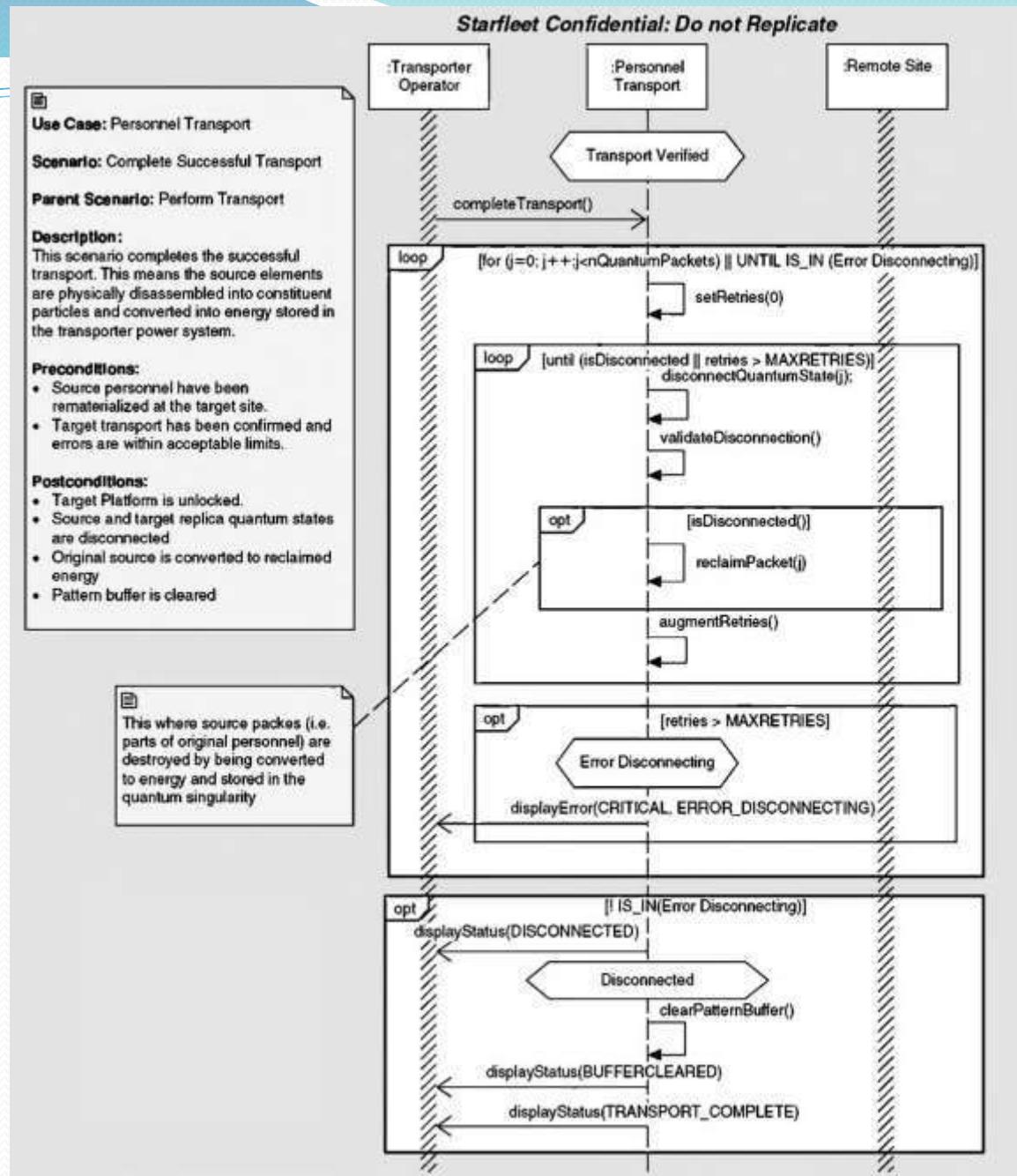


Tele- portacijos sistemos pvz.: pirminio scenarijaus detalizacija (3)

Tele- portacijos sistemos pvz.: pirminio scenarijaus detalizacija (4)



Tele- portacijos sistemos pvz.: pirminio scenarijaus detalizacija (5)



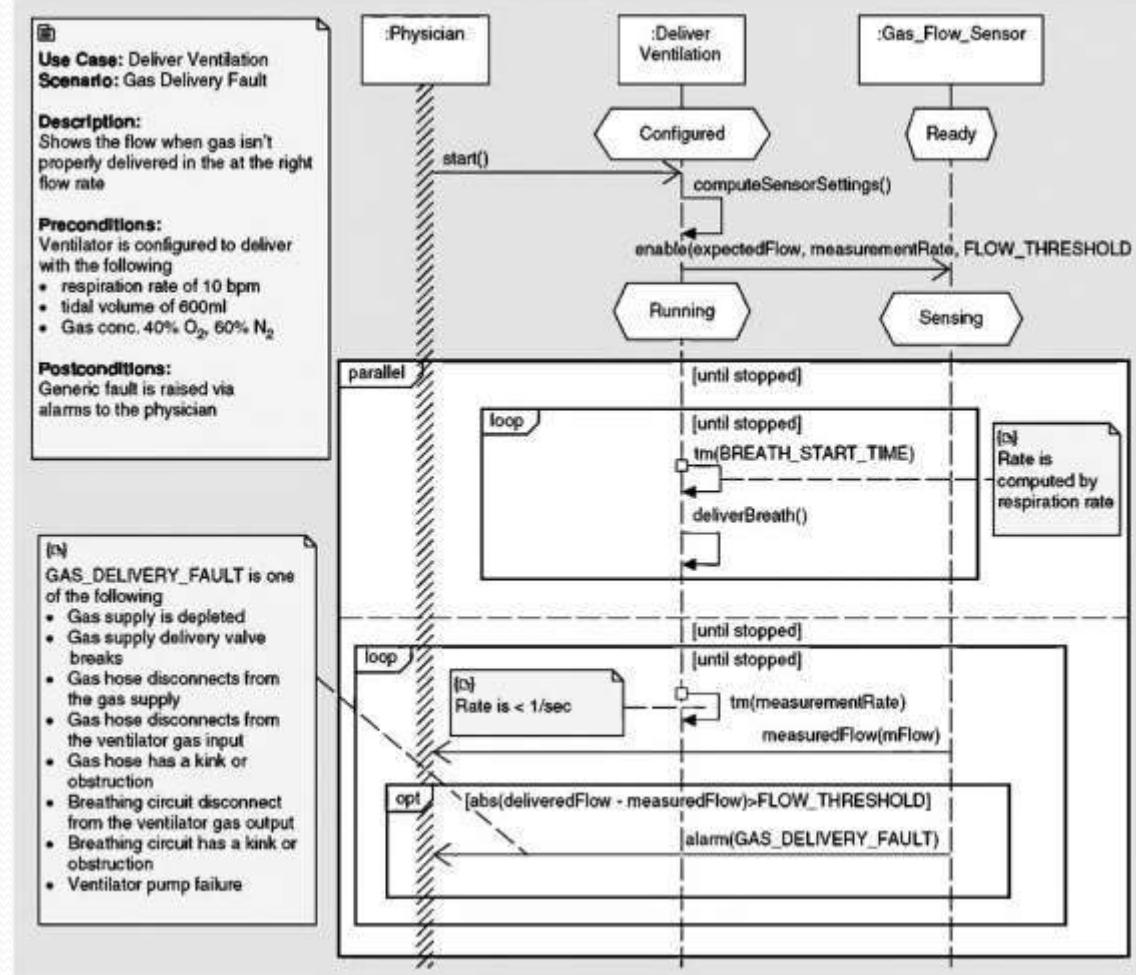
Antriniai, klaidų scenarijai

- Aprašo, kas nutinka, kai įvyksta kažkas nenumatyto, klaidingo.
- Parodo, kaip klaidos aptinkamos ir apdorojamos.
- Kaip taisyklė, kiekvieną gerąjį scenarijų atitinka daugybę antrinių...
- Pavyzdys: medicininis ventilatorius (plaučių), kuris užtikrina ritmingą kvėpavimą, paduoda nustatyta deguonies kiekį, užtikrina paduodamo srauto slėgi ir mišinio kokybę.

Medicininio ventilatoriaus pvz.: antriniai scenarijai (1)

- Išskirtinė situacija: deguonies mišinys nepaduodamas pacientui.
- Priežastys:
 - Dujų balionas tuščias;
 - Sugedo baliono vožtuvas;
 - Atsijungė dujų padavimo žarna;
 - Žarna užsikimšo;
 - Atsijungė žmogaus kvėpavimo įtaisas;
 - Žmogaus kvėpavimo įtaisas užsikimšo;
 - Sugedo ventilatoriaus pompa.

Medicininio ventiliatoriaus pvz.: antriniai scenarijai (2)



Kokie modeliai (UML diagramos) labiausiai tinkami įterptinių sistemų elgsenos modeliavimui? (1)

- Almost all embedded systems exhibit ***reactive behavior***;
 - the system waits for events of interest and then reacts to them by executing some set of actions, changing state, and then waiting for the next event.
 - This kind of behavior is precisely what state machines excel at specifying.
- State machines are *executable* – a characteristic that is invaluable in determining the consistency, correctness, and accuracy of complex sets of requirements.
- State machines are *testable*, which allows requirements to be verified before the system is actually designed.
 - Tests can be automatically generated from such formal requirements and be applied to the completed system later.

Kokie modeliai (UML diagramos) labiausiai tinkami įterptinių sistemų elgsenos modeliavimui? (2)

- Because use cases group requirements into coherent clumps, state machines will be applied a use case at a time.
- Each use case performs a set of control and data transformations triggered by events of interest – an ideal application of state machines.
- For use cases that simply run until they are done – that is, they execute algorithms – activity diagrams may be more appropriate.

Kokie modeliai (UML diagramos) labiausiai tinkami įterptinių sistemų elgsenos modeliavimui? (3)

- Scenarios for the use case are nothing more (or less) than various transition “paths” through the state machine.
 - For minimal coverage, every state machine transition and action should be represented in at least one scenario.
 - As mentioned before, use case scenarios form the basis for the verification test suite of the final system product.

Kokie modeliai (UML diagramos) labiausiai tinkami įterptinių sistemų elgsenos modeliavimui? (4)

- Scenarios are a very useful, *operational* view of the system behavior.
- However, scenarios are only “partially constructive,” meaning that they only tell part of the story.
- A (usually large) set of scenarios is required to show the complete behavior of the use case.
- A state machine is “fully constructive” and specifies all of the behavior for the use case in one place.
 - If necessary, these state machines can be decomposed in the standard ways, but whether a nested state is shown on one diagram or decomposed on another, it is still logically a singular view of the behavior.

Reikalavimų peržiūra

- Būtinas etapas – vartotojai/užsakovai privalo patvirtinti (ir pasirašyti ☺), kad jie tikisi būtent tokio funkcionalumo.
- Sekų diagramos yra geras būdas „pražaisti“ visus scenarijus. Jos gana lengvai suprantamos ir ne IT specialistų.

Reikalavimai viso Harmony proceso kontekste

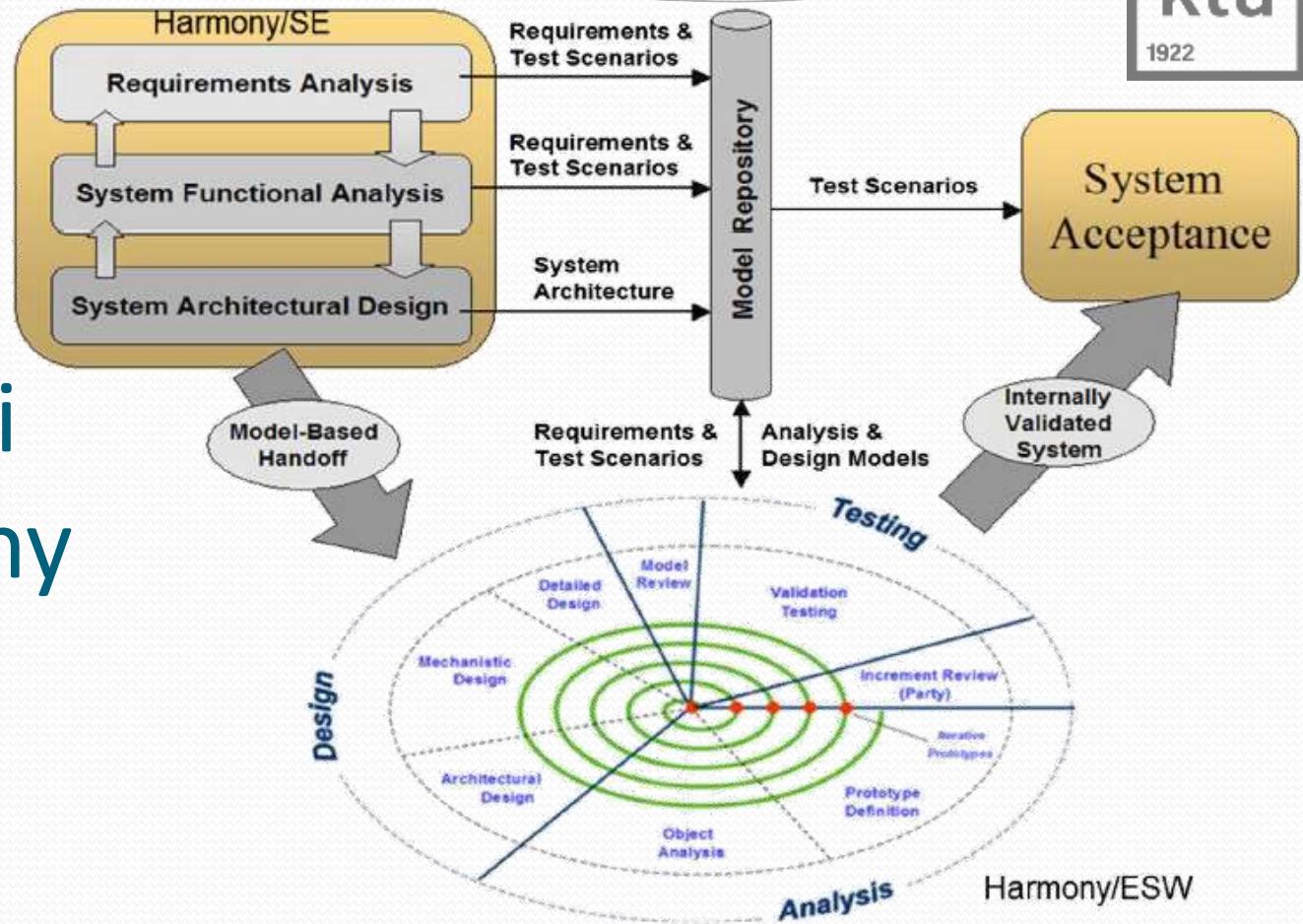


Figure 4.4
Harmony process overview.

Pirminis sistemos modeliavimas viso mikrociklo kontekste

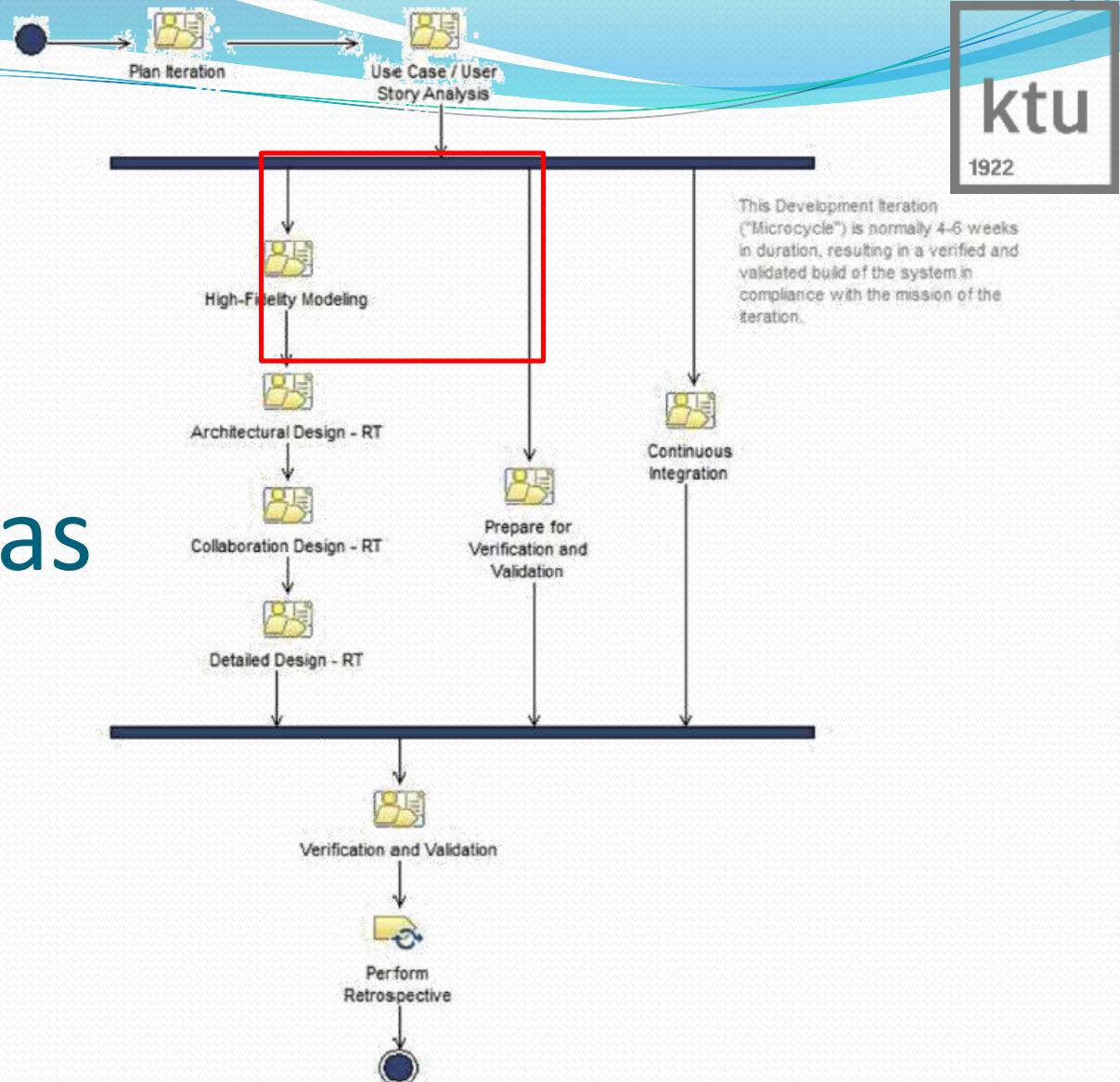


Figure 2.5

Harmony development iteration overview.

High-Fidelity Modeling fazė

Rezultatas – veikiantis (executable) klasių modelis.

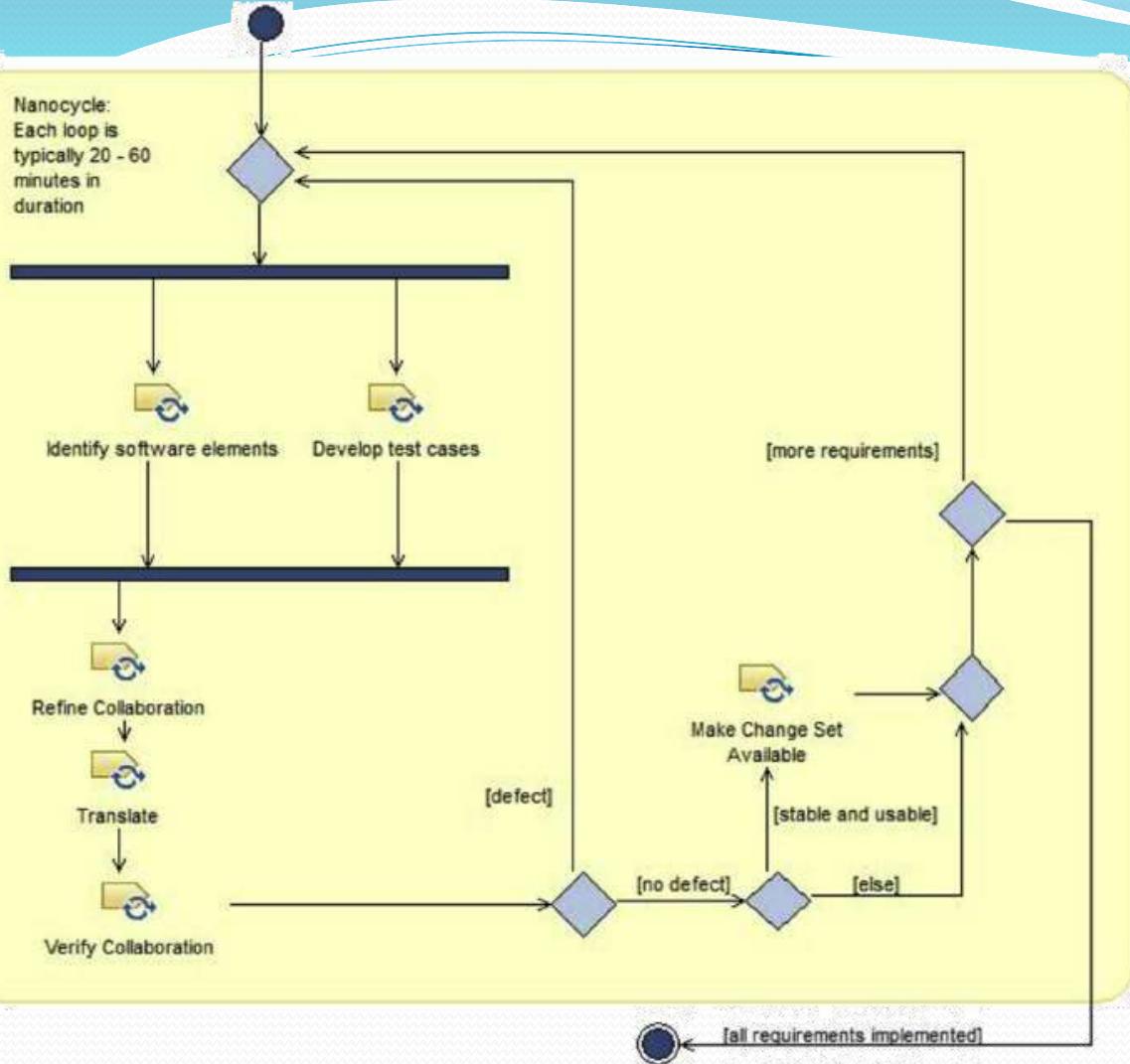


Figure 2.13
High-Fidelity Modeling workflow.

High-Fidelity Modeling paskirtis

- The purpose of the *high-fidelity modeling* phase is to
 - Identify the software elements essential to realizing the use cases
 - Identify relations among the essential software elements
 - With an object-oriented design
 - Identify attributes and allocate them to classes
 - Identify operations and allocate them to classes
 - Specify the behavior of reactive classes with state machines
 - Specify operation behavior with flow charts or activity diagrams
 - With a structured design
 - Identify variables and allocate them to files
 - Identify functions and allocate them to files
 - Identify and specify reactive behavior with state machines
 - Specify function behavior with flow charts or activity diagrams
 - Verify the correctness of the analysis model

Architektūros modeliavimas struktūriname (ne objektniame) programavime



Figure 7.1
File-based C UML model.

Objektų / klasių identifikavimo strategijos

Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p. ISBN13: 9780124077812

Table 7.1 Object Discovery Strategies.

Strategy	Description
Underline the Nouns	Used to gain a first-cut object list, the analyst underlines each noun or noun phrase in the problem statement and evaluates it as a potential object, class, or attribute.
Identify Causal Agents	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify Services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify Messages and Information Flow	Messages must have an object that sends them and an object that receives them as well as, possibly, other objects that process the information contained in the messages.
Identify Real-World Items	Real-world items are entities that exist in the real world, but are not necessarily electronic devices. Examples include objects such as respiratory gases, air pressures, forces, anatomical organs, patient information, chemicals, vats, etc.
Identify Physical Devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. In the internal architecture, they are processors or ancillary electronic "widgets." Note: This is a special category of the Identify Real World Items strategy.
Identify Key Concepts	Key concepts may be modeled as objects. Bank accounts exist only conceptually, but are important objects in a banking domain. Frequency bins for an online autocorrelator may also be objects. This strategy is an antipode to the Identify Real-World Items strategy.
Identify Transactions	Transactions are finite instances of interactions between objects that persist for some significant period of time. Examples include bus messages and queued data. This may be done with ACID (Atomic, Complete, Isolated and Durable) transactions or any kind of transactions – state-based or otherwise.
Identify Persistent Information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify Visual Elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify Control Elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply Scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects and relations.

Underline the Nouns

- Underline each noun or noun phrase in the problem or mission statement and treat it as a potential object.
- Objects identified in this way can be put into different categories:
 - Objects of interest
 - Classes of interest
 - Actors
 - Uninteresting objects
 - Attributes of objects
 - Events
 - Synonyms for any of the above

Identify the Causal Agents

- For every effect, there must be an element that is the cause.
- This strategy looks to identify the objects that cause things to happen, i.e., the ones that are behaviorally active ones.
- These are objects which
 - Produce or control actions
 - Produce or analyze data
 - Provide interfaces to people or devices
 - Store information
 - Provide services to people or devices
 - Contain more fundamental objects as parts
- A causal object is an object that autonomously performs actions, coordinates the activities of component parts, or generates events.
- Whenever there is some initiating action, an object somewhere in the system must provide that service.

Identify Services (Passive Contributors or Server Objects)

- In an object-oriented analysis, all (or at least the great majority of) services will be provided by objects.
- They may provide passive control (that is, they do what is requested of them when it is requested), data storage, or both.
 - A simple switch is a passive control object. It provides a service to the causal objects (it turns the light on or off upon request), but does not initiate actions by itself.
- Passive objects are also known as *servers* because they provide services to client objects.
- Incidentally, servers almost always are at the receiving end of unidirectional associations from their clients;
 - clients know about the servers, but the servers do not know their clients.

Identify Messages and Information Flows

- For each message, there is an
 - object that sends it,
 - an object that receives it,
 - and potentially, other objects that process it.
- Messages correspond to information and control flows and are realized by either operation calls or event receptions.
- In either case, for every message, there is at least a sender object and a receiver object.
- The message itself is often an object.
 - That is true for remote communications with network packets and datagrams.
 - It is also true when the message must be remembered or processed in some way (see the Identify Transactions strategy, below).

Identify Real-World Items

- Embedded systems need to model the information or behavior of real-world objects even though they are not part of the system *per se*.
 - An anesthesia system must model the relevant properties of patients (name, weight, condition, billing information, etc.), even though customers are clearly outside the anesthesia system.
 - A tracking system would typically model relevant aspects of the things that it tracks, such as combat ID, target type, location, velocity and acceleration.
- These internal representations of real-world elements are normally not simulations of those systems but representations of information it is important for the system to represent.
- However, in some cases, such as simulation systems, it is crucial to model the physics of such systems as well, such as airflow in aerodynamic modeling.

Identify Physical Devices

- Real-time systems interact with their environment using sensors and actuators in a hardware domain.
- The system controls and monitors such physical devices inside and outside the system and these objects are modeled as objects.
- Devices must also be configured, calibrated, enabled, and controlled so that they can provide services to the system.
- When device information and state must be maintained or services invoked, the devices are modeled as objects.
- Objects representing physical devices almost always represent the *interfaces* to those devices and serve as a means of encapsulating the invocation of services from them.

Identify Key Concepts

- Key concepts are important abstractions within the domain that have interesting attributes and behaviors.
- These abstractions often do not have physical realizations, but must nevertheless be modeled by the system.
 - E.g. Within the user interface domain, a *window* is a key concept.
 - In the banking domain, an *account* is a key concept.
 - In an autonomous manufacturing robot, the *task plan*, a set of steps required to implement the desired manufacturing process, is a key concept.
 - In the design of a C compiler, *functions*, *data types*, and *pointers* are key concepts.
- Each of these objects has no physical manifestation. They exist only as abstractions modeled within the appropriate domains as objects or classes.

Identify Transactions

- Transactions are objects that represent the interactions of other objects and must persist for a nontrivial period of time.
- Examples of transactions include
 - bank account deposits and withdrawals,
 - elevator requests,
 - target designations for fire control systems,
 - and objects that manage reliable message delivery.
- In many cases, the transactional objects disappear once the interaction has concluded (so-called *volatile transactions*)
- while in other cases, the transactional objects must be retained for long periods of time (also known as *persistent transactions*).

Identify Persistent Information

- Persistent information is typically held within passive container objects such as
 - stacks, queues, trees, tables, or databases.
 - Configuration data for various devices is one kind of persistent data.
- Many systems also acquire information from their environments and store this information.
 - Objects hold that information in their attributes. Examples include a system that remembers patient data or usernames and passwords, calibration tables for sensors, and flight data for a black box recorder.

Identify Visual Elements

- Many real-time systems interact directly or indirectly with human users.
- Real-time system displays may be as simple as a single blinking LED to indicate power status, or as elaborate as a full Windows-like GUI with buttons, windows, scroll bars, icons, and text.
- Visual elements used to convey information to the user are objects within the user interface domain.
- These visual elements are themselves objects, and have both attributes and behavior.

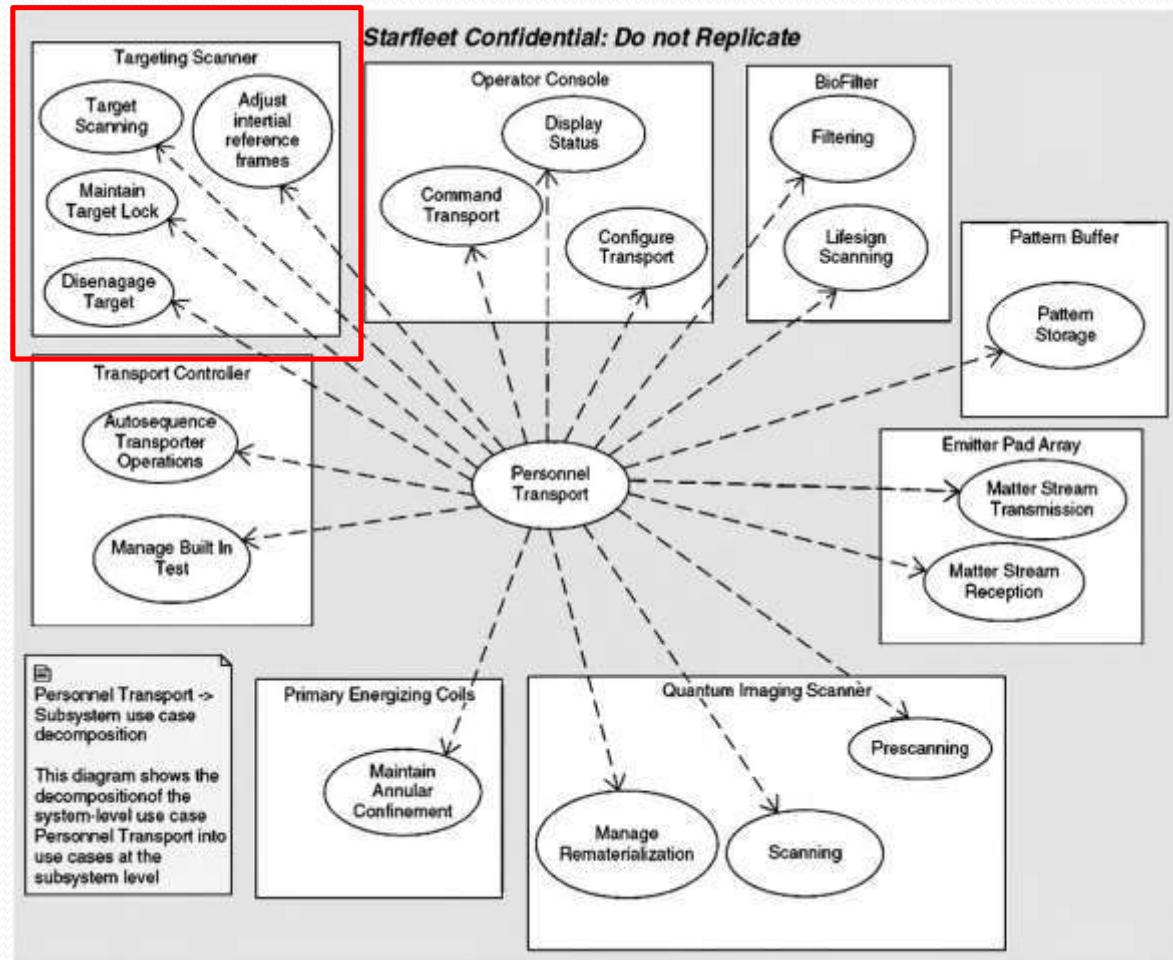
Identify Control Elements

- Control elements are entities that control other objects.
- These are specific types of causal objects.
- Some objects, called *composites*, often orchestrate the behaviors of their part objects.
- These may be simple objects or may be elaborate control systems, such as
 - PID control loops
 - Fuzzy logic inference engines
 - Expert system inference engines
 - Neural network simulators
 - State-based systems
 - Algorithmic systems

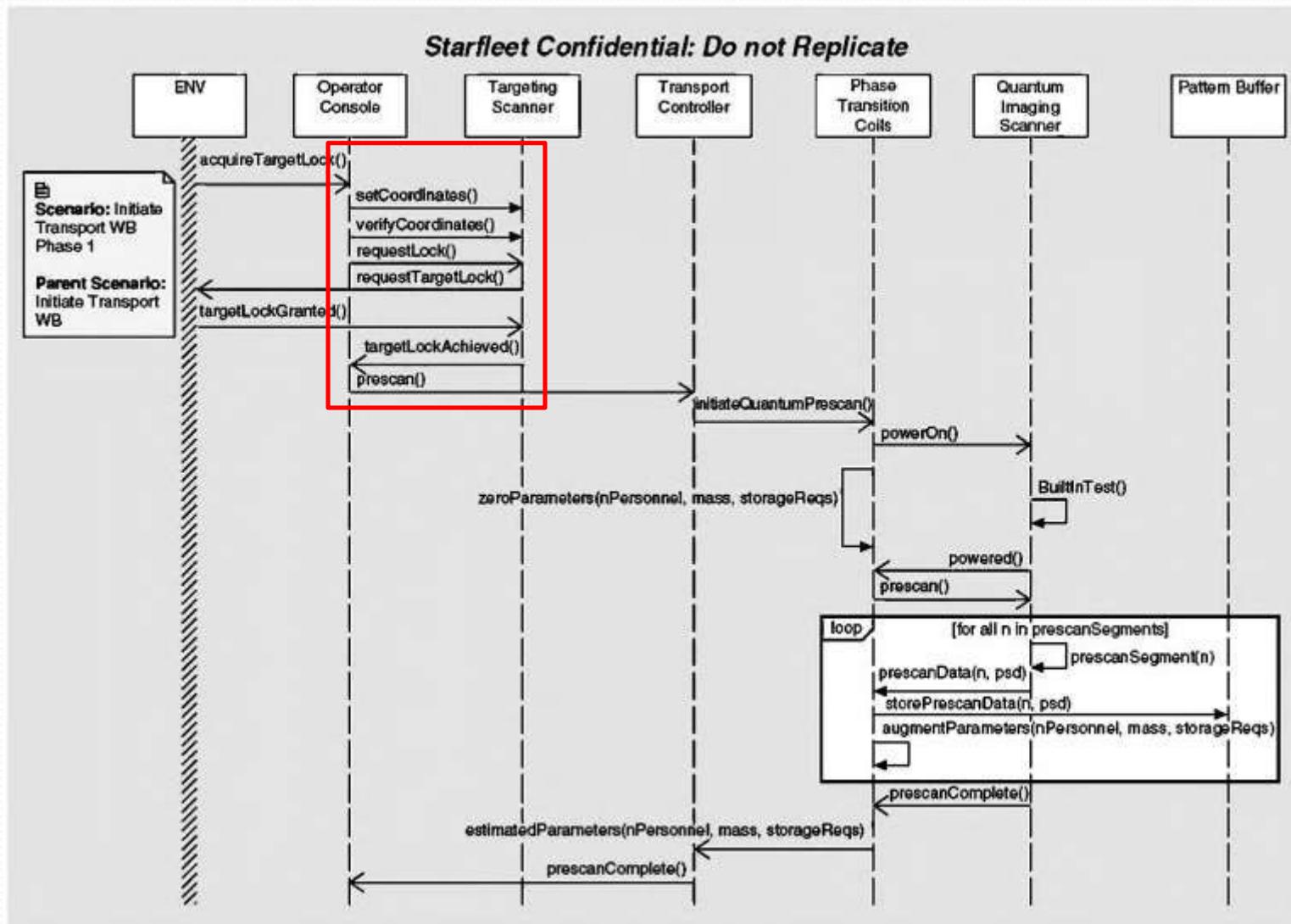
Apply Scenarios

- The application of use case scenarios is another strategy to
 - identify missing objects as well as
 - to test that an object collaboration adequately realizes a use case.
- Using only known objects, step through the messages to implement the scenario.
- The object collaboration structure must support the elaborated use case scenarios – including providing the information to be manipulated and the services realizing the messages in the scenarios.
- This is one of the most useful strategies because it allows you to identify elements and behavior missing from your collaboration.

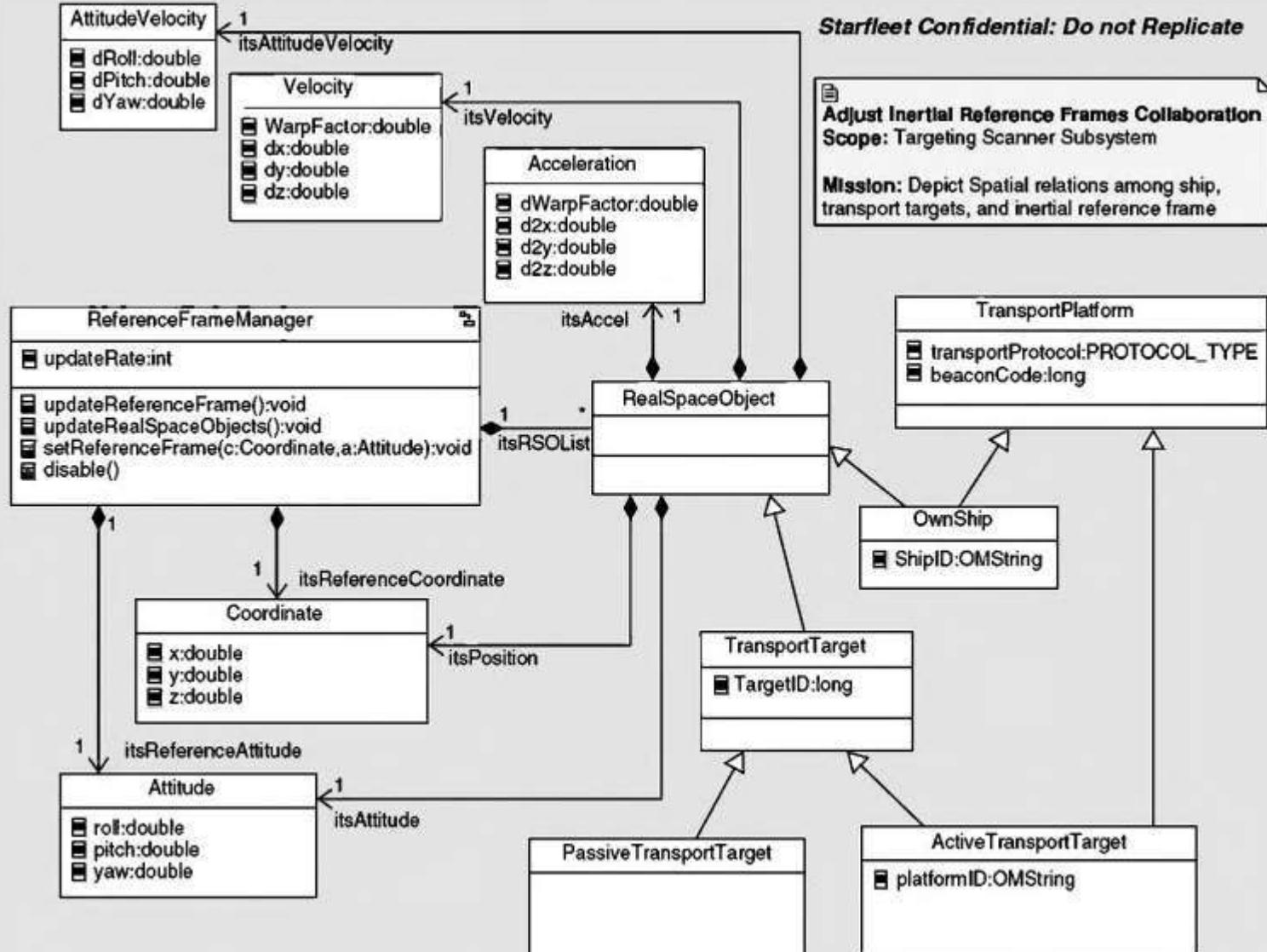
Objektų identifikacijos p.vz.



Objektų identifikavimas iš vartojimo atvejų ir jų scenarijų (1)

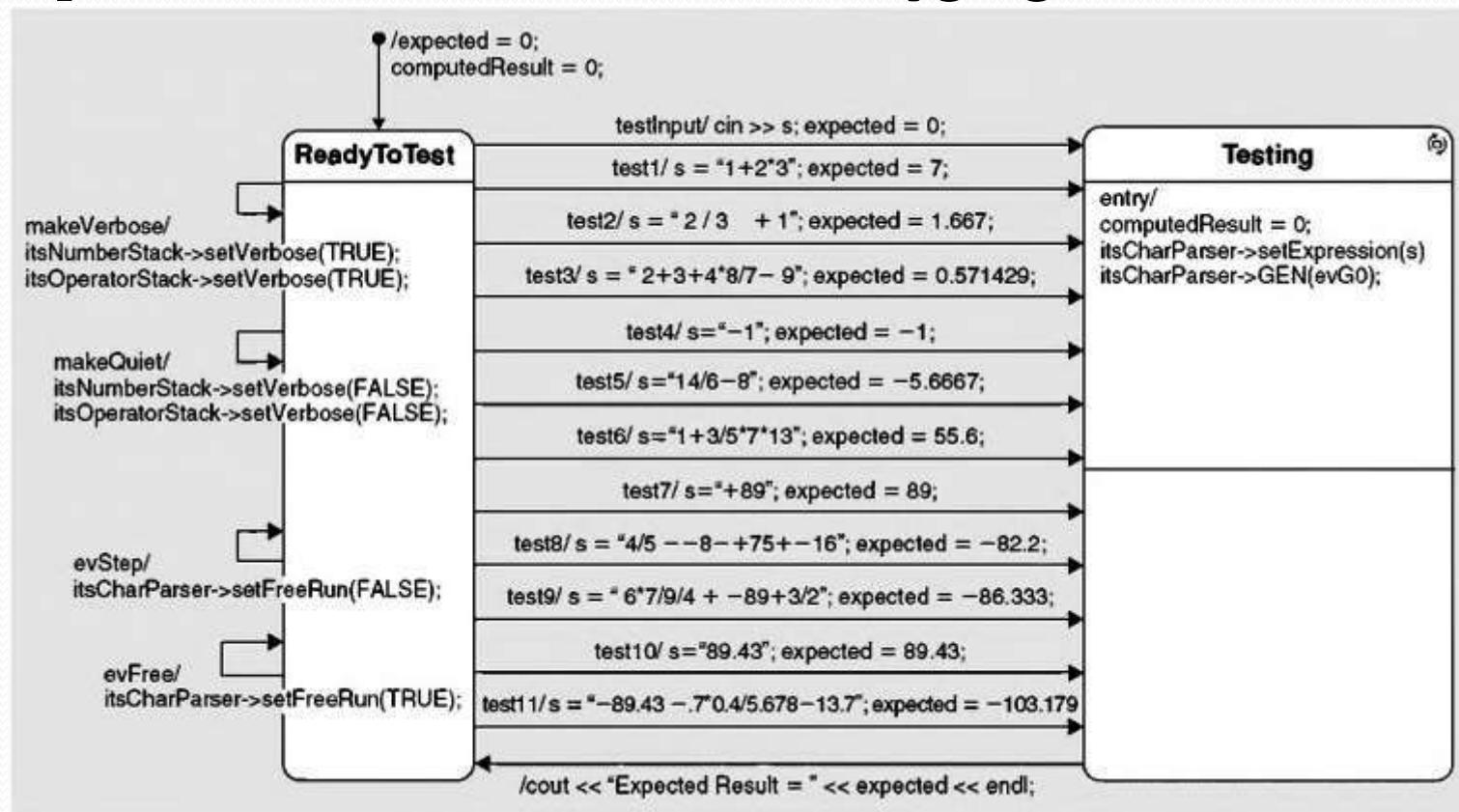


Objektų identifikavimas iš vartojimo atveju ir jų scenarijų (2)



Testavimo atvejų sudarymas

- Nepamirštam, kad Testai kuriami lygiagrečiai su modeliu:



Kodo generavimas, vykdymas, testavimas, integravimas

- Jei įmanoma, reikia naudoti automatines priemones, bet net ir tokų neturint, visada galima parašyti modelį atitinkantį kodą ... rankiniu būdu ☺
- Sukompiliuotą kodą testuojame ir **būtinai** įsitikiname, kad viskas veikia OK ir visi testai praėjo prieš įdėdami savo komponentą į CVS sistemą.
 - Nėra nieko blogiau, kai neatsakingi programuotojai paviršutiniškai ištестуоja (tiksliau neištестуоja) savo modelių/kodo ir atiduoda kitiams. Iškart nepastebėjus tokios klaidos, vėliau ypač sunku atsekti, kas ir kada padarė klaidą, nes kiti kolegos tikisi, kad tas komponentas tikrai veikia korektiškai ir klaidų ieško pas save ☺

Projektavimas viso mikrociklo kontekste

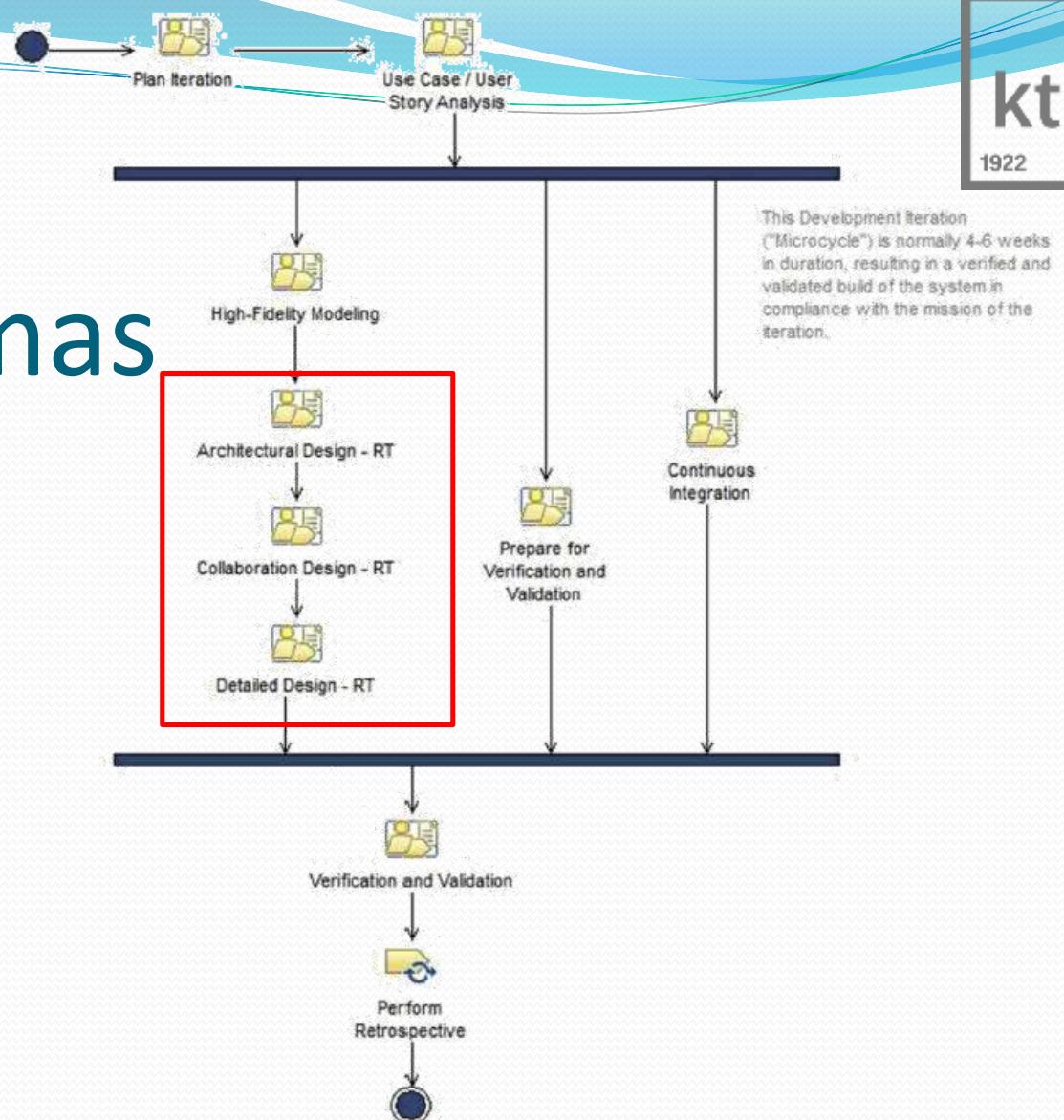


Figure 2.5
Harmony development iteration overview.

Kas jau padaryta prieš pradedant projektavimo fazę ?

- Nemažom su tradiciniais kūrimo procesais:
 - Harmony projektavimas – tai **jau sukurtos** įrangos tobulinimas-optimizavimas.
- Prieš pradedant projektavimą mes jau turime:
 - Mikrociklo misijos aprašą;
 - Tvarkaraštį su resursų paskirstymu;
 - Mikrocikle realizuojamų vartojimo atvejų specifikacijas;
 - Pirminio projektavimo modelius (architektūrą ir sąveiką);
 - Programinį kodą, sugeneruotą iš modelių;
 - Testavimo atvejų modelius.

Projektavimas

- Trys projektavimo lygiai:
 - **Architektūrinis** (angl. architectural) – sistemos lyguo.
 - **Bendradarbiavimo** (angl. mechanistic) – vartojimo atvejo lyguo.
 - **Detalus** (angl. detailed) – klasės/metodo lyguo.
- Tikslas – **Optimizacija**.
- Etapai:
 - Identifikuoti svarbius optimizacijos kriterijus;
 - Surikiuoti kriterijus pagal svarbą;
 - Parinkti projektavimo šablonus ir technologijas, kurie optimizuoją svarbesnius kriterijus mažiau svarbių sąskaita;
 - Pritaikyti projektinius sprendimus (patobulinti modelius);
 - Verifikasioti ir Validuoti patobulintą sprendimą.

Ką reiškia optimizuoti?

- Tikslas padaryti sukurtą įrangą:
 - Pakankamai greitą;
 - Gerai palaikomą;
 - Pakankamai patikimą;
 - Saugią ir t.t.

Dažniausios optimizacijos klaidos

- Pradedamas optimizuoti kodas ir modeliai dar ne iki galio sukurtos ir sumodeliuotos sistemos.
 - Kuo daugiau „judančių“ (nebaigtų) sistemos dalių, tuo didesnė tikimybė, kad ne tik nepavyks ką nors optimizuoti, bet bus dar labiau pakenkta visam projektui.
- Tiksliai nežinoma, ką ir kodėl reikia optimizuoti.
 - Pvz. bandoma sumažinti kodo eilučių kiekį, bet ar tikrai to reikia, jeigu įterptinė sistema turi pakankamai RAM ar Flash atminties?
 - Daugeliui paprastų sistemų apskritai nereikia nieko optimizuoti, o optimizavimas vardan jo paties – tai tik laiko ir kitų resursų švaistymas.
 - Geriausias būdas nustatyti, ką ir kodėl reikia optimizuoti – **išbandyti priminę sistemos versiją ir pamatuoti jos veikimo parametrus!**

Projektavimo šablonų nauda (1)

- Nereikia išradinėti dviračio – pasinaudojame kitų jau atrastais ir išbandytais sprendimais.
- Nepamirštam, kad kaip taisyklė, šablonai turi ir neigiamų pasekmių!
- Dažniausiai šablonai padeda optimizuoti QoS parametrus:
 - Našumas (blogiausiu atveju, vidutiniu atveju, nuspėjamumas);
 - Galimybė suplanuoti (schedulability);
 - Pralaidumas (vidutinis, ilgalaikis, staigios apkrovos (burst));
 - Patikimumas (klaidų atžvilgiu, gedimų atžvilgiu);

Projektavimo šablonų nauda (2)

- Dažniausiai šablonai padeda optimizuoti QoS parametrus:
 - Saugumas;
 - Pakartotinio panaudojimo galimybė;
 - Paskirstomumas;
 - Pernešamumas;
 - Palaikomumas;
 - Plečiamumas;
 - Sudėtingumas;
 - Resursų naudojimas;
 - Energijos naudojimas;
 - Kaina.

Architektūrinis projektavimas

- Architecture encompasses the largescale design decisions that affect most or all of the system.
 - Architecture is a part of design.
- Design focuses on optimization of the analysis model against what are collectively termed the *design criteria*:
 - The set of aspects of the system against which different designs and technological choices are measured, evaluated, and, ultimately, selected.
- Design criteria may refer to:
 - system performance (such as worst-case execution time, bandwidth, or throughput),
 - runtime usage of system resources (such as memory),
 - design-time “goodness” metrics (such as complexity or encapsulation),
 - design properties (such as maintainability, reusability, or portability)
 - or even project properties (such as work effort required).

Architektūrinio projektavimo veiklos

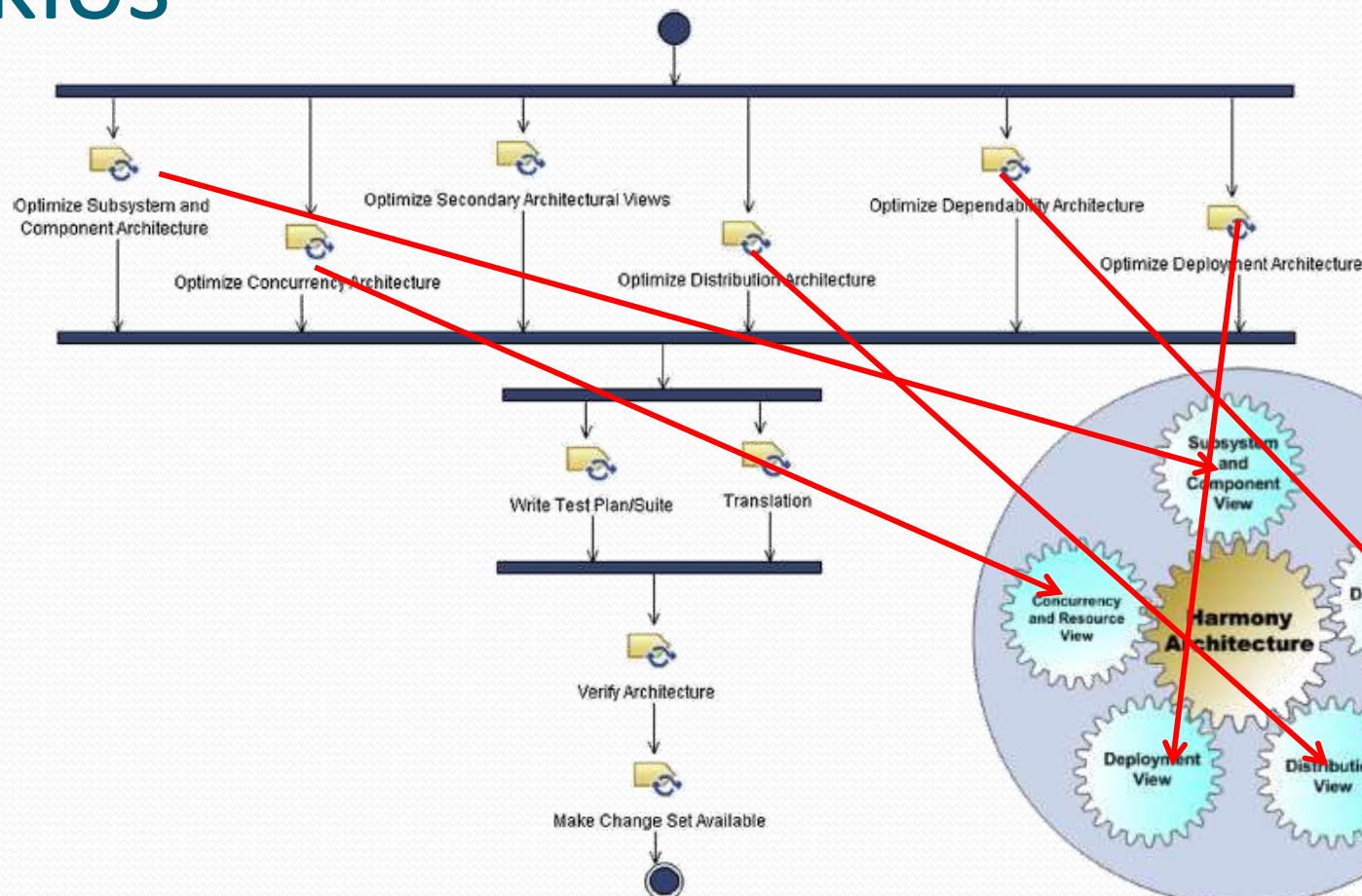


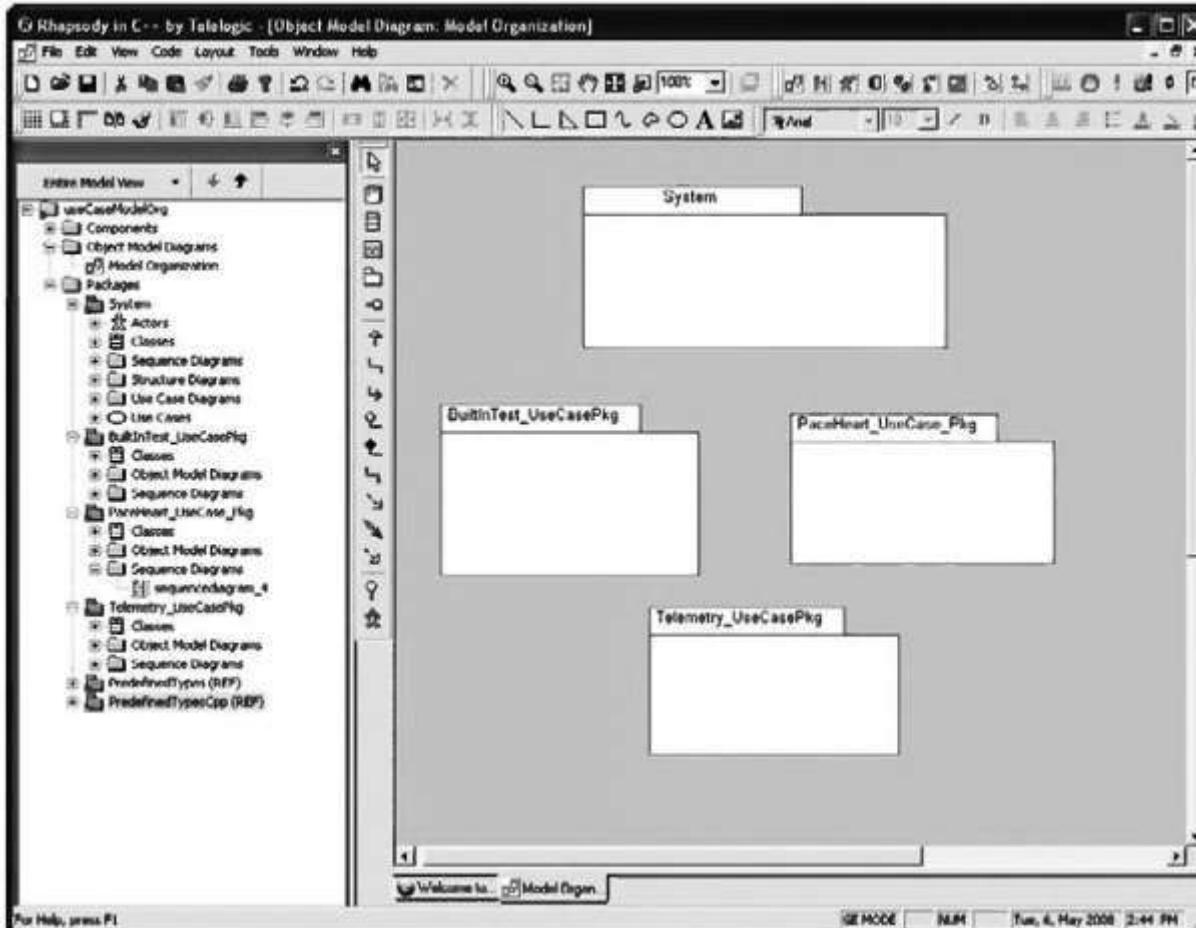
Figure 2.15
Architectural Design workflow.

Figure 2.14
Harmony architectural views.

Loginės architektūros specifikavimas

- Modelių ir modeliavimo proceso organizavimas.
- Užtikrinama, kad:
 - Projektuotojai galės tobulinti modelius, nepakenkdami prieš tai atliktam darbui.
 - Projektuojams bus prieinamos modelių dalys, už kurias jie nėra atsakingi.
 - Bus sudaryta efektyvi modelių apjungimo ir pavertimo į galutinę sistemą procedūra.
 - Bus galima dirbti su įvairiais modelių elementais.
 - Bus galima pakartotinai panaudoti modelius ir jų dalis.

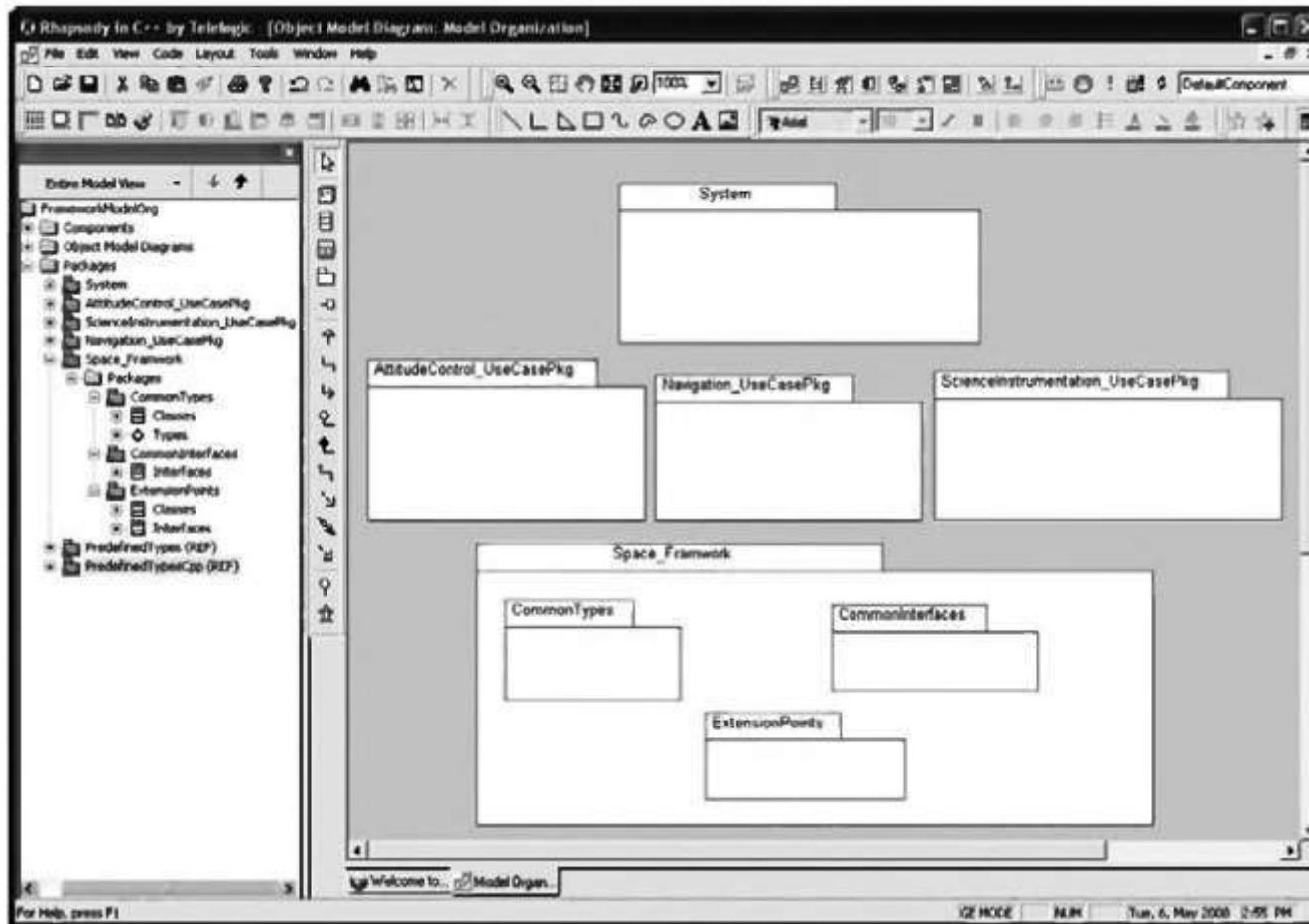
Paprasčiausias modelių organizavimas: pagal vartojimo atvejus (1)



organizavimas: pagal vartojimo atvejus (2)

- Tinka nedidelėms sistemoms su kelias vartojimo atvejais ir keliais kūrėjais.
- Paprasta atsekti, kaip realizuojamas kiekvienas vartojimo atvejis.
- Minusai:
 - Netinka vidutinėms sistemoms;
 - Nesimato bendrų elementų – tenka kaskart kartoti tą patį ir išradinėti dviratį;
 - Negalimas pakartotinis panaudojimas.

Modelių organizavimas: karkaso principo naudojimas (1)



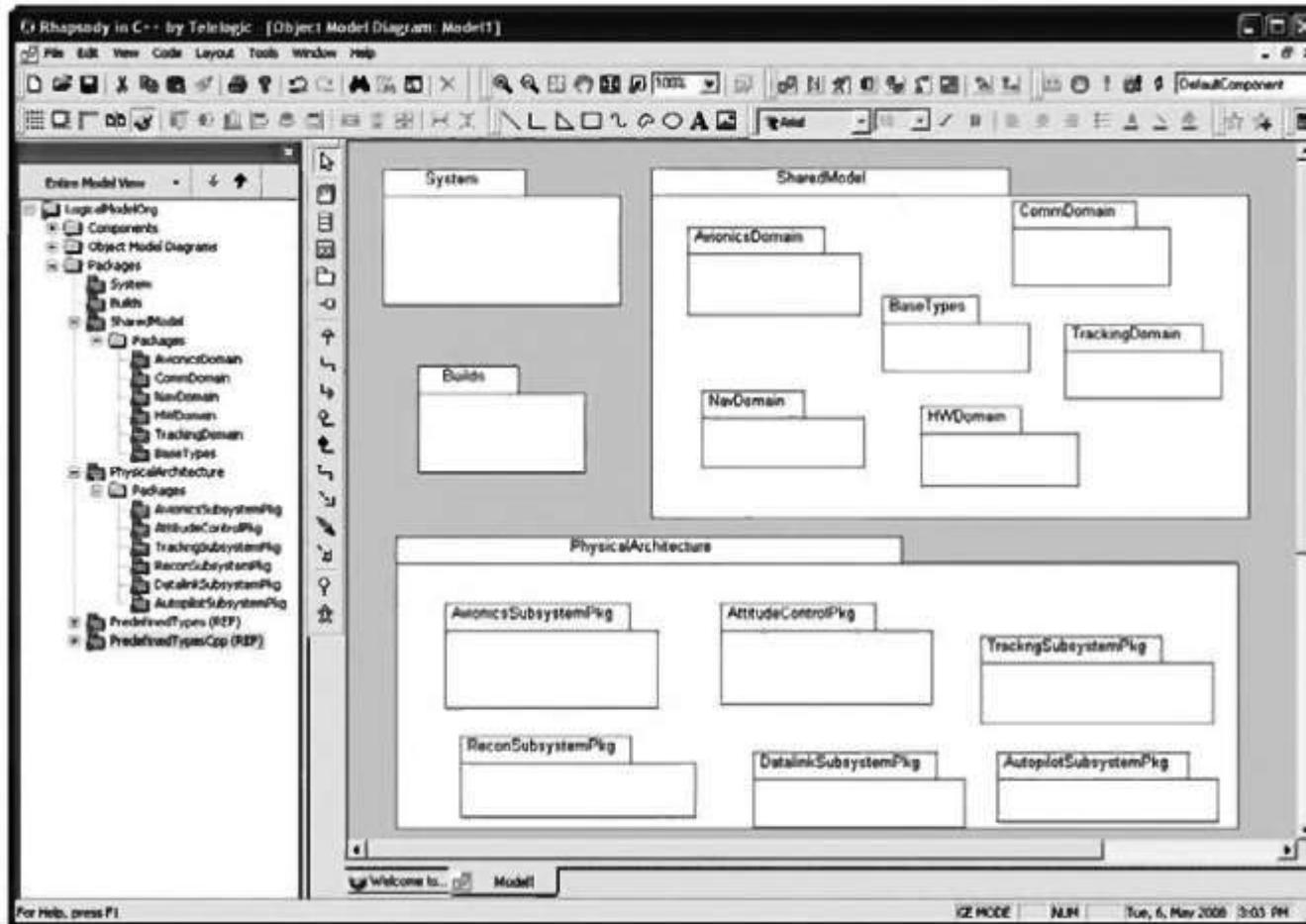
Modelių organizavimas: karkaso principo naudojimas (2)

- Tinka mažoms sistemoms;
- Turi paketą, skirtą bendriems elementams;
- Minusai:
 - Dar vis mažas pakartotino panaudojimo ir plečiamumo laipsnis.

Loginis ir fizinis modelis

- Loginis modelis – tai tipai ir klasės (PIM).
- Fizinis modelis – tai komponentai, objektai, užduotys, posistemės (PSM).
- Norint užtikrinti pakartotinį panaudojimą, naudinga atskirti šiuos modelius.

Modelių organizavimas: loginiu modeliu pagristas organizavimas (1)



Modelių organizavimas: loginiu modeliu pagrįstas organizavimas (2)

- Tinka didelėms sistemoms;
- 5 pagrindiniai paketai:
 - Sistema;
 - Fizinė architektūra;
 - Posistemės;
 - Bendras modelis;
 - Versijos (builds).
- Pliusai:
 - Pakartotinis panaudojimas;
 - Plečiamumas.

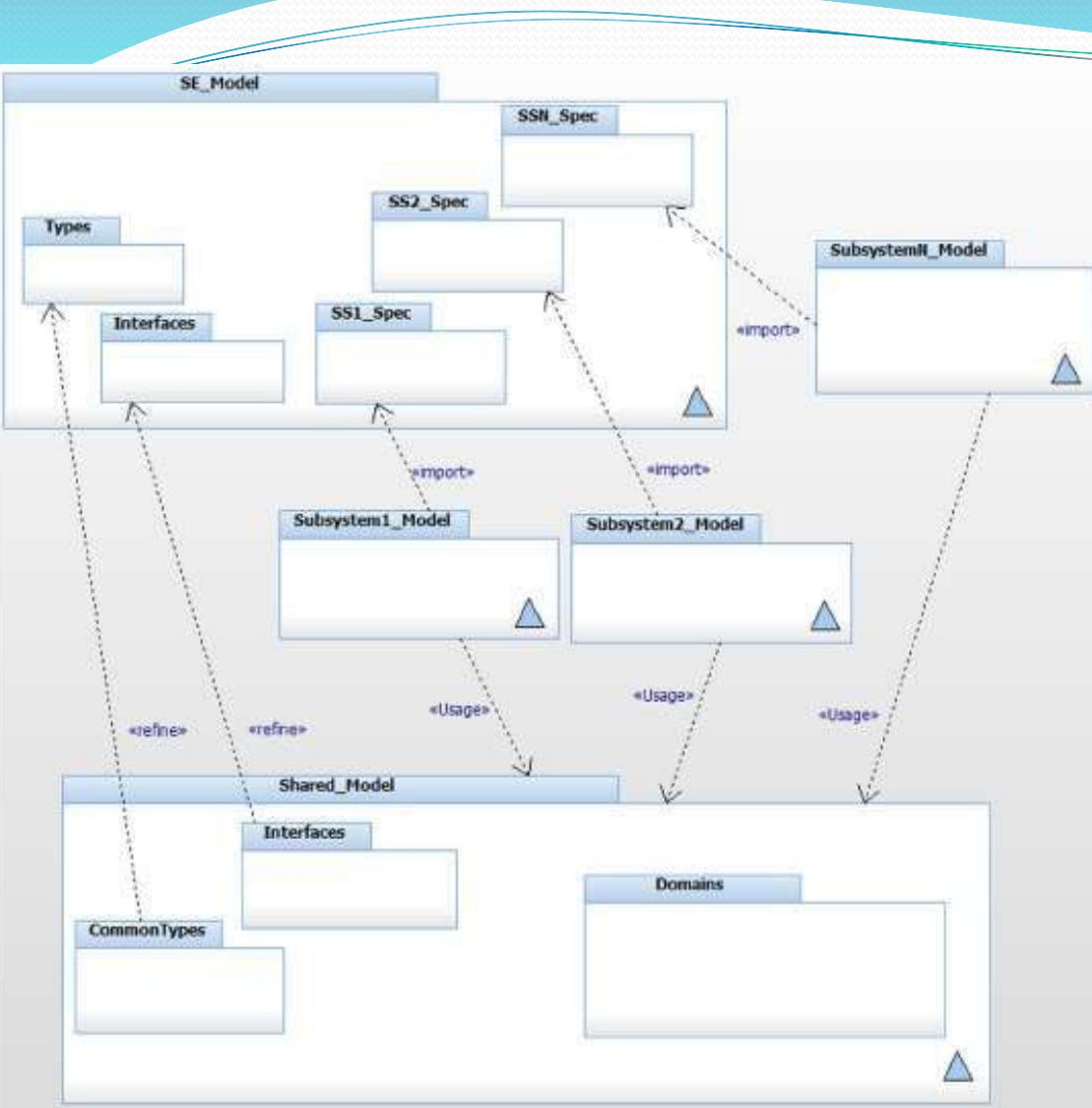


Figure 5.2
Recommended large project organization.

Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p.
 ISBN13: 9780124077812

Rekomenduojama modelių organizavimo struktūra

Kas yra posistemė?

- Posistemė – tai didesnės apimties sistemas architektūrinis vienetas.
- Gali būti sudarytas tik iš programines įrangos, arba ir iš techninės.
- Turi aiškiai apibrėžtas programines sąsajas bendravimui su kitomis posistemėmis.
- Loosely coupled (silpno susiejimo) principas – kitos posistemės, norėdamos naudotis programinėmis sąsajomis, neturi žinoti ir neprivalo suprasti, kaip veikia naudojama posistemė (juoda dėžė).

Posistemės vartojimo atvejų identifikavimas

- Paprastai posistemes realizuoja skirtinges inžinierių komandas,
 - kurios privalo žinoti reikalavimus keliamus jų posistemei iš
 - vartotojų ir kitų sistemos dalių.
- Įejimas:
 - Use case modelis ir scenarijai;
 - Sistemos architektūra.
- Du būdai:
 - „Iš viršaus į apačią“ dekomponuojant sistemos vartojimo atvejus į posistemes, kurios atsakingos už jų vykdymą.
 - „Iš apačios į viršų“, kai detalūs reikalavimai, t.y. use case modelio įvykiai ir žinutės yra priskiriami posistemėms (pagal scenarijų sekų diagramas), o tada šie reikalavimai apjungiami į posistemės vartojimo atvejus.

Posistemės vartojimo atvejų modelis

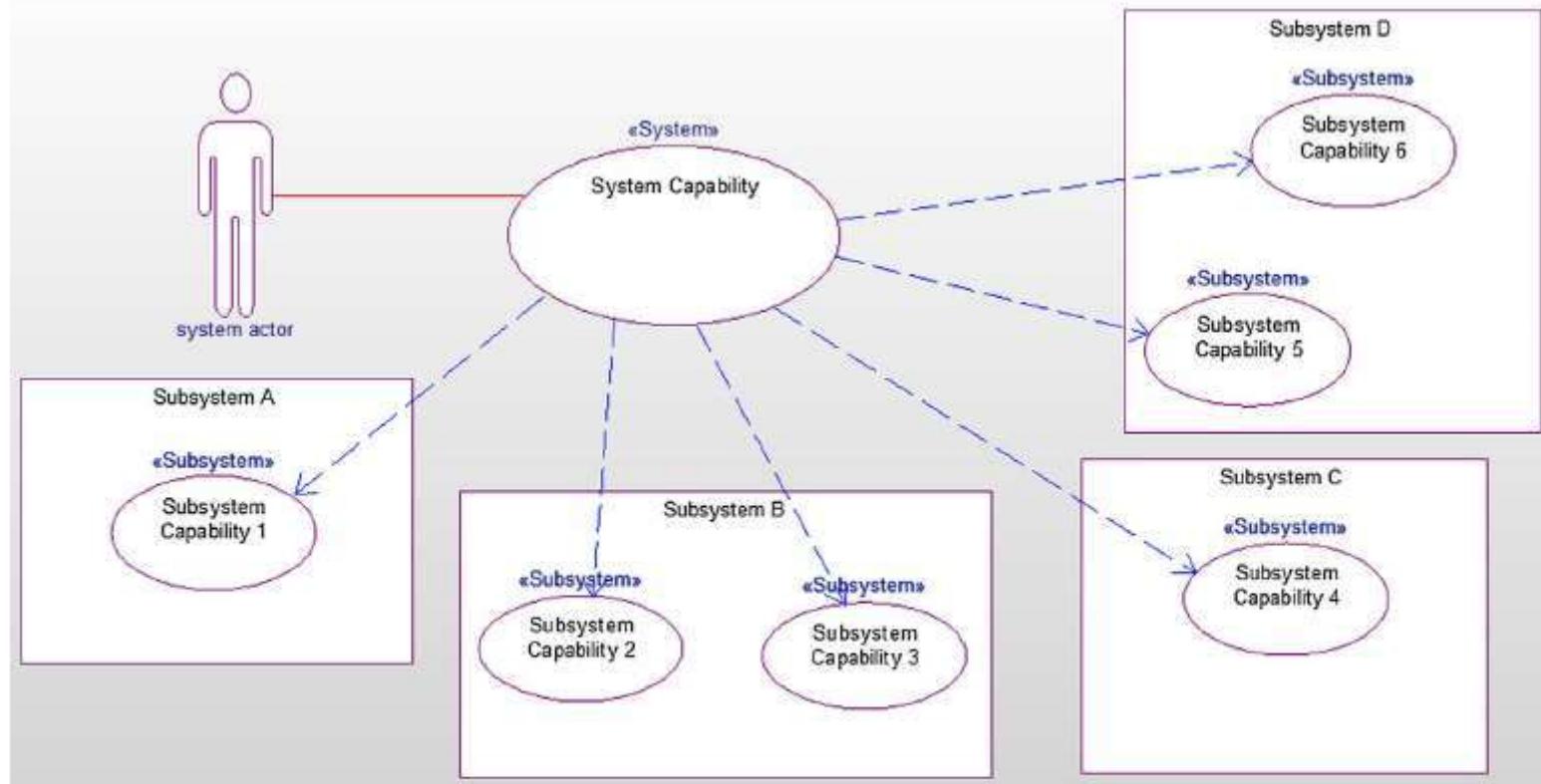
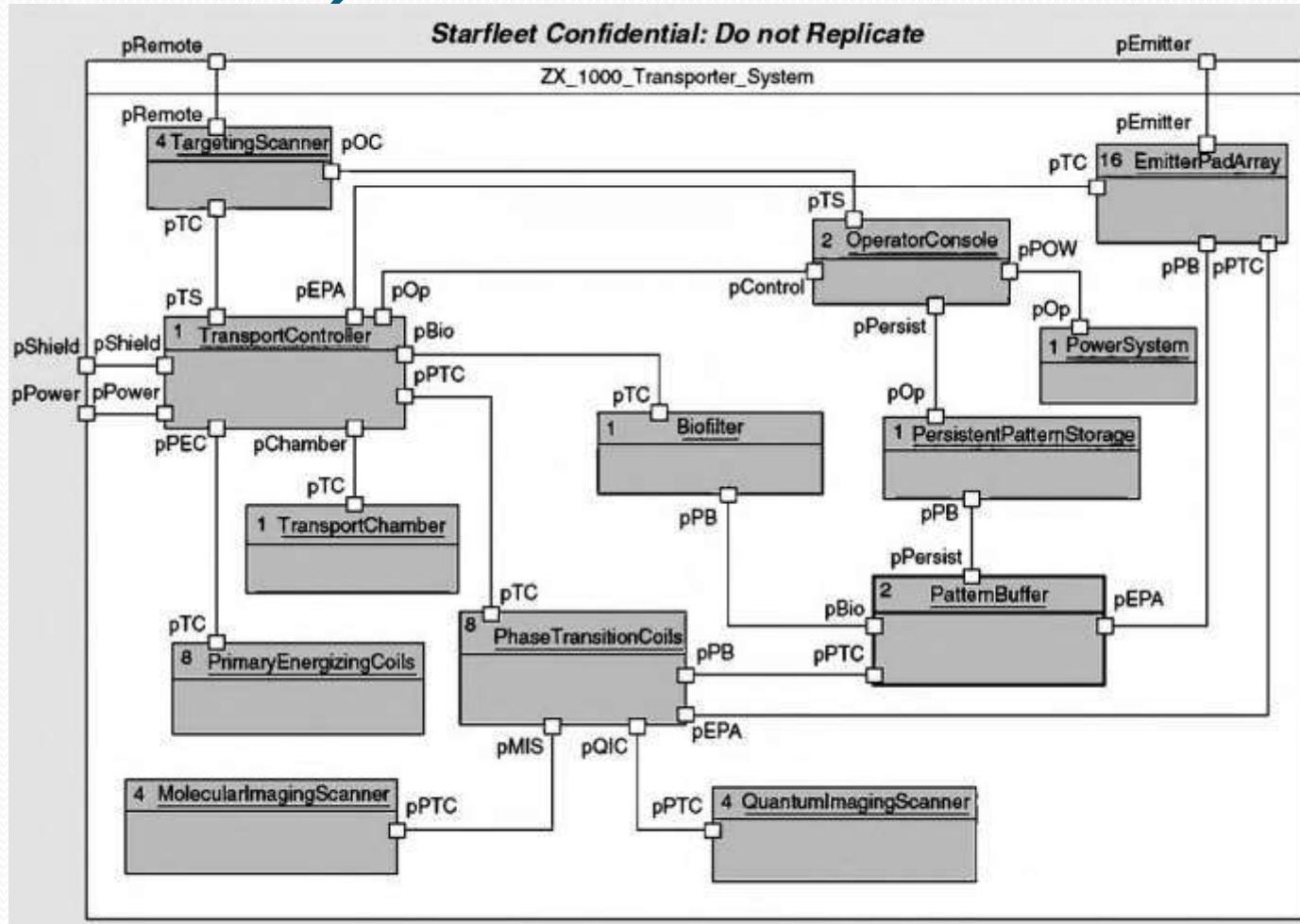
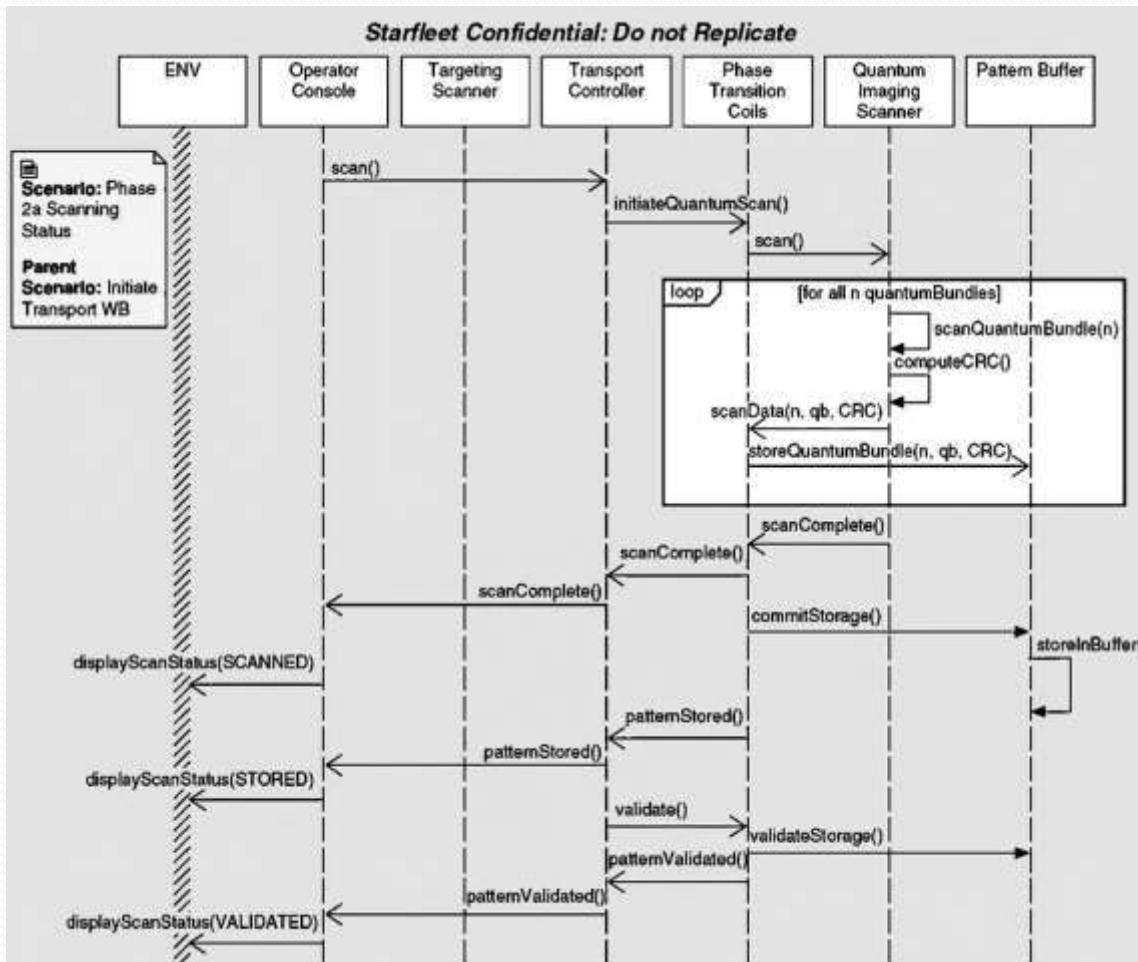


Figure 5.17
Subsystem use cases.

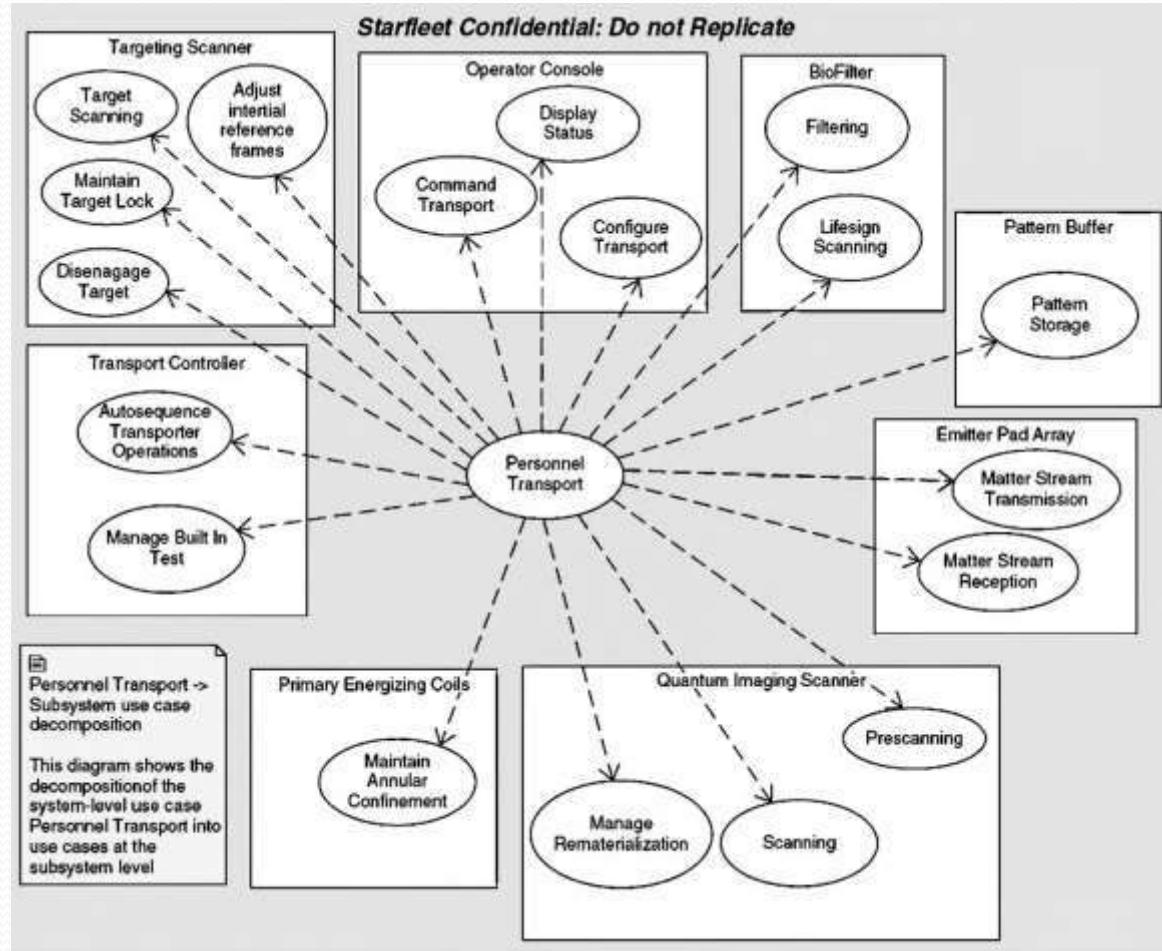
Teleportacijos sistemos pvz.: posistemiu architektūra



Teleportacijos sistemos pve. posistemų elgsena („iš apačios į viršų“)



Teleportacijos sistemos puz.: vartojimo atvejų dekompozicija („iš viršaus į apačią“)



Diegimo (deployment) architektūra: UML deployment diagrama

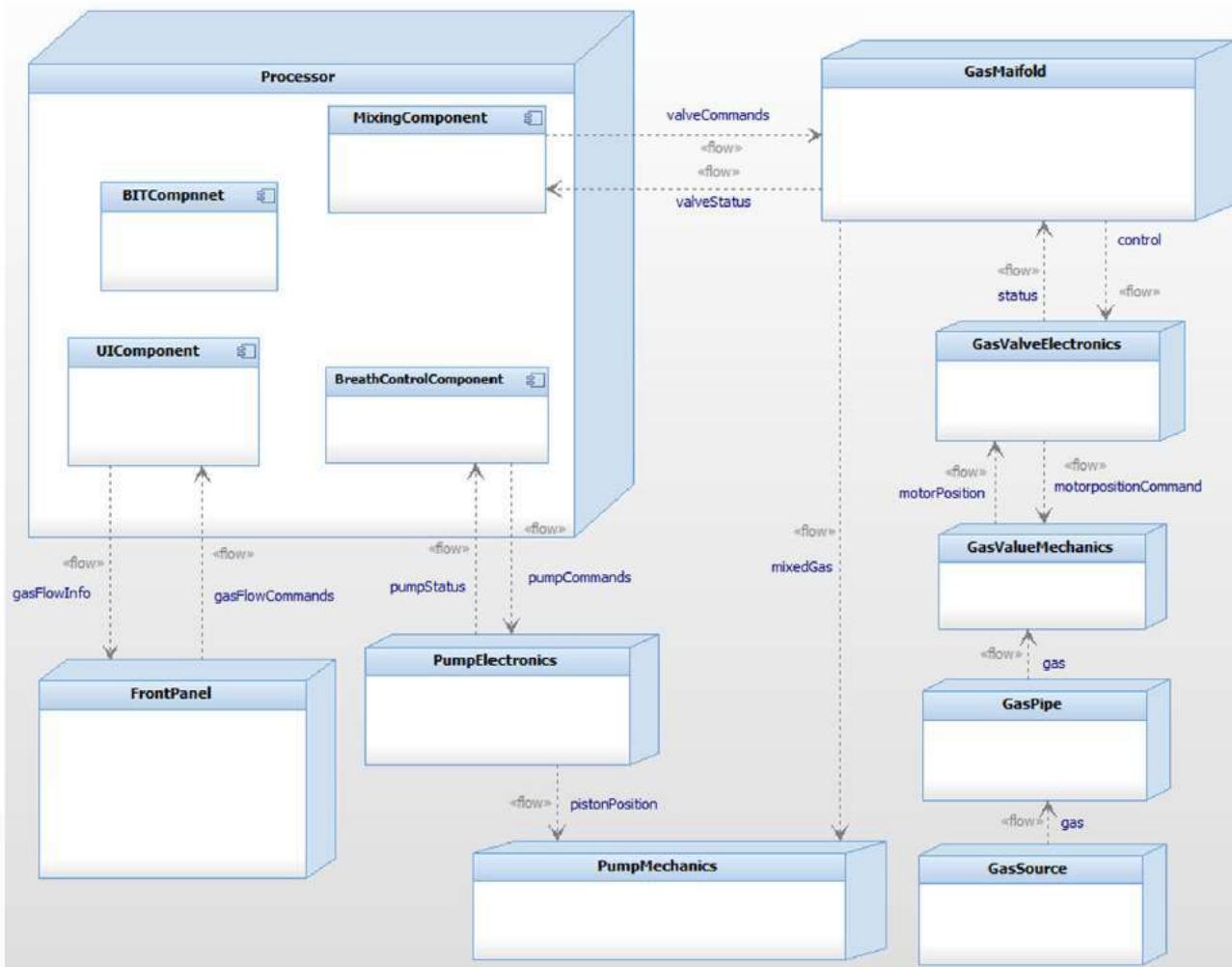


Figure 5.27
Deployment diagram for a patient ventilator.

Diegimo architektūra: UML komponentų diagrama

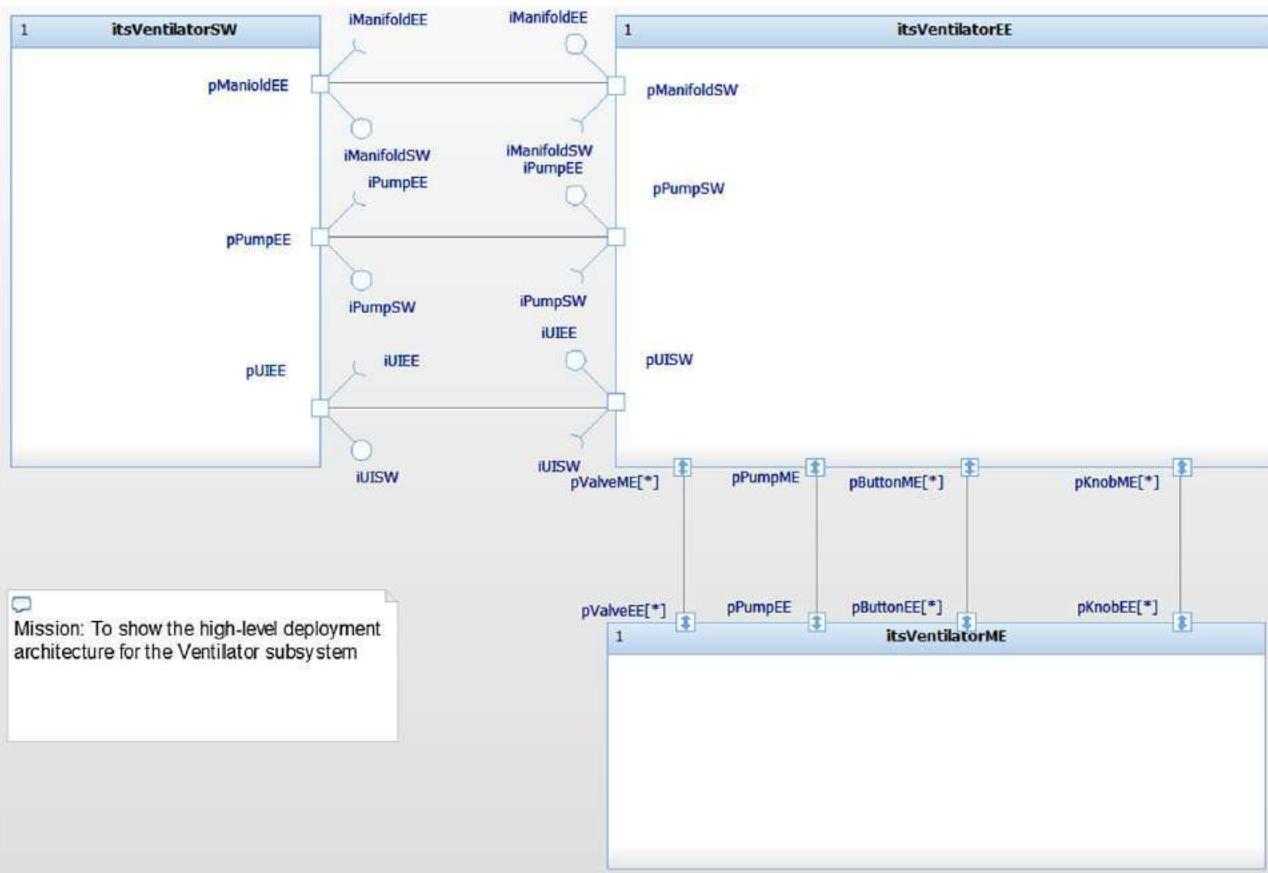


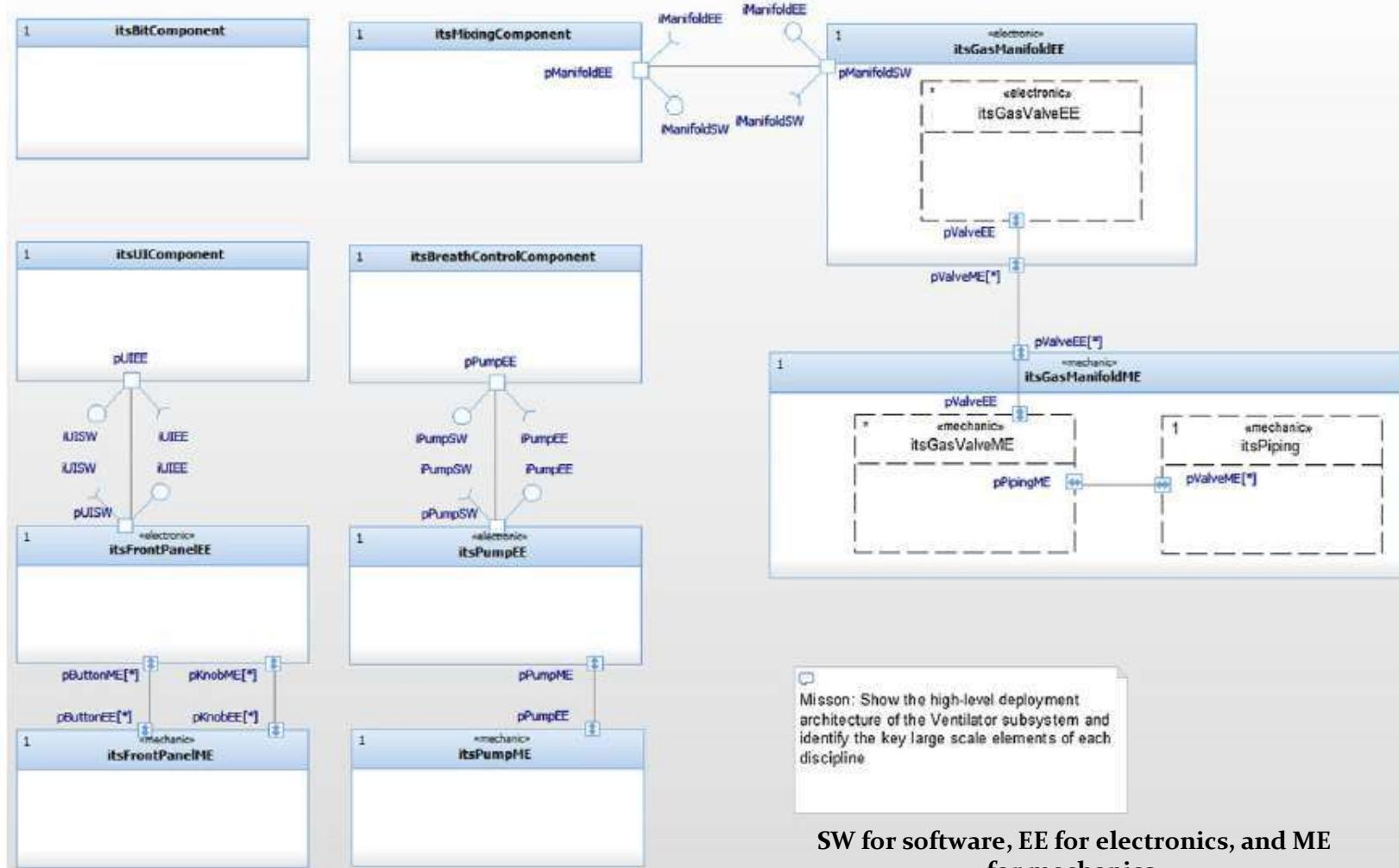
Figure 5.28
Ventilator high-level deployment architecture.

SW for software, EE for electronics, and ME for mechanics.

Reikalavimų dekomponavimas pagal inžinerijos disciplinas/sritis

- **The volume of each breath delivered by the ventilator shall be settable in 1 ml increments from 100 ml to 1200ml, delivered with an accuracy of +/- 0.5 ml.**
 - (Software) SW shall command the breath control pump in a range of 100 to 1200 ml, in increments of 1 ml.
 - (Software) SW shall command the pump to follow the standard volume curve updating the pump speed at least every 100ms.
 - (Electronics) The electronic pump control shall command the pump piston to a new commanded position within 1ms.
 - (Mechanical) The gas manifold shall deliver mixed gas continuously to the pump.
 - (Mechanical) The pump piston shall complete its response to an electronic signal to change position within 10ms with a positional accuracy of at least 5%.

Diegimo architektūra: UML detalii komponentų diagramma



Patikimumo architektūra (Dependability architecture)

- Dependability refers to the confidence with which we can entrust our lives to automated systems.
- Dependability has three primary aspects:
 - Safety;
 - Reliability;
 - Security.

Patikimumas (reliability)

- *Reliability* is a measure of the “uptime” or “availability” of a system
 - probability that a computation will successfully complete before the system fails.
 - It is normally estimated with mean time between failure (MTBF) or a related measure known as *availability*.
- **Redundancy** is one design approach that increases availability, because if one component fails, another takes its place.
- Redundancy only improves reliability when the failures of the redundant components are independent.

Saugumas (safety)

- *Safety* is very different from reliability, but a great deal of analysis affects both safety and reliability.
- A safe system is one that does not incur too much risk of loss, either to persons or equipment.
- A *hazard* is an undesirable event or condition that can occur during system operation.
- *Risk* is a quantitative measure of how dangerous a system is and is usually specified as:
 - $Risk = Hazard_{severity} * Hazard_{likelihood}$

Incidentas, avarija (fault)

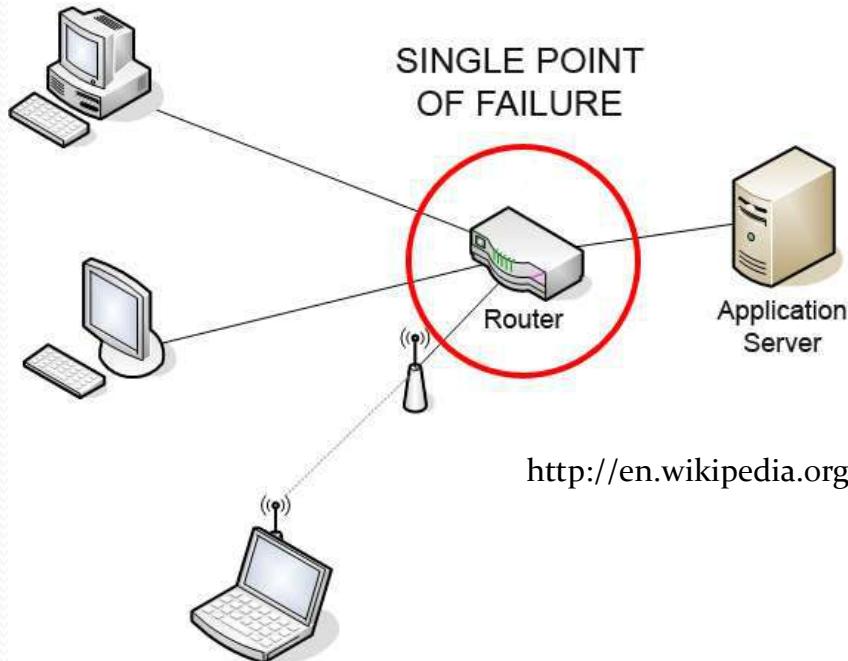
- Faults come in two flavors: ***errors*** and ***failures***.
- ***Errors*** are systematic faults introduced in analysis, design, implementation, or deployment.
 - By “systematic,” we mean that the error is *always* present, even though it may not be always be manifest.
- In contrast, ***failures*** are random errors that occur when something breaks.
- Hardware exhibits both errors and failures, but software exhibits only errors.
- The distinction between error and failure is important because different design patterns optimize the system against these concerns differently.

Gedimai (failure)

- Gedimas apibrėžia situaciją, kai iki tol teisingai veikusi sistema staiga nustoja veikti korektiškai.
- Programinė įranga negenda!
 - Ji arba būna su defektais (klaidomis), arba ne.
 - Prisimenam gynybinio kūrimo principus!
- Techninė įranga genda dėl:
 - Projektavimo klaidų;
 - Dėl susidėvėjimo ar išorinių ir vidinių poveikių.

Single point of failure

- Tai sistemos dalis, kuriai sugedus, nustoja veikti visa sistema.
- Kritinėse sistemose tokiu vietų neturėtų būti.



http://en.wikipedia.org/wiki/File:Single_Point_of_Failure.png

Safe vs Reliable

- Šaunamasis ginklas – dažnai labai patikimas, bet labai nesaugus.
- Sugedės (nevažiuojantis) automobilis – labai saugus, bet absoliučiai nepatikimas ☺
- Saugumas gali prieštarauti patikimumui, ypač jei sistema turi „fail-safe“ būseną, t.y. yra atjungiamas ar apribojamas jos funkcionalumas, kad ji nesukeltų neigiamų padarinių aplinkai.

„Fail-safe“ būsenos ir jų tipai

- Off būsena:
 - Emergency stop – iškart atjungiam maitinimą;
 - Production stop – išjungiam, kai tik pabaigiam einamąją užduotį;
 - Protection stop – išjungiam iškart, bet ne maitinimą;
- Partial shutdown – apribojam funkcionalumą;
- Hold – jokio funkcionalumo, tik automatinės apsaugos procedūros;
- Manual/ External control – sistema funkcionuoja tik pagal išorines komandas;
- Restart – sistema perkraunama.

Fail-safe būsenų pavyzdžiai

- Lėktuvo variklio išjungimas (off);
- Raketos nešėjos susprogdinimas (hold);
- Medicinos prietaisų išjungimas ir aliarmas (emergency stop);
- Medicinos prietaisų pervedimas tik į stebėjimo režimą (partial shutdown);
- Robotas baigia savo užduotis prieš atsijungdamas, kai žmogus įeina į jo veikimo (pavojingą) zoną (production stop).

Pertekliškumas (redundancy)

- The key to managing both *safety* and *reliability* is **redundancy**.
- Redundancy improves reliability because it allows the system to continue to work in the presence of *faults*.
- For improving safety
 - additional elements are needed to monitor the system to ensure that it is operating properly
 - and possibly other elements are needed to either shut down the system in a safe way or take over the required functionality.
- The goal of redundancy used for safety – to ensure that there is no loss (to either persons or equipment).

Pertekliškumo tipai

- Homogeninis – naudojami pilnai identiški papildomi kanalai.
- Heterogeninis – tai ne paprastas kanalo klonas, bet kitokia architektūra ir sprendimai (pvz. vienas kanalas pagrįstas PID logika, o kitas – miglota (fuzzy) logika).

Homogeninis vs heterogeninis pertekliškumas

- Homogeninis apsaugo tik nuo atsitiktinių klaidų.
- Heterogeninis apsaugo papildomai ir nuo sisteminių klaidų (projektuotojo ar realizuotojo įneštų).
 - Visi pertekliniai kanalai gali turėti gedimų, tačiau tai nebus tie patys gedimai, todėl kanalai **nesuges tuo pat metu ar tokiu pat būdu**.
- Programinės įrangos pertekliškumas:
 - Duomenų;
 - Valdymo;
 - Duomenų ir valdymo.

Saugumo užtikrinimo būdai (1)

- **Išvengimas** – sudaromos tokios sąlygos, kad pavojus niekad neatsirastų;
- **Apmokymai** – sistemos vartotojai apmokomi, kad nesukeltų kritinių situacijų;
- **Aliarmai** – vartotojams pranešama apie kilusį pavojų, kad jie imtusi veiksmų;
- **Blokavimas** – pavojus pašalinamas, panaudojant papildomus (antrinius) įrenginius ar logiką;

Saugumo užtikrinimo būdai (2)

- **Vidinis patikrinimas** – sistemoje realizuoti patikrinimai, leidžiantys nustatyti, kad kilo pavojus;
- **Speciali ekipiruotė** – vartotojai dėvi specialią aprangą (pirštinės, akiniai ir t.t.).
- **Prieigos apribojimai** – tik autorizuoti vartotojai gali dirbti su potencialiai pavojingom sistemos dalimis.
- **Žymėjimas/Perspėjimai** – pavojų įvardija specialūs užrašai, pvz. „Neliesk – nutrenks ☺“

Sauga (security)

- **Security** is the degree of resistance to, or protection from, harm (<https://en.wikipedia.org/wiki/Security>):
- Kompiuterinių sistemų sauga (kibernetinis saugumas):
 - includes hardware, software, data, people, and procedures by which digital equipment, information and services
 - are protected from unintended or unauthorized access, change or destruction, and is of growing importance due to the increasing reliance of computer systems in most societies.
 - It includes physical security to prevent theft/damage of equipment and information security to protect the data on that equipment.

Projektavimo procesas

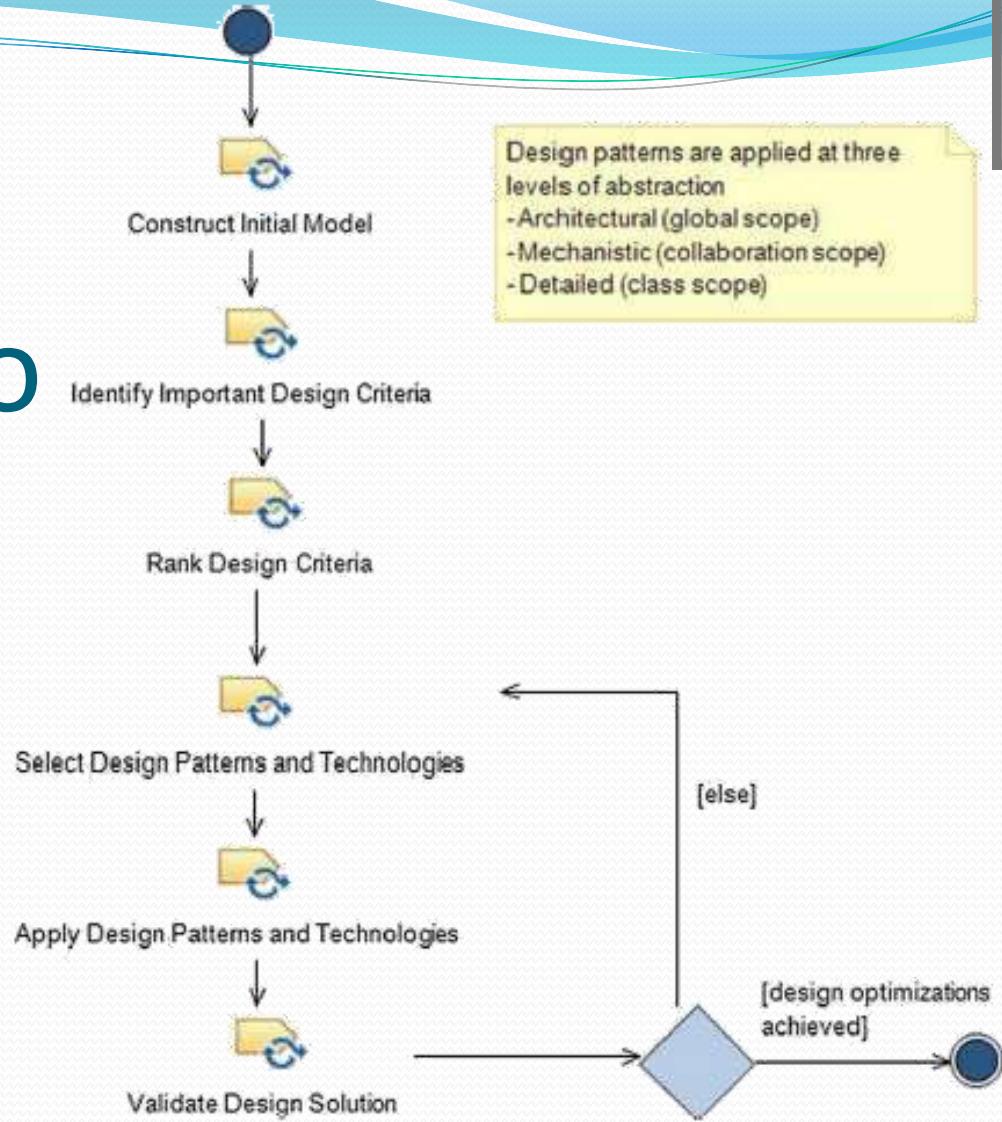


Figure 6.1
Basic Harmony design workflow.

Projektavimo etapų paaiškinimai

(1)

- Pradinio modelio sudarymas – tai analizės rezultatas (jau turim).
- Svarbių kriterijų identifikavimas, ištraukiant vieną ar kelis aukščiau aptartus galimus kriterijus.
- Kriterijų rangavimas leidžia nustatyti, kurie jų yra kritiskiausi/svarbiausi.

Projektavimo etapų paaiškinimai

(2)

- Pagal kriterijų svarbą pasirenkami projektavimo šablonai ir technologijos (operacinės sistemos, perdavimo protokolai ir pan.), kurios leidžia optimizuoti norimus kriterijus.
- Šablonų pritaikymas vykdomas pakeičiant šablono abstrakčius elementus analizės etapo modeliu elementais.
- Validavimas apima dvi veiklas:
 - Įsitikinam, kad pritaikę šabloną, nesugadinom prieš sukurto funkcionalumo;
 - Įsitikinam, kad pavyko pasiekti užsibrėžtų kriterijų reikšmes.

Tinkamiausio projektinio sprendimo parinkimas pagal kriterijus

Table 6.1 Design Trade-off Spreadsheet.

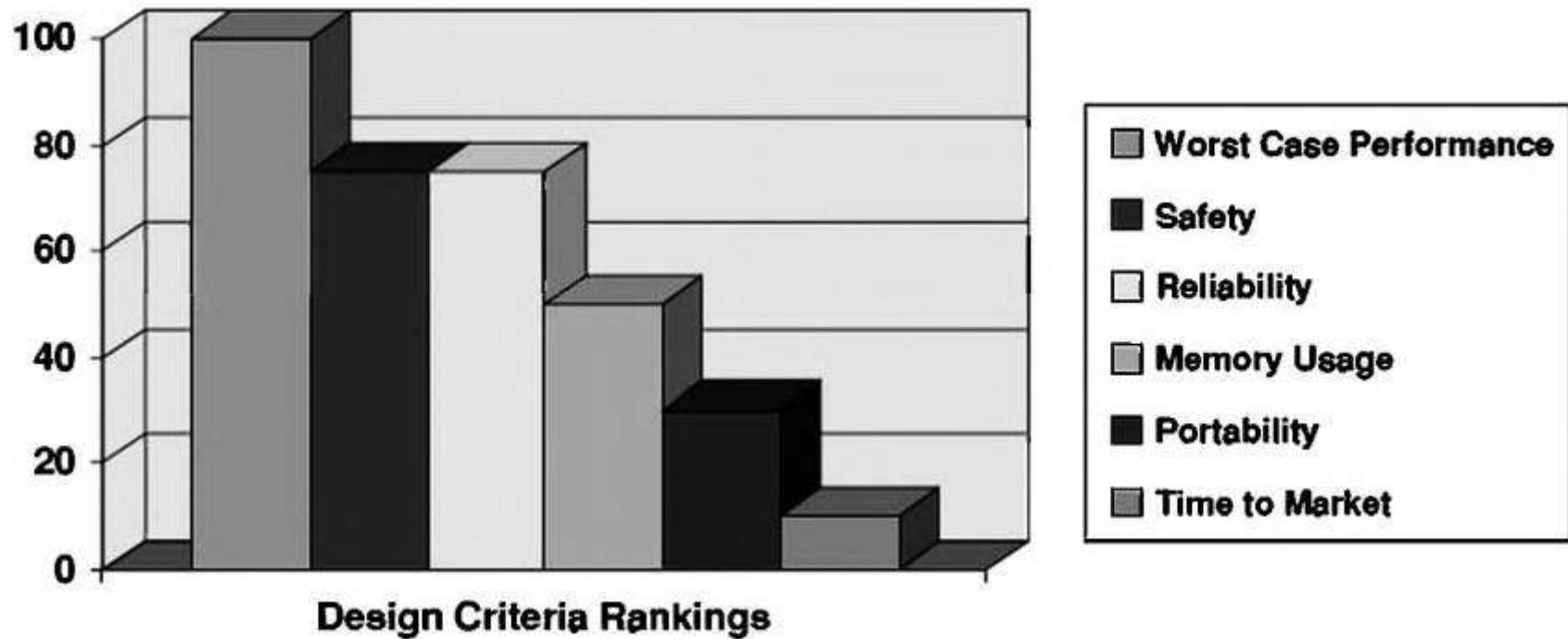
Design Solution	Design Criteria					Total Weighted Score
	Criteria 1 Weight = 7	Criteria 2 Weight = 5	Criteria 3 Weight = 3	Criteria 4 Weight = 2	Criteria 5 Weight = 1.5	
	Score	Score	Score	Score	Score	
Alternative 1	7	3	6	9	4	106
Alternative 2	4	8	5	3	4	95
Alternative 3	10	2	4	8	8	120
Alternative 4	2	4	9	7	6	84

Diegimo architektūros alternatyvų įvertinimo pvz.

	Performance	Low Power	Recurring Cost	Development Cost	Reliability	Schedule Risk	Weighted Results
Criticality	39%	13%	10%	20%	3%	15%	100%
Alternative A	100	60	100	60	100	100	86.80
Alternative B	60	100	100	30	60	100	69.20
Alternative C	60	60	100	30	60	60	58.00

Weights:	Excellent	100
	Acceptable	60
	Marginal	30
	Unacceptable	0

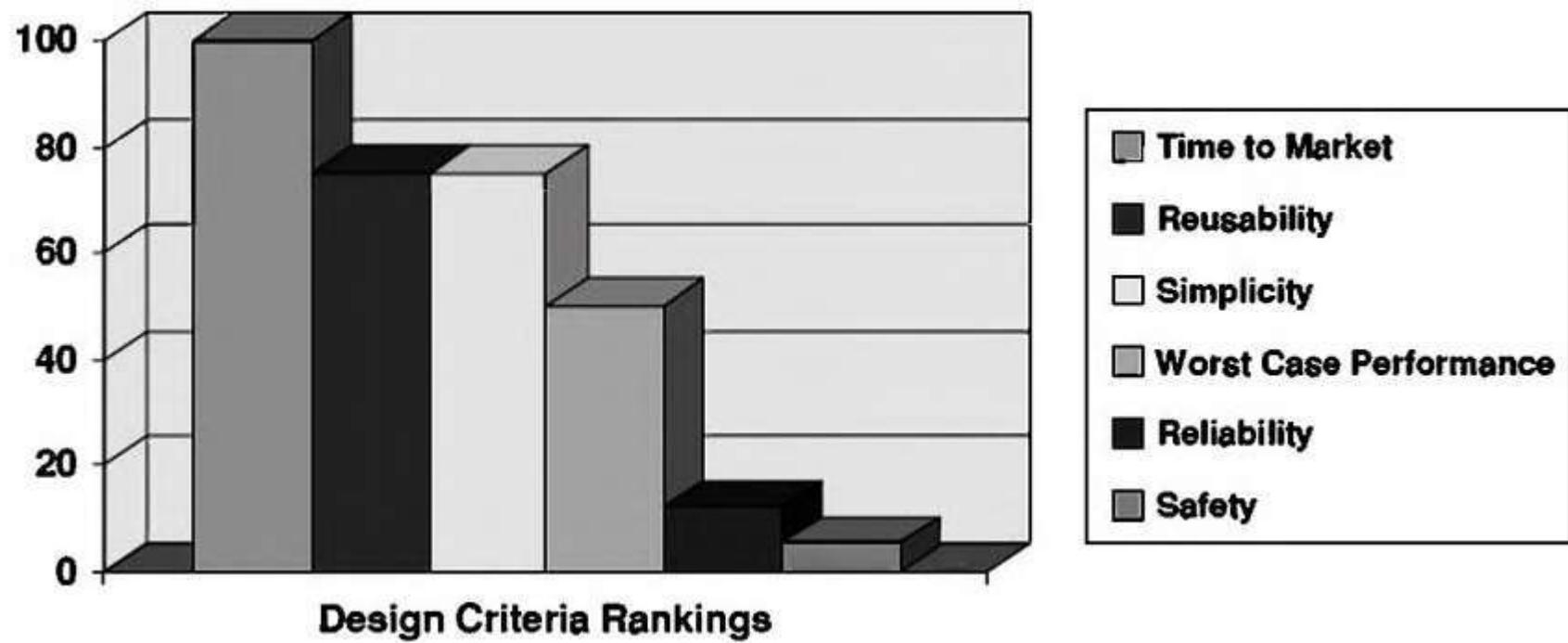
Kriterijų optimizacijos pavyzdys (1)



Kriterijų optimizacijos pavyzdys (2)

- Galimi šablonai:
 - Static Priority Pattern
 - Geras atsako į įvykius laikas, bet sudėtingas, sunkiai nuspėjamas.
 - Fixed Block Memory Allocation Pattern
 - Didina patikimumą, nes nėra atminties fragmentacijos, tačiau pačios atminties reikia daugiau.
 - Channel Pattern
 - Užtikrina patikimumą, panaudojant pertekliškumo principus.
 - Tripple Modular Redundancy (TMR) Pattern
 - Naudoja 3 kanalus ir balsavimo mechanizmą patikimumui užtikrinti, bet sprendimas brangus, naudoja daug energijos, išskiria šilumą, didelis svoris.

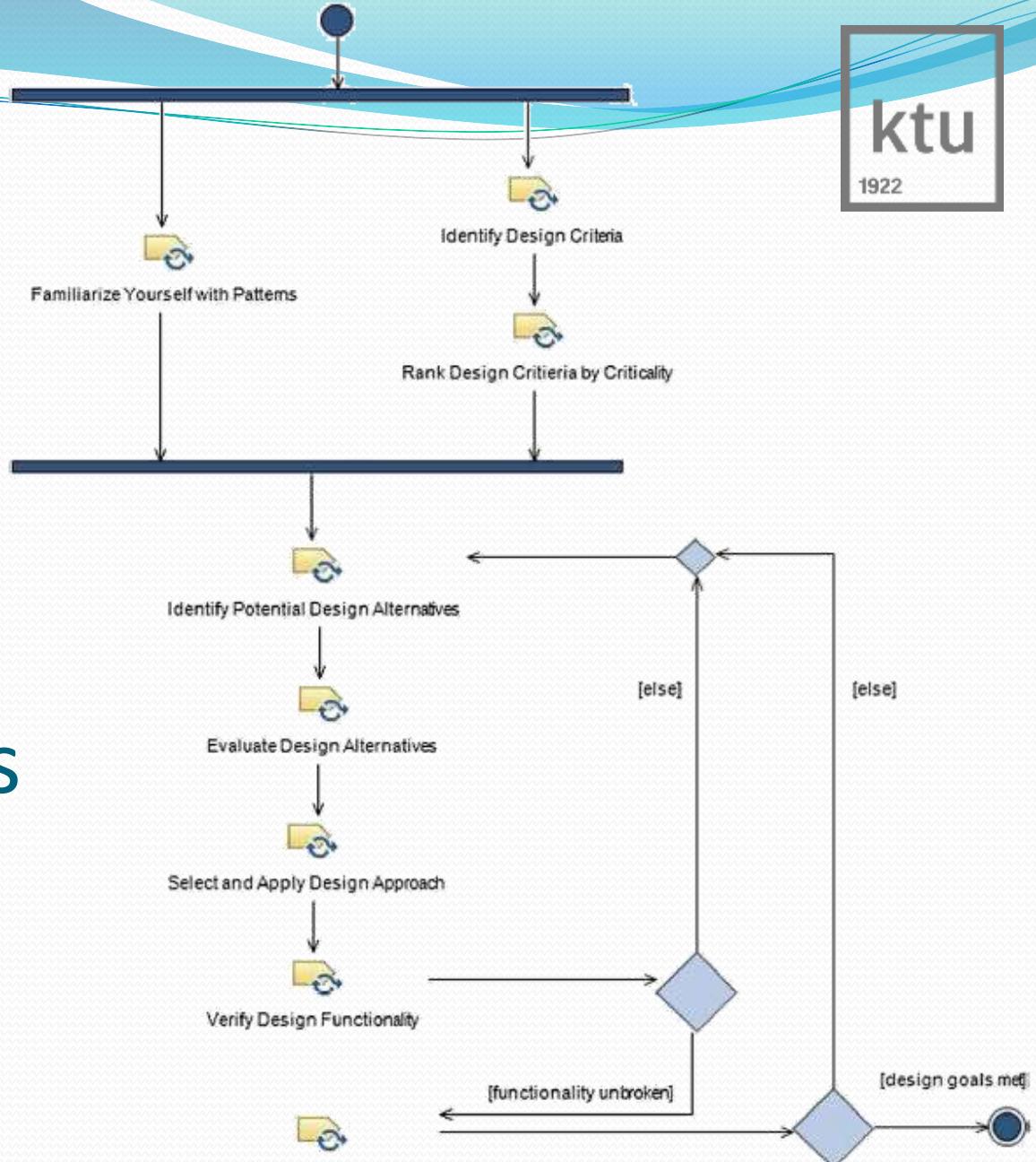
Kriterijų optimizacijos pavyzdys (3)



Kriterijų optimizacijos pavyzdys (4)

- Galimi šablonai:
 - Cyclic Executive Scheduling Pattern
 - Lygiagrečių procesų šablonas užtikrina nuspėjamumą ir patikimumą, bet létai reaguoja į įvykius.
 - Dynamic Memory Pattern
 - Automatiškai užtikrina šiuolaikiniai kompiiliatoriai, bet veda prie atminties klaidų ir fragmentavimo. Garantuoja paprastumą, bet ne patikimumą.
 - Recursive Containment Pattern
 - Užtikrina paprastą rekursyvią sistemos architektūrą, tačiau sunkiai dera su pertekliškumu.
 - Virtual Machine Pattern
 - Užtikrina pernešamumą, bet aukoja našumą.

Projektavimo šablonų naudojimo rekomendacijos



Šaltinis: Bruce Powel Douglass. Real-time UML workshop for embedded systems. Newnes; 2 edition (March 12, 2014), 576 p. ISBN13: 9780124077812

Figure 6.2
Pattern hatching.

Saugumo architektūra (Safety architecture)

- Specifies how the architecture **identifies, isolates, and corrects faults** at runtime with the goal of **preventing harm**.
- Assessing the adequacy of the design solution is done with safety analysis (such as fault tree analysis (FTA)) and/or reliability analysis (such as failure mode and effects analysis (FMEA)).
- FTA supports reasoning about what combinations of events and conditions result in hazards.

Pavojų identifikavimas ir analizė

Hazard	Fault	Severity (1 (low)-10 (high)		Likelihood (0.0-1.0)	Computed Risk	Time Units	Tolerance Time	Detection Time	Control Measure	Control Action Time	Exposure Time	Is Safe?
Hypoventilation	Breathing tube disconnect	10	0.2	2	minutes	5	0.5	Blood oxygen sensor	2	2.5	TRUE	
Hypoventilation	Ventilator timer error	10	0.2	2	minutes	5	0.5	Independent pressure sensor with alarming	2	2.5	TRUE	
Hypoventilation	Gas supply failure	10	0.4	4	minutes	5	0.05	Ventilator incoming gas pressure sensor	2	2.05	TRUE	
Hypoxia	Gas mixer failure	10	0.6	6	minutes	5	0.05	Inspiratory limb O ₂ sensor	2	2.05	TRUE	
Hyperventilation	Ventilator timer error	8	0.1	0.8	minutes	20	0.5	Blood oxygen sensor	2	2.5	TRUE	
Overpressure	Pump failure; expiratory tube blockage	10	0.3	3	ms	200	10	Secondary pressure sensor with autorelease valve	5	15	TRUE	

Exposure time = Detection Time + Action Time.

"Is Safe" is computed as = (Exposure Time <= Tolerance Time).

Klaidų, vedančių prie pavojų, identifikavimas: FTA (Fault Tree Analysis) (1)

	An event that results from a combination of events through a logic gate		A condition that must be present to produce the output of a gate
	A basic fault event that requires no further development		Transfer
	An "undeveloped fault" event not elaborated because the event is trivial or more decomposition is not necessary		AND gate
	An event that is expected to occur normally		NAND Gate
	NOT Gate		OR Gate
			NOR Gate
			XOR Gate

Figure 6.3
FTA notational elements.

Klaidų, vedančių prie pavojų, identifikavimas: FTA (Fault Tree Analysis) (2)

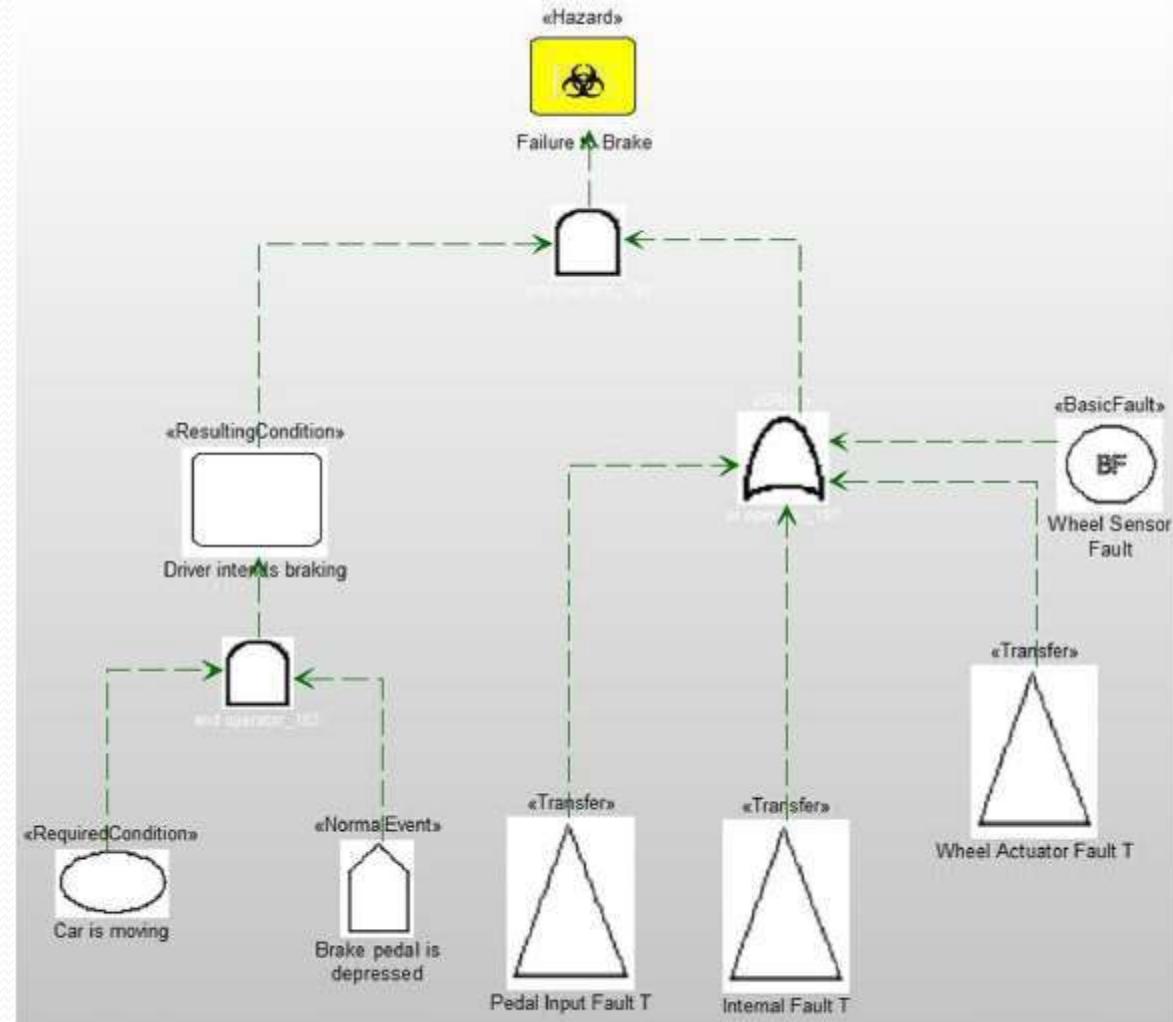
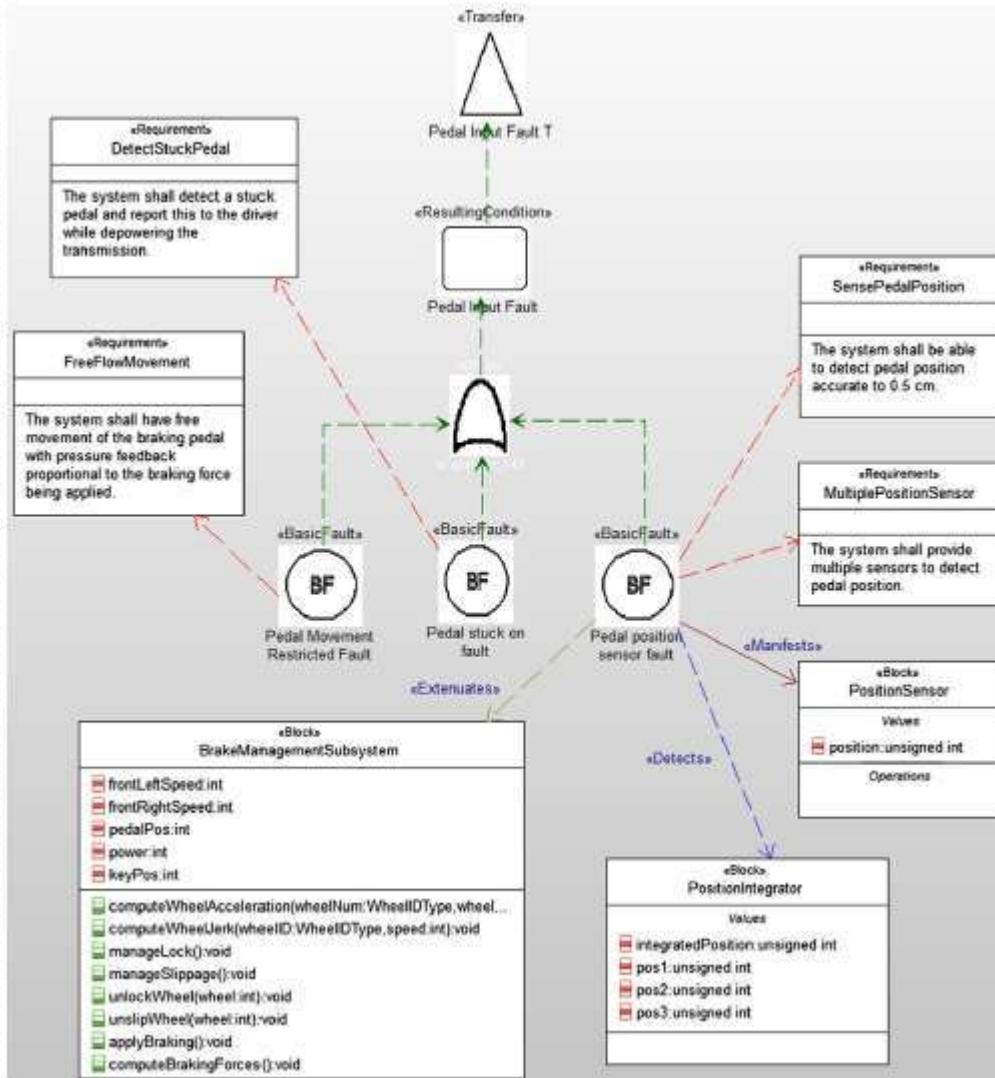


Figure 6.6
Failure to Brake FTA.

Klaidų, vedančių prie pavojų, identifikavimas: FTA (Fault Tree Analysis) (3)



Klaidų, vedančių prie pavojų, identifikavimas: FTA (Fault Tree Analysis) (4)



- We can make the hazard **less likely** to occur by adding multiple sensors;
 - only if all of the associated sensors fail does the depression of the brake pedal fail to be detected.
- We can make the hazard **less severe** by adding seat belts and an air bag.
- Both address the hazard risk but do so in different ways.
- The advantage of the FTA is that it allows us to reason about where to add safety measures and their effectiveness.

Kanalo (channel) projektavimo šablonas saugumui ir patikimumui užtikrinti

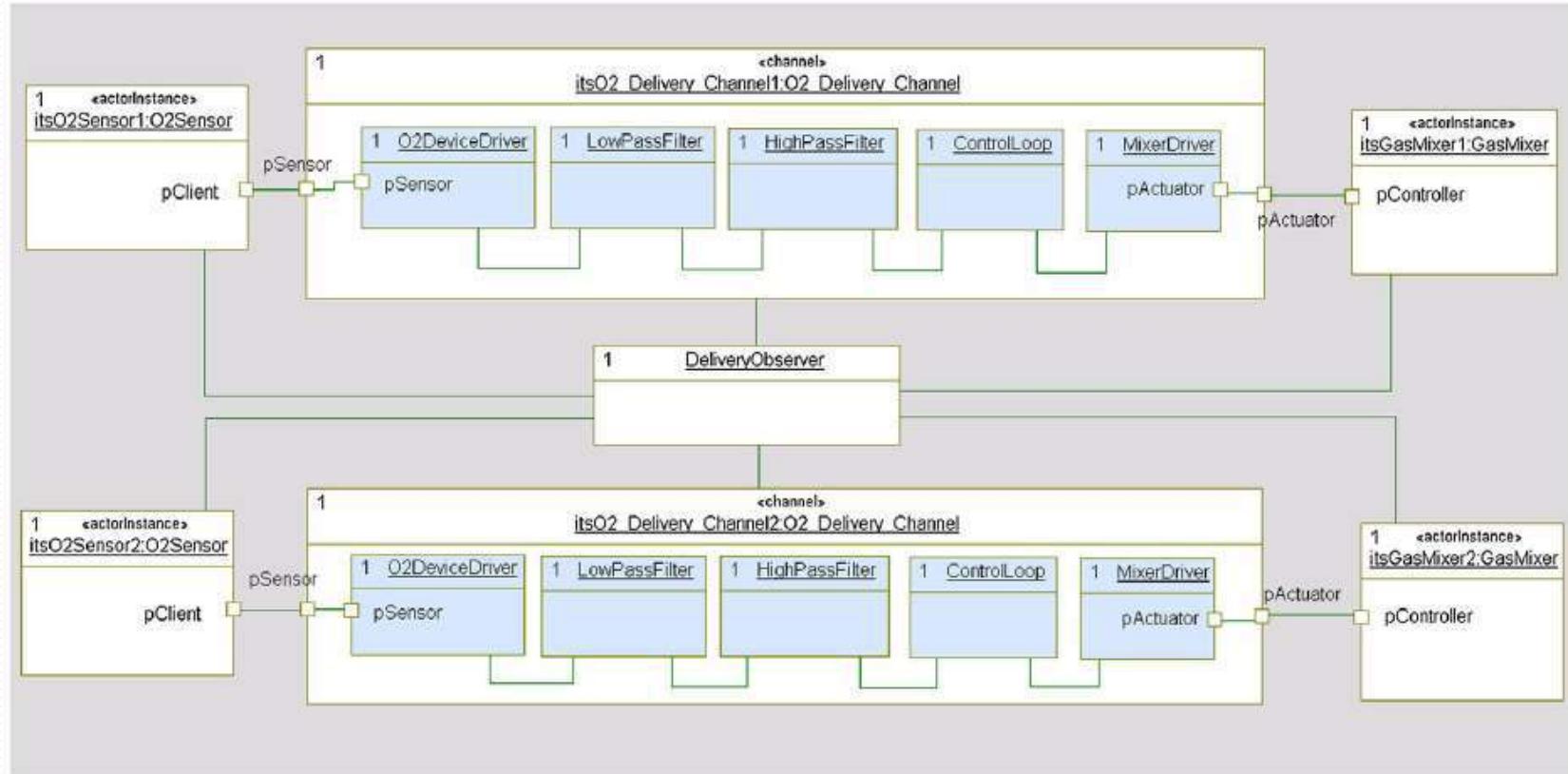


Figure 6.9
Example of channel architecture pattern.

Stebėtojo-vykdklio (Monitor-actuator) projektavimo šablonas

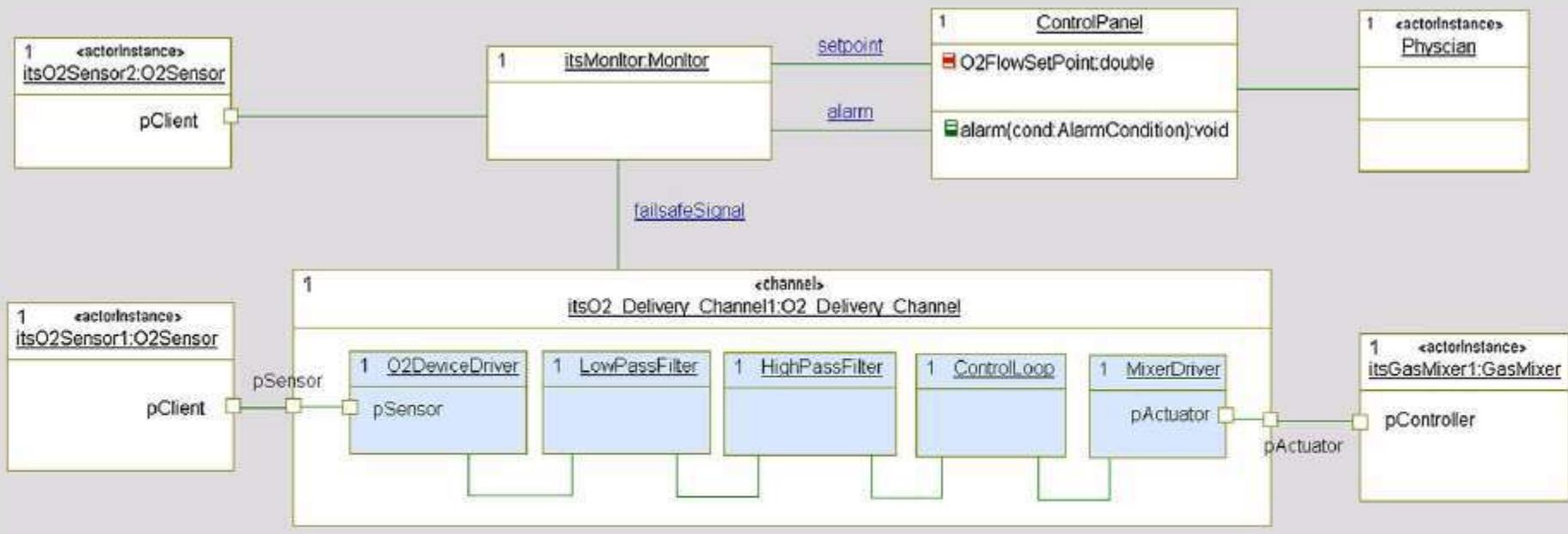


Figure 6.11
Example monitor-actuator pattern.

Patikimumo (reliability) architektūra

- Refers to the availability of system services.
- **Reliability analysis identifies the requirements for availability of system services.**
 - This results in reliability specifications at the component level.
- Reliability is improved through a combination of two design approaches:
 - Individual components may be engineered to be individually reliable by choosing more robust materials, algorithms, and methods, even if they are more expensive.
 - Introducing component redundancy so that if a component fails there are other components ready to deliver the required component services.

Failure mode and effects analysis (FMEA)

Failure Mode and Effects Analysis

Service / Function	Failure Mode	Faults	Failure Effects	Pre-action				Post-action						
				Likelihood (1=impossible, 10=certain)	Severity (1=no effect, 10=catastrophe)	Detectability (1=certain, 10=no detection)	RPN (= sev * likely * detect)	Existing Control Measures	Recommendations	Responsible	Actions	Likelihood	Severity	RPN
Pedal nonresponsive	Pedal stuck	Pedal stuck	Pedal doesn't move; no braking action occurs	3	9	10	270	none	Make pedal assembly self-lubricating	Joe	Added sealed piston with lubrication	2	9	180
		Pedal position sensor fails	Pedal depresses; no braking occurs	4	9	8	288	Start up comm check with sensor	Use three pedal position sensors	Susan	Added two more sensors with voting	2	9	144
		Braking						Continuous						

Saugos (Security and Information Assurance) architektūra: FTA

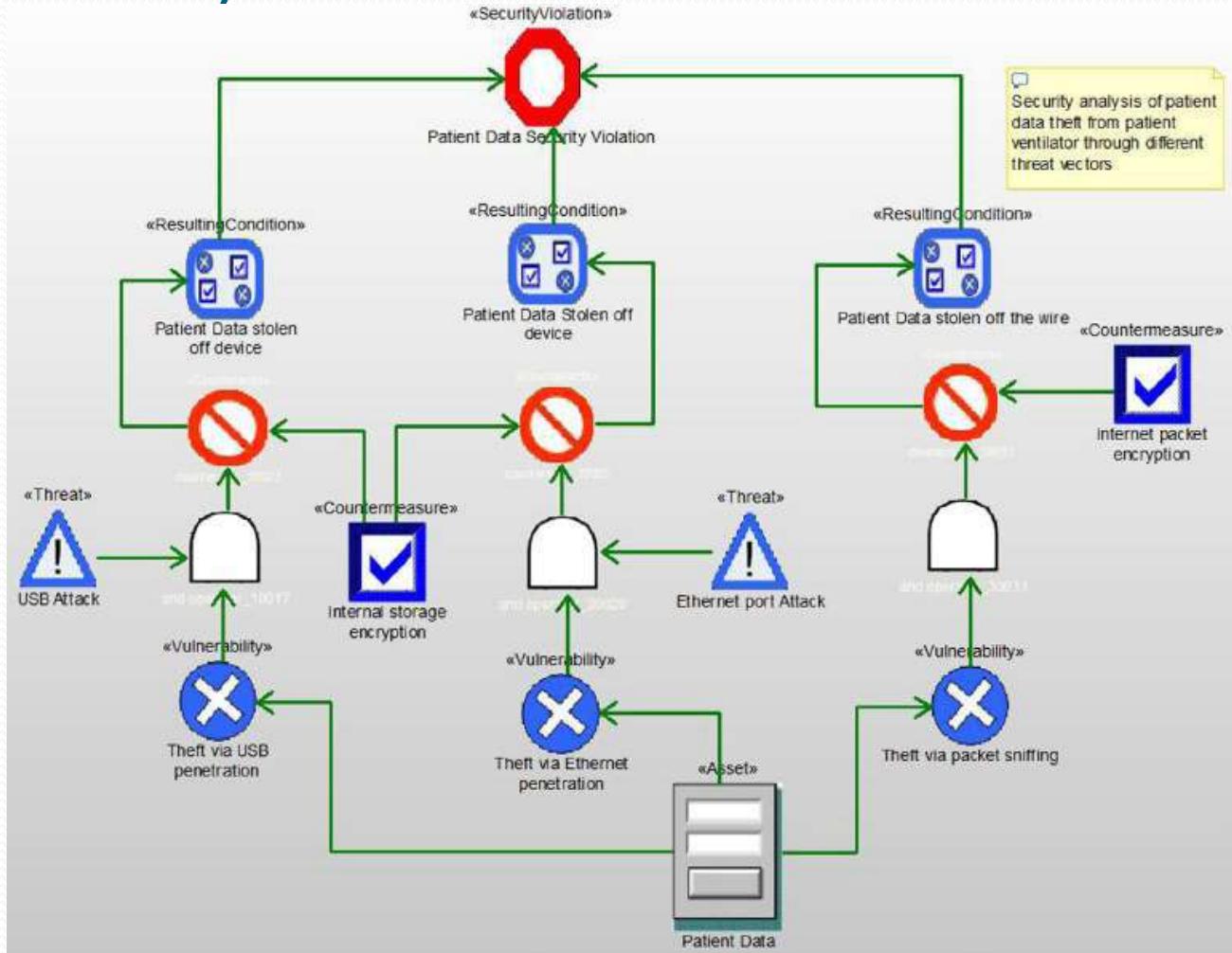


Figure 6.18

Security analysis of patient data theft.

Saugos (Security and Information Assurance Architecture) architektūra: grėsmių analizė

Threat Analysis Table

Asset value is the value of the asset to be protected (1=very low, 10=very high).										
Likelihood is the probability of the attack (1=very low, 10=certain).										
Reproducibility refers to how easy it is to reproduce the attack (for example, does it depend on timing or other circumstances?) (1=hard, 10 = very easy).										
Exploitability refers to how easy it is to launch the attack (1=very easy, 10=very hard).										
Breadth is the a measure of the extent of the attack. How widespread is it or how many systems are affected? (1=few, 10=very many).										
Discoverability is how easy is it for outsiders to find out about and exploit the vulnerability (1=very easy, 10=very hard).										
Threat Priority is the product of the above values and is used to prioritize the threats for countermeasures.										
			These are in the range of 1 -10							
Asset	Vulnerability	Threat Vector	Asset Value	Likelihood of attack	Reproducibility	Exploitability	Breadth	Discoverability	Threat Priority	Countermeasure
Patient Demographic Data	Access via Ethernet	Input validation weak	4	7	9	4	1	9	9072	Internal encryption
	Access via USB	Auto-execution of USB SW	4	7	9	3	1	9	6804	Internal encryption
	Access via packet snooping	Messages sent in plain text	4	7	9	5	1	8	10080	Message encryption

Figure 6.19
Threat analysis table.

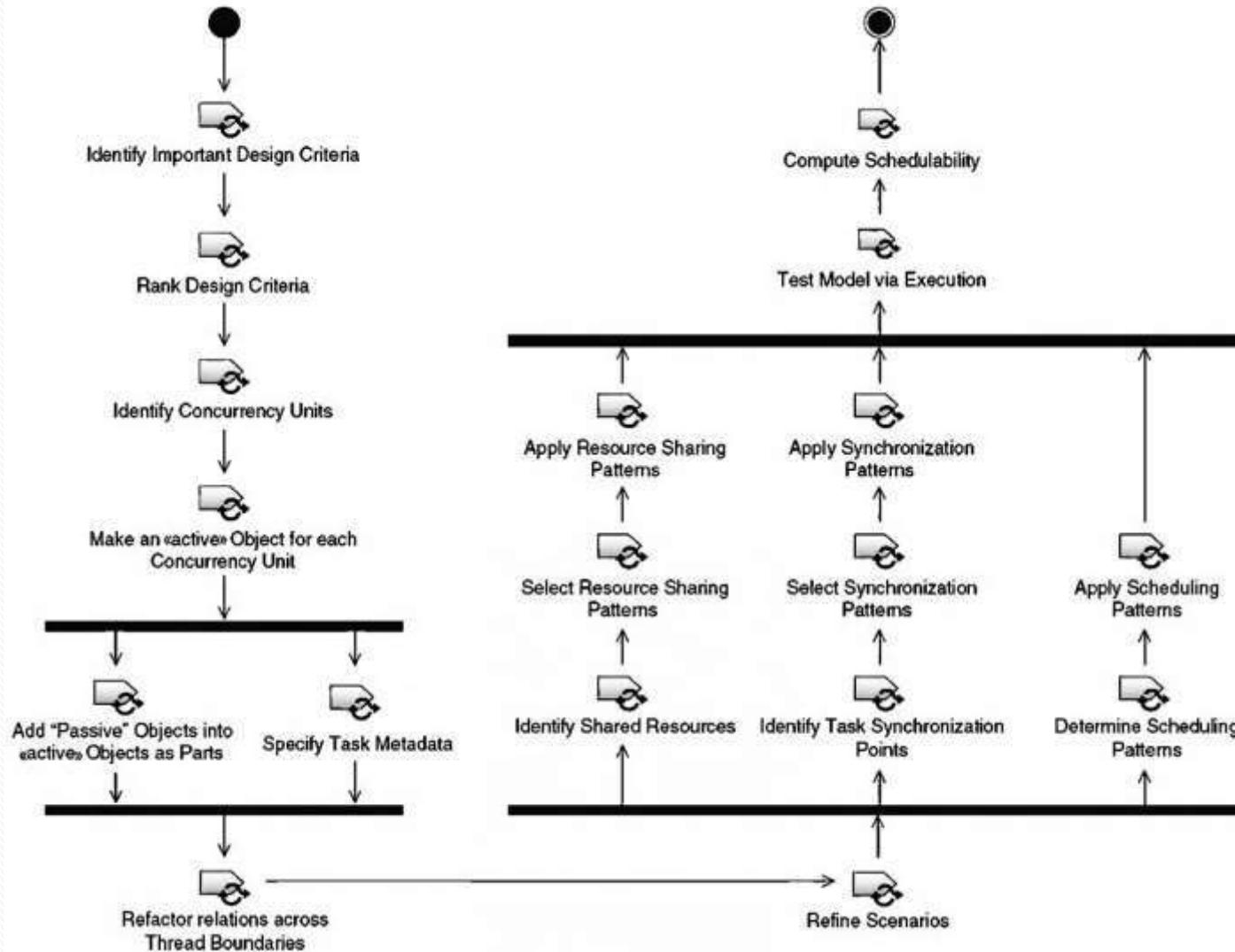
Paskirstymo architektūros optimizavimas

- Etapai:
 - Optimizacijos kriterijų pasirinkimas;
 - Žinučių srauto apkrovimo analizė;
 - Fizinės žinučių perdavimo terpės identifikavimas;
 - Technologijų ir šablonų parinkimas:
 - Šablonai, optimizuojantys žinučių perdavimą;
 - Protokolai, tinkantys fizinei aplinkai;
 - Tarpinė programinė įranga (angl. middleware);
 - Eksperimentinis įvertinimas ir analizė;
 - Patobulinimai;
 - Validavimas.

Protokolų pasirinkimo alternatyvos

- Kodėl Ethernet netinka didelio apkrautumo tinklams?
- Kodėl CAN protokolas tinkta tik trumpoms žinutėms?
- Kokie IPC (angl. inter-process communication) ir RPC (angl. remote procedure call) metodai tinkamiausi?
- Kodėl programuotojai nemégsta CORBA?
- Kodėl Web servisai reikalauja didelio duomenų pralaidumo?

Lygiagretumo ir resursų valdymo architektūros optimizavimas



Lygiagrečiai veikiančių komponentų (procesų, gijų) naudojimo strategijos (1)

Strategy	Description	Pros	Cons
Single event groups	Use a single event type per task	Very simple threading model	Doesn't scale well to many events; suboptimal performance
Interrupt handler	Use a single event type to raise an interrupt	Simple to implement for handling a small set of incoming event types; highly efficient in terms of handling urgent events quickly	Doesn't scale well to many events; interrupt handles typically must be very short and atomic; possible to drop events; difficult to share data with other tasks
Event source	Group all events from a single source so as to be handled by one task	Simple threading model	Doesn't scale well to many event sources; suboptimal performance
Related information	Group all processing of information (usually around a resource) together in a task	Useful for “background” tasks that perform low-urgency processing	Same as event source

Lygiagrečiai veikiančių komponentų (procesų, gijų) naudojimo strategijos (2)

Independent processing	Identify different sequences of actions as threads when there is no dependency between the different sequences	Leads to simple tasking model	May result in too many or too few threads for optimality; doesn't address sharing of resources or task synchronization
Interface device	A kind of event source	Useful for handling device (e.g., bus) interfaces	Same as event source
Recurrence properties	Group together events of similar recurrence properties such as period, minimum interarrival time, or deadline	Best for schedulability; can lead to optimal scheduling in terms of timely response to events	More complex
Safety assurance	Group special safety and reliability assurance behaviors together (e.g., watchdogs)	Good way to add on safety and reliability processing for safety-critical and high-reliability systems	Suboptimal performance

Kokias lygiagretumo realizacijos problemas sprendžiame?

- Kaip turi būti sudaromas užduočių/gijų tvarkaraštis?
- Kaip bus dalinamasi resursais?
- Ar bus atliekama gijų sinchronizacija ir kaip ji bus atliekama?
- Egzistuoja įvairūs šablonai, kuriuos iš dalies aptarėme pirmosiose paskaitose.
- Visą tai išmoksite T12oB111 „Realaus laiko sistemos“ modulyje kitą semestrą ☺

Lygiagretumo ir resursų valdymo architektūros modeliavimas

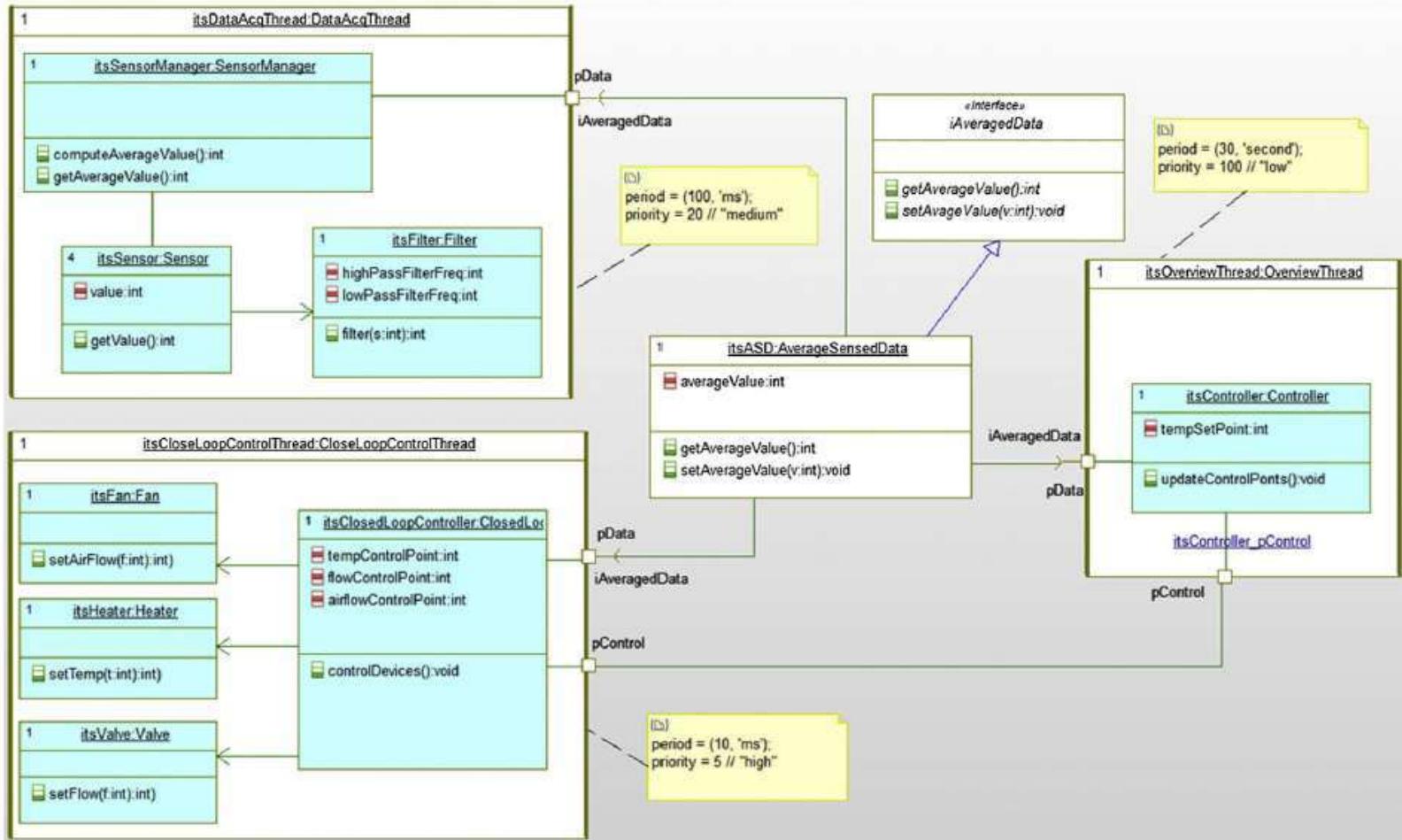


Figure 9.7
Concurrency design model.

Bendradarbiavimo ir detalusis projektavimas (Collaboration and Detailed Design)



- A *collaboration* is a set of classes and objects that work together to realize a larger scale behavior, such as the realization of a use case.
 - If there are 25 use case (“analysis-level”) collaborations, then collaboration design must be applied 25 times, once per use case.
- Detailed design applies idioms to optimize individual class features.
 - There is usually a small set of the classes – typically 3% to 5%– that require special effort.

Bendradarbiavimo projektavimas

- Tobuliname elgseną ir bendradarbiavimą tarp sistemas komponentų kiekvienu vartojimo atveju.
- Naudojam projektavimo šablonus.
- Modeliai:
 - Klasių, sekų, veiklos ir būsenų.

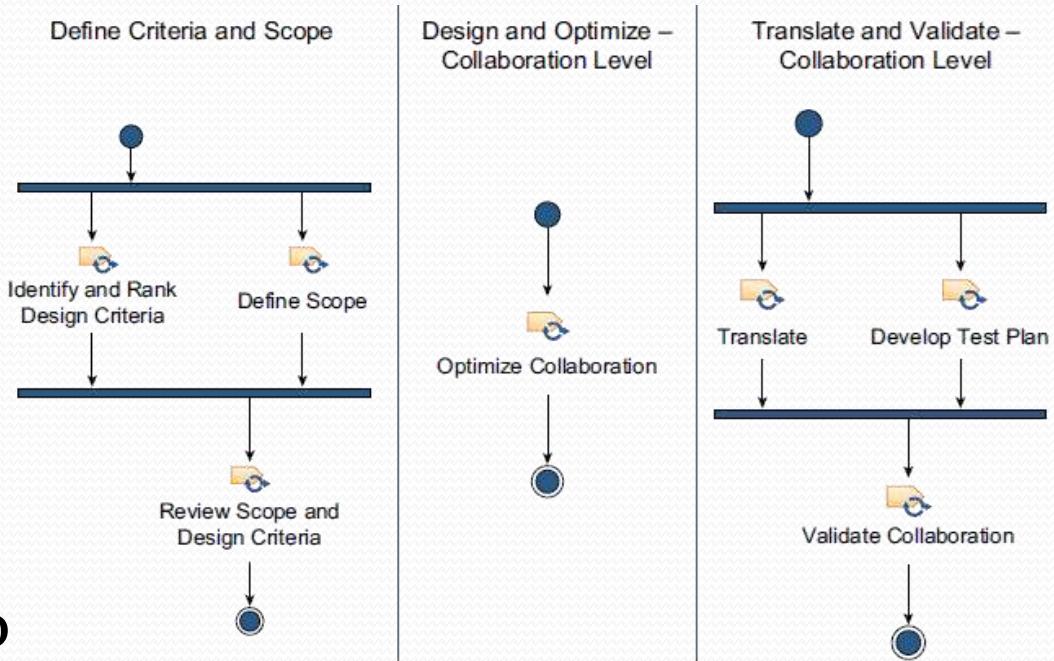
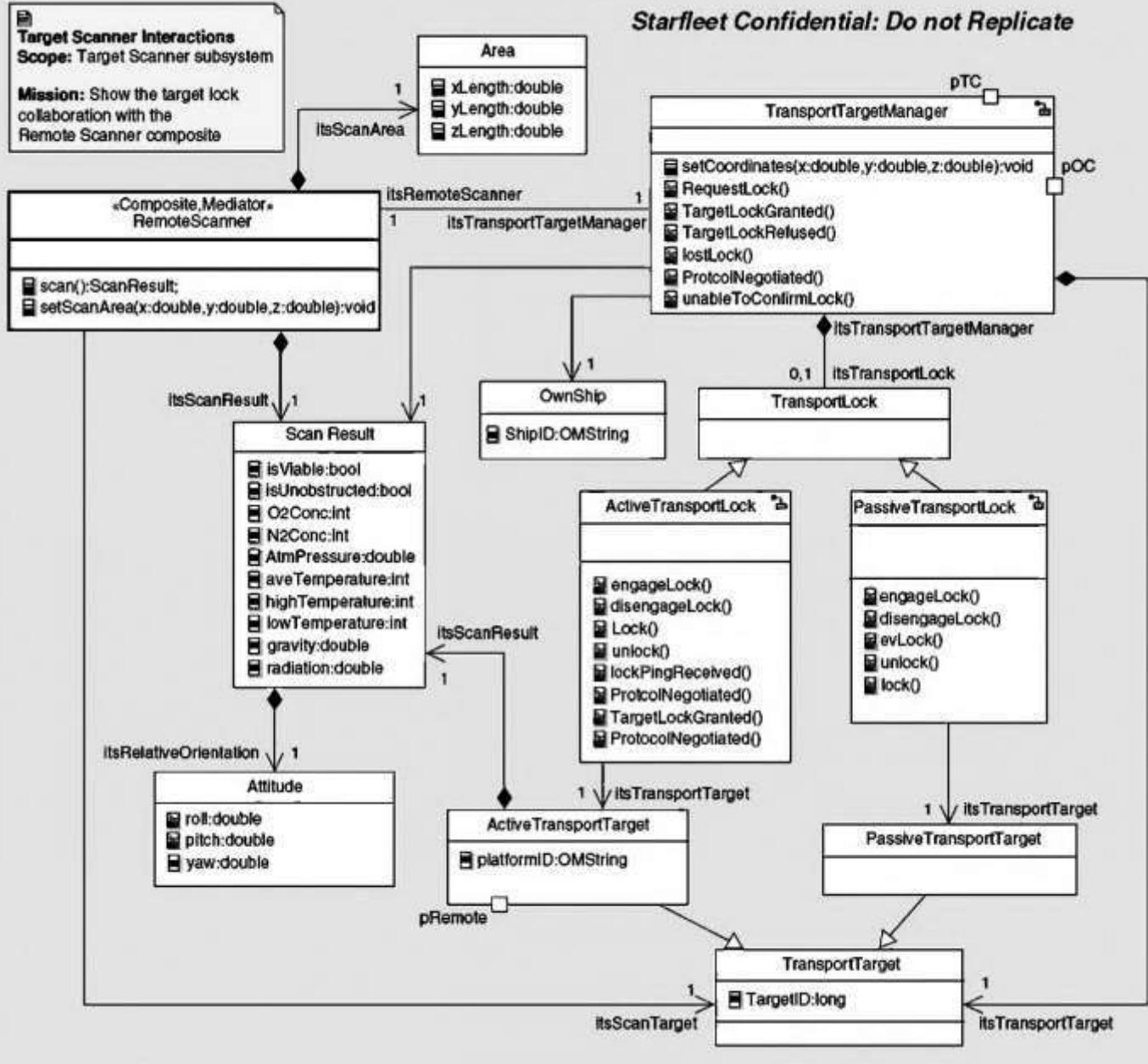


Figure 10.2
Collaboration design workflow details.

Teleportacijos sistema: taikymosi skanerio bendradarbiavimo tobulinimas

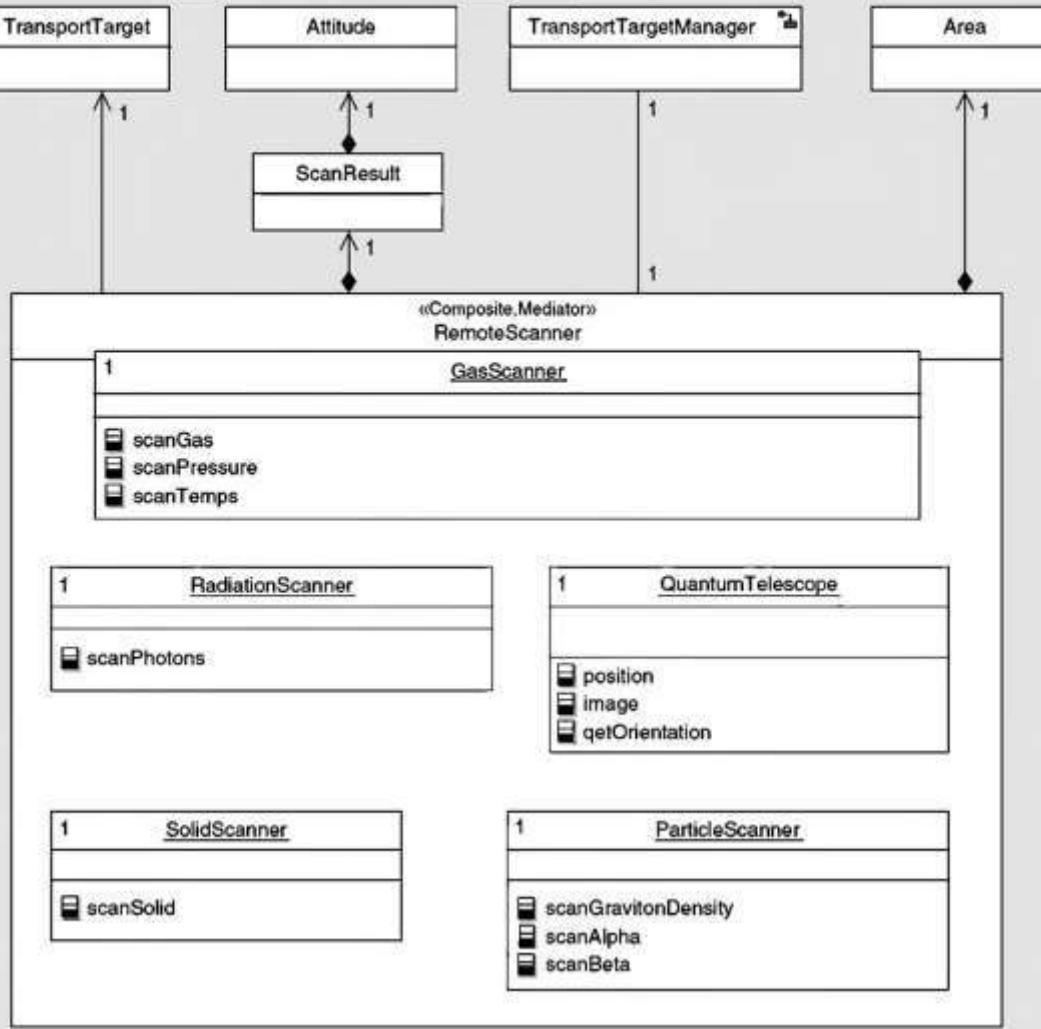
- Tikslai:
 - Apjungti smulkias sistemos dalis į stambesnes, kad supaprastinti bendrą architektūrą (Composite pattern).
 - Padaryti sistemą tinkamą naudoti skirtingų gamintojų HW (Adapter Pattern).
 - Kadangi reikės skanuoti milijardus smulkių objektų, reikia efektyvių ir sparčių valdymo sprendimų (Flyweight Pattern).
 - Optimizuodami skanavimo našumą ir greitaveiką, norime ištraukti valdantį objektą, kuris užtikrins visų skanerių bendrą veikimą tam tikroje zonoje (Mediator Pattern).

Composite Pattern pvz. (1)



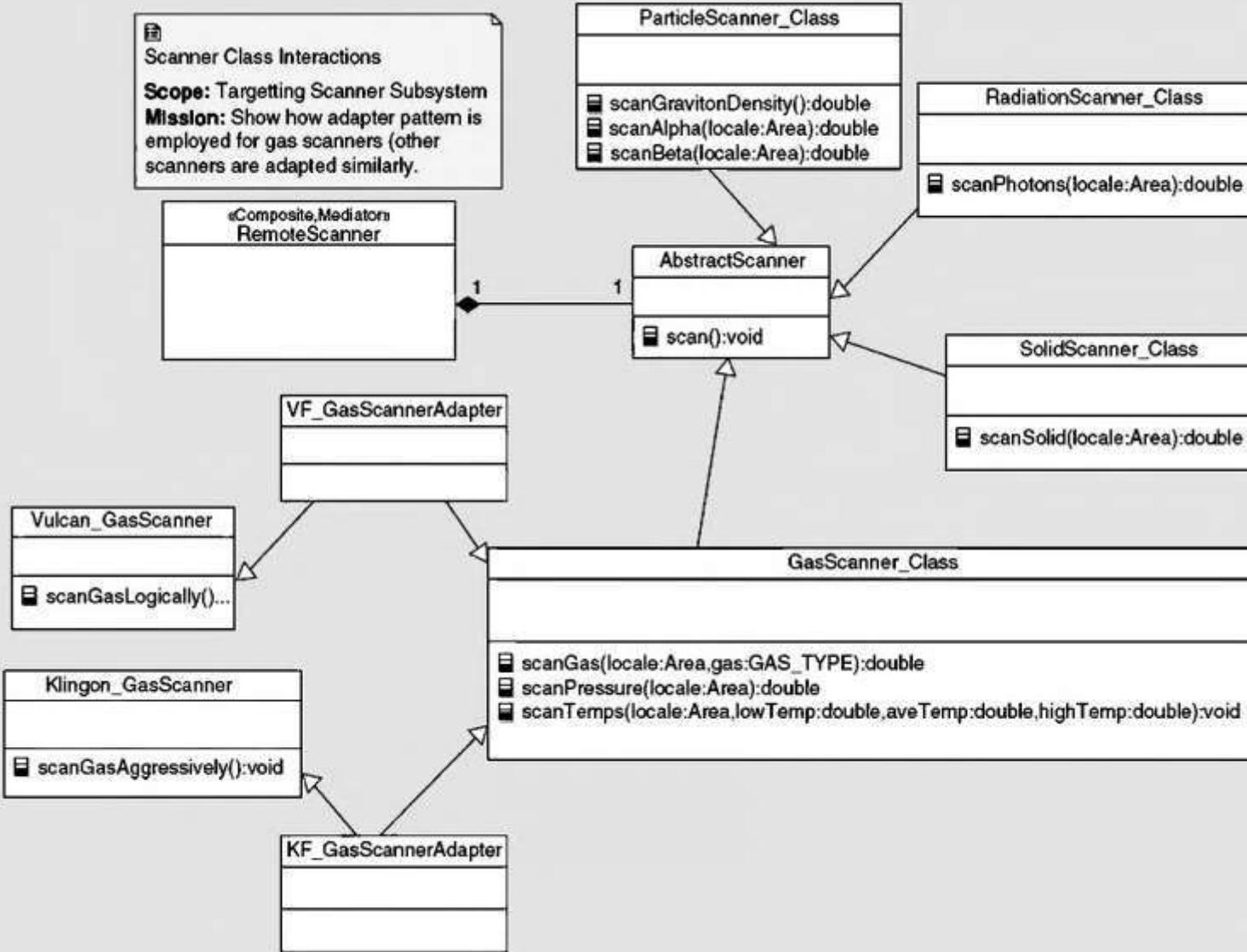
 The RemoteScanner is an assembly of multiple, special-purpose scanners that all take a target locale within the target and scan for different things.

Starfleet Confidential: Do not Replicate

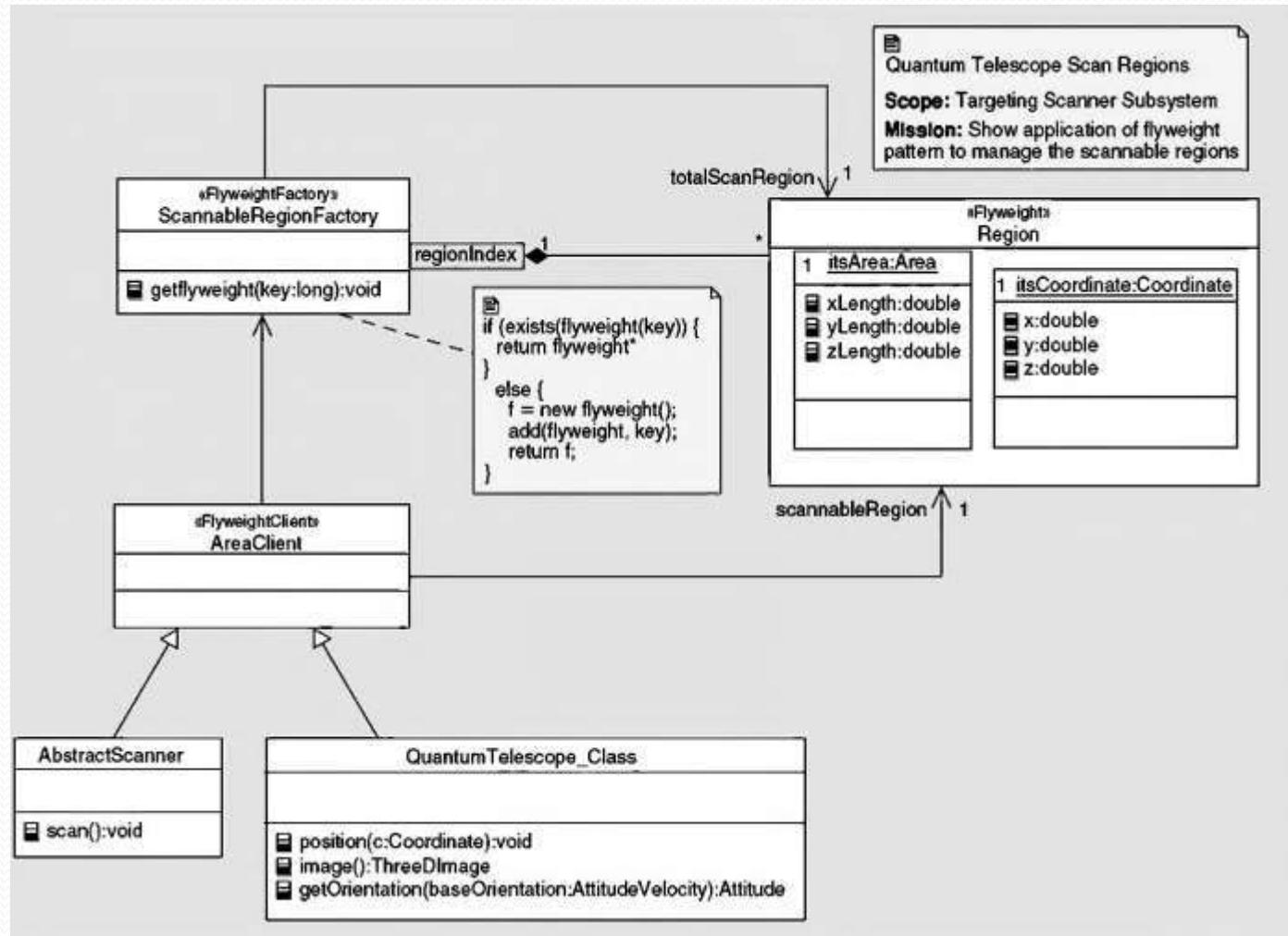


Composite Pattern pvz. (2)

Adapter Pattern pvz. (3)



Flyweight Pattern pvz. (4)



Detalusis projektavimas

- Optimizacija pačiame žemiausiaame lygmenyje:
 - Funkcijų;
 - Kintamųjų;
 - Klasių.
- Dažniausiai optimizuojamos tik kritinės vietas: 3-5% sistemos elementų.
- Optimizacijos reikalaujančių elementų identifikavimas:
 - Teorinis (tiesiog spėjimas ar intuicija);
 - Inspektavimo (įvertinant ir žiūrint į modelį ar kodą);
 - Matavimų (paleidus sistemą ir nustatius lėtai/neoptimaliai veikiančias vietas).

Detalaus projektavimo etapai

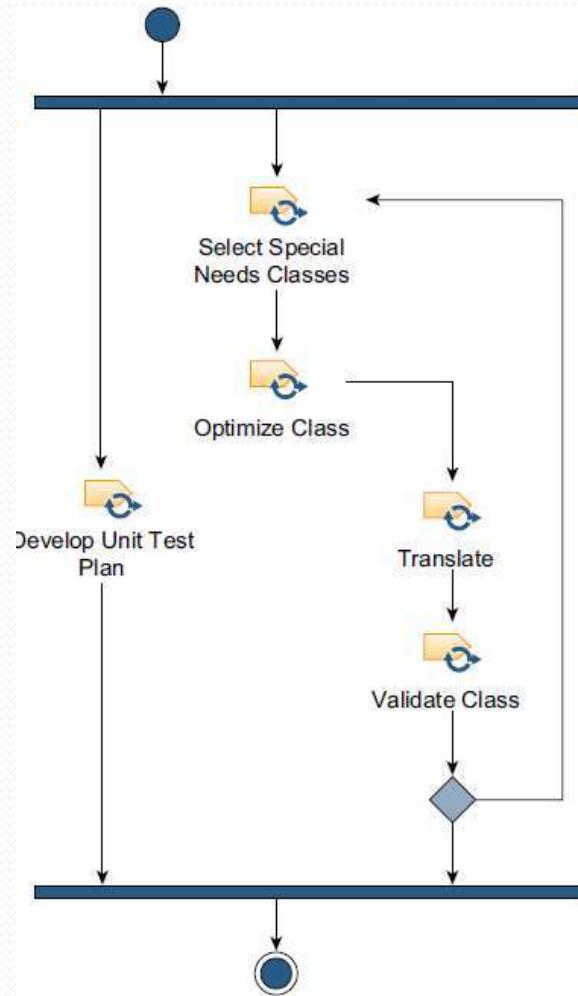
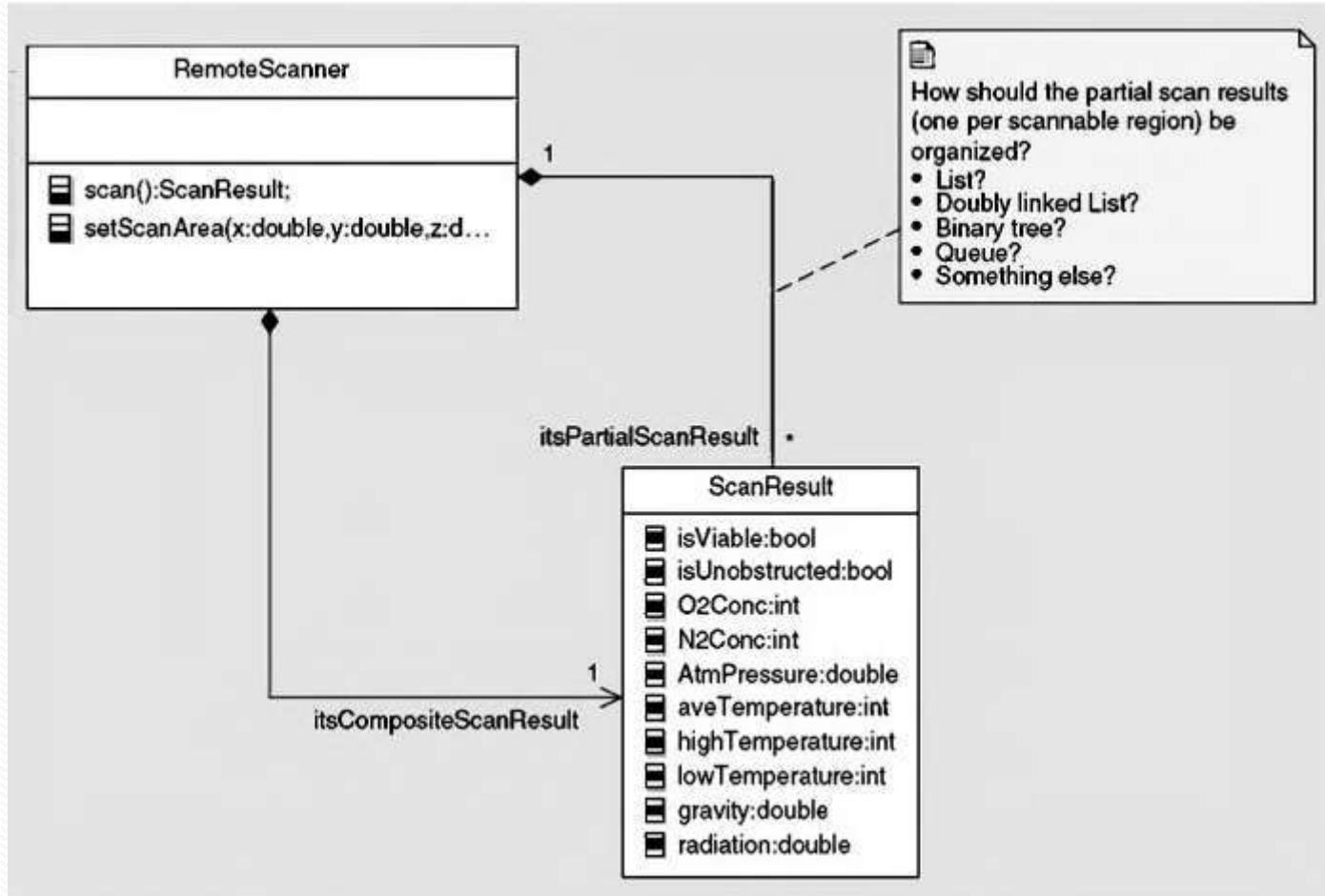
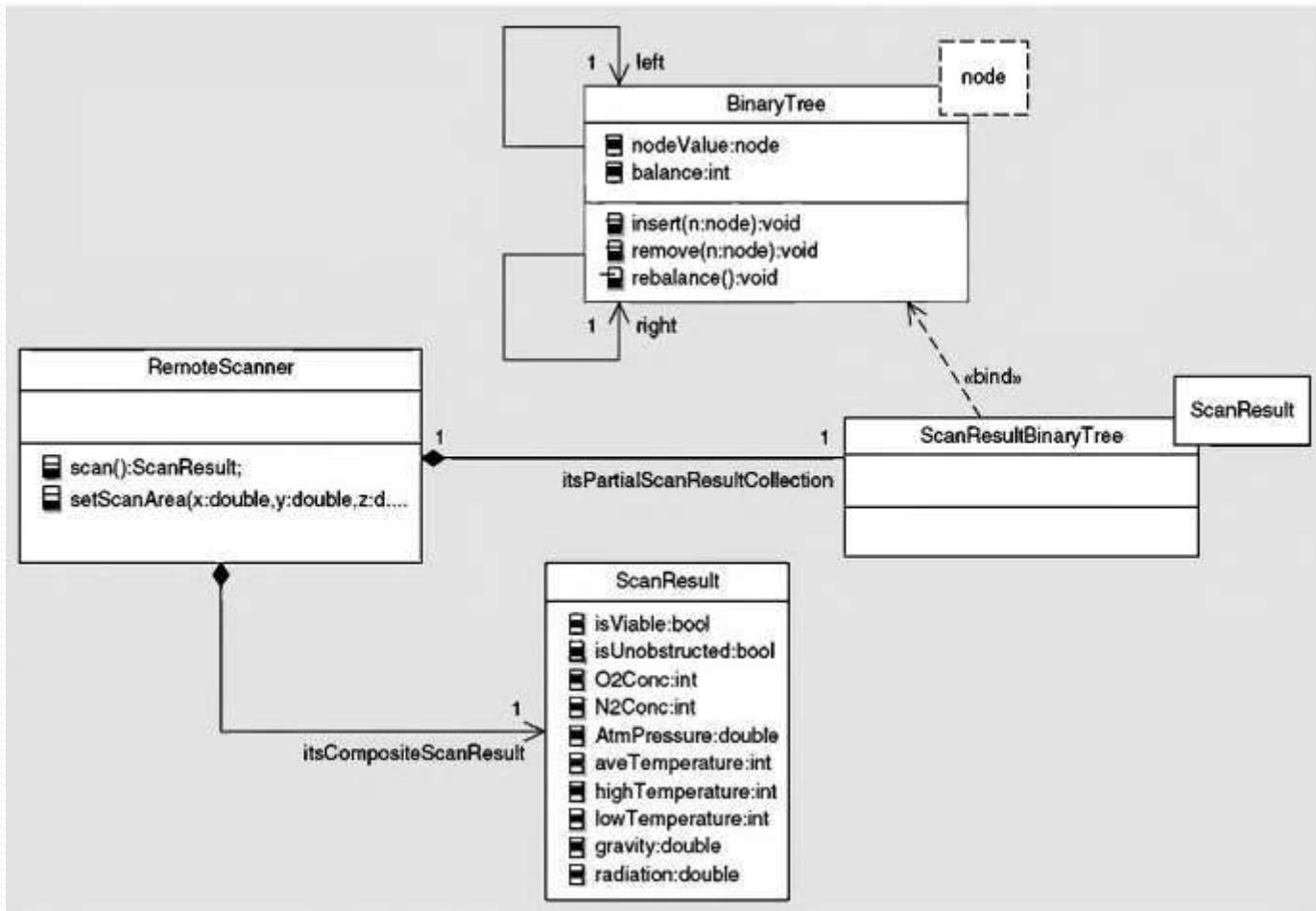


Figure 10.8
Harmony detailed design workflow.

Klasių optimizacijos pavyzdys: 1 su N savybėmis



Klasių optimizacijos pvz.: 1 su N sąryšio sprendimas, panaudojant dvejetainį medį



Ačiū už dėmesį. Klausimai?