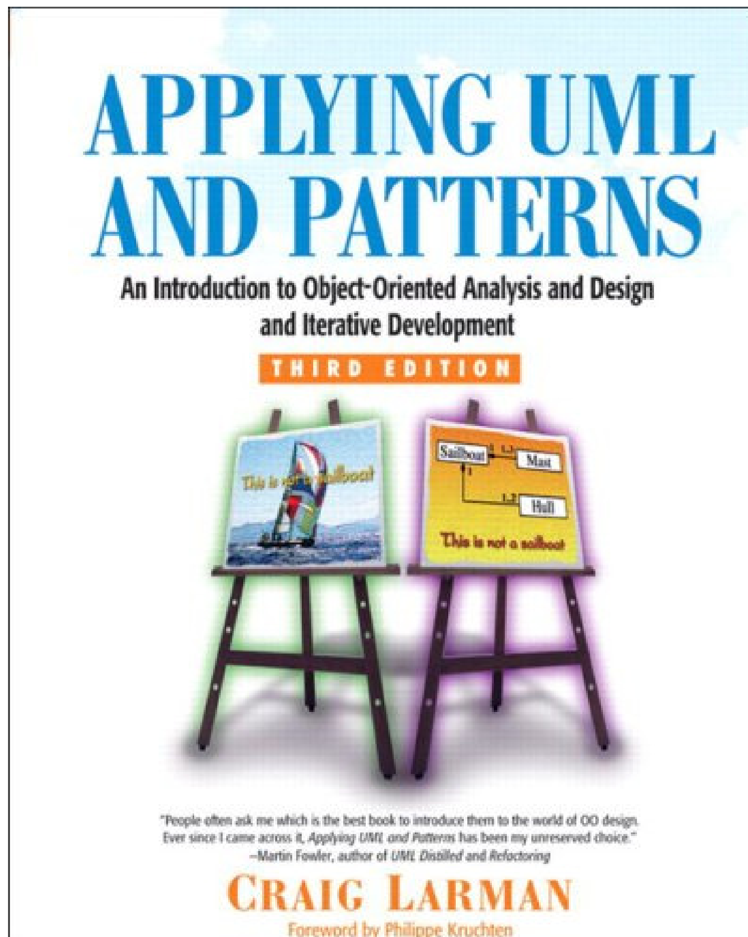
Craig Larman

# GRASP: Dizajnimi i Objekteve me Përgjegjësi

nga libri i mirënjohur për OOAD




- Botimi i dytë -

Përktheu dhe përshtati
Ridvan Bunjaku

Prishtinë
Korrik 2009

# Përmbajtja

Parathënia e librit "Applying UML and Patterns" .... 4
Komente të përkthyesit .... 6
Botimi i dytë .... 7
Kapitulli 17. GRASP: Dizajnimi i Objekteve me Përgjegjësi .... 8
17.1. UML versus Principeve të Dizajnit .... 8
17.2. Dizajni i objekteve: Shembuj të Inputave (Hyrjeve), Aktiviteteve dhe Outputave (Daljeve) .... 9
Çka janë inputat (hyrjet) në dizajnin e objekteve? .... 9
Cilat janë aktivitetet e dizajnit të objekteve? .... 11
Cilët janë outputat (daljet)? .... 12
17.3. Përgjegjësitë dhe dizajni i udhëhequr nga përgjegjësitë .... 13
17.4. GRASP: Qasje metodike ndaj dizajnit themelor OO .... 14
17.5. Cila është lidhja ndërmjet përgjegjësive, GRASP, dhe Diagrameve UML? .... 15
17.6. Çka janë Paternat? .... 16
Paternat kanë emra - E rëndësishme! .... 16
'Patern i ri' është oksimoron (kundërthënie) .... 17
Libri i paternave të dizajnit i bandës së katërshes (Gang-of-Four) .... 17
A është GRASP grup i paternave apo i principeve? .... 18
17.7. Ku jemi tash? .... 19
17.8. Një shembull i shkurtë i dizajnit të objekteve me GRASP .... 19
Krijuesi .... 19
Eksperti i Informatave (Information Expert) .... 22
Çiftimi i Ulët (Low Coupling) .... 23
Kontrolluesi .... 25
Kohezioni i Lartë .... 29
17.9. Aplikimi i GRASP në Dizajnin e Objekteve .... 30
17.10. Krijuesi .... 31
17.11. Eksperti i Informatave (ose Eksperti) .... 34
17.12. Çiftimi i Ulët .... 40
Zgjedhni betejat tuaja .... 43
17.13. Kontrolluesi .... 44
UI-të Web dhe aplikimi i Kontrolluesit server-side (nga ana e serverit) .... 49
Implementimi me Java Swing: Rich Client UI (Ndërfaqe e Përdoruesit për Klient të Pasur) .... 51

Implementimi me Java Struts: Shfletues klient dhe WebUI ........ 53

Kontrolluesit e fryrë (të mëdhenj) ........ 54

Shtresa UI nuk i trajton ngjarjet e sistemit ........ 54

Sistemet e trajtimit të mesazheve dhe Paterni Komandë ........ 57

17.14. Kohezioni i Lartë ........ 58

Një Princip Tjetër Klasik: Dizajni Modular ........ 61

Kohezioni dhe çiftimi: Jin dhe Jang ........ 62

17.15. Resurset e rekomanduara ........ 63

Kapitulli 25. GRASP: Më shumë objekte me përgjegjësi ........ 64

Hyrje ........ 64

25.1. Polimorfizmi (Shumëformësia) ........ 65

Problemi NextGen: Si me përkrahë llogaritës të taksave që janë palë të treta? ........ 65

Problemi Monopoly: Si me dizajnu për aksione të ndryshme të kutive? ........ 67

Udhëzim: Kur me dizajnu me interfejsa? ........ 73

25.2. Fabrikimi i Pastër ........ 75

Problemi NextGen: Ruajtja e objektit Shitje në Bazë të Shënimeve ........ 75

Problemi Monopoly: Trajtimi i Zareve ........ 77

25.3. Indireksioni ........ 80

AdapterPërLlogaritësTëTaksave ........ 80

RuajtjePersistente ........ 81

25.4. Variacionet e Mbrojtura ........ 82

Mekanizmat e Motivuar nga Variacionet e Mbrojtura ........ 83

Principi i Zëvendësimit i Liskovit (The Liskov Substitution Principle (LSP)) ........ 85

Shtojcë: artifakte dhe shembuj ........ 91

Studimet e rasteve (case studies) ........ 91

Rasti i përdorimit RP1: Përpunoje Shitjen ........ 93

Rasti i përdorimit RP1: Luaje lojën Monopoly ........ 101

Specifikim plotësues (suplementar) – Shembulli NextGen ........ 102

Dokument i vizionit - Shembulli NextGen: Vizioni (i Pjesshëm) ........ 107

Fjalor i projektit ........ 111

Rregullat e domenit - Shembulli NextGen ........ 112

Kërkesat e iteracionit 1 ........ 113

Model i Domenit ........ 114

Klasat konceptuale ........ 114

Lista e asociacioneve më të shpeshta ..... 115
Modeli i pjesshëm i domenit për NextGen POS ..... 116
Atributet - Modeli i pjesshëm i domenit për NextGen POS ..... 117
Diagram Sekuencial i Sistemit NextGen ..... 118
Kontrata të operacioneve ..... 119
Arkitektura logjike – shtresat ..... 120
Diagram sekuencial ..... 121
Shembull i kodit – rritja e ndryshores ..... 121
Thirrje asinkronike ..... 122
Disa kontrata me diagrame sekuenciale ..... 123
Diagram i Dizajnit të Klasave ..... 129
Implementim në kod – Hyrje në zgjidhjen NextGen POS ..... 130
Implementim në kod – Hyrje në zgjidhjen Monopoly ..... 133
Fjalor i publikimit ..... 137
Përmbledhje e GRASP ..... 141

## Parathënia e librit “Applying UML and Patterns”

Programimi është argëtues, por zhvillimi i softuerit cilësor është i vështirë. Ndërmjet ideve të mira, kërkesave apo "vizionit", dhe një produkti softuerik, ka shumë më shumë se programim. Analiza dhe dizajni, definimi se si me zgjidhë problemin, çka me programu, me mbërthy këtë dizajn në mënyra që janë të lehta për me komuniku, me rishiku, me implementu, dhe me evolu - është ajo që qëndron në thelb të këtij libri. Kjo është ajo që do ta mësoni.

Gjuha e Unifikuar e Modelimit - Unified Modeling Language (UML) është bërë gjuhë e pranuar universalisht për planet e dizajnit të softuerit. UML është gjuha vizuale që përdoret për m'i shprehë idetë e dizajnit nëpër këtë libër, që e thekson se si i aplikojnë zhvilluesit vërtet elementet e përdorura shpesh të UML, e jo veçoritë e errëta të gjuhës.

Rëndësia e paternave në përpunimin e sistemeve komplekse është pranuar moti në disciplinat tjera. Paternat e dizajnit softuerik janë ato që na lejojnë m'i përshkru fragmentet e dizajnit, dhe m'i ripërdorë idetë e dizajnit, duke i ndihmu zhvilluesit me përfitu nga ekspertiza e të tjerëve. Paternat i japin emër dhe formë eksperimenteve abstrakte, rregullave dhe praktikave më të mira të teknikave të orientuara kah objektet. Asnjë inxhinier i arsyeshëm nuk don me ia fillu nga një cung i thatë, dhe ky libër e ofron një paletë të paternave të dizajnit të gatshme m'i përdorë.

Mirëpo dizajni i softuerit duket paksa i thatë dhe misterioz kur nuk prezentohet në kontekst të një procesi të inxhinierimit të softuerit. Dhe në këtë temë, jam i impresionuar se për edicionin e tij të ri, Craig Larman ka zgjedhë me përqafu dhe me prezentu Procesin e Unifikuar (Unified Process), duke tregu se si mundet m'u përdorë ai në mënyrë relativisht të thjeshtë dhe pa ceremoni të mëdha. Duke i prezentu studimet e rasteve (case studies) në proces iterativ, të udhëhequr nga rreziqet, dhe arkitekturë-qendror, këshillat e Craig'ut kanë kontekst realistik; ai e ekspozon dinamikën e asaj që vërtet ndodh në zhvillimin e softuerit, dhe i tregon forcat e jashtme që janë në lojë. Aktivitetet e dizajnit janë të lidhura me punët tjera, dhe më nuk shfaqen si aktivitet i pastër mendor i transformimeve sistematike apo intuitës kreative. Dhe Craig dhe unë jemi të bindur në benefitet e zhvillimit iterativ, që do t'i shihni të ilustruara bujshëm nëpër libër.

Pra për mua, ky libër e ka miksin e duhur të përbërësve. Do ta mësoni një metodë sistematike për me bë Analizë dhe Dizajn të Orientuar kah Objektete (Object-Oriented Analysis and Design - OOA/D) nga një mësues i madh, një metodologjist briliant, dhe një "guru OO" që ua ka mësuar atë mijëra njerëzve nëpër botë. Craig e përshkruan metodën në kontekst të Procesit të Unifikuar (UP). Ai gradualisht i prezenton paternat më të sofistikuar të diztajnit - kjo do ta bëjë librin shumë të përdorshëm kur ballafaqoheni me sfida të dizajnit në botën reale. Dhe ai e përdor notacionin më të pranuar.

Jam i nderuar që e kam pasë shansin me punu drejtpërdrejt me autorin e këtij libri madhor. Jam kënaqë duke e lexu edicionin e parë, dhe isha i impresionuar kur më luti me rishiku dreftin e këtij edicioni të ri. Jemi taku disa herë dhe kemi shkëmby shumë e-maila. Kam mësuar shumë nga Craig, bile edhe për vetë punën tonë të procesit në Procesin e Unifikuar dhe si me përmirësu atë dhe me pozicionu në kontekste të ndryshme organizative. Jam i sigurtë se edhe ju do të mësoni shumë, gjatë leximit të këtij libri, edhe nëse tashmë jeni familiarë me OOA/D. Dhe, sikur unë, do ta gjeni veten

duke iu kthy librit, për me fresku memorjen tuaj, apo për me fitu mprehtësi të mëtejme nga shpjegimet dhe përvoja e Craig'ut.

Lexim të këndshëm!

Philippe Kruchten

Profesor i Inxhinierimit të Softuerit, Universiteti i British Columbia

më parë,

Partner i Rational dhe Drejtor i Zhvillimit të Procesit për Softuerin RUP Rational

Vancouver, British Columbia

## Komente të përkthyesit

Ky është vetëm kapitulli i 17-të nga 40 sa i ka krejt libri. Libri "Applying UML and Patterns" konsiderohet si libri më i përshtatshëm për hyrje në OOAD. Për të korrura maksimale preferohet me lexu krejt librin, ndërsa për nivel më të avansuar preferohet libri "Design Patterns".

Në kapitullin 17 janë prezentuar vetëm pesë principe apo paterna GRASP prandaj ky publikim mund të konsiderohet si hyrje në OOAD. Për me plotësu zbrastësinë e 16 kapitujve paraprakë, në fund të publikimit është një shtojcë ku janë artifaktet kryesore që janë punu deri në kapitullin 17. Ato janë modele të mira për me pa në pika të shkurtra punën që bëhet në një projekt të madh me OOAD.

Gjatë librit janë studiuar dy raste: NextGEN POS, një aplikacion për shitje në shitore me pakicë, dhe Monopoly, simulimi i lojës së njohur. Autori me qëllim ka punu në dy mënyra të ndryshme. Në NextGEN ka punu me vegla digjitale, aplikacione për gjenerim të diagrameve UML, ndërsa në Monopoly ka punu ekskluzivisht me dorë (në tabela).

Gjatë përkthimit e kam përdorë një stil gjuhësor që është më i thjeshtë dhe më i afërt për lexuesin shqiptar në përgjithësi. Në shumë raste i kam thjeshtu fjalitë duke konsideru se lexuesi është programer e jo gjuhëtar, dhe se atij/asaj i intereson informata dhe kuptimi e jo saktësia gjuhësore sintaksore e fjalisë. Dmth. për hir të semantikës më të mirë ka shmangie nga sintaksa dhe drejtshkrimi standard.

Në fund është një fjalor i publikimit ku janë dhënë shpjegime të shkurtra të shkurtesave (akronimeve). Akronimet i kam lënë origjinal për shkak të kuptueshmërisë më të mirë meqë literatura dërmuese është në anglisht.

Pas fjalorit është përmbledhja e principeve/paternave GRASP që mund të shërbejë si referencë e mirë dhe e shpejtë.

Dhe për fund, kur e kam kërku lejen e publikimit nga autori e kam përjetu një befasi të këndshme. Autori, pos që është përgjigjë shpejt (brenda 4 minutave), është përgjigjë shqip:

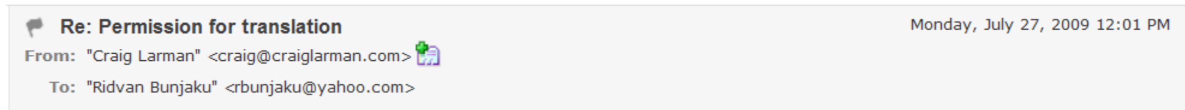
(singapore)

dear ridvan,

Po, ju lutemi përdorni atë. Faleminderit për bërjen e përkthimit.

regards, craig
craig@craiglarman.com
www.craiglarman.com
author of:
*-LATEST: Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*
*-COMING SOON: Practices for Scaling Lean & Agile Development: Large, Multisite & Offshore Products with Large-Scale Scrum*
*-Agile and Iterative Development: A Manager's Guide*
*-Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*

Në vijim është pamja e korrespondencës së shkurtë por efektive. Vëreni se Singapori është 6 orë para nesh.


**Re: Permission for translation** Monday, July 27, 2009 12:01 PM
From: "Craig Larman" <craig@craiglarman.com>
To: "Ridvan Bunjaku" <rbunjaku@yahoo.com>

(singapore)
dear ridvan,

Po, ju lutemi përdorni atë. Faleminderit për bërjen e përkthimit.

regards, craig
craig@craiglarman.com
www.craiglarman.com
author of:
*-LATEST: Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*
*-COMING SOON: Practices for Scaling Lean & Agile Development: Large, Multisite & Offshore Products with Large-Scale Scrum*
*-Agile and Iterative Development: A Manager's Guide*
*-Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*

On Jul 27, 2009, at 5:57 PM, Ridvan Bunjaku wrote:

> Dear mr. Larman,
> I was delighted to read and study your book, Applying UML and Patterns, Third Edition.
> I found it most interesting chapter 17, GRASP: Designing Objects with Responsibilities, which I translated in Albanian for my personal studying purposes.
> Now, since it's already translated, I ask your permission to publish it in web, for the Albanian community which, unfortunately, lacks Albanian literature on IT field in general, and particularly in software development.
> Please find attached a draft of the translated chapter. I will yet do paging, will add a foreword and some final tunings.
> Of course, I will mention you as author and myself as a translator, and the source book.
>
> Looking forward for your reply.
>
> Best regards,
> Ridvan
>
> ---
>
> |:( |:)
> Ridvan Bunjaku
> Web: http://kalabuli.googlepages.com/
> Shëndet dhe suksese!
> Health and success!
>
> ---
>
> <GRASP draft.pdf>

Ridvan Bunjaku

Prishtinë, 28 korrik 2009

## Botimi i dytë

Botimi i dytë e përmban edhe kapitullin 25, ku flitet për katër paternat e fundit GRASP.

Prishtinë, 7 gusht 2009

# Kapitulli 17. GRASP: Dizajnimi i Objekteve me Përgjegjësi

*Të kuptuarit e përgjegjësive është kryesor në dizajnin e mirë të orientuar kah objektet.*

*Martin Fowler*

**Qëllimet**

- Me mësu m'i apliku pesë nga principet GRASP të paternave për OOD (Object Oriented Design - Dizajni i Orientuar kah Objektet)

Ky kapitull dhe tjetri kontribuojnë dukshëm në të kuptuarit e thelbit të dizajnit OO. OOD nganjëherë mësohet si variacion i kësaj:

> Pasi t'i keni identifiku kërkesat dhe ta keni kriju modelin e domenit, shtoni metodat në klasat përkatëse, dhe definojeni dërgimin e mesazheve ndërmjet objekteve për m'i plotësu kërkesat.

Offf! Një këshillë e tillë e mjegullt nuk na ndihmon, meqë këtu janë të involvuara principe dhe çështje të thella. Me vendosë - se çfarë metoda ku e kanë vendin dhe si duhet me ndërvepru objektet - bart pasoja dhe duhet m'u marrë seriozisht. Zotërimi i Dizajnit OO - pra i hijeshisë së tij të komplikuar - përfshin grup të madh të principeve të buta, me shumë shkallë të lirisë. Nuk është magji - paternat (modelet) munden m'u emëru (e rëndësishme!), m'u shpjegu, dhe m'u apliku. Shembujt ndihmojnë. Praktika ndihmon. Dhe ky hap i vogël ndihmon: Pasi të studiohen këto studime të rasteve (case studies), provoni ta rikrijoni në mure (nga kujtesa) zgjidhjen Monopoly me partnerët, dhe t'i zbatoni principet, siç është Eksperti i Informatave (Information Expert).

## 17.1. UML versus Principeve të Dizajnit

Meqë UML është thjesht gjuhë standarde e modelimit vizual, njohja e detaleve të saj nuk ju ndihmon se si me mendu në objekte - kjo është temë e këtij libri. UML nganjëherë përshkruhet si "vegël e dizajnit" por kjo nuk është krejt e saktë...

**Vegla kritike e dizajnit për zhvillim të softuerit është një mendje e edukuar mirë me principe të dizajnit.** Nuk është UML apo cilado teknologji tjetër.

## 17.2. Dizajni i objekteve: Shembuj të Inputave (Hyrjeve), Aktiviteteve dhe Outputave (Daljeve)

Ky seksion e përmbledh një shembull në pikturë të madhe të dizajnit në një metodë iterative:

- Çka është bërë? Aktivitetet e mëhershme (p.sh. sesioni) dhe artifaktet
- Si lidhen gjërat? Ndikimi i artifakteve të mëhershme (p.sh. rastet e përdorimit) në dizajnin OO
- Sa me bë modelim të dizajnit, dhe si?
- Cili është outputi (dalja, rezultati)?

Sidomos, do të doja që ju me kuptu se si lidhen artifaktet e analizës me dizajnin e objekteve.

### Çka janë inputat (hyrjet) në dizajnin e objekteve?

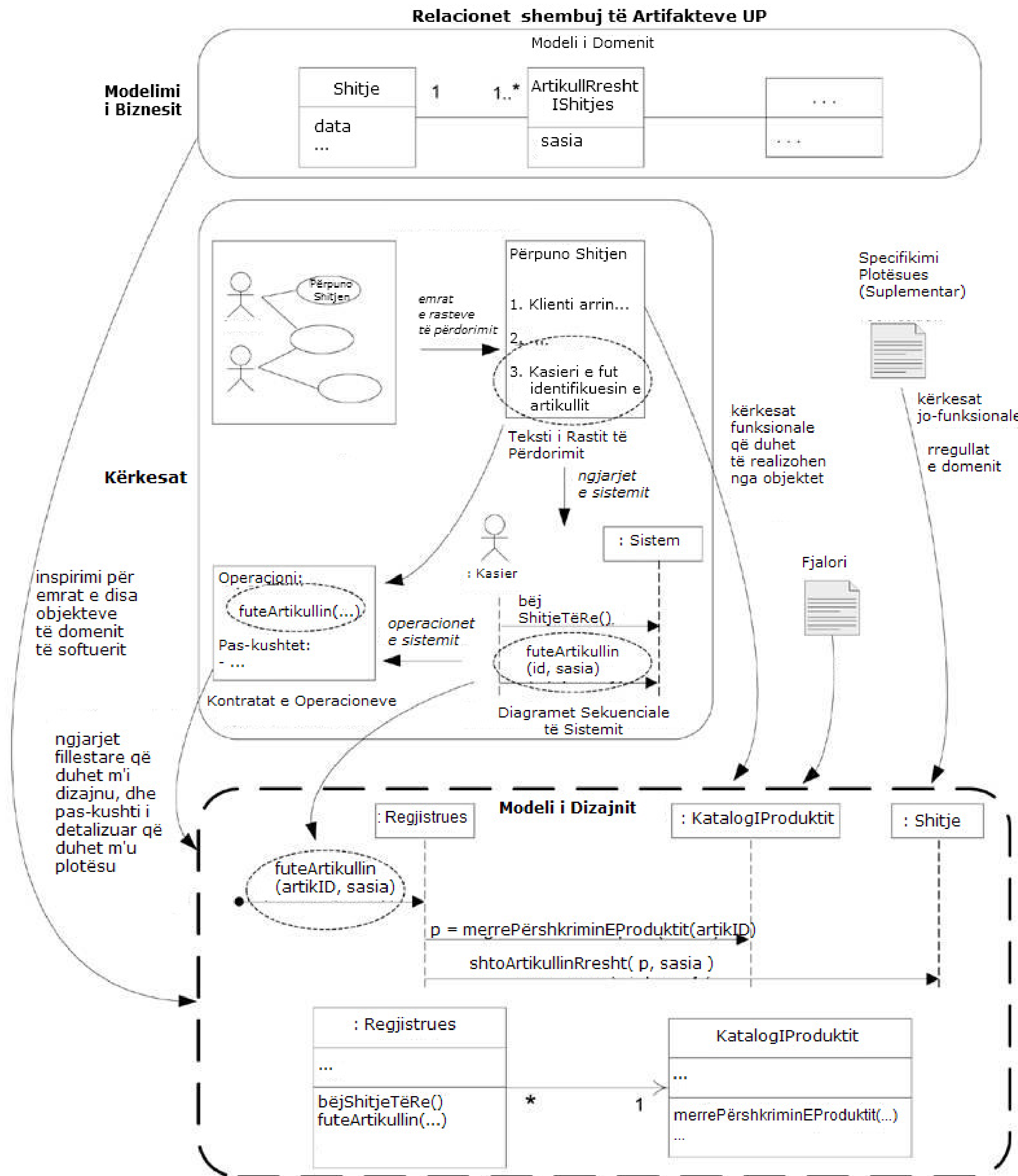
Të fillojmë me hyrjet "proces". Ta zëmë se jemi zhvillues duke punu në projektin POS NextGen (Point of Sale - pikë e shitjes, aplikacion për shitje të artikujve), dhe se skenari vijues është i vërtetë:

| | |
|---|---|
| **Sesioni i parë dy-ditor i kërkesave** është kryer. | Arkitekti kryesor dhe biznesi pajtohen m'i implementu dhe m'i testu disa **skenarë të rastit të përdorimit Përpuno Shitjen** në iteracionin e parë tre-javor. |
| **Tri nga njëzet rastet e përdorimit** - ato që janë më të rëndësishmet arkitekturalisht dhe me vlerë më të lartë të biznesit - janë analizu detalisht, duke e përfshirë, natyrisht, rastin e përdorimit Përpuno Shitjen (UP - Unified Process, Procesi i Unifikuar - rekomandon, siç është tipike me metodat iterative, që të analizohen vetëm **10%-20% e kërkesave** detalisht para se me fillu me programu.) | **Artifaktet tjera** janë nisur: Specifikimi Plotësues, Fjalori, dhe Modeli i Domenit |
| **Eksperimentet e programimit** i kanë zgjidhur pyetjet teknike që munden me ndalë procesin, si p.sh. a do të punojë Java Swing UI në ekran prekës. | Arkitekti kryesor i ka vizatu disa ide për **arkitekturën logjike të shkallës së madhe**, duke përdorë diagrame të paketave UML. Kjo është pjesë e Modelit të Dizajnit UML. |

Cilat janë hyrjet *artifakt* dhe relacionet e tyre me dizajnin e objekteve? [1] Ato janë të përmbledhura në figurën 17.1. dhe në tabelën vijuese.

[1] Hyrje tjera artifakt mund të përfshijnë dokumente të dizajnit për një sistem ekzistues që është duke u ndryshu. Po ashtu është e dobishme me inxhinieru-mbrapa - reverse-engineer - kodin në diagrame të paketave UML për me pa strukturën logjike të shkallës së lartë dhe disa diagrame të klasave dhe sekuenciale.

**Figura 17.1. Relacionet e artifakteve duke e theksu ndikimin në dizajnin OO.**



| **Teksti i rastit të përdorimit** e definon sjelljen e dukshme që duhet me përkrahë në fund objektet e softuerit - objektet dizajnohen për m'i "realizu" (implementu) rastet e përdorimit. Në UP, ky dizajn OO quhet, në mënyrë të pritshme, realizim i rasteve të përdorimit. | **Specifikacioni Suplementar** (plotësues) i definon qëllimet jo-funksionale, siç është ndërkombëtarizimi, që duhet m'i plotësu objektet tona. |
|---|---|
| **Diagramet sekuenciale të sistemit** i identifikojnë mesazhet e operacioneve të sistemit, që janë mesazhe fillestare të diagrameve tona të interaksionit (ndërveprimit) të objekteve bashkëpunuese. | **Fjalori** i sqaron detalet e parametrave apo të shënimeve që vijnë nga shtresa UI (User Interface, Ndërfaqja e Përdoruesit), shënimet që i pasohen bazës, dhe logjikën detale specifike të një elementi apo kërkesat e validimit, si p.sh. formatet legale dhe validimi për UPCtë e produkteve (UPC, universal product code – kod univerzal i produktit, kod që e identifikon një produkt që shitet). |
| **Kontratat e operacioneve** munden me plotësu tekstin e rasteve të përdorimit për me qartësu se çka duhet me arritë objektet e softuerit në një operacion të sistemit. Pas-kushtet i definojnë arritjet e detalizuara. | **Modeli i Domenit** i sugjeron disa emra dhe atribute të objekteve të domenit të softuerit në shtresën e domenit të arkitekturës së softuerit. |

Jo të gjitha këto artifakte janë të nevojshme. Rikujtoni se në UP të gjitha elementet janë opsionale, që krijohen sipas rastit, për me zvogëlu ndonjë rrezik.

## Cilat janë aktivitetet e dizajnit të objekteve?

Jemi gati m'i heqë kapelat e analistëve dhe m'i vendosë kapelat e dizajnerëve-modeluesve.

Duke i pasë një apo më shumë nga hyrjet e përmendura, zhvilluesit 1) fillojnë menjëherë me kodu (idealisht me **zhvillim testi-së-pari** – test-first development), 2) fillojnë pak modelim UML për dizajn të objekteve, ose 3) fillojnë me ndonjë teknikë tjetër të modelimit, siç janë kartelat CRC[2].

Në rastin UML, poenta reale nuk është UML, por modelimi vizual - duke e përdorë një gjuhë që na lejon të eksplorojmë më vizualisht se sa që mundemi vetëm me tekst të thjeshtë. Në këtë rast, për shembull, i vizatojmë edhe diagramet e interaksionit, edhe diagramet plotësuese të klasave (modelimi dinamik dhe statik) **gjatë një dite të modelimit**. Dhe më e rëndësishmja, gjatë aktivitetit të vizatimit (dhe të kodimit) aplikojmë principe të ndryshme OO, siç janë **paternat e dizajnit GRASP dhe Gang-Of Four** (GoF - Banda e Katërshes). Qasja e përgjithshme ndaj modelimit të dizajnit OO do të bazohet në *metaforën* e **dizajnit të udhëhequr nga përgjegjësitë** (responsibility-driven design, RDD), duke mendu se si me iu nda përgjegjësi objekteve bashkëpunuese.

| Ky dhe kapitujt vijues e eksplorojnë se çka domethënë me zbatu RDD, GRASP dhe disa nga paternat e dizajnit GoF. |
|---|

Në ditën e modelimit, mbase ekipi punon 26 orë në grupe të vogla ose në mure (në dërrasa të zeza apo letra të mëdha) ose me vegla të modelimit të softuerit, duke bërë lloje të ndryshme të modelimit për pjesët e vështira, kreative të dizajnit. Kjo mundet me përfshi modelim të UI (user interface – ndërfaqja e përdoruesit), OO, dhe të bazës së shënimeve me vizatime UML, vegla të prototipimit, skica, etj.

Gjatë vizatimit UML, adoptojmë qëndrim realistik (po ashtu i përkrahur në modelimin e shkathët - agile), që ne po i vizatojmë modelet kryesisht për *me kuptu dhe me komuniku*, e jo për me dokumentu. Natyrisht, presim që disa nga diagramet UML me qenë hyrje e dobishme në definicionin e kodit (apo gjenerimin automatik të kodit me vegël UML).

Të martën - ende herët në iteracionin tri-javor të caktuar - ekipi e ndal modelimin dhe i vendos kapelat e programerëve për me iu shmangë mentalitetit ujëvarë të tej-modelimit (modelimit të tepruar) para programimit.

## Cilët janë outputat (daljet)?

Figura 17.1 i ilustron disa inpute (hyrje) dhe relacionin e tyre me outputin (daljen) e një diagrami UML të interaksionit dhe të klasave. Vëreni se ndoshta mundemi m'iu referu këtyre inputeve të analizës gjatë dizajnit; për shembull, duke i ri-lexu tekstin e rasteve të përdorimit apo të kontratave të operacioneve, duke e skenu me sy modelin e domenit, dhe duke e rishiku Specifikimin Plotësues.

Çka është kriju (për shembull) gjatë ditës së modelimit?

- specifikisht për dizajn të objekteve, diagramet UML të interaksionit, diagramet e klasave, dhe diagramet e paketave për pjesët e vështira të dizajnit që kemi dashtë m'i eksploru para kodimit
- skicat dhe prototipet e UI (ndërfaqes së përdoruesit)
- modelet e bazës (me notacion të profilit UML të modelimit të shënimeve)
- skicat dhe prototipet e raporteve

## 17.3. Përgjegjësitë dhe dizajni i udhëhequr nga përgjegjësitë

Një mënyrë e njohur e të menduarit për dizajnin e objekteve të softuerit dhe po ashtu të komponentave të shkallës së lartë[3] është në terma të përgjegjësive, roleve, dhe bashkëpunimeve. Kjo është pjesë e një qasje më të gjerë, të quajtur **Dizajn i Drejtuar nga Përgjegjësitë** apo **RDD** (responsibility-driven design, RDD).

Në RDD, ne i mendojmë objektet e softuerit ashtu që ato kanë përgjegjësi - një abstraksion i asaj se çka bëjnë ata. UML e definon përgjegjësinë si "kontratë ose obligim i një klasifikuesi". Përgjegjësitë janë të lidhura me obligimet ose sjelljen e një objekti në terma të rolit të vet. Në parim, këto përgjegjësi janë të këtyre dy llojeve: *bërje* dhe *ditje*.

Përgjegjësitë e **bërjes** të një objekti përfshijnë:

- Me bë diçka vetë, siç është krijimi i një objekti apo bërja e një llogarie.
- Me inicu aksion në objekte tjera
- M'i kontrollu dhe m'i koordinu aktivitetet në objektet tjera

Përgjegjësitë e **ditjes** të një objekti përfshijnë:

- me ditë për shënimet private të enkapsuluara
- me ditë për objektet e afërta
- me ditë për gjërat që mundet m'i nxjerrë (derivu) apo m'i llogaritë

Përgjegjësitë i ndahen klasave të objekteve gjatë dizajnit të objekteve. Për shembull, unë mundem me deklaru që "një *Shitje* është përgjegjëse për krijimin e *ArtikujveRreshtTëShitjes* (SalesLineItems)" (bërje, veprim), ose "një *Shitje* është përgjegjëse për me ditë totalin e vet" (ditje).

***Udhëzim***: Për objektet e domenit të softuerit, modeli i domenit, për shkak të atributeve dhe asociacioneve që i ilustron, shpesh i inspiron përgjegjësitë relevante që kanë të bëjnë me "ditje". Për shembull, nëse klasa Shitje e modelit të domenit e ka një atribut të kohës, është e natyrshme sipas qëllimit të **zbraztësisë së ulët të reprezentimit** (low represantional gap, LRG), që një klasë *Shitje* e softuerit me ditë kohën e vet.

Translacioni i përgjegjësive në klasa dhe metoda ndikohet nga *granulariteti* (niveli i imtësimit) i përgjegjësisë. Përgjegjësitë e mëdha kapin qindra klasa dhe metoda. Përgjegjësitë e vogla munden me kapë vetëm një metodë. Për shembull, përgjegjësia për "me ofru qasje në bazat relacionale" mundet m'i përfshi dyqind klasa dhe mijëra metoda, të paketuara në një nënsistem. Në kontrast të kësaj, përgjegjësia për "me kriju një Shitje" mundet me përfshi vetëm një metodë në një klasë.

Përgjegjësia nuk është gjëja e njëjtë si metoda – ajo është abstraksion - mirëpo metodat i realizojnë përgjegjësitë.

RDD po ashtu e përfshin idenë e **bashkëpunimit** (collaboration). Përgjegjësitë implementohen përmes metodave që ose veprojnë vetëm ose bashkëpunojnë me metoda dhe objekte tjera. Për

[3] Të menduarit në terme të përgjegjësive mundet m'u apliku në çfarëdo shkalle të softuerit - nga një objekt i vogël deri në sistem të sistemeve.

shembull, klasa *Shitje* mundet me definu një apo më shumë metoda që e dinë totalin e saj; të themi, një metodë me emrin *merreTotalin*. Për me plotësu këtë përgjegjësi, *Shitja* mundet me bashkëpunu me objekte tjera, p.sh. duke ua dërgu nga një mesazh *merreNëntotalin* secilit *ArtikullRreshtTëShitjes* (*SalesLineItem*) duke ua kërku nëntotalin e vet.

**RDD është metaforë**

RDD është metaforë e përgjithshme për me mendu për dizajnin OO të softuerit. Mendoni objektet e softuerit si të ngjashme me njerëzit me përgjegjësi, të cilët bashkëpunojnë me njerëzit tjerë për me kry punën. RDD çon në të pamurit e një dizajni OO si *komunitet të objekteve të përgjegjshme bashkëpunuese.*

***Poenta kryesore:*** GRASP i emëron dhe i përshkruan disa principe themelore për ndarjen e përgjegjësive, prandaj është e dobishme me ditë - për me përkrahë RDDnë.

## 17.4. GRASP: Qasje metodike ndaj dizajnit themelor OO

GRASP: Asistencë e të Mësuarit për Dizajnin OO me Përgjegjësi

Është e mundur m'i emëru dhe m'i shpjegu principet detale dhe rezonimin e nevojshëm, për me zotëru dizajnin themelor të objekteve duke iu nda përgjegjësi objekteve. Principet **GRASP** apo paternat janë asistencë e të mësuarit për me ju ndihmu me kuptu dizajnin esencial të objekteve dhe me apliku rezonimin e dizajnit në mënyrë metodike, racionale, të shpjegueshme. Kjo qasje në kuptimin dhe përdorimin e principeve të dizajnit bazohet në *paternat e ndarjes së përgjegjësive*.

Ky kapitull - dhe disa të tjerë - e përdorin GRASPin si vegël për me ndihmu në zotërimin e bazave të OOD (Object Oriented Design – Dizajn i Orientuar kah Objektet) dhe të kuptimit të ndarjes së përgjegjësive në dizajnin e objekteve.

Kuptimi se si me apliku GRASPin për dizajnin e objekteve është qëllimi kryesor i këtij libri.

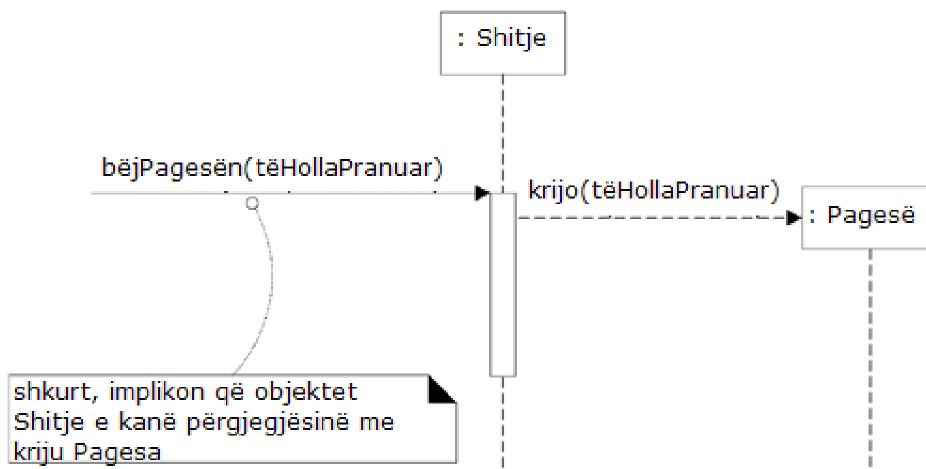
Pra, GRASP është relevant, por në anën tjetër, është vetëm ndihmë e të mësuarit për m'i strukturu dhe m'i emëru principet - pasi t'i zotëroni bazat, termat specifike të GRASP (Eksperti i Informatave, Krijuesi, ...) nuk janë të rëndësishme.

## 17.5. Cila është lidhja ndërmjet përgjegjësive, GRASP, dhe Diagrameve UML?

Ju mundeni me mendu për me ua nda përgjegjësitë objekteve gjatë kodimit apo gjatë modelimit. Brenda UML, vizatimi i diagrameve të interaksionit bëhet shansë për konsiderimin e këtyre përgjegjësive (të realizuara si metoda).

Figura 17.2 tregon se objekteve *Shitje* (*Sale*) iu është dhënë përgjegjësia me kriju *Pagesa* (*Payments*), që konkretisht thirret me një mesazh *bëjPagesë* (*makePayment*) dhe trajtohet me metodën korresponduese *bëjPagesë*. Gjithashtu, plotësimi i kësaj përgjegjësie e kërkon bashkëpunimin për me kriju objektin *Pagesë* dhe për me thirrë konstruktorin e tij.

**Figura 17.2. Përgjegjësitë dhe metodat janë të lidhura**




Prandaj, kur vizatojmë diagram UML të interaksionit, ne jemi duke vendosë për ndarjen e përgjegjësive. Ky kapitull i thekson principet fundamentale - të shprehura në GRASP - për m'i udhëheqë zgjedhjet për ndarjen e përgjegjësive. Si rrjedhim, ju mundeni m'i zbatu principet GRASP gjatë vizatimit të diagrameve UML të interaksionit, dhe po ashtu gjatë kodimit.

## 17.6. Çka janë Paternat?

Zhvilluesit OO me përvojë (dhe zhvilluesit tjerë të softuerit) e ndërtojnë një repertoar edhe të principeve të përgjithshme edhe të zgjidhjeve idiomatike që i udhëheqin ata në krijimin e softuerit. Këto principe dhe idioma, nëse kodifikohen në format të strukturuar që e përshkruan problemin dhe zgjidhjen dhe janë të emëruara, munden m'u qujtë **paterna** (modele, mostra). Për shembull, qe një patern ilustrues:

| | |
|---|---|
| Emri i paternit: | **Eksperti i Informatave** |
| Problemi: | Cili është principi themelor sipas të cilit iu japim përgjegjësi objekteve? |
| Zgjidhja: | Jepja një përgjegjësi klasës që i ka informatat e nevojshme për me plotësu atë. |

Në dizajnin OO, një **patern** është përshkrim i emëruar për një problem dhe për një zgjidhje që mundet m'u apliku në kontekste të reja; idealisht, një patern na këshillon për atë se si me apliku zgjidhjen e vet në rrethana të ndryshme dhe i konsideron forcat dhe balansimet (trade-off-at). Shumë paterna, për një kategori specifike të dhënë të problemit, udhëzojnë në ndarjen e përgjegjësive për objektet.

Më së thjeshti, një **patern** i mirë është një çift problem/zgjidhje i *emëruar* dhe i *mirënjohur* që mundet m'u apliku në kontekste të reja, me këshillën se si me apliku në situata të reja dhe me diskutimin e balansimeve, implementimeve, variacioneve të saj etj.

### Paternat kanë emra - E rëndësishme!

Zhvillimi i softuerit është fushë e re. Fushat e reja kanë mungesë të emrave të stabilizuar mirë të principeve të veta - dhe kjo e bën komunikimin dhe edukimin të vështirë. Paternat kanë emra, si *Eksperti i Informatave* dhe *Fabrika Abstrakte*. Emërimi i një paterni, i një ideje të dizajnit, apo i një principi i ka përparësitë vijuese:

- E përkrah copëzimin dhe inkorporimin e atij koncepti në kuptimin dhe memorjen tonë
- E ndihmon komunikimin

Kur një patern është i emëruar dhe i publikuar gjerësisht - dhe të gjithë ne pajtohemi ta përdorim atë emër - ne mundemi me diskutu një ide komplekse të dizajnit në fjali më të shkurtëra (apo diagrame më të shkurtëra), një virtyt i abstraksionit. Konsiderojeni diskutimin në vijim të dy zhvilluesve të softuerit, duke përdorë fjalor të emrave të paternave:

**Jill**: "Hej Jack, për nënsistemin e ruajtjes së shënimeve, t'i ekspozojmë serviset me *Fasadë*. Do ta përdorim një *Fabrikë Abstrakte* për *Mapuesit*, dhe *Proxit* për materializimin përtac.

**Jack**: "Çka dreqin the?!?"

**Jill**: "Qe, lexoje këtë..."

## 'Patern i ri' është oksimoron (kundërthënie)

*Paterni i ri* duhet m'u konsideru oksimoron nëse e përshkruan një ide të re. Vetë termi "patern" e sugjeron një gjë që përsëritet për kohë të gjatë. Poenta e paternave të dizajnit *nuk* është me shprehë ide të reja të dizajnit. Krejt e kundërta - paternat e mëdhenj tentojnë me kodifiku njohurinë, idiomat dhe principet *ekzistuese* të provuara dhe të vërteta; sa më i mprehur, më i vjetër, dhe i përdorur më gjerë, aq më mirë.

Si pasojë, paternat GRASP nuk shpallin ide të reja; ato emërojnë dhe kodifikojnë principe themelore të përdorura gjerësisht. Për një ekspert të dizajnit OO, paternat GRASP - sipas idesë nëse jo sipas emrit - do të duken fundamentale dhe familiare. Kjo është poenta!

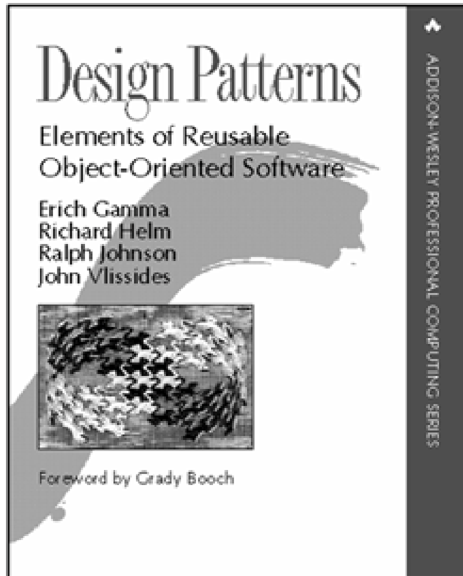
## Libri i paternave të dizajnit i bandës së katërshes (Gang-of-Four)

Ideja e paternave të emëruar në softuer vjen nga Kent Beck (i njohur edhe nga **Programimi Ekstrem**) në mes të 1980tave[4]. Mirëpo, 1994 ishte gur madhor kilometrik në historinë e paternave, në dizajnin OO, dhe në librat e dizajnit të softuerit: u publikua libri i shitur masovikisht dhe influencuesi i madh, *Design Patterns*[5], me autorë Gamma, Helm, Johnson, dhe Vlissides. Libri, i konsideruar si "Bibla" e librave të paternave të dizajnit, i përshkruan 23 paterna për dizajnin OO, me emra si Strategji dhe Adapter. Këto 23 paterna, me autorë katër njerëz, prandaj quhen paternat e dizajnit të **Bandës së Katërshes**[6] (Gang of Four (apo **GoF**) design patterns).

[4] Nocioni i paternave e ka origjinën nga ndërtimi i paternave arkitekturalë i Christopher Alexander. Paternat për softuer e kanë origjinën në 1980tat me Kent Beck, që u bë i vetëdijshëm për punimin e Alexanderit për paterna në arkitekturë, dhe pastaj janë zhvilluar nga Beck me Ward Cunningham në Tektronix.

[5] Publikuesit e listojnë datën e publikimit si 1995, mirëpo ai pat dalë në tetor 1994.

[6] Po ashtu një barcoletë e hollë e lidhur me politikën kineze pas vdekjes së Mao-s.




Mirëpo, *Design Patterns* nuk është libër hyrës; ai supozon njohuri të konsiderueshme paraprake të dizajnit OO dhe të programimit, dhe shumica e shembujve në kod janë në C++.

Është qëllim kryesor i këtij libri (Applying UML and Patterns) m'i ***mësu edhe paternat GRASP edhe paternat esenciale GoF***.

## A është GRASP grup i paternave apo i principeve?

GRASP i definon nëntë principe themelore të dizajnit OO, apo blloqe themelore të ndërtimit në dizajn. Disa kanë pyetur "A nuk i përshkruan GRASP principet e jo paternat?" Një përgjigjje është në fjalët e autorëve Banda e Katërshes, nga parathënia e librit të tyre influencues *Design Patterns*:

> *Paterni i një personi është bllok primitiv i ndërtimit i një personi tjetër.*

Në vend se m'u fokusu në emërtime, ky tekst fokusohet në vlerën pragmatike të përdorimit të stilit të paternave si një *ndihmë e shkëlqyeshme* e të mësuarit për emërtimin, prezentimin, dhe mbajtjen mend të ideve themelore, klasike të dizajnit.

## 17.7. Ku jemi tash?

Deri tash, ky kapitull e ka përmbledhë background'in për dizajnin OO:

1. Background'i i **procesit iterativ** - Artifaktet e mëhershme? Si lidhen ato me modelet e dizajnit OO? Sa kohë duhet me kalu duke e modelu dizajnin?
2. **RDD** si metaforë për dizajnin e objekteve - një komunitet i objekteve të përgjegjshme bashkëpunuese.
3. **Paternat** si mënyrë për m'i emëru dhe m'i shpjegu idetë e dizajnit OO - GRASP për paterna themelorë të ndarjes së përgjegjësive, dhe GoF për ide më të avansuara të dizajnit. Paternat munden m'u zbatu gjatë modelimit dhe gjatë kodimit.
4. **UML** për **modelimin vizual** të dizajnit OO, gjatë të cilës kohë munden m'u zbatu paternat GRASP dhe GoF.

Pasi t'i kemi kuptu këto, është koha m'u fokusu në disa detale të dizajnit të objekteve.

## 17.8. Një shembull i shkurtë i dizajnit të objekteve me GRASP

Seksionet vijuese e eksplorojnë GRASPin më detalisht, por le të fillojmë me një shembull më të shkurtë për m'i pa idetë e mëdha, të aplikuara në studimin e rastit Monopoly. Janë nëntë paterna GRASP; ky shembull e aplikon nëngrupin vijues:

- Krijuesi
- Eksperti i Informatave
- Çiftimi i Ulët
- Kontrolluesi
- Kohezioni i Lartë

### Krijuesi

**Problemi: Kush e krijon objektin Katror (Square)?**

Një nga problemet e para që duhet m'i konsideru në dizajnin OO është: Kush e krijon objektin X? Kjo është përgjegjësi e tipit *bërje*. Për shembull në studimin e rastit Monopoly, kush e krijon objektin e softuerit *Katror*? Tash, *cilido* objekt mundet me kriju një *Katror*, por çka do të zgjedhnin shumë zhvillues? Dhe pse?

Çka po mendoni ta marrim një objekt *Qen* (dmth. një klasë arbitrare) ta kemi si krijues? Jo! Mundemi me ndi deri në eshtra. Pse? Sepse - dhe kjo është pikë kritike - nuk e josh modelin tonë mendor të domenit. *Qeni* nuk e përkrah **zbraztësinë e ulët të reprezentimit** (**LRG**) ndërmjet asaj se si e mendojmë domenin dhe një korrespondence të drejtpërdrejtë me objektet e softuerit. E kam punu këtë problem me mijëra zhvillues, dhe pothuajse të gjithë, prej Indisë e deri në Amerikë, do të thonë,

"Le t'i krijojë objekti *Tabelë* (*Board*) *Katrorët*." Interesante! Reflekton në "intuitë" që zhvilluesit OO të softuerit shpesh (përjashtimet janë eksploru më vonë) duan që "përmbajtësit" m'i kriju gjërat "e përmbajtura", si *Tabelat* m'i kriju *Katrorët*.

Meqë ra fjala, pse po i definojmë klasat e softuerit me emrat *Katror* dhe *Tabelë*, e jo me emrat *AB324* dhe *ZC17*? Përgjigjja: sipas LRG. Kjo e lidh Modelin e Domenit të UP me Modelin e Dizajnit të UP, apo modelin tonë mendor të domenit me realizimin e tij në *shtresën e domenit* të arkitekturës së softuerit.

Me këtë si background, ja definicioni i paternit **Krijues**[7]:

| | |
|---|---|
| Emri: | **Krijues** |
| Problemi: | Kush e krijon një A? |
| Zgjidhja (kjo mundet m'u shiku si këshillë) | Ndaja klasës B përgjegjësinë e krijimit të një instance të klasës A nëse njëra nga këto është e vërtetë (sa më shumë aq më mirë):<br>• B "e përmban" ose në mënyrë kompozite e agregon A-në<br>• B e regjistron (e ruan) A-në<br>• B e përdor ngushtësisht (në mënyrë të afërt) A-në<br>• B i ka shënimet inicializuese për A. |

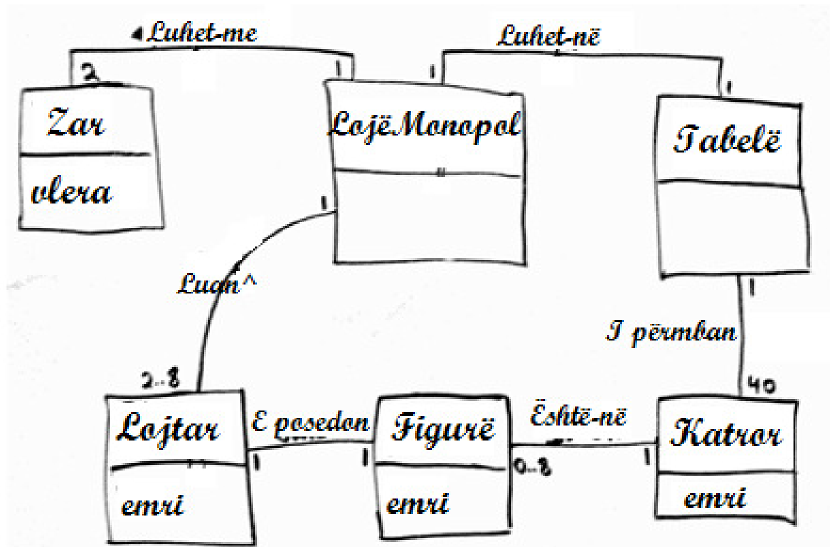
Vëreni se kjo ka të bëjë me ndarjen e përgjegjësive. Ta shohim se si me zbatu Krijuesin.

Së pari, një poentë delikate por e rëndërishme në zbatimin e Krijuesit dhe të paternave tjerë GRASP: B dhe A i referohen objekteve të *softuerit*, jo objekteve të modelit të domenit. Së pari tentojmë me apliku Krijuesin duke i shiku objektet ekzistuese të *softuerit* që e plotësojnë rolin e B. Por çka nëse sapo jemi duke e fillu dizajnin OO, dhe ende s'kemi definu klasa të softuerit? Në këtë rast, sipas LRG, *shih në modelin e domenit* për inspirim.

Kështu, për problemin e krijimit të *Katrorit* (Square), meqë ende nuk janë definu klasa të softuerit, shikojmë në modelin e domenit në Figurën 17.3 dhe shohim se një *Tabelë (Board) përmban Katrorë (Squares)*. Kjo është perspektivë konceptuale, jo e softuerit, por natyrisht ne mundemi me pasqyru atë në Modelin e Dizajnit ashtu që objekti i softuerit *Tabelë* përmban objekte të softuerit *Katror*. Dhe pastaj, konsistent me LRG dhe me këshillën e Krijuesit, *Tabela* do t'i krijojë *Katrorët*. Po ashtu, *Katrorët* do të jenë gjithmonë pjesë e një *Tabele*, dhe *Tabela* e menaxhon krijimin dhe shkatërrimin e tyre; prandaj, ato janë në asociacion të agregimit kompozit me *Tabelën*.

[7] Paternat alternativë të krijimit, siç është Fabrika Konkrete dhe Fabrika Abstrakte, diskutohen më vonë.

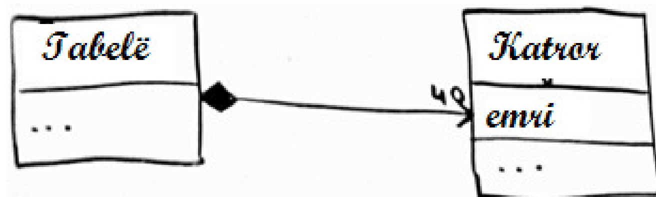
**Figura 17.3. Iteracioni-1 i modelit të domenit të Monopolyt**




Rikujtoni se një praktikë agjile (e shkathët) e modelimit është me kriju modele paralele komplementare dinamike dhe statike. Prandaj, e kam vizatu *edhe* një diagram të pjesshëm sekuencial *edhe* një diagram të klasave për me reflektu këtë vendim të dizajnit në të cilin e kam apliku një patern GRASP gjatë vizatimit të diagrameve UML. Shih Figurën 17.4. dhe Figurën 17.5. Vëreni në Figurën 17.4. se kur krijohet një *Tabelë*, e krijon një *Katror*. Për hir të shkurtësisë në këtë shembull, do ta injoroj çështjen anësore të vizatimit të ciklit për m'i kriju të 40 katrorët.

**Figura 17.4. Aplikimi i paternit Krijues në model dinamik.**

**Figura 17.5. Në një DCD të Modelit të Dizajnit, *Tabela* ka asociacion kompozit të agregimit me *Katrorët*. Jemi duke e apliku Krijuesin në model statik.**



## Eksperti i Informatave (Information Expert)

**Problemi: Kush di për një objekt *Katror*, nëse është dhënë çelësi?**

Paterni Eksperti i Informatave (shpesh i shkurtuar si Eksperti) është një nga principet më themelore të ndarjes së përgjegjësive në dizajnin e objekteve.

Supozojmë se objektet duhet me qenë të afta m'iu referu një *Katrori* të caktuar, duke e ditë emrin e tij. Kush duhet me qenë përgjegjës për me njoftë një *Katror*, duke e pasë çelësin e saj? Natyrisht, kjo është përgjegjësi e *dijes*, por Eksperti aplikohet edhe në *bërje*.

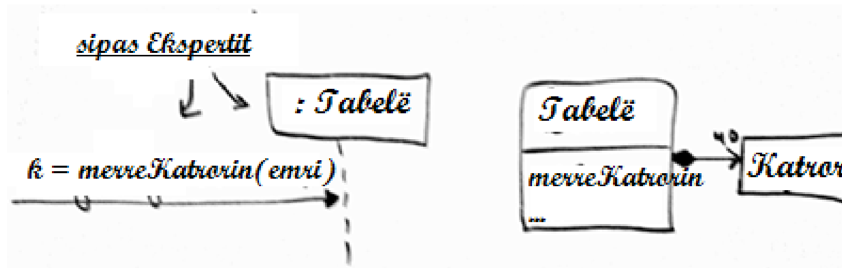
Sikur me Krijuesin, *cilido* objekt mundet me qenë përgjegjës, por çka do të zgjedhnin shumë zhvillues OO? Dhe pse? Sikur me problemin e Krijuesit, shumë zhvillues e zgjedhin objektin *Tabelë*. Duket disi trivialisht e qartë me ia nda këtë përgjegjësi *Tabelës*, mirëpo është edukative me dekonstruktu se pse, dhe me mësu me apliku këtë princip në raste më delikate. Shembujt e mëvonshëm do të bëhen më delikatë.

Eksperti i Informatave e shpjegon se pse është zgjedhë *Tabela*:

| | |
|---|---|
| Emri: | **Eksperti i Informatave** |
| Problemi: | Cili është principi themelor sipas të cilit i ndahen përgjegjësitë objekteve? |
| Zgjidhja (këshilla): | Ndaja përgjegjësinë klasës që e ka informatën e nevojshme për me plotësu atë përgjegjësi. |

Një përgjegjësi ka nevojë për informatat për plotësimin e vet - informatat për objektet tjera, gjendjen e vetë objektit, botën përreth një objekti, informatat që mundet m'i nxjerrë një objekt, e kështu me radhë. Në këtë rast, për me mujtë me lexu dhe me prezantu cilindo *Katror* - duke e pasë emrin e tij - një objekt duhet me ditë (m'i pasë informatat) për të gjithë *Katrorët*. Ne më herët vendosëm, siç shihet në figurën 17.5., se një *Tabelë* softuerike do t'i agregojë të gjitha objektet *Katror*. Prandaj, *Tabela* i ka informatat e nevojshme për me plotësu këtë përgjegjësi. Figura 17.6 e ilustron aplikimin e Ekspertit në kontekstin e vizatimit.

**Figura 17.6. Aplikimi i Ekspertit**




Principi tjetër, Çiftimi i Ulët, e shpjegon se pse Eksperti është princip i dobishëm, thelbësor i dizajnit OO.

## Çiftimi i Ulët (Low Coupling)

**Pyetje: Pse Tabela e jo Qeni?**

Eksperti na udhëzon me ia nda objektit Tabelë përgjegjësinë për me njoftë një Katror të caktuar, duke e pasë emrin unik, sepse Tabela din për të gjithë Katrorët (i ka informatat - është Ekspert i Informatave). Por pse Eksperti e jep këtë këshillë?

Përgjigjja gjendet në principin e Çiftimit të Ulët (Low Coupling). Shkurt dhe joformalisht, **çiftimi** është masë se sa fort është i lidhur një element, ka dijeni, apo varet nga elementet tjera. Nëse ka çiftim apo varësi, atëherë kur elementi varës ndryshon, i varuri mundet m'u ndiku. Për shembull një nënklasë është e çiftuar fuqishëm me një mbiklasë. Një objekt A që i thirr operacionet e objektit B ka çiftim me shërbimet e B.

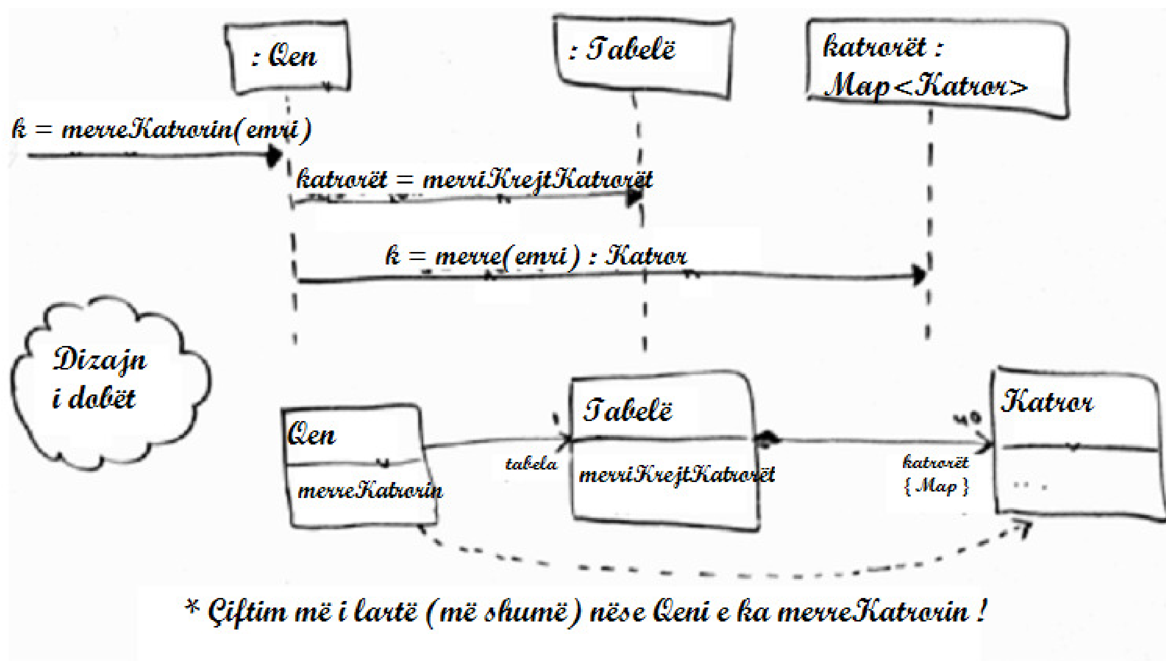
Principi i Çiftimit të Ulët zbatohet në shumë dimensione të zhvillimit të softuerit; është vërtet njëri nga qëllimet kryesore në ndërtimin e softuerit. Në terma të dizajnit të objekteve dhe të përgjegjësive, mundemi me përshkru këshillën si vijon:

| | |
|---|---|
| Emri: | **Çiftimi i Ulët** |
| Problemi: | Si me reduktu ndikimin e ndryshimit? |
| Zgjidhja (këshilla): | Ndaji përgjegjësitë ashtu që çiftimi (i panevojshëm) të mbetet i ulët. Përdore këtë princip për m'i vlerësu alternativat. |

Ne e përdorim Çiftimin e Ulët për m'i *vlerësu* dizajnet ekzistuese ose për me vlerësu zgjedhjen ndërmjet alternativave - kur të gjitha gjërat tjera janë të njëjta, duhet me preferu dizajnin që e ka çiftimin më të ulët se sa alternativat.

Për shembull, siç vendosëm në Figurën 17.5., një objekt Tabelë përmban shumë Katrorë. Pse mos me ia nda merreKatrorin Qenit (dmth. një klase tjetër arbitrare)? Konsideroje ndikimin në terma të çiftimit të ulët. Nëse një Qen e ka merreKatrorin, siç është tregu në skicën UML në Figurën 17.7, ai duhet me bashkëpunu me Tabelën për me marrë koleksionin e të gjithë Katrorëve në Tabelë. Ato janë me gjasë të vendosura në një objekt të koleksionit Map, që lejon marrje sipas çelësit. Pastaj, Qeni mundet m'iu qasë dhe ta kthejë një Katror të caktuar sipas emrit të çelësit.

**Figura 17.7. Vlerësimi i efektit të çiftimit në këtë dizajn.**




Por le ta vlerësojmë çiftimin total me këtë dizajn të dobët me Qen ndaj dizajnit tonë origjinal ku Tabela e bën veprimin merreKatrorin. Në rastin me Qen, edhe Qeni edhe Tabela duhet me ditë për objektet Katror (dy objekte kanë çiftim me Katrorin). Prandaj, çiftimi i përgjithshëm është më i ulët me dizajnin me Tabelë, dhe me të gjitha tjerat të njejta, është më i mirë se sa dizajni me Qen, në termat e përkrahjes së qëllimit të Çiftimit të Ulët.

Në një nivel më të lartë të qëllimit, pse është i dëshirueshëm Çiftimi i Ulët? Me fjalë të tjera, pse do të donim me reduktu ndikimin e ndryshimit? *Sepse Çiftimi i Ulët ka tendencë me zvogëlu kohën, përpjekjet, dhe defektet në ndryshimin e softuerit.* Kjo është përgjigjje e shkurtë, por asi përgjigjje me implikime të mëdha në ndërtimin dhe mirëmbajtjen e softuerit!

## Poentë Kryesore: Eksperti e përkrah Çiftimin e Ulët

Për m'u kthy te motivimi për Ekspertin e Informatave: ai na udhëzon në zgjedhjen që e përkrah Çiftimin e Ulët. Eksperti kërkon nga ne ta gjejmë objektin që i ka informatat më të shumta të kërkuara për përgjegjësinë (p.sh. Tabela) dhe ta ndajmë përgjegjësinë atje.

Nëse e vendosim përgjegjësinë kudo tjetër (p.sh. te Qeni), çiftimi i përgjithshëm do të jetë më i lartë për shkak se më shumë informata apo objekte duhet të shpërndahen tutje nga burimi i vet origjinal ose nga shtëpia, si katrorët në koleksionin Map që duhej të ndahen me Qenin, larg nga shtëpia e tyre në Tabelë.

***Aplikim i UML***: Ju lutem vëreni disa elemente UML në diagramin sekuencial në Figurën 17.7:

- Ndryshorja e vlerës kthyese sqs nga mesazhi merriKrejtKatrorët (getAllSquares) është përdorë edhe për me emëru objektin lifeline në sqs : Map<Square> (sqs : Map<Katror>) (p.sh. një koleksion i tipit Map që mban objekte Katrorë). Referencimi i ndryshores së vlerës kthyese në kutinë e vijës së jetës (lifeline box), për m'i dërgu mesazhe, është gjë e shpeshtë.
- Ndryshorja s në fillimin e mesazhit getSquare dhe ndryshorja s në mesazhin e mëvonshëm s i referohen objektit të njëjtë.
- Shprehja e mesazhit s = get(name) : Square, apo, s = merre(emri) : Katror, tregon se tipi i s është referencë në një instancë Katror.

## Kontrolluesi

Një arkitekturë e thjeshtë me shtresa e ka një shtresë UI (User Interface – Ndërfaqe e Përdoruesit) dhe një shtresë të domenit, ndër të tjerat. Aktorët, siç është vëzhguesi njeri në lojën Monopoly, gjenerojnë ngjarje UI, siç është klikimi në një buton me maus për me lujtë lojën. Objektet UI të softuerit (në Java, për shembull, një dritare JFrame dhe një buton JButton) duhet pastaj me reagu në ngjarjen e klikut të mausit dhe përfundimisht me shkaktu lojën me lujtë.

Nga Principi i Ndarjes Model-Pamje, e dimë se objektet UI duhet mos me përmbajtë logjikë të aplikacionit apo të "biznesit" siç është llogaritja e lëvizjes së një lojtari. Prandaj, posa ta kapin ngjarjen e mausit objektet UI, ato duhet m’ia **delegu** (m'ia përcjellë punën një objekti tjetër) kërkesën objekteve të *domenit* në *shtresën e domenit*.

Paterni Kontrollues i përgjigjjet kësaj pyetje të thjeshtë: Cili objekt *i pari* pas ose përtej shtresës UI duhet me pranu mesazhin nga shtresa UI?

Për me lidhë këtë me diagramet sekuenciale të sistemit, siç tregon edhe rishikimi i figurës 17.8, operacioni kryesor i sistemit është luajeLojën. Disi vëzhguesi njeri e gjeneron një kërkesë luajeLojën (me siguri duke e kliku një buton GUI me tekstin "Luaje lojën") dhe sistemi përgjigjet.

**Figura 17.8. Diagrami sekuencial i sistemit (SSD) për lojën Monopoly. Vëreje operacionin luajeLojën (playGame)**

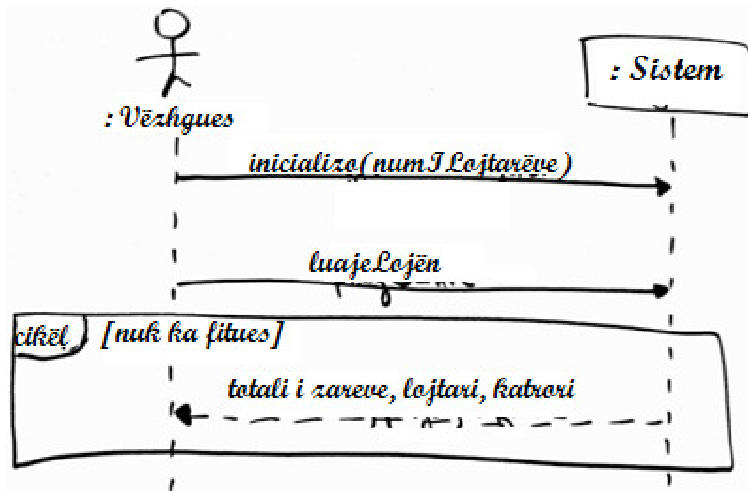
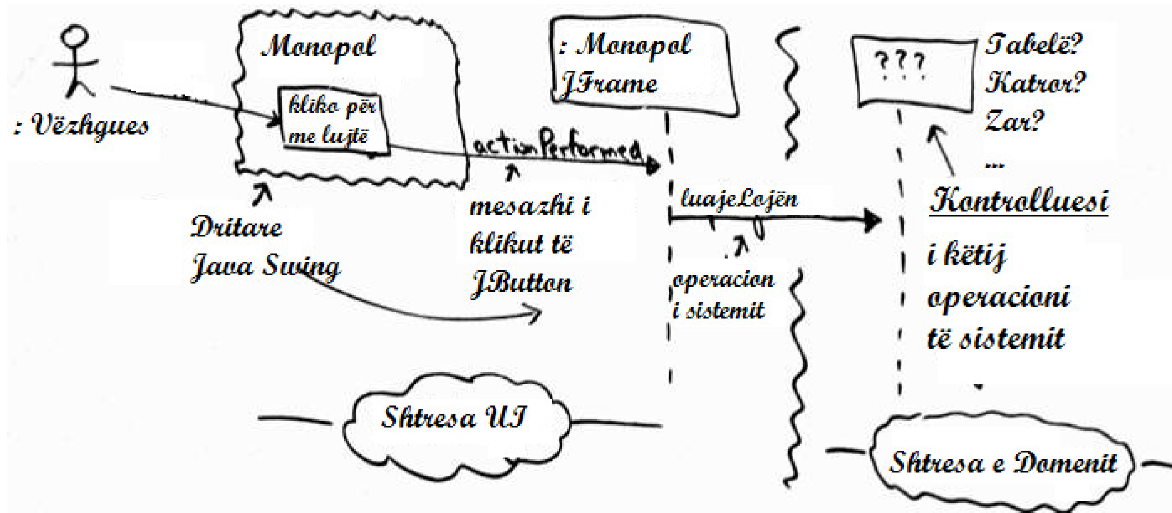



Figura 17.9 e ilustron një pamje më të imtësuar të asaj se çka po ndodh, duke e supozu një dritare JFrame të Java Swing GUI, dhe një buton JButton[8]. Klikimi në JButton ia dërgon një mesazh actionPerformed (aksioni u performua) një objekti, shpesh vetë dritares JFrame, siç e shohim në Figurën 17.9. Pastaj - dhe kjo është **poenta kryesore** - ditarja JFrame duhet me adaptu atë mesazh actionPerformed në diçka semantikisht më kuptimplote, siç është një mesazh luajeLojën (për m'i korrespondu analizës SSD), dhe me ia delegu mesazhin luajeLojën një objekti të domenit në shtresën e domenit.

[8] Objekte, mesazhe dhe paterna të bashkëpunimit të ngjashme vlejnë në .NET, Python etj.

**Figura 17.9. Kush është Kontrolluesi për operacionin e sistemit luajeLojën?**




> A po e shihni lidhjen ndërmjet operacioneve të sistemit të Diagramit Sekuencial dhe dizajnit detal të objekteve prej Ndërfaqjes së Përdoruesit deri te shtresa e domenit? Kjo është e rëndësishme.

Kështu, Kontrolluesi mirret me një çështje të thjeshtë në dizajnin OO: Si me lidhë shtresën UI me shtresën e logjikës së aplikacionit? A duhet me qenë Tabela objekti i parë që e pranon mesazhin luajeLojën nga shtresa UI? Apo diçka tjetër?

Në disa metoda OOA/D, emri *kontrollues* i është dhënë objektit të logjikës së aplikacionit që e ka pranu dhe e ka "kontrollu" (koordinu) trajtimin e kërkesës.

Paterni Kontrollues e ofron këshillën vijuese:

Emri: **Kontrollues**

Problemi: Cili objekt përtej shtresës UI e pranon dhe e koordinon ("e kontrollon") një operacion të sistemit?

Zgjidhja (këshilla): Ndaja përgjegjësinë një objekti që e përfaqëson njërën nga këto zgjedhje:

- E përfaqëson "sistemin" e përgjithshëm, një "objekt rrënjë", një pajisje brenda të cilës ekzekutohet softueri (këto të gjitha janë variacione të *kontrolluesit fasadë*).
- Përfaqëson skenar të një rasti të përdorimit brenda të cilit ndodh operacioni i sistemit (rast i përdorimit apo *kontrollues i sesionit*)

T'i konsiderojmë këto opsione:

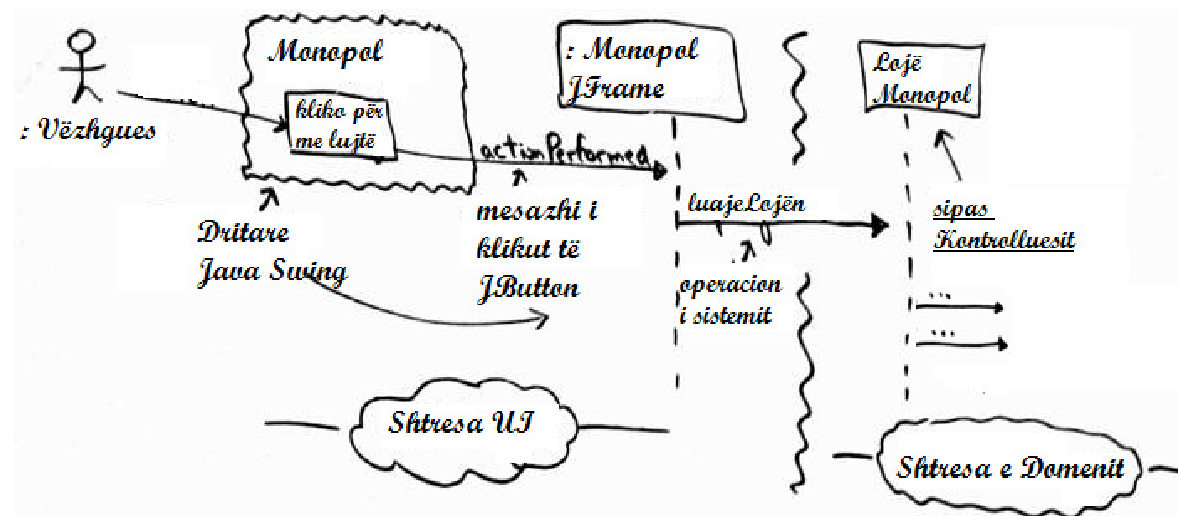
**Opsioni 1**: E përfaqëson "sistemin" e përgjithshëm, apo "objektin rrënjë" siç është një objekt i quajtur LojaMonopoly

**Opsioni 2**: E përfaqëson një pajisje brenda të cilës ekzekutohet softueri - ky opsion i përket pajisjeve të specializuara harduerike siç është një telefon apo një makinë e bankës për para (p.sh. klasa e softuerit Telefon apo MakinëEBankësPërPara); nuk aplikohet në këtë rast.

**Opsioni 3**: E përfaqëson rastin e përdorimit apo sesionin. Rasti i përdorimit brenda të cilit ndodh operacioni i sistemit quhet Luaje Lojën Monopoly. Pra, një klasë e softuerit siç është TrajtuesPërLuajeLojënMonopoly. (Shtesa "TrajtuesPër..." ose "Sesion..." është idiomë në dizajnin OO në rastet kur përdoret ky version).

Opsioni #1, klasa LojaMonopoly, është i arsyeshëm nëse ka vetëm disa operacione të sistemit (më shumë për balansimet kur ta diskutojmë Kohezionin e Lartë). Prandaj, Figura 17.10 e ilustron vendimin e dizajnit të bazuar në Kontrollues.

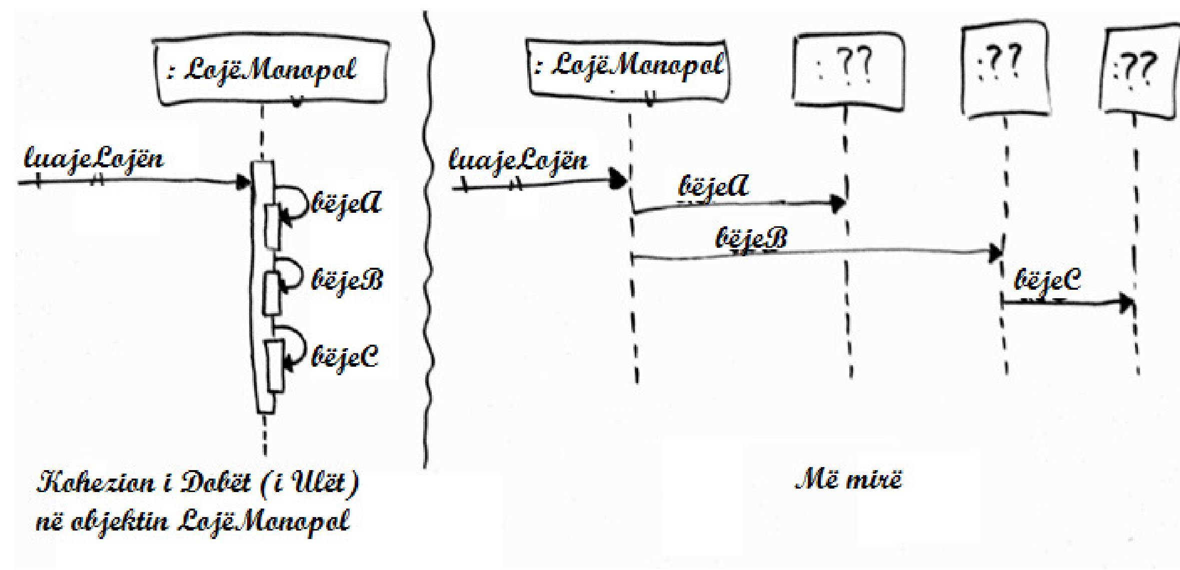
**Figura 17.10. Aplikimi i paternit Kontrollues - duke e përdorë klasën LojaMonopoly. Lidhja e shtresës UI me shtresën e domenit të objekteve të softuerit.**



## Kohezioni i Lartë

Bazuar në vendimin e Kontrolluesit, tash jemi në pikën e dizajnit të treguar në diagramin sekuencial djathtas. Diskutimi detal i dizajnit për atë se çka vjen me radhë - duke e apliku në mënyrë konsistente dhe metodike GRASP - eksplorohet në një kapitull tjetër, por tani për tani i kemi dy qasje kontrastuese të dizajnit që ia vlen m'i konsideru, të ilustruara në Figurën 17.11.

**Figura 17.11. Kontrastimi i nivelit të kohezionit në dizajne të ndryshme.**




Vëreni në versionin e anës së majtë se objekti LojëMonopol e bën punën krejt vetë, dhe në versionin e anës së djathtë e delegon dhe e koordinon punën për kërkesën luajeLojën. Në dizajn të softuerit një cilësi themelore e njohur si **kohezion** e mat joformalisht sa janë të lidhura funksionalisht operacionet e një elementi të softuerit, dhe po ashtu e mat se sa punë po bën një element i softuerit. Si një shembull i thjeshtë kontrastues, një objekt IMadh me 100 metoda dhe 2000 rreshta kod bën shumë më shumë se sa një objekt IVogël me 10 metoda dhe me 200 rreshta kod. Dhe nëse 100 metodat e objektit IMadh janë duke mbulu shumë zona të ndryshme të përgjegjësisë (si qasja në bazë dhe gjenerimi i numrave të rastit), atëherë objekti IMadh ka më pak fokus apo kohezion funksional se sa objekti IVogël. Në mënyrë të përmbledhur, edhe sasia e kodit, edhe lidhshmëria e kodit janë indikator i kohezionit të një objekti.

Për me qenë të qartë, kohezioni i keq (kohezioni i ulët) nuk e ka si pasojë vetëm atë që një objekt e bën punën krejt vetë; në fakt, një objekt me kohezion të ulët me 2000 rreshta kod me siguri bashkëpunon me shumë objekte tjera. Tani, ja ku është një *poentë kryesore*: i tërë ai interaksion ka tendencë me kriju edhe çiftim të keq (të lartë). Kohezioni i keq dhe çiftimi i keq shpesh shkojnë dorë-për-dore.

Në terma të dizajneve kontrastuese në figurën 17.11, versioni i anës së majtë i objektit LojaMonopoly ka kohezion më të keq se sa versioni i anës së djaththë, meqë versioni i anës së majtë po e bën objektin LojaMonopoly me kry vetë krejt punën, në vend se me delegu dhe me shpërnda punën ndërmjet objekteve. Kjo çon në principin e Kohezionit të Lartë, që përdoret për m'i vlerësu zgjedhjet e ndryshme të dizajnit. Kur të gjitha të tjerat janë të barabarta, prefero dizajn me kohezion më të lartë.

| | |
|---|---|
| Emri: | **Kohezioni i Lartë** |
| Problemi: | Si m'i mbajtë objektet të fokusuara, të kuptueshme, dhe të menaxhueshme, dhe si efekt anësor, me përkrahë Çiftimin e Ulët? |
| Zgjidhja (këshilla): | Ndaji përgjegjësitë ashtu që kohezioni të mbetet i lartë. Përdore këtë për m'i vlerësu alternativat. |

Mundemi me thënë se dizajni i anës së djathtë e përkrah më mirë Kohezionin e Lartë se sa versioni i anës së majtë.

## 17.9. Aplikimi i GRASP në Dizajnin e Objekteve

GRASP është shkurtesë për **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns[9] (Paternat e Përgjithshme të Softuerit për Ndarjen e Përgjegjësive). Emri është zgjedhë për me sugjeru rëndësinë e zotërimit (grasping) të këtyre principeve për me dizajnu në mënyrë të suksesshme softuer të orientuar kah objektet.

M'i kuptu dhe me qenë të aftë m'i apliku idetë prapa GRASP - gjatë kodimit apo gjatë vizatimit të diagrameve të interaksionit dhe të klasave - ua mundëson zhvilluesve që janë të rinj në nevojat e teknologjisë së objekteve, m'i zotëru këto principe themelore sa më shpejt që është e mundur; ato formojnë themel për dizajnimin e sistemeve OO.

Janë nëntë paterna GRASP:

| | | |
|---|---|---|
| Krijuesi | Kontrolluesi | Fabrikimi i Pastër |
| Eksperti i Informatave | Kohezioni i Lartë | Indireksioni |
| Çiftimi i Ulët | Polimorfizmi | Variacionet e Mbrojtura |

Pjesa e mbetur e këtij kapitulli i rishqyrton pesë të parat më detalisht; katër tjerat prezentohen në Kapitullin 25.

[9] Teknikisht, duhet me shkru "GRAS Patterns" e jo "GRASP Patterns", mirëpo e dyta tingëllon më mirë.

## 17.10. Krijuesi

**Problemi**

Kush duhet me qenë përgjegjës për krijimin e një instance të re të ndonjë klase?

Krijimi i objekteve është një nga aktivitetet më të shpeshta në një sistem të orientuar kah objektet (OO). Rrjedhimisht, është e dobishme me pasë një princip gjeneral për ndarjen e përgjegjësive të krijimit. I ndarë mirë, dizajni mundet me përkrahë çiftimin e ulët, qartësinë e rritur, enkapsulimin, dhe ripërdorshmërinë.

**Zgjidhja**

Ndaja klasës B përgjegjësinë për me kriju instancë të klasës A nëse njëra nga këto është e vërtetë (sa më shumë aq më mirë)[10]:

- B "e përmban" apo e agregon në mënyrë kompozite A-në
- B e regjistron A-në
- B e përdor ngushtësisht A-në
- B i ka shënimet inicializuese për A që do t'i pasohen A-së kur të krijohet. Prandaj B është Ekspert në aspektin e krijimit të A-së.

B është *krijues* i objekteve A.

Nëse vlen më shumë se një opsion, zakonisht preferoni një klasë B që e agregon apo që e përmban klasën A.

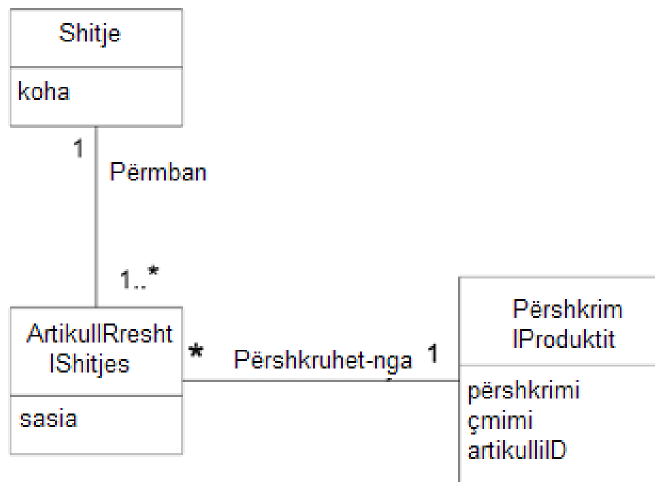
**Shembull**

Në aplikacionin NextGen POS, kush duhet me qenë përgjegjës për krijimin e një instance ArtikullRreshtIShitjes (SalesLineItem)? Sipas Krijuesit, duhet me kërku klasën që agregon, që përmban, etj, instanca ArtikullRreshtIShitjes. Konsiderojeni modelin e pjesshëm të domenit në figurën 17.12.

---

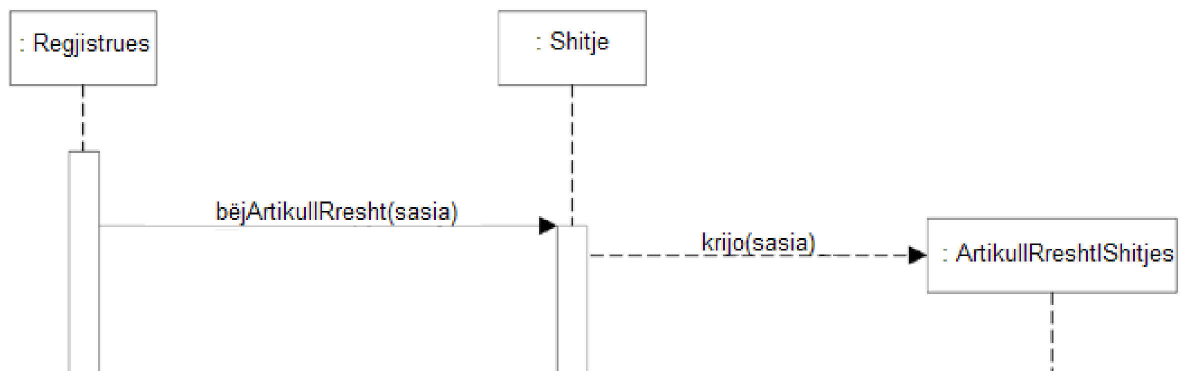
[10] Paternat tjerë të krijimit, siç është Fabrika Konkrete dhe Fabrika Abstrakte, eksplorohen më vonë.

**Figura 17.12. Modeli i pjesshëm i domenit**




Meqë një Shitje përmban (në fakt, agregon) shumë objekte ArtikullRreshtIShitjes, paterni Krijues sugjeron që Shitja është kandidat i mirë për me pasë përgjegjësinë e krijimit të instancave ArtikullRreshtIShitjes. Kjo çon në dizajnin e interaksioneve të objekteve në Figurën 17.13.

**Figura 17.13. Krijimi i një objekti ArtikullRreshtIShitjes (SalesLineItem)**




Kjo ndarje e përgjegjësive kërkon që një metodë krijoArtikullRresht (makeLineItem) të definohet në Shitje. Edhe një herë, konteksti në të cilin e konsideruam dhe vendosëm për këto përgjegjësi ishte gjatë vizatimit të një diagrami të interaksionit. Seksioni i metodës i një diagrami të klasave pastaj mundet m'i përmbledhë rezultatet e ndarjes së përgjegjësive, konkretisht të realizuara si metoda.

**Diskutim**

Krijuesi udhëzon në ndarjen e përgjegjësive që kanë të bëjnë me krijimin e objekteve, një punë shumë e shpeshtë. Qëllimi themelor i paternit Krijues është me gjetë një krijues që ka nevojë m'u lidhë me objektin e krijuar në rastin e cilësdo ngjarje. Zgjedhja e tij si krijues e përkrah çiftimin e ulët.

Kompoziti *e agregon* Pjesën, Përmbajtësi *e përmban* Përmbajtjen, dhe Regjistruesi *regjistron*. Krijuesi sugjeron se përmbajtësi ose klasa regjistruese është kandidat i mirë për përgjegjësinë e krijimit të gjësë që përmbahet apo regjistrohet. Natyrisht, kjo është vetëm këshillë.

Vëreni se i jemi kthyer konceptit të kompozimit gjatë konsiderimit të paternit Krijues. Një objekt kompozit është kandidat i shkëlqyer për m'i ndërtu pjesët e veta.

Nganjëherë ju e identifikoni një krijues duke e kërku klasën që i ka shënimet inicializuese që do të dërgohen gjatë krijimit. Ky është në fakt shembull i paternit Ekspert. Shënimet inicializuese dërgohen gjatë krijimit përmes ndonjë lloj metode të inicializimit, siç është konstruktori Java që ka parametra. Për shembull, supozoni se një instancë Pagesë, kur krijohet, ka nevojë m'u inicializu me totalin e Shitjes. Meqë Shitja e di totalin, Shitja është kandidat krijues i Pagesës.

**Kundërindikacionet**

Shpesh, krijimi kërkon kompleksitet të konsiderueshëm, siç është përdorimi i instancave të recikluara për performansë, krijimi i kushtëzuar i një instance nga një anëtar i një familje të klasave të ngjashme duke u bazu në ndonjë vlerë të jashtme të vetisë, etj. Në këto raste, është e këshillueshme me ia delegu krijimin një klase ndihmëse që quhet *Fabrikë Konkrete* apo *Fabrikë Abstrakte* e jo me përdorë klasën e sugjeruar nga *Krijuesi*.

**Përfitimet**

- Përkrahet Çiftimi i Ulët, që implikon varësi më të ulëta të mirëmbajtjes dhe shanse më të larta për ripërdorim. Çiftimi sigurisht nuk rritet sepse klasa *e krijuar* me gjasë është tashmë e dukshme për klasën *krijues*, për shkak të asociacioneve ekzistuese që e motivuan zgjedhjen e saj si krijues.

**Paternat apo Principet e afërta**

- Çiftimi i ulët
- Fabrika Konkrete dhe Fabrika Abstrakte
- Pjesa e Plotë (Whole-Part) e përshkruan një patern për m'i definu objektet agregate që e përkrahin enkapsulimin e komponentave.

## 17.11. Eksperti i Informatave (ose Eksperti)

**Problemi**

Cili është principi i përgjithshëm për me ua nda përgjegjësitë objekteve?

Një Model i Dizajnit mundet me definu qindra apo mijëra klasa të softuerit, dhe një aplikacion mundet me kërku m'u plotësu qindra apo mijëra përgjegjësi. Gjatë dizajnit të objekteve, kur interaksionet ndërmjet objekteve janë të definuara, ne bëjmë zgjedhje për me ua nda përgjegjësitë klasave të softuerit. Nëse kemi zgjedhë mirë, sistemet kanë tendencë me qenë më lehtë të kuptueshme, m'i mirëmbajtë, dhe m'i zgjeru më lehtë, dhe zgjedhjet tona na lejojnë më shumë shanse m'i ripërdorë komponentat në aplikacionet e ardhshme.

**Zgjidhja**

Ndaja përgjegjësinë ekspertit të informatave - klasës që i ka *informatat* e nevojshme me plotësu përgjegjësinë.

**Shembull**

Në aplikacionin NextGEN POS, një klasë duhet me ditë totalin e plotë për një shitje.

| Fillo t'i ndash përgjegjësitë duke e deklaruar qartë përgjegjësinë. |
|---|

Sipas kësaj këshille, deklarata është:

> Kush duhet me qenë përgjegjës për me ditë totalin e plotë për një shitje?

Sipas Ekspertit të Informatave, duhet me kërku atë klasë të objekteve që e ka informatën e nevojshme me përcaktu totalin.

Tash vijmë te një pyetje kryesore: A shikojmë në Modelin e Domenit apo në Modelin e Dizajnit për m'i analizu klasat që e kanë informatën e duhur? Modeli i Domenit i ilustron klasat konceptuale të domenit të botës reale; Modeli i Dizajnit i ilustron klasat e softuerit.

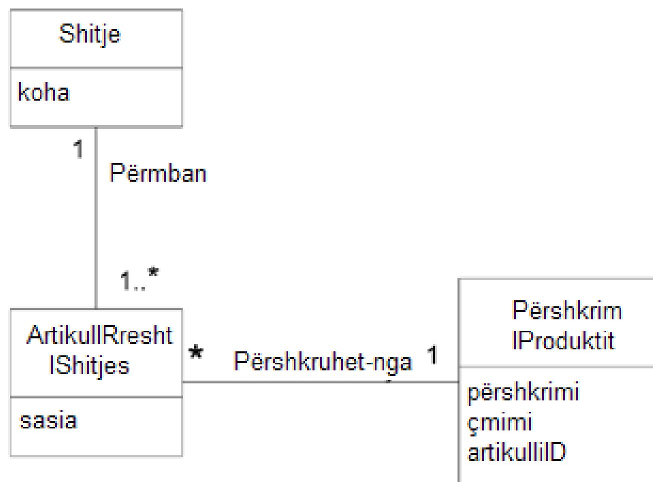
Përgjigjja:

1. Nëse ka klasa relevante në Modelin e Dizajnit, së pari shikoni aty.
2. Përndryshe, shikoni në Modelin e Domenit, dhe tentoni m'i përdorë (apo m'i zgjeru) reprezentimet e tij për me inspiru krijimin e klasave korresponduese të dizajnit.

Për shembull, e zëmë se sapo po fillojmë punën e dizajnit, dhe nuk ka, apo ka Model minimal të Dizajnit. Atëherë, në Modelin e Domenit kërkojmë Ekspertë të Informatave; ndoshta Shitja e botës reale është një i tillë. Pastaj, e shtojmë një klasë të softuerit në Modelin e Dizajnit ngjashëm me emrin Shitje, dhe ia japim përgjegjësinë me ditë totalin e vet, të shprehur me metodën me emrin merreTotalin. Kjo qasje e përkrah *zbraztësinë e ulët reprezentuese* në të cilën dizajni softuerik i objekteve lidhet me konceptet tona se si është i organizuar domeni real.

Për me shqyrtu këtë rast detalisht, konsideroje Modelin e pjesshëm të Domenit në Figurën 17.14.

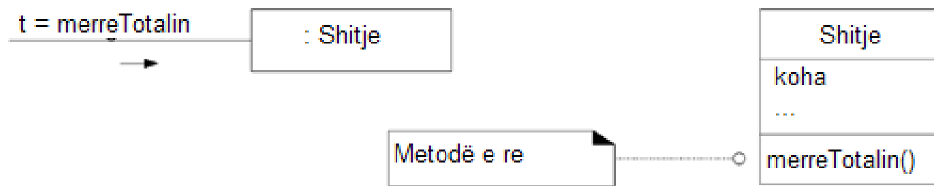
**Figura 17.14. Asociacionet e Shitjes**




Çfarë informate na duhet me përcaktu totalin e plotë? Duhet me ditë për të gjitha instancat ArtikullRreshtIShitjes për një shitje dhe shumën e nëntotaleve të tyre. Një instancë Shitje i përmban këto; prandaj, sipas udhëzimit të Ekspertit të Informatave, Shitja është klasë e përshtatshme e objektit për këtë përgjegjësi; është *ekspert i informatave* për atë punë.

Ashtu si u përmend, është në kontekstin e krijimit të diagrameve të interaksionit ku shpesh ngriten këto çështje të përgjegjësisë. Paramendoni se po fillojmë me punu nëpër vizatimin e diagrameve për me ua nda përgjegjësitë objekteve. Një diagram i pjesshëm i interaksionit dhe një diagram i klasave në Figurën 17.15 i ilustrojnë disa vendime.

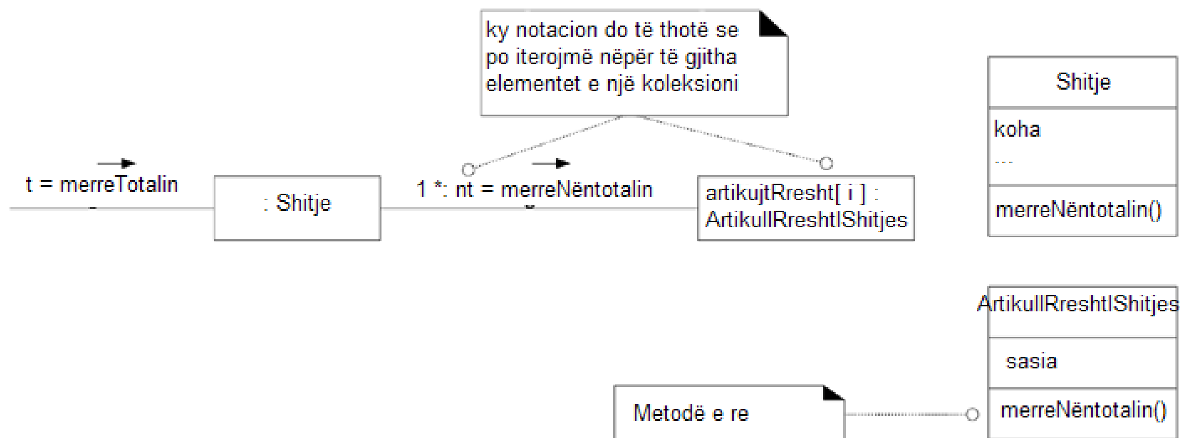
**Figura 17.15. Diagramet e pjesshme të interaksionit dhe të klasave**




Ende s'e kemi kry. Çfarë informate na duhet me përcaktu nëntotalin e artikullit të rreshtit? ArtikullRreshtIShitjes.sasia dhe PërshkrimIProduktit.çmimi. Objekti ArtikullRreshtIShitjes e din sasinë e vet dhe PërshkriminEProduktit të vet të asociuar; prandaj, sipas Ekspertit, objekti ArtikullRreshtIShitjes duhet me përcaktu nëntotalin; është *eksperti i informatave*.

Në terma të një diagrami të interaksionit, kjo domethënë se Shitja duhet me ua dërgu mesazhet merreNëntotalin secilit nga objektet ArtikullRreshtIShitjes dhe m'i mbledhë rezultatet; ky dizajn është tregu në Figurën 17.16.

**Figura 17.16. Llogaritja e totalit të Shitjes.**

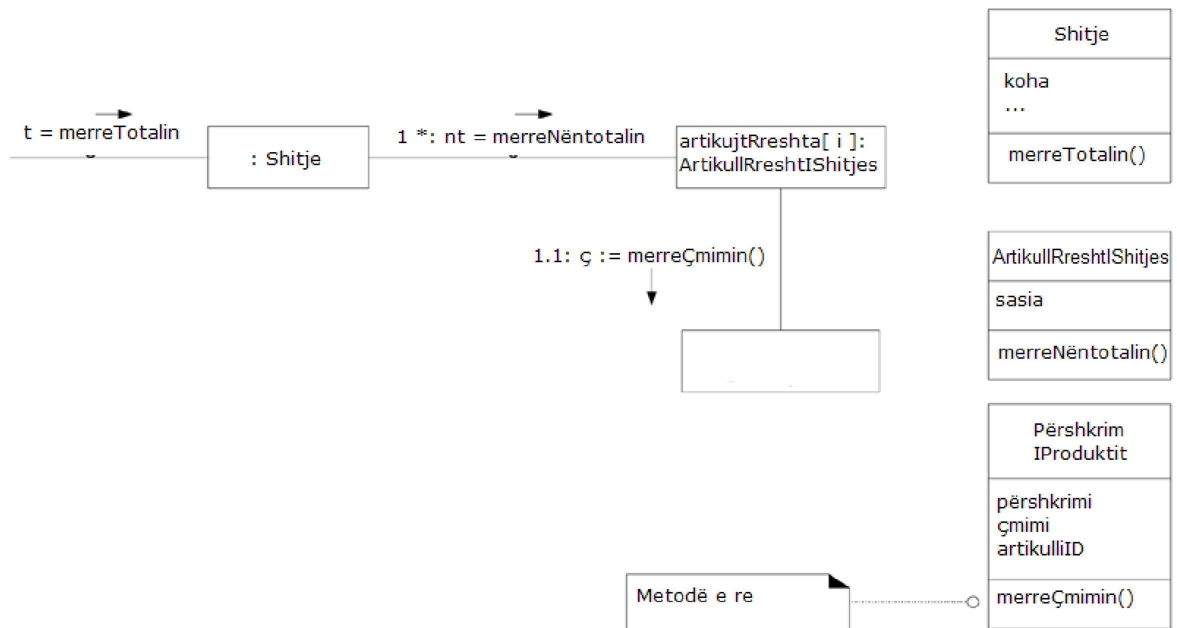



Për me plotësu përgjegjësinë për me ditë dhe m'u përgjigjë për nëntotalin e vet, një ArtikullRreshtIShitjes duhet me ditë çmimin e produktit.

PërshkrimiIProduktit është ekspert i informatave për m'u përgjigjë për çmimin e vet; prandaj, objekti ArtikullRreshtIShitjes ia dërgon një mesazh duke e pytë për çmimin e produktit.

Dizajni është tregu në Figurën 17.17.

**Figura 17.17. Llogaritja e totalit të Shitjes.**




Si përfundim, për me plotësu përgjegjësinë për me ditë dhe m'u përgjegjë për totalin e shitjes, ua kemi nda tri përgjegjësi tri klasave të dizajnit të objekteve si vijon.

| **Klasa e Dizajnit** | **Përgjegjësia** |
|---|---|
| Shitje | e din totalin e shitjes |
| ArtikullRreshtIShitjes | e din nëntotalin e artikullit të rreshtit |
| PërshkrimIProduktit | e din çmimin e produktit |

E konsideruam dhe vendosëm për këto përgjegjësi në kontekstin e vizatimit të një diagrami të interaksionit. Pastaj kishim mundë m'i përmbledhë metodat në seksionin e metodave në diagramin e klasave.

Principi sipas të cilit e kemi nda secilën përgjegjësi ishte Eksperti i Informatave - duke e vendosë përgjegjësinë te objekti që e kishte informatën e duhur për me plotësu atë përgjegjësi.

**Diskutim**

Eksperti i Informatave përdoret shpesh në ndarjen e përgjegjësive; është princip themelor udhëzues që përdoret vazhdimisht në dizajnin e objekteve. Eksperti nuk është mendu me qenë ide e mjegullt apo ekstravagante; ai e shpreh "intuitën" e zakonshme që objektet i bëjnë gjërat në lidhje me informatat që i kanë.

Vëreni se plotësimi i një përgjegjësie shpesh kërkon informatë që është e shpërndarë nëpër klasa të ndryshme të objekteve. Kjo implikon që shumë ekspertë "të pjesshëm" të informatave do të bashkëpunojnë në detyrë. Për shembull problemi i totalit të shitjes në fund e kërkoi bashkëpunimin e tri klasave të objekteve. Kurdo që informata shpërndahet nëpër objekte të ndryshme, ata do të duhet me ndërvepru përmes mesazheve për me nda punën.

Eksperti zakonisht çon në dizajne ku një objekt i softuerit i bën ato operacione që normalisht bëhen në gjënë e pajetë në botën reale të cilën e prezenton ai; Peter Coad këtë e quan strategjia "E Bëj Vetë" (Do It Myself). Për shembull, në botën reale, pa përdorimin e ndihmave elektro-mekanike, një shitje nuk jua tregon totalin e vet; është gjë e pajetë. Dikush e llogarit totalin për një shitje. Por në botën e softuerit të orientuar kah objektet, të gjitha objektet softuerike janë "të gjalla" apo "që jetojnë", dhe munden me marrë përgjegjësi dhe me bë gjëra. Fundamentalisht, ata i bëjnë gjërat që kanë lidhje me informatën që e dinë. Unë e quaj këtë principi i "animacionit" në dizajnin e objekteve; është si me qenë në një film të vizatuar ku çdo gjë është e gjallë.

Paterni Eksperti i Informatave - si shumë gjëra në teknologjinë e objekteve - e ka analogjinë e botës reale. Ne shpesh iu japim përgjegjësi individëve që e kanë informatën e nevojshme për me kry ni punë. Për shembull, në një biznes, kush duhet me qenë përgjegjës për me kriju një deklaratë të profitit-dhe-humbjes? Personi që ka qasje në të gjitha informatat e nevojshme për me kriju - ndoshta oficeri kryesor financiar. Dhe ashtu sikur objektet e softuerit bashkëpunojnë sepse informata është e shpërndarë rreth e rrotull, ashtu është edhe me njerëzit. Oficeri kryesor financiar i kompanisë mundet me kërku nga kontabilistët me gjeneru raporte për kreditë dhe debitë.

**Kundërindikacionet**

Në disa situata, zgjidhja e sugjerar prej Ekspertit është e padëshirueshme, zakonisht për shkak të problemeve me çiftim dhe kohezion (këto principe janë të diskutuara më vonë në këtë kapitull).

Për shembull, kush duhet me qenë përgjegjës për me rujtë një Shitje në bazë? Sigurisht, shumica e informatave që do të ruhen janë në objektin Shitje, dhe kështu Eksperti mundet me argumentu se përgjegjësia qëndron te klasa Shitje. Dhe, sipas zgjerimit logjik të këtij vendimi, secila klasë duhet m'i pasë shërbimet e veta për ruajtë vetëveten në bazë. Mirëpo veprimi sipas atij rezonimi çon në probleme në kohezion, çiftim dhe duplikim. Për shembull, klasa Shitje tash duhet me përmbajtë logjikë që ka të bëjë me trajtimin e bazës, siç është ajo që ka të bëjë me SQL dhe JDBC (Java Database Connectivity - Lidhshmëria e Java me Bazë). Klasa më nuk fokusohet vetëm në logjikën e pastër të aplikacionit "me qenë shitje". Tash lloje tjera të përgjegjësive e ulin kohezionin e saj. Klasa duhet m’u çiftu me shërbimet teknike të bazës të një nënsistemi tjetër, siç janë serviset JDBC, e jo vetëm m'u çiftu me objektet tjera në shtresën e domenit të objekteve të softuerit, kështu që çiftimi i

saj rritet. Dhe me siguri logjikë e ngjashme e bazës do të duplikohej në shumë klasa të persistencës (ruajtjes së qëndrueshme të shënimeve).

Të gjitha këto probleme indikojnë shkelje të një principi themelor arkitektural: dizajni për ndarje të çështjeve madhore të sistemit. Mbaje logjikën e aplikacionit në një vend (siç janë objektet e domenit të softuerit), mbaje logjikën e bazës në një vend tjetër (siç është një nënsistem i veçantë persistent i serviseve), e kështu me radhë, e jo m'i përzi çështjet e ndryshme të sistemit në komponentën e njëjtë[11].

Përkrahja e ndarjes së çështjeve të mëdha e përmirëson çiftimin dhe kohezionin në dizajn. Prandaj, edhe pse sipas Ekspertit mundemi me gjetë justifikim për me vendosë përgjegjësinë për serviset e bazës në klasën Shitje, për arsye tjera (zakonisht kohezioni dhe çiftimi), kishim me përfundu me dizajn të dobët.

**Përfitimet**

- Mirëmbahet enkapsulimi i informatave meqë objektet e përdorin informatën e vet për m'i plotësu detyrat. Kjo zakonisht e përkrah çiftimin e ulët, që çon në sisteme më të fuqishme dhe më të mirëmbajtshme. Çiftimi i Ulët është po ashtu patern GRASP që diskutohet në një seksion vijues.
- Sjellja është e shpërndarë nëpër klasa që e kanë informatën e e kërkuar, duke inkuraju kështu definicione më kohezive "më të lehta për kah pesha" të klasave që janë më të lehta m'i kuptu dhe m'i mirëmbajtë. Zakonisht përkrahet Kohezioni i Lartë (një tjetër patern që diskutohet më vonë).

**Paternat apo Principet e Afërta**

- Çiftimi i Ulët
- Kohezioni i Lartë

**E Njohur edhe Si; E Ngjashme Me**

"Vendosi përgjegjësitë me shënimet", "Ai që e din e bën", "E bëj vetë", "Vendosi serviset me atributet në të cilat punojnë".

[11] Shih Kapitullin 33 për një diskutim për ndarjen e çështjeve (apo shqetësimeve - separation of concerns).

## 17.12. Çiftimi i Ulët

**Problemi**

Si me përkrahë varësinë e dobët, ndikimin e dobët të ndryshimit, dhe ripërdorimin e rritur?

**Çiftimi** është masë se sa fort një element është lidhë, ka njohuri për, apo mbështetet në elementet tjera. Një element me çiftim të ulët (apo të dobët) nuk varet në shumë elemente tjera; "shumë" varet nga konteksti, por e shqyrtojmë sido që të jetë. Këto elemente përfshijnë klasa, nënsisteme, sisteme, etj.

Një klasë me çiftim të lartë (apo të fortë) mbështetet në shumë klasa të tjera. Klasat e tilla munden me qenë të padëshirueshme; disa vuajnë nga problemet vijuese:

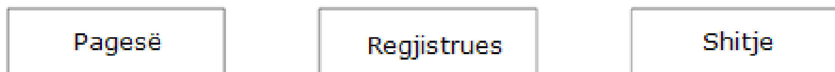
- Ndryshime të detyruara lokale për shkak të ndryshimeve në klasat e afërta
- Më të vështira m'i kuptu të izoluara
- Më të vështira m'i ripërdorë meqë përdorimi i tyre kërkon prezencë të klasave shtesë në të cilat varen

**Zgjidhja**

Ndaje përgjegjësinë ashtu që çiftimi mbetet i ulët. Përdore këtë princip për m'i vlerësu alternativat.

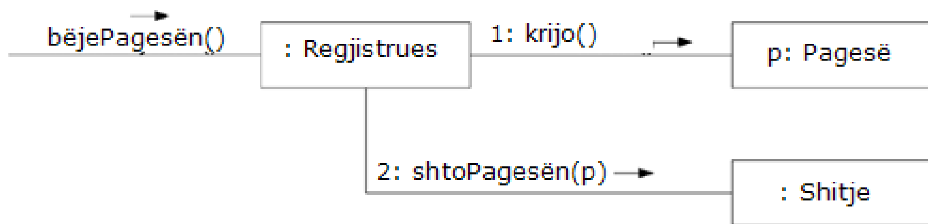
**Shembull**

Konsideroje diagramin vijues parcial (të pjesshëm) nga një studim i rastit NextGen:




Supozojmë se nevojitet me kriju një instancë Pagesë dhe me asociu me Shitjen. Cila klasë duhet me qenë përgjegjëse për këtë? Meqë një Regjistrues "e regjistron" një Pagesë në domenin e botës reale, paterni Krijues e sugjeron Regjistruesin si kandidat për krijimin e Pagesës. Instanca Regjistrues pastaj mundet me ia dërgu Shitjes një mesazh shtoPagesë, duke ia dërgu Pagesën e re si parametër. Një diagram i mundshëm parcial i interaksionit që e reflekton këtë është tregu në Figurën 17.18.

**Figura 17.18. Regjistruesi e krijon Pagesën**

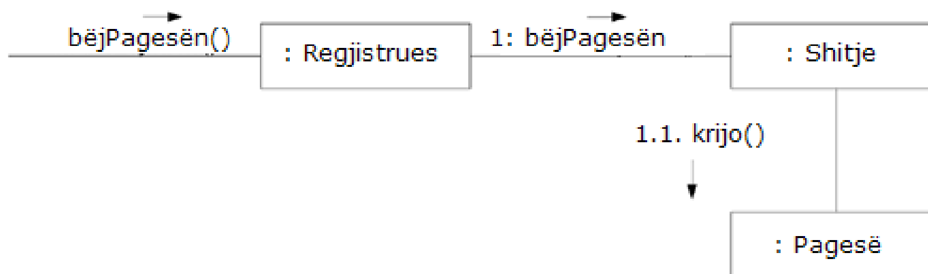



Kjo ndarje e përgjegjësive e çifton klasën Regjistrues me dijeninë për klasën Pagesë.

Zbatim i UML: Vëreni se instanca Pagesë është emëru në mënyrë eksplicite me p ashtu që në mesazhin 2 mundet m'u referencu si parametër.

Figura 17.19. e tregon një zgjidhje alternative për krijimin e Pagesës dhe asociimin e saj me Shitjen.

**Figura 17.19. Shitja e krijon Pagesën.**




Cili dizajn, duke u bazu në ndarjen e përgjegjësive, e përkrah Çiftimin e Ulët? Në të dy rastet supozojmë se Shitja duhet eventualisht m'u çiftu me dijeninë për një Pagesë. Dizajni 1, në të cilin Regjistruesi e krijon Pagesën, e shton çiftimin e Regjistruesit me Pagesën; Dizajni 2, në të cilin Shitja e bën krijimin e pagesës, nuk e rrit çiftimin. Thjesht nga pikëpamja e çiftimit, preferoni Dizajnin 2 meqë e mirëmban çiftimin e përgjithshëm të ulët. Ky shembull e ilustron se si dy paterna - Çiftimi i Ulët dhe Krijuesi – munden me sugjeru zgjidhje të ndryshme.

> Në praktikë, vetëm niveli i çiftimit nuk mund të konsiderohet i izoluar nga principet tjera siç janë Eksperti dhe Kohezioni i Lartë. Megjithatë, është faktor me konsideru në përmirësimin e dizajnit.

**Diskutim**

Çiftimi i Ulët është princip për me mbajtë mend gjatë të gjitha vendimeve të dizajnit: është një qëllim themelor që duhet me konsideru vazhdimisht. Është **princip vlerësues** që e aplikoni gjatë vlerësimit të të gjitha vendimeve të dizajnit.

Në gjuhët e orientuara kah objektet si C++, Java, dhe C#, format më të shpeshta të çiftimit nga TipiX në TipinY i përfshijnë këto:

- TipiX e ka një atribut (shënim anëtar apo ndryshore instancë) që i referohet një instance TipY, apo vetë TipitY
- Një objekt TipX i thirr serviset e një objekti TipY
- Objekti TipX e ka një metodë që i referohet një instance të TipitY, apo vetë TipitY, në çfarëdo mënyre. Këto zakonisht e përfshijnë një parametër apo ndryshore lokale të tipit TipY, apo objekti i kthyer nga një mesazh me qenë instancë e TipitY.
- TipiX është nënklasë direkte apo indirekte e TipitY
- TipiY është interfejs, dhe TipiX e implementon atë interfejs.

Çiftimi i Ulët ju inkurajon me nda një përgjegjësi ashtu që vendosja e saj nuk e rrit çiftimin në një nivel që çon në rezultate negative që mundet m'i prodhu çiftimi i lartë.

Çiftimi i Ulët e përkrah dizajnin e klasave që janë më të pavarura, që e redukton ndikimin e ndryshimit. Ai nuk mundet m'u konsideru i izoluar nga paternat tjerë si Eksperti dhe Kohezioni i Lartë, por duhet m'u përfshi si një nga disa principe të dizajnit që ndikojnë në një zgjedhje në ndarjen e përgjegjësive.

Një nënklasë është fuqishëm e çiftuar me mbiklasën e vet. Konsideroni me kujdes cilindo vendim që e nxjerrni nga një mbiklasë meqë është një formë kaq e fortë e çiftimit. Për shembull, supozojmë se objektet duhet m'u rujtë në mënyrë të qëndrueshme në një bazë relacionale apo të objekteve. Në këtë rast, mundeni me ndjekë praktikën relativisht të shpeshtë të dizajnit, me kriju një mbiklasë abstrakte të quajtur ObjektIQëndrueshëm (persistent) nga e cila nxirren klasat tjera. Mangësia e këtij nënklasimi është se i çifton shumë objektet e domenit me një servis të caktuar teknik dhe i përzien çështjet e ndryshme arkitekturale, derisa përparësia është trashëgimi automatik i sjelljes persistente.

Nuk mundeni me pasë masë absolute se kur është çiftimi shumë i lartë. Ajo çka është me rëndësi është se mundeni me matë shkallën aktuale të çiftimit dhe me vlerësu se a do të çojë në probleme të reja rritja e saj. Në përgjithësi, klasat që janë në mënyrë qenësore të përgjithshme për nga natyra dhe me gjasë të madhe për m'u ripërdorë duhet me pasë çiftim posaçërisht të ulët.

Rasti ekstrem i Çiftimit të Ulët është pa çiftim ndërmjet klasave. Ky rast e shkel metaforën qendrore të teknologjisë së objekteve: një sistem i objekteve të lidhura që komunikojnë përmes mesazheve. Çiftimi i Ulët i tepruar jep dizajn të dobët - asi me disa objekte jokohezive, të fryra, dhe komplekse aktive që e bëjnë krejt punën, dhe me shumë objekte pasive me zero-çiftim që veprojnë si depo të thjeshta të informatave. Një shkallë e mesme e çiftimit ndërmjet klasave është normale dhe e nevojshme për me kriju sistem të orientuar kah objektet, në të cilin punët kryhen nga një bashkëpunim ndërmjet objekteve të lidhura.

**Kundërindikacionet**

Çiftimi i lartë me elemente stabile dhe të përhapura është rrallë problem. Për shembull, një aplikacion J2EE mundet me çiftu veten në mënyrë të sigurtë me libraritë Java (java.util, etj.), sepse janë stabile dhe gjerësisht të përhapura.

## Zgjedhni betejat tuaja

Nuk është problem çiftimi i lartë per se (vetëvetiu); është problem çiftimi i lartë me elementet që janë *jostabile* në ndonjë dimension, siç është interfejsi, implementimi, apo thjesht prezenca e tyre.

Kjo është pikë e rëndësishme: Si dizajnues, ne mundemi me shtu fleksibilitet, m'i enkapsulu detalet dhe implementimet dhe në përgjithësi me dizajnu çiftim më të ulët në shumë zona të sistemit. Por, nëse i përqendrojmë përpjekjet tona në "sigurinë e të ardhmes" apo në zvogëlimin e çiftimit kur nuk kemi motivim real, kjo nuk është kohë e kaluar mirë.

Duhet m'i zgjedhë betejat tuaja në zvogëlimin e çiftimit dhe enkapsulimin e gjërave. Fokusohuni në pikat e jostabilitetit real apo të evolucionit të lartë. Për shembull, në projektin NextGen, e dimë se me sistemin duhet m'u lidhë llogaritës të ndryshëm të taksave që janë palë të treta (me interfejsa unikë). Prandaj, dizajni për çiftim të ulët në këtë pikë të variacionit është praktik.

**Përfitimet**

- nuk ndikohet nga ndërrimet në komponentat tjera
- i thjeshtë me kuptu të izoluar
- i përshtatshëm për me ripërdorë

**Background'i (Historia)**

Çiftimi dhe kohezioni janë vërtet principe fundamentale në dizajn, dhe duhet m'u vlerësu dhe m'u apliku si të tilla nga të gjithë zhvilluesit e softuerit. Larry Constantine, po ashtu themelues i dizajnit të strukturuar në 1970tat dhe avokat aktual për më shumë vëmendje në inxhinierimin e përdorshmërisë, ka qenë kryesisht përgjegjës në 1960tat për identifikimin dhe komunikimin e çiftimit dhe të kohezionit si principe kritike.

**Paternat e afërt**

- Variacioni i mbrojtur

## 17.13. Kontrolluesi

### Problemi

Cili objekt përtej shtresës UI (ndërfaqja e përdoruesit) i pari e pranon dhe e koordinon ("kontrollon") një operacion të sistemit?

**Operacionet e sistemit** së pari janë hulumtu gjatë analizës së SSD. Këto janë ngjarjet kryesore input (hyrëse) për sistemin tonë. Për shembull, kur një kasier që e përdor një terminal POS e trus butonin "Kryeje Shitjen", ai e gjeneron një ngjarje të sistemit që domethënë "shitja u krye". Ngjashëm, kur një shkrues duke e përdor një përpunues të fjalëve (word-processor) e trus butonin "drejtshkrimi" (spelling), ai e gjeneron një ngjarje të sistemit që domethënë "kontrolloje drejtshkrimin".

Një **kontrollues** është objekti i parë përtej shtresës UI që është përgjegjës për me pranu apo me trajtu një mesazh të operacionit të sistemit.

### Zgjidhja

Ndaja përgjegjësinë një klase që e prezenton një nga zgjedhjet vijuese:

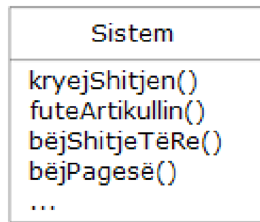
- E prezenton tërë "sistemin", një "objekt rrënjë", një pajisje brenda të cilës ekzekutohet softueri, apo një nënsistem i madh - këto të gjitha janë varianta të një kontrolluesi fasadë.
- Prezenton skenar të një rasti të përdorimit brenda të cilit ndodh ngjarja e sistemit, shpesh i emëruar si TrajtuesI<EmriIRastitTëPërdorimit>, KoordinuesI<EmriIRastitTëPërdorimit>, apo SesionI<EmriIRastitTëPërdorimit> (kontrollues i sesionit të rastit të përdorimit).
    - Përdore klasën e njëjtë kontrolluese për të gjitha ngjarjet e sistemit në skenarin e njëjtë të rastit të përdorimit.
    - Joformalisht, një sesion është instancë e një bisede me një aktor. Sesionet munden me qenë të çfarëdo gjatësie por shpesh organizohen në terma të rasteve të përdorimit (sesionet e rasteve të përdorimit).

*Përfundim*: Vëreni se klasat "dritare", "pamje", dhe "dokument" nuk janë në këtë listë. Klasat e tilla nuk duhet m'i kry punët e asociuara me ngjarje të sistemit; ato zakonisht i pranojnë këto ngjarje dhe ia delegojnë një kontrolluesi.

### Shembull

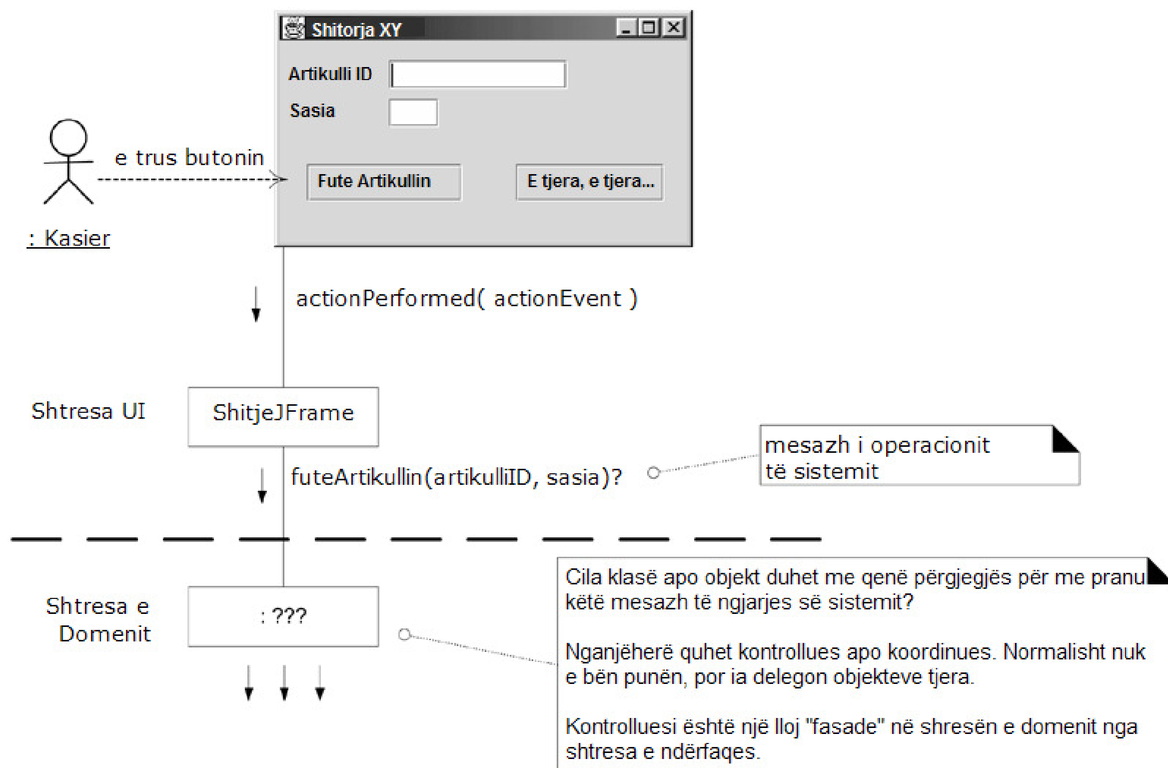
Aplikacioni NextGen përmban disa operacione të sistemit, të ilustruara në Figurën 17.20. Ky model e tregon vetë sistemin si klasë (që është legale dhe nganjëherë e dobishme gjatë modelimit)

**Figura 17.20. Disa operacione të sistemit në aplikacionin NextGenPOS**




Gjatë analizës, operacionet e sistemit munden m'iu nda klasës Sistem në ndonjë model të analizës, për me tregu se ato janë operacione të sistemit. Mirëpo, kjo nuk domethënë se një klasë e softuerit Sistem i kryen ato gjatë dizajnit. Në vend të kësaj, gjatë dizajnit, një klase kontrollues i ndahet përgjegjësia për operacionet e sistemit (shih Figurën 17.21).

**Figura 17.21. Çfarë objekti duhet me qenë Kontrolluesi për futeArtikullin?**




Kush duhet me qenë kontrolluesi për ngjarjet e sistemit si futeArtikullin dhe përfundoShitjen?

Sipas paternit Kontrollues, qe disa zgjedhje:

| | |
|---|---|
| E përfaqëson "sistemin" e përgjithshëm, "objektin rrënjë", apo nënsistemin | Regjistrues, SistemPESH (POSSystem) |
| E përfaqëson një pranues apo trajtues të të gjitha ngjarjeve të sistemit të një skenari të rastit të përdorimit | TrajtuesPërPërpunoShitjen, SesionPërPërpunoShitjen |

Vëreni se në domenin e PESH (Pikë e Shitjes, Point of Sale, POS), një Regjistrues (i quajtur POS Terminal) është pajisje e specializuar me softuerin që ekzekutohet në të.

Në terma të diagramit të interaksionit, një nga shembujt në Figurën 17.22 mundet me qenë i dobishëm.

**Figura 17.22. Alternativat për kontrollues**

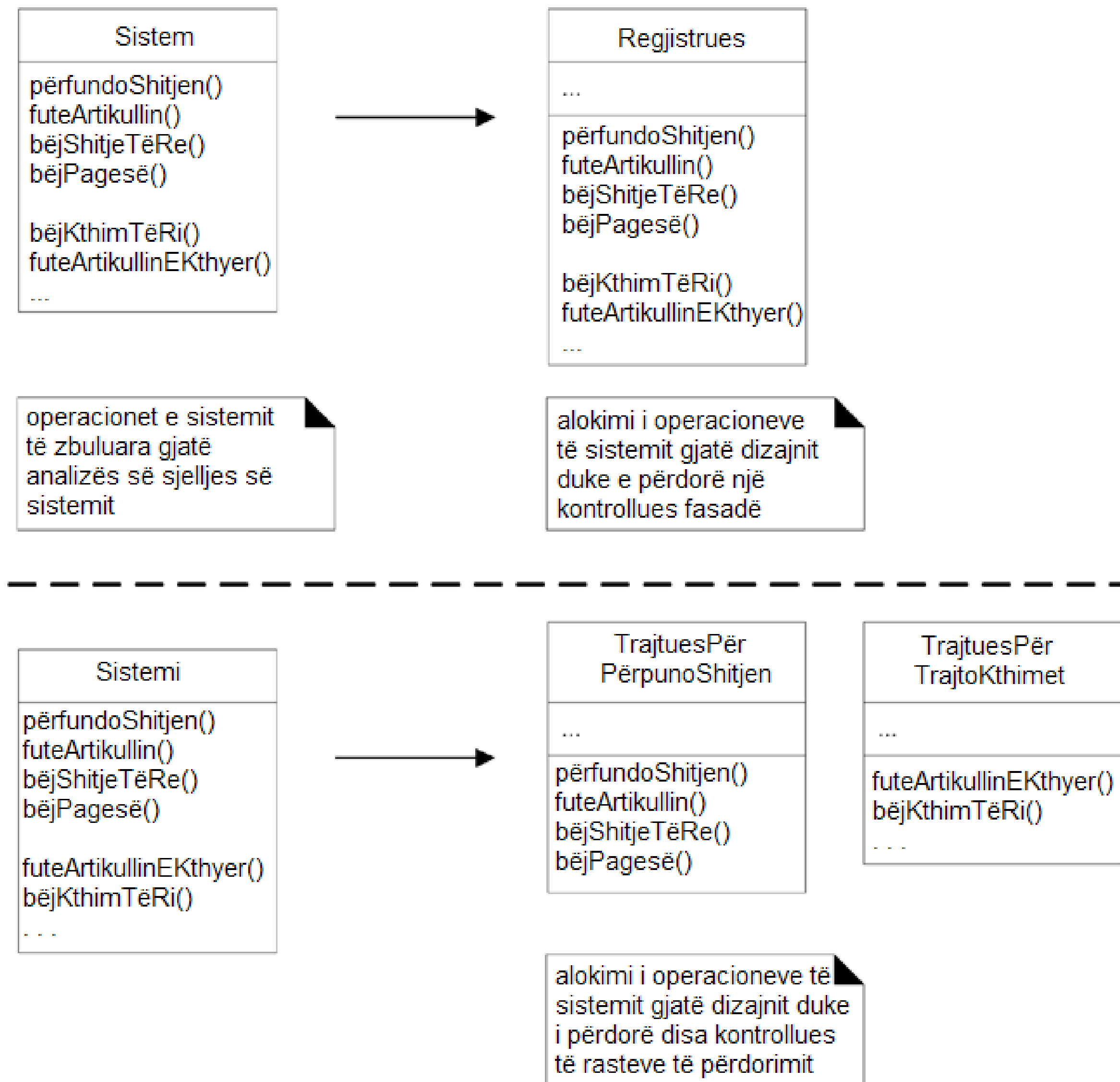
futeArtikullin(id, sasia)
: Regjistrues

futeArtikullin(id, sasia)
: TrajtuesPër PërpunoShitjen

Zgjedhja se cila nga këto klasa është kontrolluesi më i përshtatshëm ndikohet nga faktorë të tjerë të cilët i hulumton seksioni vijues.

Gjatë dizajnit, operacionet e sistemit të identifikuara gjatë analizës së sjelljes së sistemit i ndahen një apo më shumë klasave kontrollues, siç është Regjistruesi, si në Figurën 17.23.

**Figura 17.23. Alokimi i operacioneve të sistemit**




**Diskutim**

Thjesht, ky është patern i delegimit. Në pajtim me të kuptuarit se shtresa UI nuk duhet me përmbajtë logjikë të aplikacionit, objektet e shtresës UI duhet m'ia delegu kërkesat e punës një shtrese tjetër. Kur "shtresa tjetër" është shtresa e domenit, paterni Kontrollues i përmbledh zgjedhjet më të shpeshta që ju, si zhvillues OO, i bëni për delegatin e objektit të domenit që i pranon kërkesat e punës.

Sistemet i pranojnë ngjarjet e jashtme hyrëse (input), zakonisht duke e involvu një GUI që operohet nga një person. Mediumet tjera të inputit (hyrjes) përfshijnë mesazhet e jashtme, siç janë në një switch të telekomunikimeve për përpunim të thirrjeve apo sinjale nga sensorët si në sistemet e kontrollit të proceseve.

Në të gjitha rastet, duhet me zgjedhë një trajtues për këto ngjarje. Kthehuni te paterni Kontrollues për udhëzim drejt zgjedhjeve përgjithësisht të pranueshme dhe të përshtatshme. Siç është ilustru në Figurën 17.21, kontrolluesi është një lloj fasade në shtresën e domenit nga shtresa UI.

Shpesh do të doni ta përdoni klasën e njëjtë kontrollues për të gjitha ngjarjet e sistemit të një rasti të përdorimit ashtu që kontrolluesi mundet m'i mirëmbajtë informatat për gjendjen e rastit të përdorimit. Kjo informatë është e dobishme, për shembull, m'i identifiku ngjarjet e sistemit jashtë-rendit (për shembull, një operacion kryejPagesën para një operacioni përfundoShitjen). Kontrollues të ndryshëm munden m'u përdorë për raste të ndryshme të përdorimit.

Një defekt më i shpeshtë në dizajnin e kontrolluesve rezulton nga tej-ndarja e përgjegjësive. Kontrolluesi pastaj vuan nga kohezioni i keq (i ulët), duke e thyer principin e Kohezionit të Lartë.

### Udhëzim

Normalisht, një kontrollues duhet t'ia delegojë objekteve tjera punën që duhet të bëhet; ai e koordinon ose e kontrollon aktivitetin. Ai nuk bën shumë punë vetë.

Ju lutem shihni seksionin "Çështje dhe Zgjidhje" për elaborim.

Kategoria e parë e kontrolluesve është kontrolluesi fasadë që e përfaqëson sistemin e përgjithshëm, pajisjen, apo një nënsistem. Ideja është me zgjedhë ndonjë emër të klasës që sugjeron mbulesë apo fasadë, mbi shtresat tjera të aplikacionit dhe që e ofron pikën kryesore të thirrjeve të serviseve nga shtresa UI nëpër shtresat tjera të poshtme. Fasada mundet me qenë abstraksion i një njësie të përgjithshme fizike, siç është Regjistrues[12], TelecommSwitch, Telefon, apo Robot; një klasë që e përfaqëson tërë sistemin e softuerit, siç është një SistemPESH; ose cilido koncept tjetër të cilin e zgjedh dizajneri për me përfaqësu sistemin e tërësishëm apo një nënsistem, bile edhe, për shembull, LojëEShahut, nëse është softuer lojë.

Kontrolluesit fasadë janë të përshtatshëm kur nuk ka "shumë" ngjarje të sistemit, apo kur ndërfaqja e përdoruesit (UI) nuk mundet m'i ridrejtu mesazhet e ngjarjeve të sistemit në kontrolluesit alternativë, si p.sh. në një sistem të përpunimit të mesazheve.

Nëse e zgjedhni kontrollues të rastit të përdorimit, atëherë do të keni kontrollues të ndryshëm për secilin rast të përdorimit. Vëreni se ky lloj i kontrolluesit nuk është objekt i domenit; është konstrukt artificial për me përkrahë sistemin (Fabrikim i Pastër në terma të paternave GRASP). Për shembull, nëse aplikacioni NextGen përmban raste të përdorimit siç është Përpuno Shitjen dhe Trajto Kthimet, atëherë mundet me qenë një klasë TrajtuesPërPërpunoShitjen e kështu me radhë.

Kur duheni me zgjedhë kontrollues të rastit të përdorimit? Konsiderojeni si alternativë kur vendosja e përgjegjësive në kontrollues fasadë çon në dizajne me kohezion të ulët apo çiftim të lartë,

---

[12] Terma të ndryshme përdoren për njësi fizike të POS, duke përfshirë regjistër, point-of-sale terminal (POST) - terminal i pikës së shitjes, e kështu me radhë. Gjatë kohës, "regjistri" e ka mishëru nocionin edhe të njësisë fizike edhe të abstraksionit logjik të gjësë që i regjistron shitjet dhe pagesat.

zakonisht kur kontrolluesi fasadë bëhet "i fryrë" me përgjegjësi të tepruara. Një kontrollues i rastit të përdorimit është zgjedhje e mirë kur ka shumë ngjarje të sistemit nëpër procese të ndryshme; ai e faktorizon trajtimin e tyre në klasa të menaxhueshme të ndryshme dhe po ashtu ofron bazë për njohjen dhe rezonimin mbi gjendjen e skenarit aktual në progres.

Në UP (Unified Process, Procesi i Unifikuar) dhe metodën më të vjetër të Jacobson, janë konceptet opsionale të klasave kufi, kontrollues dhe entitet. **Objektet kufitare** janë abstraksione të interfejsave, **objektet entitet** janë objektet e domenit të softuerit që janë të pavarura nga aplikacioni (dhe zakonisht persistente), dhe **objektet e kontrollit** janë trajtues të rasteve të përdorimit siç janë përshkru në këtë patern Kontrollues.

Një konkluzion i rëndësishëm i paternit Kontrollues është se objektet UI (për shembull, objektet dritare apo buton) dhe shtresa UI duhet mos me pasë përgjegjësi për kryerjen e ngjarjeve të sistemit. Me fjalë tjera, operacionet e sistemit duhet m'u trajtu në shtresat e logjikës së aplikacionit apo të domenit e jo në shtresën UI të sistemit. Shihni një shembull te "Çështje dhe Zgjidhje".

### UI-të Web dhe aplikimi i Kontrolluesit server-side (nga ana e serverit)

Një qasje e ngjashme e delegimit mundet m'u përdorë në ASP.NET dhe WebForma: "Kodi prapa" (code behind) fajllit që i përmban trajtuesit e ngjarjeve për klikat e butonave të shfletuesit Web do të ofrojë referencë për një objekt të kontrolluesit të domenit (p.sh. një objekt Regjistrues në rastin e studimit POS), dhe pastaj do ta delegojë kërkesën për punë. Kjo është në kontrast me qasjen e zakonshme, stili delikat i programimit në të cilin zhvilluesit fusin logjikë të aplikacionit duke e trajtu në fajllin "kodi prapa", duke e përzi kështu logjikën e aplikacionit në shtresën UI.

Kornizat (frameworks) server-side të Web UI (kornizat nga ana e serverit për ndërfaqet e përdoruesit Web), siç është Struts, e mishërojnë konceptin e paternit Web-MVC (Model-View-Controller, Model-Pamje-Kontrollues). "Kontrolluesi" në Web-MVC dallon nga ky kontrollues GRASP. I pari është pjesë e shtresës UI dhe e kontrollon interaksionin UI dhe rrjedhën e faqes. Kontrolluesi GRASP është pjesë e shtresës së domenit dhe e kontrollon apo e koordinon trajtimin e kërkesës së punës, esencialisht i pavetëdijshëm se çfarë teknologjie UI po përdoret (p.sh., Web UI, Swing UI, ...).

Po ashtu, te dizajnet nga ana e serverit (server-side) kur përdoren teknologjitë Java, është i shpeshtë delegimi nga shtresa Web UI (p.sh. nga një klasë Struts Action) në një objekt Enterprise JavaBeans (**EJB**) Session. Varianti #2 i paternit kontrollues - një objekt që e përfaqëson një sesion të përdoruesit apo skenar të rastit të përdorimit - e mbulon këtë rast. Në këtë rast, objekti EJB Session mundet me delegu vetë më larg në shtresën e domenit të objekteve, dhe prapë, mundeni me apliku paternin Kontrollues për me zgjedhë një pranues të përshtatshëm në shtresën e pastër të domenit.

Pas krejt kësaj që u tha, trajtimi i duhur i operacioneve të sistemeve nga ana e serverit ndikohet fuqishëm nga kornizat e zgjedhura teknike të serverit dhe vazhdon të jetë cak që lëviz. Por principi themelor i Ndarjes Model-Pamje mundet dhe vazhdon të vlejë.

Edhe me rich-client UI (ndërfaqe e klientit të pasur me kontrolla), p.sh. një UI Swing, që ndërvepron me një server, paterni Kontrollues prapë vlen. UI në anën e klientit e bart kërkesën në kontrolluesin lokal nga ana e klientit, dhe kontrolluesi e bart krejt ose një pjesë të trajtimit të kërkesës në serviset në distancë. Ky dizajn e zvogëlon çiftimin e UI me serviset në distancë dhe e bën më të lehtë, për shembull, me siguru servise ose lokalisht ose në distancë, përmes indireksionit të kontrolluesit nga ana e klientit.

**Përfitimet**

- *Potenciali i rritur për ripërdorim dhe për interfejsa të integrueshëm (pluggable).* Këto përparësi sigurojnë që logjika e aplikacionit nuk trajtohet në shtresën e ndërfaqes. Përgjegjësitë e një kontrolluesi munden m'u trajtu teknikisht në një objekt interfejs, mirëpo një dizajn i tillë implikon që kodi i programit dhe përmbushja e logjikës së aplikacionit do të futej në ndërfaqe apo objekte dritare. Një dizajn i tipit "interfejsi-si-kontrollues" e redukton shansin me ripërdorë logjikën në aplikacionet e ardhshme, meqë logjika që është e lidhur me një interfejs të veçantë (për shembull, objektet dritare) është rrallë e aplikueshme në aplikacionet tjera.
  Në kontrast të kësaj, delegimi i përgjegjësisë së një operacioni të sistemit te një kontrollues e përkrah ripërdorimin e logjikës në aplikacionet e ardhshme. Dhe meqë logjika e aplikacionit nuk është e lidhur me shtresën e ndërfaqes, ajo ndërfaqe mundet m'u ndërru me ndërfaqe tjetër.
- *Shansa me rezonu për gjendjen e rastit të përdorimit.* Nganjëherë ne duhet m'u siguru që operacionet e sistemit ndodhin në sekuencë legale, ose duam të jemi të aftë me rezonu për gjendjen aktuale të aktivitetit dhe të operacioneve brenda rastit të përdorimit që është në proces e sipër. Për shembull, mundet me na u dashtë me garantu se operacioni kryejPagesën nuk mundet me ndodhë para operacionit përfundoShitjen. Nëse është kështu, na duhet me kapë këtë informatë të gjendjes diku; kontrolluesi është një zgjedhje e arsyeshme, sidomos nëse e përdorim kontrolluesin e njëjtë përgjatë tërë rastit të përdorimit (siç rekomandohet).

**Implementimi**

Shembujt vijues i përdorin teknologjitë Java për dy raste më të shpeshta, një klient i pasur (rich client) në Java Swing dhe një Web UI me Struts në server (Servlet engine).

Ju lutem vëreni se duhet me apliku qasje të ngjashme në **WinFormat** .NET dhe **WebFormat** ASP.NET. Një praktikë e mirë në .NET të dizajnuar mirë (shpesh e injoruar nga programerët e Microsoft që e shkelin Principin e Ndarjes Model-Pamje) është *mos* me futë kod të logjikës së aplikacionit në trajtuesit e ngjarjeve apo fajllat "code behind" (kodi prapa) - ato të dyjat janë pjesë e shtresës UI. Në vend të kësaj, në trajtuesit e ngjarjeve .NET apo fajllat "code behind", thjesht pajisuni me një referencë në një objekt të domenit (p.sh. një objekt Regjistrues), dhe delegoni te ai.

### Implementimi me Java Swing: Rich Client UI (Ndërfaqe e Përdoruesit për Klient të Pasur)

Ky seksion supozon se jeni familiarë me Swing-un themelor. Kodi përmban komente për m'i shpjegu poentat kryesore. Disa komente: Vëreni te ❶ se dritarja PerpunoShitjenJFrame e ka një referencë në objektin kontrollues të domenit, Regjistrues. Në ❷ e definoj trajtuesin për klikun e butonit. Në ❸ e tregoj mesazhin kryesor - dërgimi i mesazhit futeArtikullin te kontrolluesi në shtresën e domenit.

```
package com.craiglarman.nextgen.ui.swing;

// importet...

// ne Java, JFrame eshte dritare tipike
public class PerpunoShitjenJFrame extends JFrame
{
    // dritarja ka reference ne objektin 'kontrollues' te domenit

❶     private Regjistrues regjistruesi;

    // dritares i jepet regjistruesi, gjatë krijimit
    public PërpunoShitjenJFrame(Regjistrues _regjistruesi)
    {
        regjistruesi = _regjistruesi;
    }

    //ky buton klikohet per ta kryer
    // operacionin e sistemit "futeArtikullin"
    private JButton BTN_FUTE_ARTIKULLIN;

    // kjo eshte metoda e rendesishme!
    // ketu e tregoj mesazhin nga shtresa UI te shtresa e domenit
    private JButton merreBTN_FUTE_ARTIKULLIN()
    {
        // a ekziston butoni?
        if ( BTN_FUTE_ARTIKULLIN != null )
        {
            return BTN_FUTE_ARTIKULLIN;
        }

        // PERNDRYSHE butoni duhet m’u inicializu...
        BTN_FUTE_ARTIKULLIN = new JButton();
        BTN_FUTE_ARTIKULLIN.setText("Fute artikullin");
```

```
        // KY ESHTE SEKSIONI KRYESOR!
        // ne Java, keshtu e definoni
        // nje trajtues te klikut per nje buton
❷        BTN_FUTE_ARTIKULLIN.addActionListener
        (
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    // Transformues është klasë ndihmëse
                    // për transformimin e Stringjeve
                    // në tipe tjera të shënimeve
                    // sepse pjesët GUI të JTextField kanë Stringje
          IDeArtikullit id = Transformues.neIDteArtikullit(getTXT_ID().getText());
          int sasia = Transformues.neInt(getTXT_SASIA().getText());

                    // ketu e kalojme kufirin nga shtresa UI
                    // ne delegatin e shtreses se domenit te 'kontrolluesi'
                    // > > > KY ESHTE URDHERI KRYESOR < < <

❸                   regjistruesi.futeArtikullin(id, sasia);
                }
            }
        ); // fundi i thirrjes addActionListener
    }
}
```

## Implementimi me Java Struts: Shfletues klient dhe WebUI

Kjo pjesë supozon se jeni familiarë me Struts themelor. Vëreni te ❶ që për m'u pajisë me referencë në objektin e domenit Regjistrues në anën e serverit, objekti Action duhet me hallakatë (me gërmu) në kontekstin e Servletit. Në ❷ e tregoj mesazhin kryesor - dërgimi i mesazhit futeArtikullin te objekti i kontrolluesit të domenit në shtresën e domenit.

```
package com.craiglarman.nextgen.ui.web;

// ... importat

// ne Struts, nje objekt Action asociohet me nje klik te butonit ne web-shfletues,
// dhe thirret (ne server) kur klikohet butoni

public class AksionFuteArtikullin extends Action
{
    // kjo eshte metoda qe thirret ne server
    // kur klikohet butoni ne shfletuesin e klientit
    public ActionForward execute(   ActionMapping mapingu,
                                    ActionForm forma,
                                    HttpServletRequest kerkesa,
                                    HttpServletResponse pergjigjja )
                                throws Exception
    {
        // serveri e ka nje objekt Depo qe
        // i mban referencat ne shume gjera, duke e perfshire
        // objektin "regjistrues" POS
        Depo depoja = (Depo)getServlet().getServletContext().
               getAttribute(Constants.REPOSITORY_KEY);

❶       Regjistrues regjistruesi = depoja.merreRegjistruesin();

        // nxjerri artikulliID dhe sasia nga forma web
        String txtId = ((FormeShitje)forma).merreArtikullinID();
        String txtSasia = ((FormeShitje)forma).merreSasine();

        // Transformuesi eshte klase ndihmese...
        ArtikullID id = Transformues.neArtikullID(txtID);
        int sasia = Transformues.neInt(txtSasia);

        // ketu e kalojme kufirin nga shtresa
             // UI ne shtresen e delegatit te domenit
        // > > > KY ESHTE URDHERI KRYESOR < < <
❷       regjistruesi.futeArtikullin(id, sasia);

        // ...
    }
}
```

## Kontrolluesit e fryrë (të mëdhenj)

### Çështjet dhe Zgjidhjet

E dizajnuar dobët, një klasë kontrollues do të ketë koezion të ulët - e pafokusuar dhe duke trajtu tepër zona të përgjegjësisë; ky quhet **kontrollues i fryrë**. Shenjat e fryrjes janë:

- Është një klasë *e vetme* kontrollues që i pranon të gjitha ngjarjet e sistemit në sistem, dhe ka shumë të tilla. Kjo ndodh nganjëherë nëse është zgjedhë një kontrollues fasadë.
- Kontrolluesi vetë i kryen shumicën e punëve të nevojshme për me trajtu ngjarjen e sistemit, pa e delegu punën. Kjo zakonisht përfshin shkelje të Ekspertit të Informatave dhe të Kohezionit të Lartë.
- Një kontrollues ka shumë atribute, dhe mirëmban informata të konsiderueshme për sistemin apo domenin, që është dashtë m'u shpërnda te objektet tjera, ose e dyfishon informatën që gjendet diku tjetër.

Ndërmjet mjekimeve për kontrollues të fryrë janë këto dyja:

1. Shto më shumë kontrollues - një sistem nuk duhet me pasë nevojë vetëm për një. Në vend të kontrolluesve fasadë, angazho kontrollues të rasteve të përdorimit. Për shembull, konsideroje një aplikacion me shumë ngjarje të sistemit, siç është një sistem i rezervimit të fluturimeve. Ai mundet m'i përmbajtë kontrolluesit vijues:

| **Kontrolluesit e rasteve të përdorimit** |
|---|
| TrajtuesPërRezervo |
| TrajtuesPërMenaxhoOraret |
| TrajtuesPërMenaxhoPagesat |

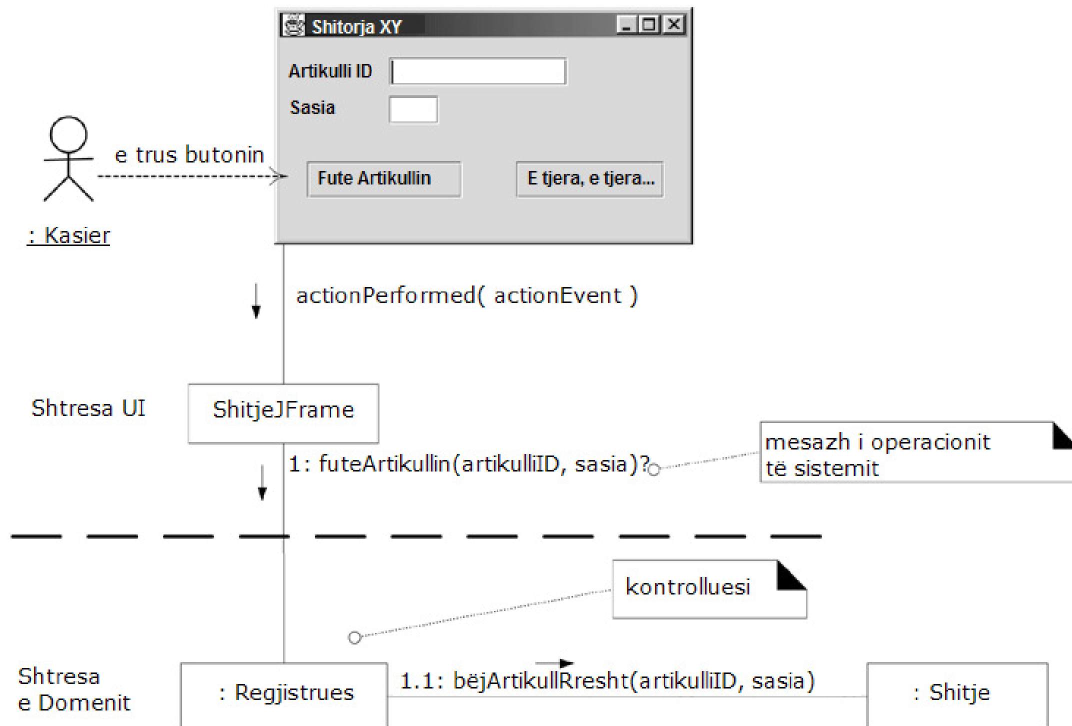
2. Dizajnoje kontrolluesin ashtu që kryesisht ua delegon objekteve tjera kryerjen e përgjegjësisë së secilit operacion të sistemit.

## Shtresa UI nuk i trajton ngjarjet e sistemit

Për me ripërsëritë: Një konkluzion i rëndësishëm i paternit Kontrollues është se objektet UI (për shembull, objektet dritare) dhe shtresa UI nuk duhet me pasë përgjegjësinë për trajtimin e ngjarjeve të sistemit. Si shembull, konsiderojeni një dizajn në Java që e përdor një JFrame për me shfaqë informatën.

Supozojmë se aplikacioni NextGen e ka një dritare që i shfaq informatat e shitjes dhe i regjistron operacionet e kasierit. Duke e përdorë paternin Kontrollues, Figura 17.24 e ilustron një relacion të pranueshëm ndërmjet JFrame dhe kontrolluesit dhe objekteve tjera në një pjesë të sistemit POS (PESH) (me thjeshtime).

**Figura 17.24. Çiftimi i dëshirueshëm i shtresës UI me shtresën e domenit**

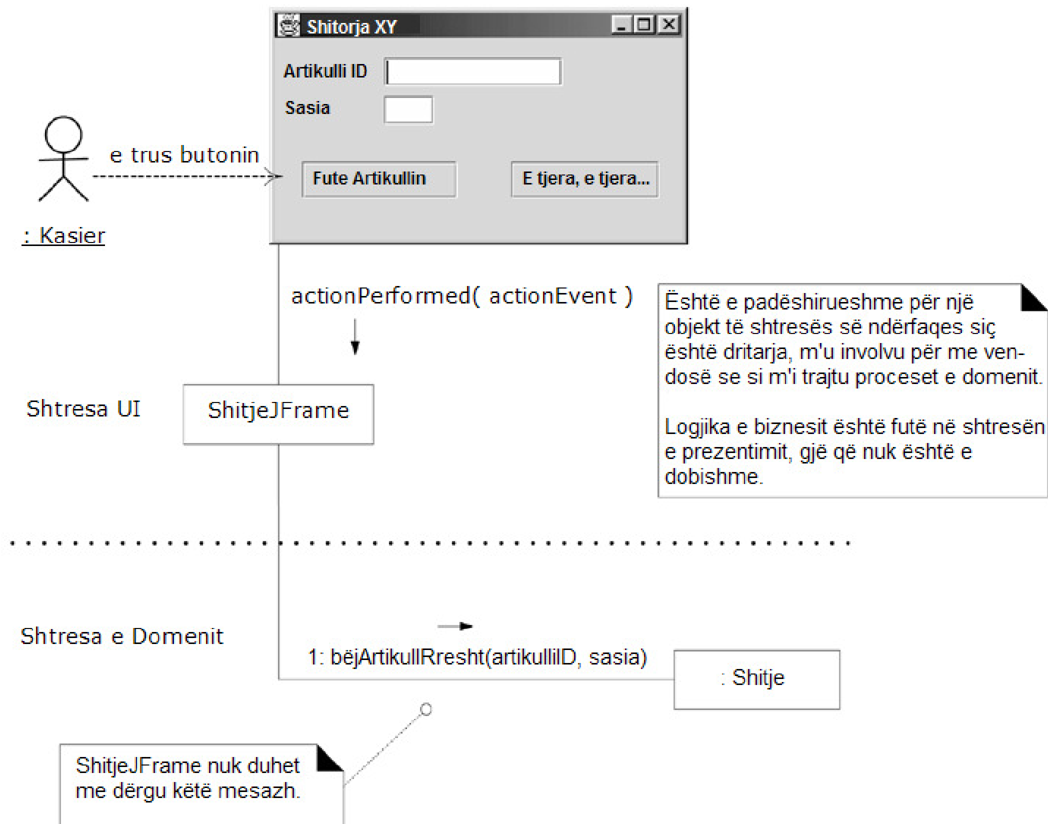



Vëreni se klasa ShitjeJFrame (SaleJFrame) - pjesë e shtresës UI - ia delegon kërkesën futeArtikullin (enterItem) objektit Regjistrues. Ajo nuk është përzier në përpunimin e operacionit apo me vendosë se si me trajtu atë; dritarja vetëm e ka delegu atë në shtresën tjetër.

Ndarja e përgjegjësisë për operacionet e sistemit te objektet në shtresën e aplikacionit apo të domenit duke e përdorë paternin Kontrollues në vend të shtresës UI mundet me rritë potencialin e ripërdorimit. Nëse një objekt i shtresës UI (si ShitjeJFrame) e trajton një operacion të sistemit që paraqet pjesë të një procesi të biznesit, atëherë logjika e procesit të biznesit do të përmbahej në një objekt ndërfaqe (për shembull, dritare); shansi për me ripërdorë logjikën e biznesit atëherë shuhet për shkak të çiftimit të saj me një ndërfaqe dhe një aplikacion specifik.

Si pasojë, dizajni në Figurën 17.25 është i padëshirueshëm.

**Figura 17.25. Çiftim më pak i dëshiruesëm i shtresës së ndërfaqes me shtresën e domenit.**




Vendosja e përgjegjësisë së operacionit të sistemit në një objekt kontrollues të domenit e lehtëson ripërdorimin e logjikës së programit duke e përkrahë procesin e asociuar të biznesit në aplikacionet e ardhshme. Po ashtu e bën më të lehtë me shkyçë shtresën UI dhe me përdorë një kornizë apo teknologji tjetër UI, apo me ekzekutu sistemin në mënyrë offline "batch".

## Sistemet e trajtimit të mesazheve dhe Paterni Komandë

Disa aplikacione janë sisteme apo serverë të trajtimit të mesazheve që pranojnë kërkesa nga proceset tjera. Një switch i telekomunikimeve është shembull më i shpeshtë. Në asi sisteme, dizajni i ndërfaqes dhe i kontrolluesit është disi më ndryshe. Detalet eksklorohen në një kapitull më vonë, mirëpo në esencë, një zgjidhje më e shpeshtë është me përdorë paternin Komandë dhe paternin Përpunues i Komandave.

**Paternat e afërt**

- **Komanda**. Në sistem të trajtimit të mesazheve, secili mesazh mundet m'u përfaqësu dhe m'u trajtu nga një objekt i veçantë Komandë.
- **Fasada**. Një kontrollues fasadë është një lloj i Fasadës.
- **Shtresat**. Ky është patern POSA. Vendosja e logjikës së domenit në shtresën e domenit në vend të shtresës së prezentimit është pjesë e paternit Shtresat.
- **Fabrikimi i Pastër**. Ky patern GRASP është krijim arbitrar i dizajnerit, jo i një klase të softuerit emri i të cilës inspirohet nga Modeli i Domenit. Një kontrollues i rastit të përdorimit është një lloj i Fabrikimit të Pastër.

## 17.14. Kohezioni i Lartë

**Problemi**

Si m'i mbajtë objektet të fokusume, të kuptueshme, dhe të menaxhueshme, dhe si efekt anësor, me përkrahë Çiftimin e Ulët?

Në terma të dizajnit të objekteve, **kohezioni** (apo në mënyrë më specifike, kohezioni funksional) është masë se sa fort janë të lidhura dhe të fokusuara përgjegjësitë e një elementi. Një element me përgjegjësi shumë të lidhura që nuk bën sasi marramendëse të punës ka kohezion të lartë. Këto elemente përfshijnë klasa, nënsisteme, e kështu me radhë.

**Zgjidhja**

Ndaje një përgjegjësi ashtu që kohezioni të mbetet i lartë. Përdore këtë për m'i vlerësu alternativat.

Një klasë me kohezion të ulët bën shumë gjëra të palidhura apo bën tepër punë. Klasat e tilla janë të padëshirueshme; ato vuajnë nga problemet vijuese:

- vështirë m'i kuptu
- vështirë m'i përdorë
- vështirë m'i mirëmbajtë
- delikate; vazhdimisht të ndikuara nga ndryshimi

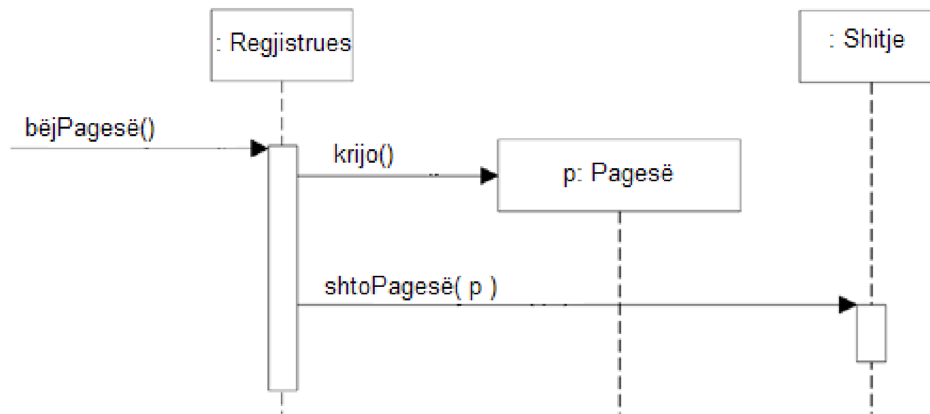
Klasat me kohezion të ulët shpesh paraqesin "kokërr shumë të madhe" të abstraksionit apo kanë marrë përgjegjësi që është dashtë me iu delegu objekteve tjera.

**Shembull**

Ta shohim edhe një herë problemin shembull që u përdor në paternin Çiftimi i Ulët dhe ta analizojmë lidhur me Kohezion të Lartë.

Supozojmë se kemi nevojë me kriju një instancë Pagesë (me para të gatshme) dhe me asociu me Shitjen. Cila klasë duhet me qenë përgjegjëse për këtë? Meqë Regjistruesi e regjistron Pagesën në domenin e botës reale, paterni Krijues e sugjeron Regjistruesin si kandidat për me kriju Pagesën. Instanca Regjistrues pastaj mundet me ia dërgu Shitjes një mesazh shtoPagesë, duke ia bartur Pagesën e re si parametër, siç është tregu në Figurën 17.26.

**Figura 17.26. Regjistruesi e krijon Pagesën**




Kjo ndarje e përgjegjësive e vendos përgjegjësinë për bërjen e pagesës në Regjistrues. Regjistruesi po e merr një pjesë të përgjegjësisë për me përmbushë operacionin e sistemit bëjePagesën.

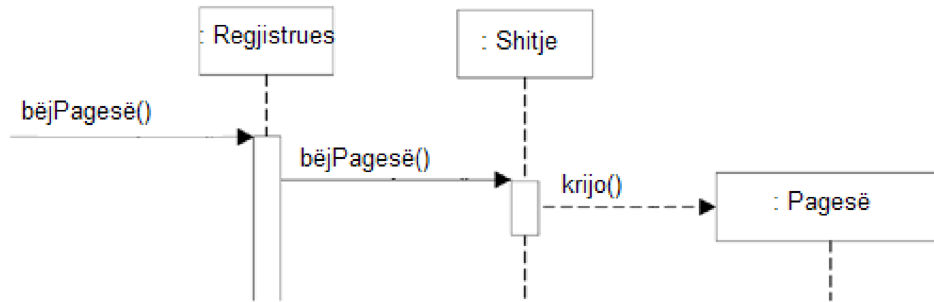
Në këtë shembull të izoluar, kjo është e pranueshme; por nëse vazhdojmë ta bëjmë klasën Regjistrues të përgjegjshme për m'i kry disa apo shumicën e punëve të lidhura me shumë e shumë operacione të sistemit, ajo do të bëhet gjithnjë e më e ngarkuar me punë dhe do të bëhet jokohezive.

Paramendoni pesëdhjetë operacione të sistemit, të gjitha të pranuara nga Regjistruesi. Nëse Regjistruesi e bën punën lidhur me secilën, do të bëhet objekt "i fryrë" jokoheziv. Poenta nuk është se kjo punë e vetme e krijimit të Pagesës vetëm për vetëm e bën Regjistruesin jokoheziv, por si pjesë e një pikture më të madhe të ndarjes së përgjithshme të përgjegjësive, mundet me sugjeru trend drejt kohezionit të ulët.

Dhe më e rëndësishmja në terma të zhvillimit të shkathtësive si dizajnues të objekteve, pavarësisht zgjedhjes finale të dizajnit, është arritja e vlefshme se të paktën dimë me konsideru ndikimin në kohezion.

Si kontrast, siç tregohet në Figurën 17.27, dizajni i dytë ia delegon Shitjes përgjegjësinë e krijimit të pagesës, që e përkrah kohezionin më të lartë në Regjistrues.

**Figura 17.27. Shitja e krijon Pagesën**




Meqë dizajni i dytë e përkrah edhe kohezionin e lartë edhe çiftimin e ulët, ai është i dëshirueshëm.

| Në praktikë, vetëm niveli i kohezionit nuk mund të konsiderohet i izoluar nga përgjegjësitë tjera dhe principet tjera siç është Eksperti dhe Çiftimi i Ulët. |
|---|

**Diskutim**

Si Çiftimi i Ulët, edhe Kohezioni i Lartë është princip për me mbajtë mend gjatë të gjitha vendimeve të dizajnit; është qëllim themelor që duhet me konsideru vazhdimisht. Është princip vlerësues që një dizajner e aplikon gjatë vlerësimit të të gjitha vendimeve të dizajnit.

Grady Booch e përshkruan kohezionin e lartë funksional si ekzistues kur elementet e një komponente (siç është një klasë) "të gjitha punojnë bashkë për me ofru një sjellje të lidhur mirë (të afërt, related)".

Qe disa skenarë që i ilustrojnë shkallët e ndryshme të kohezionit funksional:

1. *Kohezion shumë i ulët*. Një klasë është vetë përgjegjëse për shumë gjëra në zona shumë të ndryshme funksionale.
    - Supozojmë ekzistencën e një klase të quajtur Interfejs-RDB-RPC që është plotësisht përgjegjëse për ndërveprimin me bazat relacionale dhe për trajtimin e thirrjeve të procedurave në distancë (Relational Database-Remote Procedure Call). Këto janë dy zona shumë të ndryshme funksionale, dhe secila kërkon shumë kod përkrahës. Përgjegjësitë duhet m'u nda në një familje të klasave që kanë të bëjnë me qasjen RDB (me baza relacionale) dhe një familje që ka të bëjë me përkrahjen RPC (remote procedure call - thirrje e procedurës në distancë).
2. *Kohezion i ulët*. Një klasë e vetme e ka përgjegjësinë për një detyrë komplekse në një zonë funksionale.
    - Supozojmë ekzistencën e një klase të quajtur InterfejsRDB që është plotësisht përgjegjëse për ndërveprimin me bazat relacionale. Metodat e klasës janë të gjitha të afërta (related), mirëpo ka shumë, dhe ka sasi marramendëse të kodit përkrahës; munden me qenë qindra apo mijëra metoda. Klasa duhet m'u nda në familje të klasave të lehta që e ndajnë punën për me ofru qasje në RDB.

3. *Kohezion i lartë*. Një klasë ka përgjegjësi mesatare në një zonë funksionale dhe bashkëpunon me klasa tjera për m'i përmbushë detyrat.
    - Supozojmë ekzistencën e një klase InterfejsRDB që është vetëm pjesërisht përgjegjëse për ndërveprimin me bazat relacionale. Ajo ndërvepron me nja dhjetë klasa tjera që kanë të bëjnë me qasjen në RDB për m'i thirrë dhe m'i ruajtë objektet.
4. *Kohezion mesatar*. Një klasë ka përgjegjësi të lehta dhe të vetme në disa zona të ndryshme që janë të afërta logjikisht me konceptin e klasës por jo me njëra-tjetrën.
    - Supozojmë ekzistencën e një klase të quajtur Kompani që është plotësisht përgjegjëse për (a) m'i ditë nëpunësit e vet dhe (b) m'i ditë informatat e veta financiare. Këto dy zona nuk janë të afërta fort me njëra-tjetrën, edhe pse të dyja janë të afërta logjikisht me konceptin e një kompanie. Pastaj, numri total i metodave publike është i vogël, si edhe sasia e kodit përkrahës.

Si rregull e përgjithshme, një klasë me kohezion të lartë ka numër relativisht të vogël të metodave, me funksionalitet shumë të afërt, dhe nuk bën shumë punë. Ajo bashkëpunon me objektet tjera për me nda përpjekjen nëse detyra është e madhe.

Një klasë me kohezion të lartë është e dobishme sepse është relativisht lehtë me mirëmbajtë, me kuptu, dhe me ripërdorë. Shkalla e lartë e funksionalitetit të afërt, e kombinuar me numër të vogël të operacioneve, po ashtu e thjeshton mirëmbajtjen dhe përmirësimet. Kokrra e vogël e funksionalitetit shumë të afërt po ashtu e përkrah potencialin e rritur të ripërdorimit.

Paterni Kohezioni i Lartë - si shumë gjëra në teknologjinë e objekteve - e ka analogjinë e botës reale. Vërehet shpesh se nëse një person merr tepër përgjegjësi jo të afërta - sidomos ato që me të drejtë duhet m'iu delegu të tjerëve - atëherë personi nuk është efektiv. Kjo vërehet në disa menaxherë që nuk janë mësu si me delegu. Këta njerëz vuajnë nga kohezioni i ulët; ata janë gati të bëhen "të shkoqur".

## Një Princip Tjetër Klasik: Dizajni Modular

Çiftimi dhe Kohezioni janë principe të vjetra në dizajnin e softuerit; dizajnimi me objekte nuk e përfshin injorimin e fundamenteve me themel të fortë. Një nga to - që është e lidhur fort me çiftimin dhe kohezionin - është me promovu (përkrahë) **dizajnin modular**. Për me citu:

> Modulariteti është vetia e një sistemi që është dekompozu në një grup të moduleve kohezive dhe të çiftuara dobët.

Ne e promovojmë dizajnin modular duke kriju metoda dhe klasa me kohezion të lartë. Në nivelin themelor të objekteve, ne e arrijmë modularitetin duke e dizajnu secilën metodë me qëllim të pastër, të vetëm dhe duke e grupu një grup të afërt të çështjeve në një klasë.

## Kohezioni dhe çiftimi: Jin dhe Jang

Kohezioni i keq zakonisht e shkakton çiftimin e keq dhe e anasjellta. Unë i quaj kohezionin dhe çiftimin *jini dhe jangu i inxhinierimit të softuerit* për shkak të ndikimit të tyre të ndërvarur. Për shembull, konsiderojeni një klasë GUI vegël që e përfaqëson dhe e vizaton një vegël, i ruan shënimet në bazë, dhe i thirr serviset e objekteve në distancë. Kjo jo vetëm që është thellësisht jokohezive, por është e çiftuar me shumë elemente që janë edhe të pangjashme.

### Kundërindikacionet

Në disa raste, pranimi i një kohezioni më të ulët justifikohet.

Një rast është grupimi i përgjegjësive apo kodit në një klasë apo komponentë për me thjeshtu mirëmbajtjen nga një person - edhe pse të jeni të paralajmëruar se një grupim i tillë mundet edhe me përkeqësu mirëmbajtjen. Por supozojmë se një aplikacion përmban urdhëra të futur SQL që, nga principet tjera të mira, duhet m'u shpërnda nëpër dhjetë klasa, si p.sh. dhjetë klasa "mapues të bazës". Tash, zakonisht vetëm një apo dy ekspertë të SQL e dinë se si me definu dhe si me mirëmbajtë më së miri këtë SQL. Edhe nëse dhjetëra programerë të orientuar kah objektet (OO) punojnë në projekt, vetëm disa programerë OO munden me pasë aftësi të forta SQL. Supozojmë se eksperti SQL nuk është as programer komod me OO. Arkitekti i softuerit mundet me vendosë m'i grupu të gjithë urdhërat SQL në një klasë, OperacionetRDB, ashtu që është lehtë për ekspertin SQL me punu në SQL në një vend.

Një rast tjetër për komponentat me kohezion më të ulët është me objektet e shpërndara të serverëve. Për shkak të implikimeve të mbingarkesës dhe të performansës që asociohen me objektet në distancë dhe komunikimin në distancë, është nganjëherë e dëshirueshme me kriju objekte serverike më të pakëta dhe më të mëdha, më pak kohezive që ofrojnë interfejs për shumë operacione. Kjo qasje është e lidhur edhe me paternin e quajtur **Interfejs i Imtësuar Trash për Distancë**. Në atë patern operacionet në distancë bëhen të imtësuara në mënyrë të trashë ashtu që munden me bë apo me kërku më shumë punë në thirrje në distancë të operacioneve për me lehtësu disadvantazhin e performansës për thirrjet në distancë përmes rrjetës. Si shembull i thjeshtë, në vend të një objekti në distancë me tri operacione të imtësuara mirë caktoEmrin, caktoRrogën, dhe caktoDatënEPunësimit, është vetëm një operacion në distancë caktoShënimet, që e pranon një grup të shënimeve. Kjo rezulton në më pak thirrje në distancë dhe performansë më të mirë.

### Përfitimet

- Rritet qartësia dhe lehtësia me kuptu dizajnin
- Thjeshtohet mirëmbajtja dhe përmirësimet
- Shpesh përkrahet çiftimi i ulët
- Rritet ripërdorimi i funksionalitetit të imtësuar mirë dhe shumë të afërt për shkak se një klasë kohezive mundet m'u përdorë për një qëllim shumë specifik.

## 17.15. Resurset e rekomanduara

Metafora e RDD (Responsibility-Driven Design, Dizajn i Udhëhequr nga Përgjegjësitë) është shfaqë sidomos nga punimi influencues për objekte në Smalltalk në Tektronix në Portland, nga Kent Beck, Ward Cunningham, Rebecca Wirfs-Brock, dhe të tjerë. *Designing Object-Oriented Software* (*Dizajnimi i Softuerit të Orientuar kah Objektet*) është teksti i mirënjohur, dhe është relevant sot si në kohën kur është shkruar. Wirfs-Brock e ka publiku më vonë një tjetër tekst RDD, "*Object Design: Roles, Responsibilities, and Collaborations*" (*Dizajni i Objekteve: Rolet, Përgjegjësitë dhe Bashkëpunimet*).

Dy tekste tjera të rekomanduara me theks në principet themelore të dizajnit të objekteve janë *Object-Oriented Heuristics* nga Riel dhe *Object Models* nga Coad.

# Kapitulli 25. GRASP: Më shumë objekte me përgjegjësi

*Fati është ajo çka mbetet jashtë dizajnit.*

*Branch Rickey*

## Qëllimet

- Me mësu m'i apliku paternat e mbetura GRASP.

## Hyrje

Më herët, i aplikuam pesë paterna GRASP:

- Eksperti i Informatave, Krijuesi, Kohezioni i Lartë, Çiftimi i Ulët, dhe Kontrolluesi

Katër paternat e fundit GRASP mbulohen në këtë kapitull. Ata janë:

- Polimorfizmi
- Indireksioni
- Fabrikimi i Pastër
- Variacionet e Mbrojtura

Posa të jenë shpjeguar këto, do të kemi fjalor të pasur dhe të bashkëpërdorur për m'i diskutu dizajnet. Dhe derisa disa nga paternat e dizajnit "Banda-e-Katërshes" (Gang-of-Four, GoF), siç janë Strategjia dhe Fabrika Abstrakte, prezentohen po ashtu në kapitujt vijues, ai fjalor do të rritet. Një fjali e shkurtë, siç është "E sugjeroj një Strategji të gjeneruar nga një Fabrikë Abstrakte për m'i përkrahë Variacionet e Mbrojtura dhe çiftimin e ulët në lidhje me <X>" komunikon shumë informata për dizajnin, meqë emrat e paternave e transmetojnë në mënyrë të përmbledhur një koncept kompleks të dizajnit.

Kapitujt vijues prezentojnë paterna tjerë të dobishëm dhe i aplikojnë ata në zhvillimin e iteracionit të dytë të studimeve të rasteve.

## 25.1. Polimorfizmi (Shumëformësia)

**Problemi**

Si m'i trajtu alternativat duke u bazu në tip? Si me kriju komponenta të integrueshme (pluggable) të softuerit?

*Alternativat e bazuara në tip.* Variacioni i kushtëzuar është temë fundamentale në programe. Nëse një program është i dizajnuar me përdorë logjikë if-then-else apo me urdhër case, atëherë, nëse shfaqet një variacion, kërkon ndryshim të logjikës case - shpesh në shumë vende. Kjo qasje e bën të vështirë me zgjeru lehtësisht një program me variacione të reja sepse ka tendencë që ndryshimet kërkohen në shumë vende - kudo ku ekziston logjika e kushtëzuar.

*Komponentat e integrueshme softuerike.* Duke i shiku komponentat në relacione klient-shërbyes, si mundeni me ndërru një komponentë me një tjetër, pa e afektu klientin?

**Zgjidhja**

Kur alternativat apo sjelljet e ndërlidhura ndryshojnë sipas tipit (klasës), ndaja përgjegjësinë për sjellje - duke i përdorë operacionet polimorfike (shumëformëshe) - tipeve për të cilat ndryshon sjellja[13].

*Konkluzion*: Mos e testo tipin e një objekti dhe mos e përdor logjikën kushtëzuese për me performu alternativa të ndryshueshme të bazuara në tip.

**Shembuj**

### Problemi NextGen: Si me përkrahë llogaritës të taksave që janë palë të treta?

Në aplikacionin NextGen POS, ka shumë llogaritës të taksave që janë palë të treta dhe që duhet m'u përkrahë; sistemi duhet me qenë i aftë m'u integru me llogaritës të ndryshëm. Secili llogaritës i taksave ka interfejs tjetër, prandaj ka sjellje të ngjashme por të ndryshueshme për me iu adaptu secilit nga këta interfejsa apo API fikse të jashtme. Një produkt mundet me përkrahë protokol të thjeshtë TCP socket, një tjetër mundet me ofru interfejs SOAP, dhe një i tretë mundet me ofru interfejs Java RMI.

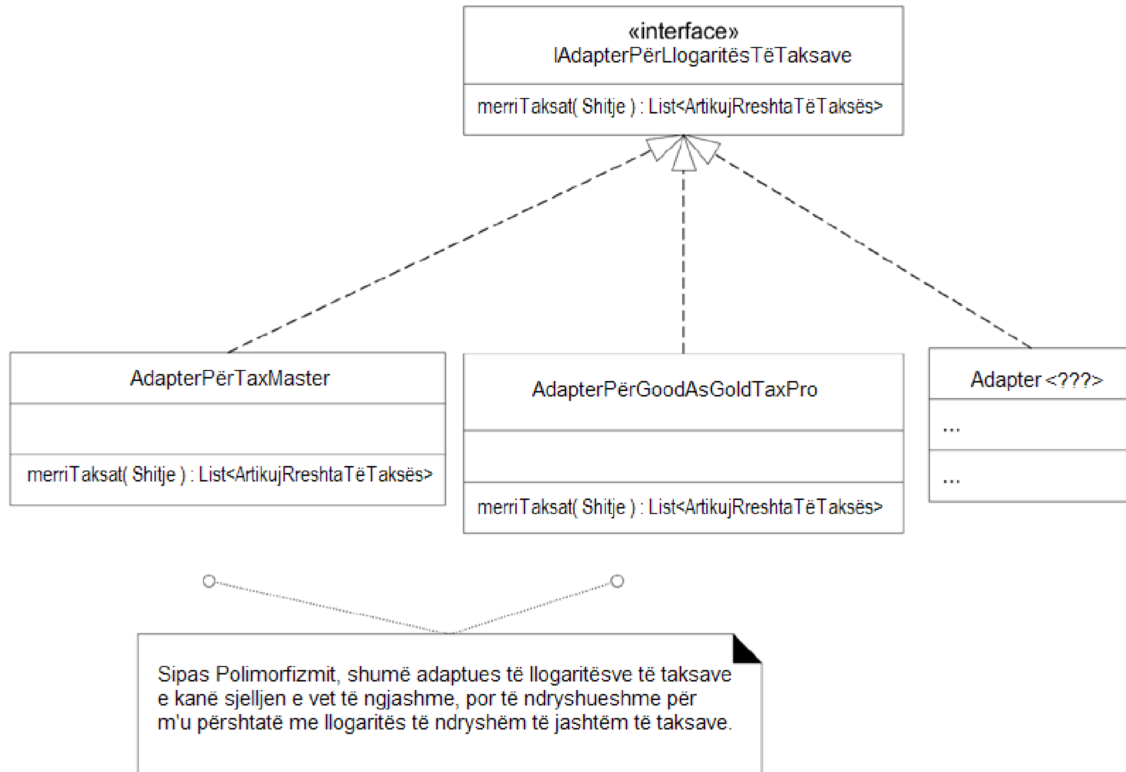
Cilat objekte duhet me qenë përgjegjëse për m'i trajtu këta interfejsa të ndryshueshëm të jashtëm të llogaritësve të taksave?

---

[13] Polimorfizmi ka disa kuptime të ndërlidhura. Në këtë kontekst domethënë "me ia dhënë emrin e njëjtë shërbimeve në objekte të ndryshme" kur shërbimet janë të ngjashme apo të ndërlidhura. Tipet e ndryshme të objekteve zakonisht e implementojnë një interfejs të përbashkët apo janë të ndërlidhura në një hierarki të implementimit me një mbiklasë të përbashkët, por kjo është e pavarur nga gjuha; për shembull, gjuhët me lidhje dinamike (dynamic binding) si Smalltalk nuk e kërkojnë këtë.

Meqë sjellja e adaptimit të llogaritësit ndryshon sipas tipit të llogaritësit, sipas Polimorfizmit, duhet me ia nda përgjegjësinë për adaptim vetë objekte të ndryshme llogaritës (apo adapterëve të llogaritësve), të implementuara me një operacion polimorfik merriTaksat (shih Figurën 25.1)

**Figura 25.1. Polimorfizmi në adaptimin me llogaritës të ndryshëm të jashtëm të taksave.**




Këto objekte adapterë të llogaritësve nuk janë llogaritësit e jashtëm, po janë objekte lokale softuerike që i përfaqësojnë llogaritësit e jashtëm, apo adapterët për llogaritës. Duke ia dërgu një mesazh objektit lokal, në fund një thirrje do të bëhet në llogaritësin e jashtëm në APInë e tij specifike.

Secila metodë *merriTaksat* e merr objektin *Shitje* si parametër, ashtu që llogaritësi mundet me analizu shitjen. Implementimi i secilës metodë *merriTaksat* do të jetë i ndryshëm: objekti *AdapterPerTaxMaster* do ta adaptojë kërkesën për APInë e Tax-Master, e kështu me radhë.

***UML***. Vëreje interfejsin dhe notacionin e realizimit të interfejsit në Figurën 25.1.

### Problemi Monopoly: Si me dizajnu për aksione të ndryshme të kutive?

Për me rikujtu, kur një lojtar ndalet te kutia Shko (Go), ai/ajo i pranon 200 Euro. Për ndaljen në kutinë "Taksa në të Ardhura" është një aksion tjetër, e kështu me radhë. Vëreje se ka rregulla të ndryshme për tipe të ndryshme të kutive (fushave të lojës). Ta rishikojmë principin e dizajnit Polimorfizmi:

> Kur alternativat apo sjelljet e ndërlidhura ndryshojnë sipas tipit (klasës), ndajau përgjegjësinë për sjellje - duke i përdorë operacionet polimorfike - tipeve për të cilat ndryshon sjellja. *Konkluzion (përfundim):* Mos testo për tipin e një objekti dhe mos përdor logjikë kushtëzuese për me performu alternativa të ndryshueshme duke u bazuar në tip.

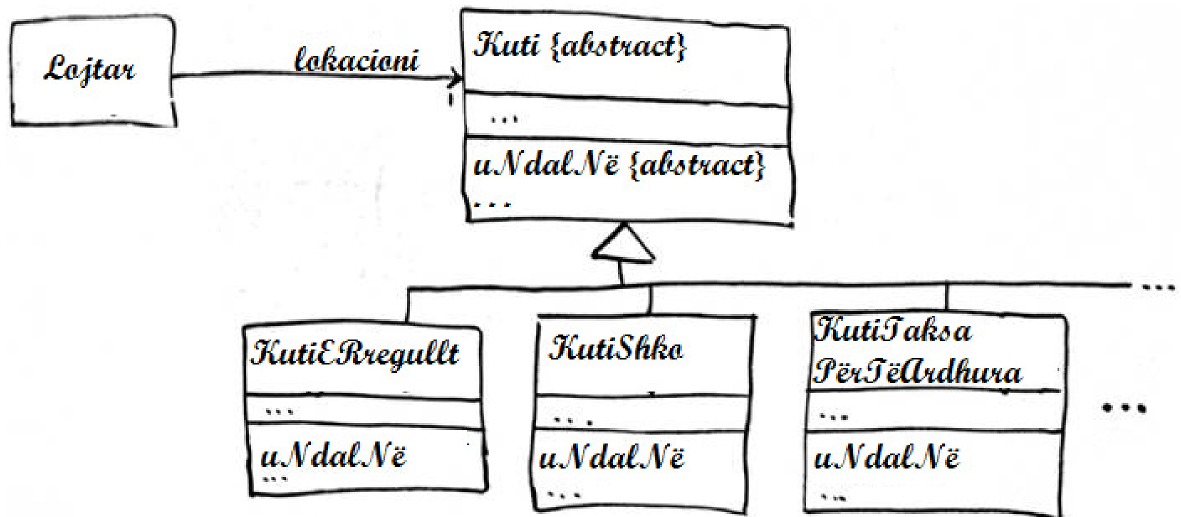
Nga konkluzioni, e dimë se nuk duhemi me dizajnu me logjikë case (urdhër switch në Java apo C#) si në pseudokodin vijues:

```
    // dizajn i keq
SWITCH ON kutia.tipi

CASE KutiShko: lojtari i pranon 200 Euro
CASE KutiTaksëNëTëArdhura: lojtari e paguan taksën
...
```

Në vend të kësaj, principi na këshillon me kriju një operacion polimorfik për secilin tip për të cilin ndryshon sjellja. Ajo ndryshon për tipet (klasat) KutiERregullt, KutiShko, e kështu me radhë. Cili është operacioni që ndryshon? Është ajo që ndodh kur lojtari ndalet në katror. Prandaj, një emër i mirë për operacionin polimorfik është *uNdalNë* apo ndonjë variacion i kësaj. Prandaj, sipas Polimorfizmit, do ta krijojmë nga një klasë të veçantë për secilin lloj të kutisë që ka përgjegjësi të ndryshme *uNdalNë*, dhe në secilën do ta implementojmë nga një metodë *uNdalNë*. Figura 25.2. e ilustron dizajnin e klasave me pamje statike.

**Figura 25.2. Aplikimi i Polimorfizmit në problemin Monopoly.**




***Aplikim i UML:*** Vëreje në Figurën 25.2. përdorimin e fjalës kyçe *{abstract}* për operacionin *uNdalNë*.

***Udhëzim:*** Nëse nuk ka sjellje të paracaktuar (default) në mbiklasë, atëherë deklaroje operacionin polimorfik në mbiklasë me qenë *{abstract}*.

Problemi i mbetur interesant është dizajni dinamik: Si duhet me evolu diagramet e interaksionit? Cili objekt duhet me dërgu mesazhin *uNdalNë* në kutinë ku është ndalë lojtari? Meqë një objekt softuerik Lojtar tashmë e din kutinë e vet të lokacionit (atë ku është ndalë), atëherë sipas principeve të Çiftimit të Ulët dhe sipas Ekspertit, klasa Lojtar është zgjedhje e mirë për me dërgu mesazhin, meqë Lojtari tashmë ka dukshmëri në kutinë korrekte.

Natyrshëm, ky mesazh duhet m'u dërgu në fund të metodës *merreRadhën*. Figura 25.3. dhe Figura 25.4. e ilustrojnë dizajnin evolues dinamik.

Figura 25.3. Aplikimi i Polimorfizmit.

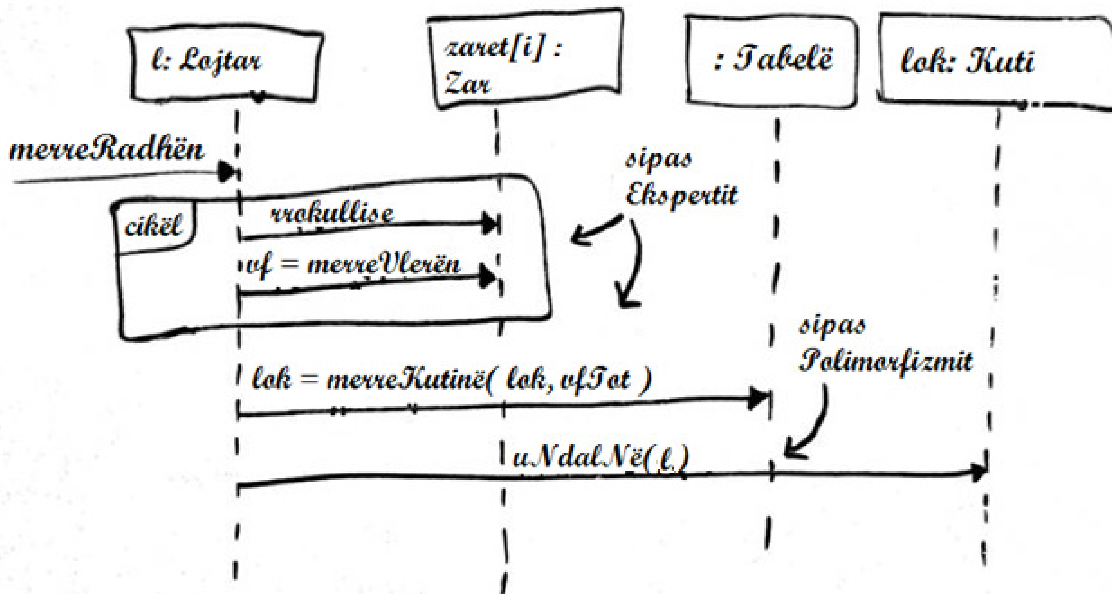
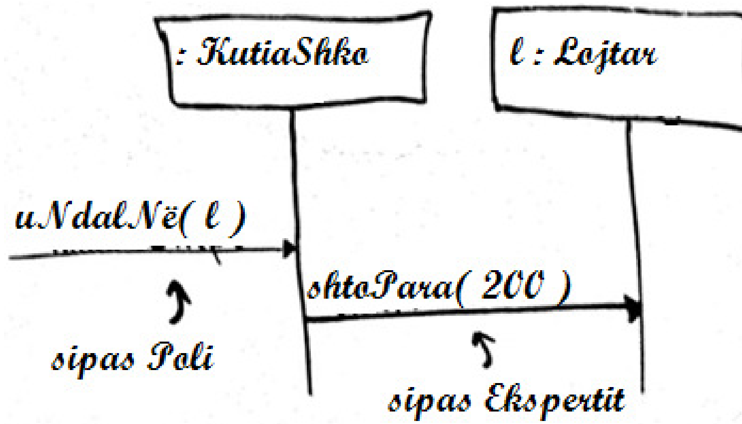



Figura 25.4. Rasti KutiShko.



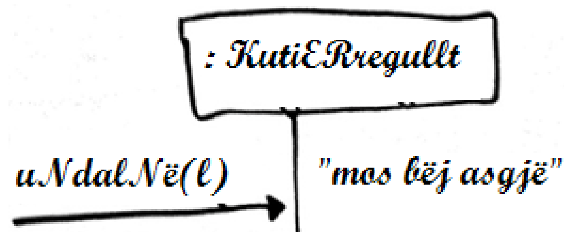
**Aplikim i UML:**

- Vëreni në Figurën 25.3. dhe Figurën 25.4. qasjen joformale për m'i tregu rastet polimorfike në diagrame të veçanta gjatë skicimit të UML-së. Një alternativë - sidomos gjatë përdorimit të veglave UML - është me përdorë korniza *sd* dhe *ref*.
- Vëreni në Figurën 25.3. se objekti *Lojtar* është shënu me 'l' ashtu që në mesazhin *uNdalNë* mundemi me iu referu atij objekti në listën e parametrave. (Do ta shihni në figurën 25.4. se është e dobishme për Kutinë me pasë dukshmëri parametrike të Lojtarit.)
- Vëreni në Figurën 25.3 se objekti *Kuti* është shënu me *lok* (shkurt për 'lokacioni') dhe ky është emri i njëjtë si i ndryshores që është vlerë kthyese në mesazhin merreKutinë. Kjo nënkupton se ato janë objekti i njëjtë.

Ta konsiderojmë secilin nga rastet polimorfike në terma të GRASP dhe çështjet e dizajnit:

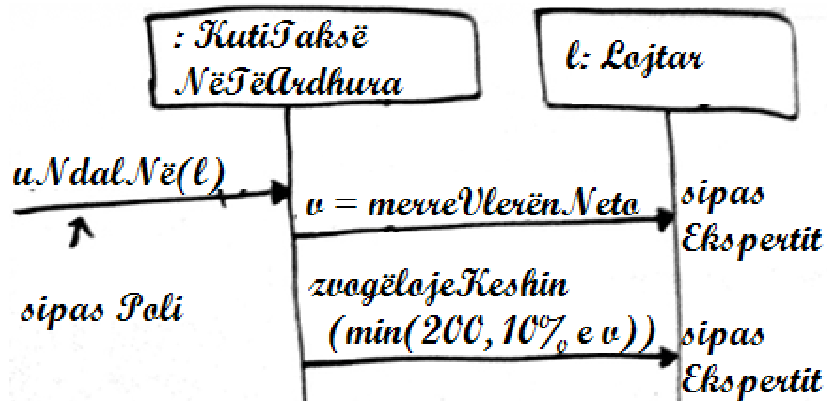
- *KutiShko*. Shih Figurën 25.4. Sipas zbraztësisë së ulët të reprezentimit (LRG), Lojtari duhet me ditë sasinë e parave të veta. Prandaj, sipas Ekspertit, atij duhet me iu dërgu një mesazh shtoPara. Prandaj kutia ka nevojë për dukshmëri te Lojtari, për me mujtë me ia dërgu mesazhin; si pasojë, Lojtari dërgohet si parametër në mesazhin *uNdalNë* për me arritë dukshmëri parametrike.
- *KutiERregullt.* Shih Figurën 25.5. Në këtë rast, nuk ndodh asgjë. Joformalisht e kam shkru në diagram për me tregu këtë, edhe pse kish mujtë m'u përdorë edhe një kuti e shënimit UML (UML note box). Në kod, trupi i kësaj metode do të jetë i zbrazët - nganjëherë quhet metodë NO-OP (no operation, pa operacion). Vëreni se për me shti me punu magjinë e polimorfizmit, duhemi me përdorë këtë qasje për me iu shmangë logjikës së rastit special.

**Figura 25.5. Rasti KutiERregullt.**



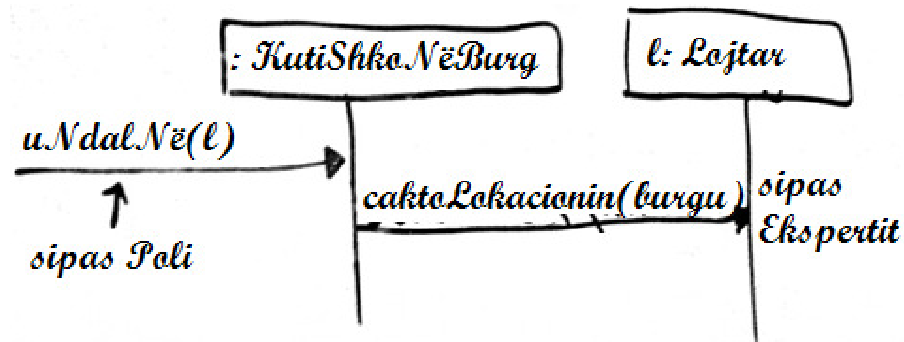
- *KutiTaksëNëTëArdhura.* Shih Figurën 25.6. Duhemi me llogaritë 10% të netos së lojtarit. Sipas zbraztësisë së ulët të reprezentimit dhe sipas Ekspertit, kush duhet me ditë këtë? Lojtari. Prandaj kutia e pyet lojtarin për vlerën, dhe pastaj e nxjerr sasinë e duhur.

**Figura 25.6 Rasti KutiTaksëNëTëArdhura.**




- *KutiShkoNëBurg.* Shih Figurën 25.7. Thjesht, duhet m'u ndërru lokacioni i Lojtarit. Sipas Ekspertit, ai duhet me pranu një mesazh caktoLokacionin. Me gjasë, objekti KutiShkoNëBurg do të inicializohet me një atribut që i referohet objektit KutiBurg, ashtu që ai mund t'ia dërgojë këtë kuti si parametër Lojtarit.

**Figura 25.7. Rasti KutiShkoNëBurg.**

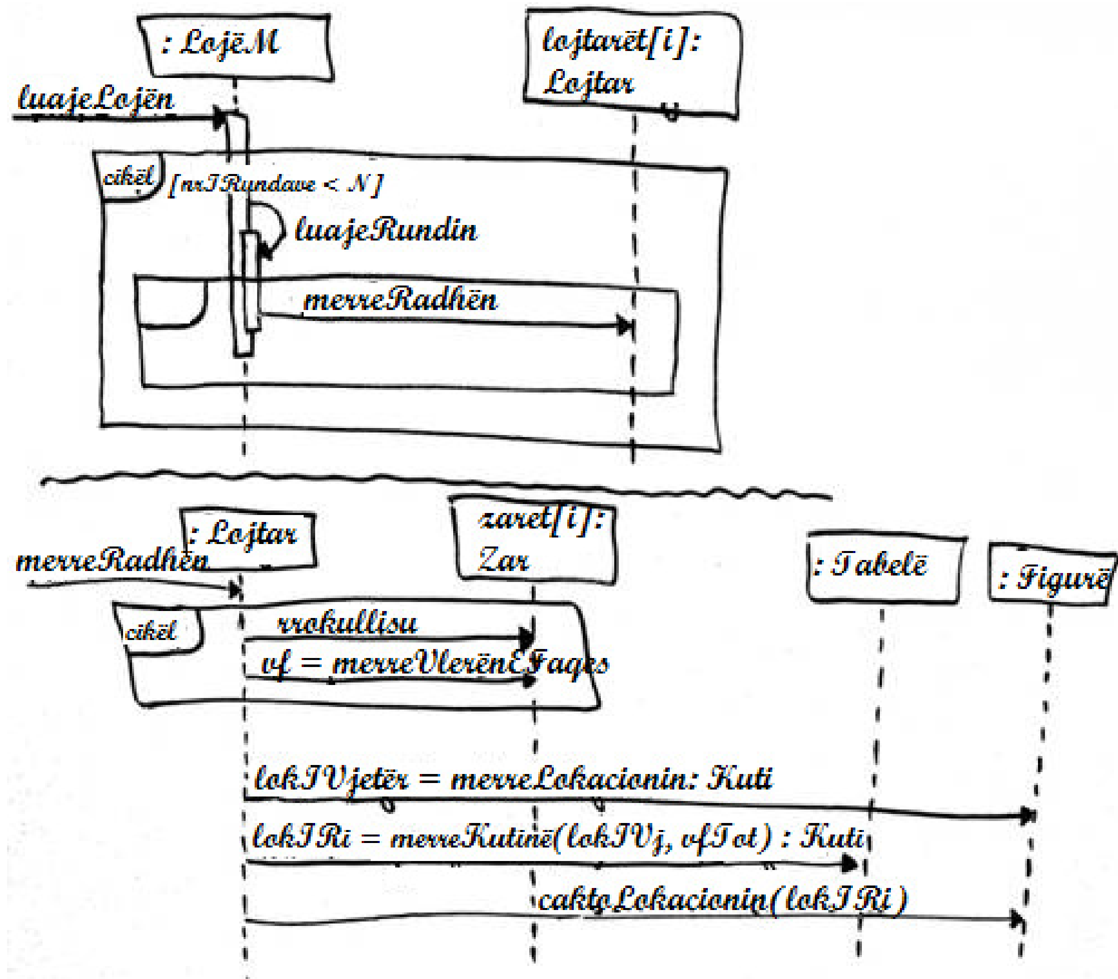



***UML si Skicë:*** Vëreni në Figurën 25.4 se vija vertikale e jetës është e vizatuar si vijë e plotë, e jo si vijë tradicionale e ndërprerë. Kjo është më e përshtatshme gjatë skicimit me dorë. Përveç kësaj, UML 2 i lejon të dy formatet - edhe pse në çfarëdo rasti përshtatja me UML-në korrekte nuk është aq e rëndësishme gjatë skicimit, mjafton që pjesëmarrësit me kuptu njëri-tjetrin.

**Përmirësimi i Çiftimit**

Si përmirësim i vogël i dizajnit OO, vëreni në Figurën 18.25 se *Figura* e mban mend lokacionin e kutisë por *Lojtari* jo, dhe kështu *Lojtari* duhet me nxjerrë informatën nga *Figura* (për me ia dërgu *Tabelës* mesazhin *merreKutinë*), dhe pastaj me ia ri-caktu *Figurës* lokacionin e ri. Kjo është pikë e dobët e dizajnit, dhe në këtë iteracion, kur lojtari duhet edhe me ia dërgu mesazhin *uNdalNë* Kutisë së vet, bëhet edhe më e dobët. Pse? Çka s'po funksionon me të? Përgjigjja: Probleme me çiftim.

**Figura 18.25. Dizajni dinamik për *luajeLojën*.**




Qartas *Lojtari*, e jo *Figura*, ka nevojë me ditë vazhdimisht lokacionin e Kutisë së vet, meqë *Lojtari* po bashkëpunon vazhdimisht me *Kutinë* e vet. Këtë duheni me pa si shans të rifaktorimit për me përmirësu çiftimin - kur objekti A vazhdimisht ka nevojë për shënimet në objektin B atëherë domethënë se ose 1) objekti A duhet m'i mbajtë ato shënime, ose 2) objekti B duhet me pasë përgjegjësinë (sipas Ekspertit) e jo objekti A.

Prandaj, në iteracionin-2 e kam përmirësu dizajnin ashtu që Lojtari e jo Figura e din kutinë e vet; kjo reflektohet edhe në DCD në Figurën 25.2. dhe në diagramin e interaksionit të Figurës 25.3.

Në fakt, dikush mundet me shqyrtu se a është Figura objekt fare i dobishëm në Modelin e Dizajnit. Në botën reale, një copë e vogël plastike që rrin në tabelë është përfaqësues (proxy) i dobishëm për një njeri, meqë ne jemi të mëdhenj dhe shkojmë në kuzhinë me marrë birrë të ftohtë! Por në softuer, objekti Lojtar (duke qenë pikë e imtë softuerike) mundet me përmbushë rolin e Figurës.

**Diskutim**

Polimorfizmi është princip fundamental në dizajnimin se si është i organizuar një sistem për m'i trajtu variacionet e ngjashme. Një dizajn i bazuar në ndarjen e përgjegjësive sipas Polimorfizmit mundet m'u zgjeru lehtësisht për me trajtu variacione të reja. Për shembull, shtimi i një klase të re adapter për llogaritës me metodën e vet polimorfike merriTaksat do të ketë ndikim minor në dizajnin ekzistues.

## Udhëzim: Kur me dizajnu me interfejsa?

Polimorfizmi e përfshin prezencën e mbiklasave apo interfejsave abstraktë në shumicën e gjuhëve OO. Kur duheni me konsideru me përdorë një interfejs? Përgjigjja e përgjithshme është me futë një interfejs kur doni me përkrahë polimorfizmin pa qenë të përkushtuar ndaj një hierarike të caktuar të klasave. Nëse një mbiklasë abstrakte KA përdoret pa interfejs, cilado zgjidhje e re polimorfike duhet me qenë nënklasë e KA, që është shumë kufizuese në gjuhët me trashëgimi të vetme (single-inheritance) si Java apo C#. Si rregull e përgjithshme, nëse ka hierarki të klasave me një mbiklasë abstrakte C1, konsiderojeni me bë një interfejs I1 që i korrespondon nënshkrimeve (signatures) të metodave publike të C1, dhe pastaj deklarojeni C1 për me implementu interfejsin I1. Pastaj, edhe nëse nuk ka motivim të menjëhershëm për me shmangë nënklasimin nën C1 për një zgjidhje të re polimorfike, prapë ekziston një pikë fleksibile e evolucionit për rastet e panjohura të së ardhmes.

**Kundërindikacionet**

Nganjëherë, zhvilluesit dizajnojnë sisteme me interfejsa dhe me polimorfizëm për "siguri të së ardhmes" që është spekulative, ndaj një variacioni të panjohur të mundur. Nëse pika e variacionit është e motivuar definitivisht nga një ndryshueshmëri e menjëhershme apo me gjasë të madhe, atëherë përpjekja për me shtu fleksibilitet përmes polimorfizmit është natyrisht e arsyeshme. Por kërkohet vlerësim kritik, sepse nuk është e pazakonshme me pa përpjekje të panevojshme që aplikohen në sigurinë e së ardhmes për një dizajn me polimorfizëm në pika të variacionit që në fakt janë me gjasë të vogël dhe në fakt kurrë nuk do të shfaqen. Jini realistikë për mundësinë e vërtetë të ndryshueshmërisë para se me investu në fleksibilitet të rritur.

**Përfitimet**

- Shtesat e kërkuara për variacionet e reja shtohen lehtë.
- Munden m'u futë implementime të reja pa i afektu klientët.

**Paternat e ndërlidhur**

- Variacionet e mbrojtura
- Një numër i paternave të njohur të dizajnit GoF, që do të diskutohen në këtë libër, mbështeten në polimorfizëm, duke e përfshi Adapterin (Përshtatësin), Komandën, Kompozitin, Përfaqësuesin (Proxy), Gjendjen (State), dhe Strategjinë.

**E njohur edhe si; E ngjashme me**

Zgjedhja e Mesazhit (Choosing Message), Mos pyet "Çfarë lloji?"

## 25.2. Fabrikimi i Pastër

**Problemi**

Cili objekt duhet me pasë përgjegjësinë, kur nuk doni me shkelë Kohezionin e Lartë dhe Çiftimin e Ulët apo qëllimet tjera, por zgjidhjet e ofruara nga Eksperti (për shembull) nuk janë të përshtatshme?

Dizajnet e orientuara kah objektet nganjëherë karakterizohen me implementim të përfaqësimeve të koncepteve në domenin e problemit të botës reale si klasë softuerike, për me zvogëlu zbraztësinë e reprezentimit (LRG); për shembull një klasë Shitje apo Klient. Mirëpo, ka shumë situata në të cilat ndarja e përgjegjësive vetëm në klasat softuerike të shtresës së domenit çon në probleme në terma të kohezionit apo çiftimit të dobët, apo potencialit të ulët të ripërdorimit.

**Zgjidhja**

Ndaja një grup me kohezion të lartë një klase artificiale apo të përshtatshme që nuk përfaqëson koncept të domenit të problemit - diçka e trilluar, për me përkrahë kohezionin e lartë, çiftimin e ulët, dhe ripërdorimin.

Një kësi klase është fabrikim i imagjinatës. Idealisht, përgjegjësitë e ndara për këtë fabrikim e përkrahin kohezionin e lartë dhe çiftimin e ulët, ashtu që dizajni i fabrikimit është shumë i qartë, apo i pastër - prej nga vjen fjala fabrikim i pastër.

Përfundimisht, një fabrikim i pastër nënkupton me trillu diçka, të cilën e bëjmë kur jemi të dëshpëruar!

**Shembuj**

### Problemi NextGen: Ruajtja e objektit Shitje në Bazë të Shënimeve

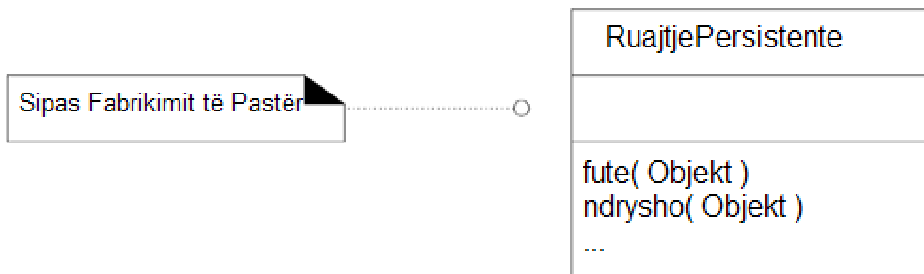
Për shembull, supozojmë se nevojitet përkrahje për m'i rujtë instancat Shitje në një bazë relacionale. Sipas Ekspertit të Informatave, ka njëfarë justifikimi me ia nda këtë përgjegjësi vetë klasës Shitje, sepse shitja i ka shënimet që duhet m'u rujtë. Mirëpo konsideroni implikimet vijuese:

- Detyra kërkon numër relativisht të madh të operacioneve përkrahëse të orientuara kah bazat, prej të cilave asnjëra nuk është e ndërlidhur me konceptin e shit-shmërisë, kështu që klasa Shitje bëhet jokohezive
- Klasa *Shitje* duhet m'u çiftu me interfejsin e bazës relacionale (siç është JDBC në teknologjitë Java), prandaj çiftimi i saj shkon lart. Dhe çiftimi nuk është as me një objekt tjetër të domenit, por me një lloj të veçantë të interfejsit të bazës së shënimeve.
- Ruajtja e objekteve në bazë relacionale është detyrë shumë e përgjithshme për të cilën kanë nevojë për përkrahje shumë klasë. Vendosja e këtyre përgjegjësive në klasën Shitje sugjeron

se do të ketë ripërdorim të dobët të shumë duplikimeve në klasat tjera që e bëjnë të njëjtën gjë.

Prandaj, edhe pse *Shitja* është kandidat logjik sipas virtytit të Ekspertit të Informatave për me rujtë veten në bazë, kjo çon në dizajn me kohezion të ulët, çiftim të lartë, dhe potencial të ulët të ripërdorimit - pikërisht lloji i situatës dëshpëruese që thërret me shpikë diçka.

Një zgjidhje e arsyeshme është me kriju një klasë që është vetë përgjegjëse për m'i rujtë objektet në njëfarë lloji të mediumit të ruajtjes së përhershme (persistente), siç është baza relacionale; quaje RuajtjePersistente[14]. Kjo klasë është Fabrikim i Pastër - trill i imagjinatës.




Vëreje emrin: RuajtjePersistente. Ky është koncept i kuptueshëm, e prapë emri apo koncepti "ruajtje persistente (e qëndrueshme)" nuk është diçka që dikush kish me gjetë në Modelin e Domenit. Dhe nëse një dizajner kish me pytë një person të biznesit në një shitore, "A punoni me objekte të ruajtjes persistente?" ai nuk kish me kuptu. Ata i kuptojnë konceptet si "shitje" dhe "pagesë". RuajtjaPersistente nuk është koncept i domenit, por diçka e trilluar apo e fabrikuar për lehtësi të zhvilluesit të softuerit.

Ky Fabrikim i Pastër i zgjidh problemet vijuese të Dizajnit:

- *Shitja* mbetet e dizajnuar mirë, me kohezion të lartë dhe çiftim të ulët
- Klasa *RuajtjePersistente* vetë është relativisht kohezive, duke e pasë qëllimin e vetëm me ruajtë apo me futë objekte në një medium të ruajtjes persistente.
- Klasa *RuajtjePersistente* është objekt shumë i përgjithshëm (gjenerik) dhe i ripërdorshëm.

Krijimi i një fabrikimi të pastër në këtë shembull është pikërisht situata në të cilën thirret përdorimi i tyre - eliminimi i një dizajni të keq bazuar në Ekspertin, me kohezion dhe çiftim të dobët, duke e zëvendësu me një dizajn të mirë në të cilin ka potencial më të madh për ripërdorim.

Vëreni se, sikur me të gjithë paternat GRASP, theksi është se ku duhet m'u vendosë përgjegjësitë. Në këtë shembull përgjegjësitë janë zhvendosë nga klasa Shitje (e motivuar nga Eksperti) në Fabrikim të Pastër.

[14] Në kornizë me persistencë reale, është përfundimisht e nevojshme më shumë se një klasë me fabrikim të pastër për me kriju dizajn të arsyeshëm. Ky objekt do të jetë fasadë ndërfaqe (front-end) për një numër të madh të objekteve ndihmëse prapa (back-end).

## Problemi Monopoly: Trajtimi i Zareve

Në kapitullin e rifaktorimit, e përdora shembullin e të sjellurit të rrokullisjes së zareve (rrokullisja dhe mbledhja e totaleve të zareve) për me përdorë Metodën Ekstraktuese në metodën Lojtar.merreRadhën. Në fund të shembullit e përmenda edhe atë se zgjidhja e rifaktoruar vetë nuk ishte ideale, dhe një zgjidhje më e mirë do të prezentohej më vonë.

Në dizajnin aktual, Lojtari i rrokullis zaret dhe e mbledh totalin. Zaret janë objekte shumë të përgjithshme, të përdorshme në shumë lojëra. Duke e vendosë këtë përgjegjësi të rrokullisjes dhe të mbledhjes në një Lojtar të lojës Monopoly, shërbimi i mbledhjes nuk është i përgjithësuar për përdorim në lojëra tjera. Një dobësi tjetër: Nuk është e mundur që thjesht me pytë për totalin aktual të zareve pa i rrokullisë zaret përsëri.

Mirëpo, zgjedhja e cilitdo objekt tjetër të inspiruar nga modeli i domenit i lojës Monopoly çon në probleme të njëjta. Dhe kjo na çon në Fabrikim të Pastër - shpik diçka për m'i ofru në mënyrë të përshtatshme shërbimet e ndërlidhura.

Edhe pse nuk ka gotë për zaret në Monopoly, shumë lojëra e përdorin një gotë të zareve në të cilën dikush i përzien të gjitha zaret dhe i rrokullis ato në tavolinë. Prandaj, e propozoj një Fabrikim të Pastër të quajtur Gotë (vëreni se ende po mundohem me përdorë fjalor të ngjashëm relevant për domenin) për m'i mbajtë të gjitha zaret, m'i rrokullisë ato dhe me ditë totalin e tyre. Dizajni i ri është tregu në Figurën 25.8. dhe Figurën 25.9. Gota e mban një koleksion të shumë objekteve Zar. Kur dikush ia dërgon Gotës një mesazh rrokullis, ajo ia çon një mesazh rrokullisu të gjitha zareve të veta.

**Figura 25.8. DCD për Gotë.**

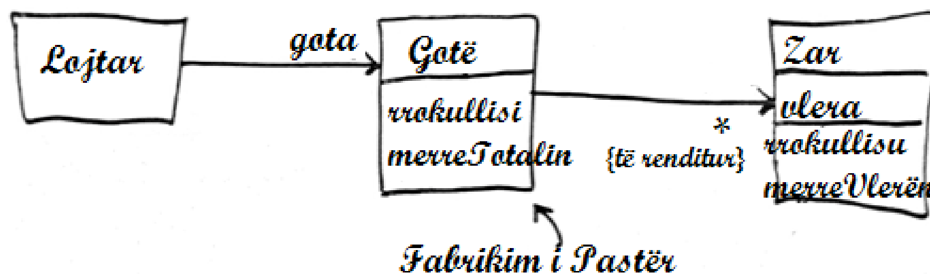
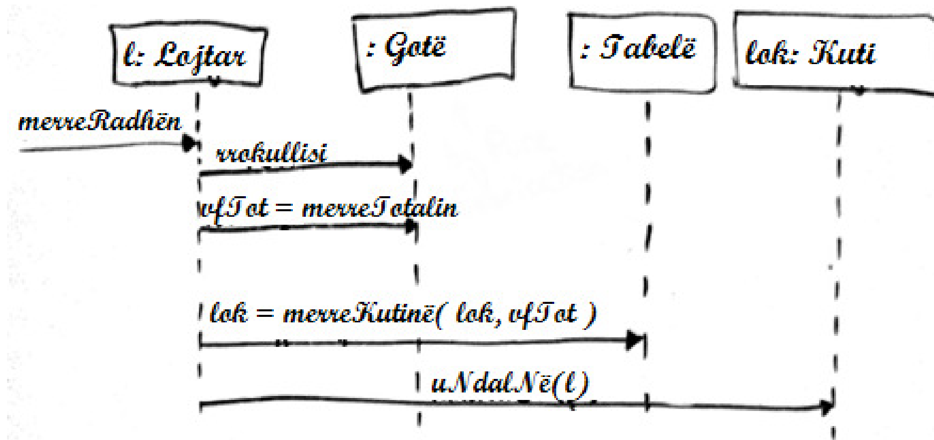


**Figura 25.9. Përdorimi i Gotës në lojën Monopoly.**




**Diskutim**

Dizajni i objekteve mundet m'u nda gjerësisht në dy grupe:

1. Ato që zgjedhen sipas **zbërthimit përfaqësues** (dekompozimit reprezentues)
2. Ato që zgjedhen sipas **zbërthimit të sjelljeve**

Për shembull, krijimi i një klase të softuerit siç është Shitja është bërë sipas zbërthimit përfaqësues; klasa softuerike është e ndërlidhur apo e përfaqëson një gjë në domen. Zbërthimi përfaqësues është strategji e shpeshtë në dizajnin e objekteve dhe e përkrah qëllimin e zbraztësisë së vogël të reprezentimit (përfaqësimit). Mirëpo nganjëherë, dëshirojmë me nda përgjegjësi sipas një grupimi të sjelljeve apo sipas një algoritmi, pa ndonjë shqetësim për krijimin e një klase me emër apo qëllim që është i ndërlidhur me një koncept të domenit të botës reale.

Një shembull i mirë është një objekt "algoritëm" siç është *GjeneruesIPërmbajtjes*, qëllimi i të cilit është (befasi!) me gjeneru përmbajtje të dokumentit dhe është kriju si klasë ndihmëse apo e volitshme nga një zhvillues, pa ndonjë shqetësim për me zgjedhë një emër nga fjalori i domenit të librave dhe dokumenteve. Ai ekziston si klasë e volitshme e konceptuar nga zhvilluesi për m'i grupu disa sjellje apo metoda të ndërlidhura, dhe prandaj është i motivuar nga *zbërthimi i sjelljeve*.

Në kontrast të kësaj, një klasë softuerike me emrin *Përmbajtja* është e inspiruar nga *zbërthimi përfaqësues*, dhe duhet me përmbajtë informata konsistente me konceptin tonë të domenit real (siç janë emrat e kapitujve).

Identifikimi i një klase si Fabrikim i Pastër nuk është kritik. Është koncept edukativ për me komuniku një ide të përgjithshme që disa klasë të softuerit inspirohen nga përfaqësimet në domen, dhe disa thjesht "trillohen" si lehtësi për dizajnuesin e objektit. Këto klasa lehtësuese zakonisht dizajnohen për me grupu bashkë ndonjë sjellje të shpeshtë, dhe prandaj inspirohen nga zbërthimi i sjelljeve e jo nga ai i përfaqësimit.

Me fjalë tjera, Fabrikimi i Pastër zakonisht copëtohet duke u bazuar në funksionalitetin e ndërlidhur, prandaj është një lloj i objektit funksion-qendror apo i sjelljeve.

Shumë paterna ekzistuese të dizajnit të orientuar kah objektet janë shembuj të Fabrikimit të Pastër: Adapteri (Përshtatësi), Strategjia, Komanda, e kështu me radhë.

Si koment final që ia vlen me ripërsëritë: Nganjëherë një zgjidhje e ofruar nga Eksperti i Informatave nuk është e dëshiruar. Edhe pse objekti është kandidat për përgjegjësinë sipas virtytit që i ka shumicën e informatave të ndërlidhura me përgjegjësinë, në anë tjera, zgjedhja e tij çon në dizajn të dobët, zakonisht për shkak të problemeve në kohezion apo çiftim.

**Përfitimet**

- Kohezioni i Lartë përkrahet sepse përgjegjësitë faktorohen në një klasë të grimcuar mirë, që fokusohet vetëm në një grup shumë specifik të detyrave të ndërlidhura.
- Potenciali i ripërdorimit mundet m'u rritë për shkak të prezencës së klasave të grimcuara mirë të Fabrikimit të Pastër, përgjegjësitë e të cilave kanë zbatueshmëri në aplikacione tjera.

**Kundërindikacionet**

Zbërthimi i sjelljeve në objekte të Fabrikimit të Pastër nganjëherë përdoret tepër nga ata që janë të rinj në dizajnin e objekteve dhe më familiarë me zbërthimin apo organizimin e softuerit në terma të funksioneve. Për me ekzagjeru, funksionet thjesht bëhen objekte. Në esencë nuk ka asgjë të keqe në krijimin e objekteve "funksionale" apo "algoritmike", mirëpo kjo duhet të jetë e balansuar me aftësinë me dizajnu me zbërthim të paraqitjes, siç është aftësia me apliku Ekspertin e Informatave ashtu që edhe një klasë përfaqësuese siç është Shitja ka përgjegjësi. Eksperti i Informatave e përkrah qëllimin e bashkë-vendosjes së përgjegjësive me objektet që e dinë informatën e nevojshme për këto përgjegjësi, gjë që ka tendencë me përkrahë çiftimin e ulët. Nëse përdoret tepër, Fabrikimi i Pastër mundet me çu në shumë objekte të sjelljes që kanë përgjegjësi jo të bashkë-vendosura me informatat e kërkuara për plotësimin e tyre, gjë që mundet me ndiku në mënyrë të pafavorshme në çiftim. Simptoma e zakonshme është se shumica e shënimeve brenda objekteve i dërgohen objekteve tjera për me rezonu me to.

**Paternat dhe principet e ndërlidhura**

- Çiftimi i Ulët
- Kohezioni i Lartë
- Një Fabrikim i Pastër zakonisht merr përgjegjësi nga klasa e domenit të cilës do t'iu ndanin ato përgjegjësi duke u bazuar në paternin Ekspert.
- Të gjitha paternat e dizajnit GoF, siç është Adapteri, Komanda, Strategjia, e kështu me radhë, janë Fabrikime të Pastërta.
- Pothuajse të gjitha paternat tjera të dizajnit janë Fabrikime të Pastërta.

## 25.3. Indireksioni

**Problemi**

Ku me nda përgjegjësinë, për me shmangë çiftimin direkt ndërmjet dy (apo më shumë) gjërave? Si m'i shçiftu (decouple, me largu çiftimin, m'i shkëputë) objektet ashtu që m'u përkrahë çiftimi i ulët dhe me mbetë i lartë potenciali i ripërdorimit ?

**Zgjidhja**

Ndaja përgjegjësinë një objekti të ndërmjetëm për me ndërmjetësu ndërmjet komponentave apo shërbimeve tjera ashtu që ato nuk janë të çiftuara direkt.

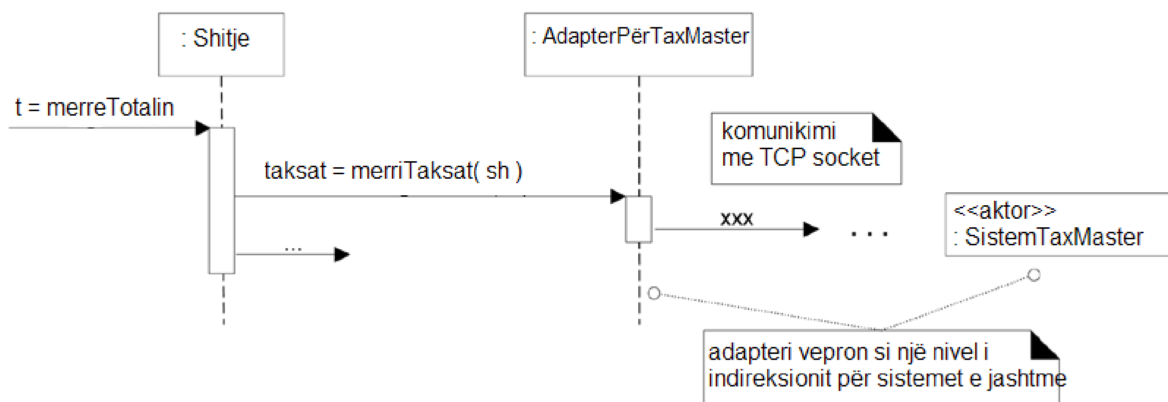
Ndërmjetësuesi e krijon një indireksion (lidhje jo të drejtpërdrejtë) ndërmjet komponentave tjera.

**Shembuj**

### AdapterPërLlogaritësTëTaksave

Këto objekte veprojnë si ndërmjetësues për llogaritësit e jashtëm të taksave. Përmes polimorfizmit, ata ofrojnë interfejs konsistent për objektet e brendshme dhe i fshehin variacionet në APItë e jashtme. Duke e shtu një nivel të indireksionit dhe duke e shtu polimorfizmin, objektet adapter e mbrojnë dizajnin e brendshëm ndaj variacioneve në interfejsat e jashtëm (shih Figurën 25.10).

**Figura 25.10. Indireksioni përmes adapterit.**




***Aplikim i UML***: Vëreni se si është modelu aplikacioni i jashtëm i servisit në distancë TaxMaster në Figurën 25.10: Është shënu me fjalën <<aktor>> për me tregu se është komponentë e jashtme softuerike për sistemin tonë NextGen.

## RuajtjePersistente

Edhe shembulli i Fabrikimit të Pastër i shkëputjes (shçiftimit) të Shitjes nga shërbimet e bazës relacionale përmes futjes së klasës RuajtjePersistente është një shembull i ndarjes së përgjegjësive për me përkrahë Indireksionin. Klasa RuajtjePersistente vepron si ndërmjetësues ndërmjet Shitjes dhe bazës.

**Diskutim**

"Shumica e problemeve në shkencë kompjuterike munden m'u zgjidhë me një nivel shtesë të indireksionit" është një fjalë e vjetër e urtë me lidhje të veçantë me dizajnet e orientuara kah objektet[15].

Sikur që shumë paterna ekzistues të dizajnit janë specializime të Fabrikimit të Pastër, shumë janë edhe specializime të Indireksionit. Adapteri, Fasada, dhe Vëzhguesi (Observer) janë shembuj. Pastaj, shumë Fabrikime të Pastërta gjenerohen për shkak të Indireksionit. Motivimi për Indireksion është zakonisht Çiftimi i Ulët; shtohet një ndërmjetësues për m'i shkëputë (shçiftu, decouple) komponentat apo shërbimet tjera.

**Përfitimet**

- Çiftimi i ulët ndërmjet komponentave.

**Paternat dhe Principet e ndërlidhura**

- Variacionet e Mbrojtura
- Çiftimi i Ulët
- Shumë paterna GoF, siç është Adapteri, Ura, Fasada, Vëzhguesi, dhe Ndërmjetësi (Mediator).
- Shumë ndërmjetësues të Indireksionit janë Fabrikime të Pastërta.

---

[15] Nga David Wheelre. Vëreni se është edhe një kundër-fjalë e urtë: "Shumica e problemeve në performancë munden m'u zgjidhë duke e heqë një shtresë të indireksionit!"

## 25.4. Variacionet e Mbrojtura

**Problemi**

Si m'i dizajnu objektet, nënsistemet, dhe sistemet ashtu që variacionet apo jostabiliteti në këto elemente nuk ka ndikim të padëshirueshëm në elementet tjera?

**Zgjidhja**

Identifikoji pikat e variacionit apo jostabilitetit të parashikuar; ndaji përgjegjësitë për me kriju interfejs stabil përreth tyre.

Shënim: Termi "interfejs" është përdorë në sensin më të gjerë të një pamje të qasjes; nuk e ka kuptimin vetëm të diçkaje si interfejsi Java.

**Shembull**

Për shembull, problemi i mëparshëm i llogaritësit të taksave dhe zgjidhja e tij me Polimorfizëm i ilustrojnë Variacionet e Mbrojtura (Figura 25.1). Pika e jostabilitetit apo e variacionit janë interfejsat apo APItë e ndryshme të llogaritësve të jashtëm të taksave. Sistemi POS ka nevojë me qenë i aftë m'u integru me shumë sisteme ekzistuese të llogaritësve të taksave, dhe po ashtu me llogaritës të ardhshëm si palë të treta që nuk ekzistojnë ende.

Duke e shtu një nivel të indireksionit, një interfejs, dhe duke e përdorë polimorfizmin me implementime të ndryshme IAdapterPërLlogaritësTëTaksave, arrihet mbrojtje brenda sistemit nga variacionet në APItë e jashtme. Objektet e brendshme bashkëpunojnë me një interfejs stabil; implementimet e ndryshme të adapterëve i fshehin variacionet e sistemeve të jashtme.

**Diskutim**

Ky është princip shumë i rëndësishëm, fundamental i dizajnit të softuerit! Pothuajse çdo trik i dizajnit softuerik apo arkitektural në libër - enkapsulimi i shënimeve, polimorfizmi, dizajnet e udhëhequra nga shënimet (data-driven), interfejsat, makinat virtuale, fajllat e konfigurimit, sistemet operative, dhe shumë më tepër - është specializim i Variacioneve të Mbrojtura.

Variacionet e Mbrojtura (PV - Protected Variations) së pari janë publiku si patern i emëruar nga Cockburn në "Patterns Languages of Program Design", edhe pse ky princip shumë fundamental i dizajnit ka qenë këtupari me dekada nën terma të ndryshme, siç është termi fshehja e informatave.

## Mekanizmat e Motivuar nga Variacionet e Mbrojtura

PV është princip rrënjë që i motivon shumicën e mekanizmave dhe paternave në programim dhe dizajn për me ofru fleksibilitet dhe mbrojtje nga variacionet - variacionet në shënime, në sjellje, në harduer, në komponenta të softuerit, në sisteme operative, e tjera.

Në një nivel, pjekuria e një zhvilluesi apo arkitekti mundet m'u pa në njohurinë e tyre të rritur të mekanizmave gjithnjë e më të gjerë për me arritë PV, m'i zgjedhë betejat e duhura PV që ia vlen m'i luftu, dhe aftësia e tyre me zgjedhë një zgjidhje të përshtatshme PV. Më vonë, dikush i mëson teknikat siç janë gjuhët e bazuara në rregulla, interpretuesit e rregullave, dizajnet reflektive dhe metashënime, makinat virtuale, dhe kështu me radhë - ku të gjitha munden m'u apliku për m'u mbrojtë nga ndonjë variacion.

Për shembull:

### *Mekanizmat Thelbësorë të Variacioneve të Mbrojtura*

Enkapsulimi i shënimeve, interfejsat, polimorfizmi, indireksioni, dhe standardet janë të motivuara nga PV. Vëreni se komponentat siç janë makinat virtuale dhe sistemet operative janë shembuj kompleksë të indireksionit për me mbërri PV.

### *Dizajnet e Udhëhequra nga Shënimet (Data-Driven)*

Dizajnet e udhëhequra nga shënimet e mbulojnë një familje të gjerë të teknikave duke i përfshi leximin e kodeve, vlerave, path'ave të fajllave të klasave, emrat e klasave, e kështu me radhë, nga një burim i jashtëm në mënyrë që me ndërru sjelljen, apo "me parametrizu" një sistem në ndonjë mënyrë gjatë kohës së ekzekutimit. Varianta tjera përfshijnë stilet e dizajnit të paragrafeve (style sheets), metashënimet për mapim objektor-relacional, fajllat e vetive, leximi në faqosje me dritare, dhe shumë më tepër. Sistemi mbrohet nga ndikimi i shënimeve, i metashënimeve apo i variacioneve deklarative duke e eksternalizu (qitë jashtë) variantën, duke lexu në të, dhe duke rezonu me të.

### *Leximi i Serviseve (Service Lookup)*

Leximi i Serviseve i përfshin teknikat siç janë përdorimi i serviseve të emërimit (për shembull JNDI-të e Java's) apo shitësit (traders) për me siguru një servis (për shembull, Jini i Java's, apo UDDI për web-shërbimet). Klientët mbrohen nga variacionet në lokacionin e serviseve, duke e përdorë interfejsin stabil të shërbimit lexues (lookup). Është rast special i dizajnit të udhëhequr nga shënimet (data-driven).

### *Dizajnet e udhëhequra nga interpretuesit (interpreter-driven)*

Dizajnet e udhëhequra nga interpretuesit i përfshijnë interpretuesit e rregullave që ekzekutojnë rregulla të lexuara nga një burim i jashtëm, një skriptë apo interpretues të gjuhës që lexojnë dhe ekzekutojnë programe, makina virtuale, motorë (engine) të rrjetave neurale që ekzekutojnë rrjeta, motorë të logjikës së kufizimeve që lexojnë dhe rezonojnë me grupe të kufizimeve, e kështu me radhë. Kjo qasje e lejon ndryshimin apo parametrizimin e sjelljes së një sistemi përmes shprehjeve të logjikës së jashtme. Sistemi mbrohet nga ndikimi i variacioneve logjike duke e eksternalizu logjikën, duke lexu në të, dhe duke e përdorë një interpretues.

### *Dizajnet reflektive apo meta-nivelore*

Një shembull i këtaj qasje është java.beans.Introspector për me marrë një objekt BeanInfo, duke e kërkuar objektin getter (marrës) Method për vetinë bean (kokërr) X, dhe duke e thirrë Method.invoke. Sistemi mbrohet nga ndikimi i logjikës apo variacionet e kodit të jashtëm nga algoritmet reflektive që i përdorin serviset e introspeksionit (analizës së vetëvetes) dhe ato meta-gjuhësore. Kjo mundet m'u konsideru si rast special i dizajneve të udhëhequra nga shënimet.

### *Qasja Uniforme*

Disa gjuhë, siç është Ada, Eiffel, dhe C#, e përkrahin një konstrukt sintaktik ashtu që edhe qasja me metodë edhe ajo me fushë shprehen në mënyrë të njëjtë. Për shembull njëRreth.rrezja mundet me thirrë metodën rrezja():float apo me iu referu drejtpërdrejt një fushe publike, varësisht nga definicioni i klasës. Ne mundemi me kalu nga fushat publike në metodat e qasjes, pa e ndryshu kodin klient.

### *Gjuhët Standarde*

Standardet zyrtare të gjuhëve siç është SQL ofrojnë mbrojtje ndaj shumimit të gjuhëve të ndryshme.

## Principi i Zëvendësimit i Liskovit (The Liskov Substitution Principle (LSP))

**LSP** (Liskov, B. 1988. Data Abstraction and Hierarchy) e formalizon principin e mbrojtjes ndaj variacioneve në implementime të ndryshme të një interfejsi, apo zgjerimeve me nënklasa të një mbiklase.

Për me citu:

> Ajo çka kërkohet këtu është diçka si vetia vijuese e zëvendësimit: Nëse për secilin objekt *o1* të tipit *S* është një objekt *o2* i tipit *T* i tillë që për të gjitha programet *P* të definuara në terma të *T*, sjellja e *P* është e pandryshueshme kur *o1* e zëvendëson *o2* atëherë *S* është nëntip i *T*.

Joformalisht, softueri (metodat, klasat, ...) që i referohet një tipi T (ndonjë interfejs apo mbiklasë abstrakte) duhet me punu ashtu si duhet apo ashtu si pritet me çfarëdo implementimi të zëvendësuar apo nënklasë të T - ta quajmë S. Për shembull:

```
public void shtoTaksat(IAdapterILlogaritësitTëTaksave llogaritësi, Shitje shitja)
{
    Listë artikujtRreshtaTëTaksës = llogaritësi.merriTaksat( shitja );
    // ...
}
```

Për këtë metodë *shtoTaksat*, pavarësisht se çfarë implementimi i *IAdapterILlogaritësitTëTaksave* dërgohet si parametër aktual, metoda duhet me vazhdu me punu "ashtu si pritet". LSP është ide e thjeshtë, intuitive për shumicën e zhvilluesve të objekteve, që e formalizon këtë intuitë.

### *Dizajnet që e fshehin strukturën*

Në edicionin e parë të këtij libri, një princip i rëndësishëm, klasik i dizajnit të objekteve i quajtur Mos Fol me Të Huajt apo Ligji i Demetrit është shprehë si një nga nëntë paternat GRASP. Shkurtimisht, domethënë me iu shmangë krijimit të dizajneve që kalojnë rrugë të gjata të strukturës së objekteve dhe i dërgojnë mesazhe (apo flasin) me objekte të largëta, indirekte (të huaja). Dizajnet e tilla janë delikate ndaj ndryshimeve në strukturat e objekteve - një pikë e shpeshtë e jostabilitetit. Por në edicionin e dytë PV që është më e përgjithshme e ka zëvendësu Mos Fol me të Huajt, sepse e dyta është rast special i të parës. Domethënë, një mekanizëm për me mbërri mbrojtjen nga ndërrimet strukturale është m'i apliku rregullat Mos Fol me Të Huajt.

Mos Fol me Të Huajt vendos kufizime në atë se cilave objekte duheni me iu dërgu mesazhe brenda një metode. Ai thotë se brenda një metode, mesazhet duhet me iu dërgu vetëm objekteve vijuese:

1. Objektit *this* (apo *self*).
2. Një parametri të metodës.
3. Një atributi të *this*.

4. Një elementi të një koleksioni që është atribut i *this*.
5. Një objekti të krijuar brenda metodës.

Qëllimi është m'i ikë çiftimit të klientit me njohurinë për objektet indirekte dhe të lidhjeve objektore ndërmjet objekteve.

Objektet direkte janë "familiarë" të klientit, objektet indirekte janë "të huaj". Një klient duhet me folë me familiarë, dhe m'i ikë të folurit me të huaj.

Qe një shembull që e shkel (butësisht) Mos Fol me Të Huajt. Komentet e shpjegojnë shkeljen.

```
class Regjistrues
{
    private Shitje shitja;

    public void metodëPaksaDelikate()
    {
      // shitja.merrePagesën ia dërgon një mesazh një "familiari"
      // (në rregull sipas #3shit)
      // por në shitja.merrePagesën().merreSasinëEOfruar()
      // mesazhi merreSasinëEOfruar është për Pagesën "e huaj"

      Para sasia = shitja.merrePagesën().merreSasinëEOfruar();

      // ...
    }

    // ...

}
```

Ky kod i përshkon lidhjet strukturale nga një objekt familiar (Shitja) në një objekt të huaj (Pagesa), dhe pastaj ia dërgon një mesazh. Është shumë pak delikate, meqë varet nga fakti se objektet Shitje janë të lidhura me objektet Pagesë. Realisht, kjo vështirë që do të jetë problem.

Por, konsiderojeni këtë fragment të radhës, që përshkon më larg përgjatë rrugës strukturale:

```
public void metodëMëDelikate()
{
    MbajtësILlogarisë mbajtësi =
        shitja.merrePagesën().merreLlogarinë().merreMbajtësinELlogarisë();

    // ...
}
```

Apo më përgjithësisht:

```
public void bëjX()
{
    F ndonjëF =
      filani.merrA().merrB().merrC().merrD.merrE().merrF();

    // ...
}
```

Shembulli është i trilluar, por po e vëreni paternin: Përshkimi më larg përgjatë një rruge të lidhjeve të objekteve për m'i dërgu mesazh një objekti të largët, indirekt - të folurit me një të huaj të largët. Dizajni është i çiftuar me një strukturë të caktuar se si janë të lidhura objektet. Sa më larg përshkon një program përgjatë një rruge, aq më delikat është. Pse? Sepse struktura e objekteve, lidhjet e tyre, munden me ndryshu. Kjo është sidomos e vërtetë në aplikacionet e reja apo iteracionet e hershme. Karl Lieberherr dhe kolegët e tij kanë bërë hulumtim për principet e mira të dizajnit të objekteve, nën ombrellën e projektit Demeter. Ky Ligj i Demetrit (Mos Fol me Të Huajt) është identifiku për shkak të frekuencës në të cilën e kanë parë ndryshimin dhe jostabilitetin në strukturën e objekteve, dhe kështu thyerjen e shpeshtë në kod që ishte e çiftuar me njohurinë e lidhjeve të objekteve.

Sidoqoftë, siç do të shqyrtohet në seksionin vijues "PV spekulative dhe zgjedhja e betejave tuaja", nuk është gjithmonë e nevojshme m'u mbrojtë prej kësaj; varet nga jostabiliteti i strukturës së objekteve. Në libraritë standarde (siç janë libraritë Java) lidhjet strukturale ndërmjet klasave të objekteve janë relativisht stabile. Në sistemet e pjekura, struktura është më stabile. Në sistemet e reja në iteracion të hershëm, nuk është stabile.

Në përgjithësi, sa më larg e përshkon dikush një rrugë, më delikat është, dhe prandaj është më e dobishme me iu përmbajtë principit Mos Fol me Të Huajt.

Zbatimi strikt i këtij ligji - mbrojtjes nga variacionet strukturale - kërkon me iu shtu operacione të reja publike "familiarëve" të një objekti; këto operacione e ofrojnë informatën e dëshiruar, dhe e fshehin se si është siguruar ajo. Për shembull, për me përkrahë Mos Fol me të Huajt për dy rastet e kaluara:

```
// rasti 1
Para sasia = shitja.merreSasinëEOfruarTëPagesës();

// rasti 2
MbajtësILlogarisë mbajtësi = shitja.merreMbajtësinELlogarisëPërPagesën();
```

**Kundërindikacionet**

### *Kujdes: PV Spekulative dhe Zgjedhja e Betejave Tuaja*

Së pari, ia vlen m'i definu dy pika të ndryshimit:

- **pika e variacionit.** Variacionet në sistemin ekzistues, aktual të kërkesave, siç janë interfejsat e shumtë të llogaritësve të taksave që duhet m'u përkrahë.
- **pika e evolucionit.** Pikat spekulative të variacionit që munden m'u shfaqë në të ardhmen por që nuk janë prezente në kërkesat ekzistuese.[16]

### *PV aplikohet në pika edhe të variacionit edhe të evolucionit.*

Një paralajmërim: nganjëherë kostoja e "sigurisë" spekulative "për të ardhmen" në pikat e evolucionit e kalon koston që shkaktohet nga një dizajn i thjeshtë më "i brishtë" që ripunohet sipas nevojës në përgjigje ndaj presioneve të ndryshimeve të vërteta. Domethënë, kostoja e mbrojtjes së inxhinieringut në pikat e evolucionit mundet me qenë më e lartë se me ripunu një dizajn të thjeshtë.

Për shembull, më kujtohet një sistem i trajtimit të mesazheve në pejxher (pager) ku arkitekti e shtoi një gjuhë skriptuese dhe një interpretues për me përkrahë fleksibilitetin dhe variacionin e mbrojtur në një pikë të evolucionit. Mirëpo, gjatë ripunimit në një version rritës (incremental), skriptimi kompleks (dhe joefikas) u pat heqë - thjesht nuk nevojitej. Dhe kur e kam nisë programimin OO (në 1980tat e hershme) kam vuajtë nga sëmundja "përgjithëso-itis" në të cilën kisha tendencë me kalu shumë orë duke kriju mbiklasa të klasave që duhej m'i shkru realisht. Kisha me bo gjithçka shumë të përgjithshme dhe fleksibile (dhe të mbrojtur ndaj variacioneve), për atë situatë të të ardhmes kur do të shpaguhej vërtet - që kurrë nuk vinte. Ishte gjykim i dobët se kur ia ka vlejtë përpjekja.

Poenta nuk është me përkrahë ripunimin dhe dizajnet e brishta. Nëse nevoja për fleksibilitet dhe mbrojtja nga ndryshimi është realistike, atëherë aplikimi i PV është i motivuar. Por nëse është për mbrojtje spekulative të të ardhmes apo për "ripërdorim" spekulativ me gjasa shumë të vogla, atëherë është koha për vetëpërmbajtje dhe të menduar kritik.

Zhvilluesit fillestarë tentojnë kah dizajnet e brishta, zhvilluesit e mesëm tentojnë kah ato tepër ekstravagante dhe fleksibile, të përgjithësuara (në asi mënyra që nuk do të përdoren kurrë). Dizajnuesit ekspertë zgjedhin me mendjehollësi; ndoshta një dizajn i thjeshtë dhe i brishtë me kosto të ndryshimit të balansuar ndaj gjasës së vet.

[16] Në UP, pikat e evolucionit munden m'u dokumentu formalisht në Rastet e Ndryshimeve; secila i përshkruan aspektet e ndryshme të një pike të evolucionit për benefitin e ndonjë arkitekti në të ardhmen.

**Përfitimet**

- Zgjerimet e kërkuara për variacionet e reja është lehtë m'i shtu
- Implementimet e reja munden m'u futë pa ndiku te klientët.
- Çiftimi zvogëlohet
- Ndikimi apo kostoja e ndryshimeve mundet m'u zvogëlu.

**Paternat dhe Principet e Ndërlidhura**

- Shumica e principeve dhe paternave të dizajnit janë mekanizma për variacionet e mbrojtura, duke e përfshi polimorfizmin, interfejsat, indireksionin, enkapsulimin e shënimeve, shumicën e paternave të dizajnit GoF, e kështu me radhë.
- Në librin "Design Patterns for Object-Oriented Software Development", pikat e variacionit dhe të evolucionit quhen "pika të nxehta" ("hot spots").

**E njohur edhe si; e ngjashme me**

PV është në thelb e njëjtë si principet "fshehja e informatës" dhe "hapur-mbyllur", që janë terma më të vjetra. Si patern "zyrtar" në komunitetin e paternave, është emëru si "Variacionet e Mbrojtura" më 1996 nga Cockburn.

## *Fshehja e Informatave*

Punimi i njohur i David Parans *"Mbi Kriteret që duhet m'u përdorë në Dekompozimin e Sistemeve në Module"* (*On the Criteria To Be Used in Decomposing Systems Into Modules*) është shembull i klasikëve që citohen shpesh por lexohen rrallë. Në të, Parnas e prezenton konceptin e **fshehjes së informatave**. Ndoshta meqë termi tingëllon si ideja e enkapsulimit të shënimeve, është *keqinterpretu* si enkapsulim i shënimeve, dhe disa libra gabimisht i definojnë konceptet si sinonime. Në vend të kësaj, Parnas ka mendu që fshehja e informatave me pasë kuptimin *"fshehi informatat për dizajnin nga modulet tjera, në pikat e ndryshimit të vështirë apo të mundshëm"*.

Për me citu diskutimin e tij për fshehjen e informatave si princip udhëzues i dizajnit:

> Ne propozojmë në vend të kësaj, që dikush me fillu me një listë të vendimeve të vështira të dizajnit apo vendimeve të dizajnit që pritet me ndryshu. Secili modul pastaj dizajnohet me fshehë atë vendim nga modulet tjera.

Domethënë, fshehja e informatave e Parnasit është principi i njëjtë i shprehur në PV, dhe jo thjesht enkapsulimi i shënimeve - që është vetëm një nga shumë teknikat për m'i fshehë informatat për dizajnin. Megjithatë, termi është riinterpretu kaq gjerësisht si sinonim për enkapsulimin e shënimeve sa që më është e pamundur me përdorë në sensin e tij origjinal pa u keqkuptu.

### *Principi Hapur-Mbyllur*

Principi Hapur-Mbyllur (Open-Closed Principle, OCP), i përshkruar nga Bertrand Meyer në "Object-Oriented Software Construction", është thelbësisht ekuivalent me paternin PV dhe me fshehjen e informatave. Një definicion i OCP ëstë:

> Modulet duhet me qenë edhe të hapura (për zgjerim; të adaptueshme) edhe të mbyllura (moduli është i mbyllur ndaj ndryshimeve në mënyrat që ndikojnë te klientët).

OCP dhe PV janë në esencë dy shprehje të principit të njëjtë, me thekse të ndryshme: mbrojtja në pikat e variacionit dhe të evolucionit. Në OCP, "moduli" i përfshin të gjitha elementet diskrete të softuerit, duke i përfshi metodat, klasat, nënsistemet, aplikacionet, e kështu me radhë.

Në kontekst të OCP, fraza "e mbyllur në lidhje me X" domethënë se klientët nuk ndikohen nëse X ndryshon. Për shembull, "klasa është e mbyllur në lidhje me definicionet e fushave të instancës" përmes mekanizmit të enkapsulimit të shënimeve me fusha private dhe me metoda publike të qasjes. Në të njëjtën kohë, ato janë të hapura ndaj modifikimit të definicioneve të shënimeve private, sepse klientët e jashtëm nuk janë të çiftuar direkt me shënimet private.

Si shembull tjetër, "adapterët e llogaritësve të taksave janë të mbyllur në lidhje me interfejsin e tyre publik" përmes implementimit të interfejsit stabil *IAdapterILlogaritësitTëTaksave*. Megjithatë, adapterët janë të hapur ndaj zgjerimit duke u ndryshu privatisht në përgjigjje të ndryshimeve në APItë e llogaritësve të jashtëm të taksave, në mënyra që nuk i dëmtojnë klientët e tyre.

# Shtojcë: artifakte dhe shembuj

Në vijim është përshkrimi i dy studimeve të rasteve që janë përdorë si shembuj gjatë tërë librit, si dhe disa artifakte më të rëndësishme nëpër kapitujt tjerë.

## Studimet e rasteve (case studies)

(kapitulli 3)

**Rasti Një: Sistemi NextGEN POS**

Studimi i rastit të parë është sistemi pikë-e-shitjes NextGen (point-of-sale, POS). Në këtë domen të problemit që në dukje të parë është i drejtpërdrejtë, do të shohim se ka probleme interesante të kërkesave dhe të dizajnit për m'i zgjidhë. Pastaj, është problem real – grupet vërtet zhvillojnë sisteme POS me teknologji të objekteve.

Një sistem POS është aplikacion i kompjuterizuar që përdoret (pjesërisht) për m'i regjistru shitjet dhe m'i trajtu pagesat; zakonisht përdoret në shitore me pakicë. Ai i përfshin komponentat e harduerit siç janë një kompjuter dhe një skener i bar-kodeve, dhe softuerin për me ekzekutu sistemin. Ai ndërvepron me interfejsa me aplikacione të ndryshme të serviseve, siç është llogaritësi i jashtëm i taksave dhe kontrolli i inventarit. Këto sisteme duhet të jenë relativisht tolerante ndaj dështimit; domethënë, edhe nëse serviset në distancë janë përkohësisht të paqasshme (siç është sistemi i inventarit), ato prapë duhet me qenë të afta m'i rujtë shitjet dhe m'i trajtu të paktën pagesat kesh (ashtu që biznesi të mos dëmtohet).

Një sistem POS në mënyrë graduale duhet t'i përkrahë shumë terminalë dhe interfejsa të ndryshëm. Këto përfshijnë një terminal me shfletues Web, një kompjuter i rregullt personal me diçka si ndërfaqja grafike e përdoruesit Java Swing, hyrje me ekran prekës, pajisje pa tela, e kështu me radhë.

Gjithashtu, jemi duke e kriju një sistem komercial POS që do t'ua shesim klientëve të ndryshëm me nevoja të pangjashme në terma të përpunimit të rregullave të biznesit. Secili klient do të dëshirojë grup unik të logjikës për m'u ekzekutu në disa pika të parashikueshme në skenarët e përdorimit të sistemit, siç është kur niset një shitje e re apo kur shtohet një artikull i ri në shitje. Prandaj, do të na duhet një mekanizëm për me ofru këtë fleksibilitet dhe mundësi të specifikimit.

Duke përdorë strategji iterative të zhvillimit, do të vazhdojmë nëpër kërkesa, analizë të orientuar kah objektet, dizajn dhe implementim.

**Rasti Dy: Sistemi i Lojës Monopoly**

Për me tregu se praktikat e njëjta të OOA/D munden m'u apliku në probleme shumë të ndryshme, e kam zgjedhë një version të softuerit të lojës Monopoly si një studim tjetër të rastit. Edhe pse domeni dhe kërkesat nuk janë fare si një sistem i biznesit siç është NextGen POS, do të shohim se modelimi i domenit, dizajni i objekteve me paterna, dhe aplikimi i UML janë ende relevante dhe të dobishme. Si me një POS, versionet softuerike të Monopoly't janë vërtet të zhvilluara dhe të shitura, edhe me UI klient-i-pasur (rich-client) edhe me UI Web.

Nuk do t'i përsëris rregullat për Monopoly; si duket pothuajse secili person, në secilin vent, e ka luajtë këtë lojë si fëmijë apo adoleshent. Nëse keni pyetje, rregullat janë online në shumë web-faqe.

Versioni softuerik i lojës do të ekzekutohet si simulim. Një person do ta fillojë lojën dhe do ta tregojë numrin e lojtarëve të simuluar, dhe pastaj do të shohë derisa loja ekzekutohet deri në përfundim duke e paraqitë një gjurmë të aktivitetit gjatë radhëve të lojtarëve të simular.

## Rasti i përdorimit RP1: Përpunoje Shitjen

(kapitulli 6)

**Niveli:** qëllim i përdoruesit

**Aktori primar, kryesor:** Kasieri

**Pjesëmarrësit (aksionarët) dhe Interesat:**

- Kasieri: Dëshiron futje të saktë dhe të shpejtë, dhe pa gabime të pagesës, meqë mungesat e keshit tërhiqen nga rroga e tij.
- Shitësi: I dëshiron komisionet (shtesat) e shitjes të freskuara
- Klienti: Dëshiron blerje dhe shërbim të shpejtë me angazhim minimal. Dëshiron shfaqje të lehtë e të dukshme të artikujve dhe çmimeve të futura. Dëshiron dëshmi të blerjes për t'i përkrahë kthimet.
- Kompania: Dëshiron t'i regjistrojë saktësisht transaksionet dhe t'i kënaqë interesat e klientit. Dëshiron të sigurojë që regjistrohen të pranueshmet e pagesës për Shërbimin e Autorizimit të Pagesës. Dëshiron ca tolerancë ndaj gabimit për me leju regjistrim të shitjeve edhe nëse komponentat e serverit (p.sh. validimi në distancë i kredisë) janë të padisponueshme. Dëshiron freskim automatik dhe të shpejtë të kontabilitetit dhe të inventarit.
- Menaxheri: Dëshiron të jetë i aftë që t'i kryejë shpejt operacionet e mbishkrimit, dhe t'i rregullojë lehtë problemet e Kasierit.
- Agjencitë Qeveritare të Tatimeve: Duan të mbledhin taksë nga secila shitje. Mund të jenë shumë agjenci, si, kombëtare, shtetorëre dhe qarkore.
- Shërbimi i Autorizimit të Pagesës: Dëshiron të pranojë kërkesa me autorizim digjital në formatin dhe protokolin korrekt. Dëshiron që të llogarisë saktë të pagueshmet e tyre për shitoren.

**Parakushtet:** Kasieri është identifikua dhe autentiku.

**Garantimi i Suksesit (ose Paskushtet):** Shitja është ruajtur. Tatimi është llogaritur në mënyrë korrekte. Kontabiliteti dhe Inventari janë freskuar. Komisionet janë regjistruar. Fatura gjenerohet. Aprovimet e autorizimit të pagesës janë regjistruar.

**Skenari Kryesor i Suksesit (apo Rrjedha Themelore):**

1. Klienti arrin te kasa me të mirat dhe/ose shërbimet që don t'i blejë
2. Kasieri e nis një shitje të re.
3. Kasieri e fut identifikuesin e artikullit.
4. Sistemi e ruan rreshtin e shitjes dhe e paraqet përshkrimin e artikullit, çmimin dhe totalin aktual. Çmimi llogaritet nga një grup i rregullave të çmimeve.

   *Kasieri i përsërit hapat 3-4 derisa ta indikojë "mjaft".*

5. Sistemi e paraqet totalin me taksat e llogaritura.
6. Kasieri ia tregon totalin klientit dhe e kërkon pagesën.

7. Klienti paguan dhe sistemi e trajton pagesën.
8. Sistemi e fut në ditar shitjen e kryer dhe i dërgon informatat e shitjes dhe të pagesës në sistemin e jashtëm të Kontabilitetit (për kontabilitet dhe komisione) dhe sistemin e Inventarit (për me freskua Inventarin).
9. Sistemi e paraqet faturën.
10. Klienti shkon me faturën dhe të mirat (nëse ka).

**Zgjerimet (ose Rrjedhat Alternative):**

*a. Në çfarëdo kohe, Menaxheri kërkon veprim të mbishkrimit:

1. Sistemi hyn në mënyrën Menaxheri i autorizuar
2. Menaxheri apo Kasieri e kryejnë një veprim të Mënyrës së menaxherit, p.sh., ndërrimin e balansit të keshit, rifillo një shitje të suspenduar në një regjistër tjetër, anuloje një shitje, etj.
3. Sistemi kthehet në mënyrën Kasieri i autorizuar.

*b. Në çfarëdo kohe, Sistemi bjen (dështon):

Për me përkrahë rikëndelljen dhe për me rregullu kontabilitetin, sigurohu që të gjitha gjendjet dhe ngjarjet e ndijshme të transaksionit mund të rikëndellen nga cilido hap i skenarit.

1. Kasieri e ristarton Sistemin, kyçet, dhe kërkon rikthim të gjendjes së mëparshme.
2. Sistemi e rikonstrukton gjendjen e mëparshme.

    2a. Sistemi detekton anomali që e parandalojnë rikëndelljen:

    1. Sistemi e sinjalizon për gabimin Kasierin, e regjistron gabimin, dhe hyn në gjendje të pastër
    2. Kasieri fillon shitje të re

1a. Klienti apo Menaxheri indikojnë ta rikthejnë një shitje të anuluar.

1. Klienti e kryen operacionin e rikthimit, dhe e fut IDnë për me lexu shitjen.
2. Sistemi e shfaq gjendjen e shitjes së rikthyer, me nëntotalin.
    2a. Shitja nuk gjendet.
    1. Sistemi e sinjalizon Kasierin për gabimin.
    2. Kasieri me gjasë fillon një shitje të re dhe i ri-fut të gjithë artikujt.
3. Kasieri vazhdon me shitjen (me siguri duke futur më shumë artikuj apo duke e trajtu pagesën).

2-4a. Klienti i tregon Kasierit se ai ka status të çlirimit nga tatimi (p.sh. seniorët, autoktonët)

1. Kasieri e verifikon, dhe pastaj e fut kodin e lirimit nga taksa
2. Sistemi e regjistron statusin (që do ta përdorë gjatë llogaritjes së tatimeve)

3a. ID jovalide e artikullit (nuk gjendet në sistem)

1. Sistemi e sinjalizon gabimin dhe e refuzon futjen
2. Kasieri reagon ndaj gabimit:

    2a. Ka ID që lexohet nga njeriu (p.sh. UPC numerike)

    1. Kasieri e fut manualisht IDnë e artikullit
    2. Sistemi e shfaq përshkrimin dhe çmimin
        2a. ID jovalide e artikullit: Sistemi e sinjalizon gabimin. Kasieri provon metodë alternative.

    2b. Nuk ka ID të artikullit, por ka çmim në etiketë:

    1. Kasieri i thotë Menaxherit me kry një operasion të mbishkrimit.
    2. Menaxheri e kryen mbishkrimin.
    3. Kasieri e indikon futjen manuale të çmimit, e fut çmimin, dhe kërkon taksim standard për këtë sasi (meqë nuk ka informatë për produktin, makina e taksës nuk mund ta nxjerrë ndryshe se si me e taksu)

    2c. Kasieri e kryen Ndihmën Gjeje Produktin për me marrë IDnë e vërtetë dhe çmimin

    2d. Përndryshe, Kasieri e lut një nëpunës për IDnë ose çmimin e vërtetë, dhe e bën ose IDnë manuale ose çmimin manual (shih më lart).

3b. Ka shumë nga kategoria e njëjtë e artikullit dhe përcjellja e identitetit unik të artikullit nuk është e rëndësishme (p.sh., 5 paqeta të burgerëve vegjetarianë):

1. Kasieri mund ta fusë identifikuesin e kategorisë së artikujve dhe sasinë.

3c. Artikulli kërkon futje manuale të kategorisë dhe të çmimit (si lulet apo kartolinat me çmime në to):

1. Kasieri e fut kodin e kategorisë speciale manuale, plus çmimin.

3-6a: Klienti i thotë Kasierit me e heqë një artikull nga blerja:
Kjo është legale vetëm nëse vlera e artikullit është më e vogël se sa limiti i heqjes për Kasierë, përndryshe nevojitet mbishkrim i Menaxherit.

1. Kasieri e fut identifikuesin e artikullit për me heqë nga shitja.
2. Sistemi e heq artikullin dhe e shfaq totalin e freskuar aktual.
    2a. Çmimi i artikullit e tejkalon kufirin e heqjes për Kasierët:
        1. Sistemi sinjalizon gabim, dhe sugjeron mbishkrim të Menaxherit
        2. Kasieri e kërkon mbishkrimin nga Menaxheri, e merr, dhe e përsërit operacionin.

3-6b: Klienti i thotë Kasierit me anulu shitjen:

1. kasieri e anulon shitjen në Sistem.

3-6c: Kasieri e suspendon shitjen:

1. Sistemi e regjistron shitjen ashtu që të jetë e disponueshme për marrje në cilindo regjistër POS.
2. Sistemi e paraqet një "faturë të suspendimit" që i përfshin artikujt e rreshtit dhe një ID të shitjes që përdoret për me çelë dhe rivendosë shitjen.

4a. Çmimi i ofruar nga sistemi është i padëshiruar (p.sh. Klienti është anku për diçka dhe i ofrohet çmim më i ulët):

1. Kasieri e kërkon aprovimin nga Menaxheri.
2. Menaxheri e kryen operacionin e mbishkrimit.
3. Kasieri e fut çmimin manual të mbishkrimit.
4. Sistemi e paraqet çmimin e ri.

5a. Sistemi detekton dështim në komunikimin me shërbimin e jashtëm të llogaritjes së taksimit:

1. Sistemi e ristarton shërbimin në nyjen e POS dhe vazhdon.
    1a. Sistemi detekton se shërbimi nuk ristartohet
        1. Sistemi sinjalizon për gabimin.

2. Kasieri mund ta llogarisë manualisht dhe ta fusë taksën, ose ta anulojë shitjen.

5b. Klienti thotë se kanë të drejtë në zbritje (p.sh. nëpunës, klient i preferuar):

1. Kasieri e sinjalizon kërkesën për zbritje.
2. Kasieri e fut identifikimin e Klientit
3. Sistemi e paraqet totalin e zbritjes, duke u bazu në rregullat e zbritjes.

5c. Klienti thotë se ka kredi në llogarinë e vet, për me apliku në shitje:

1. Kasieri e sinjalizon kërkesën për kredi
2. Kasieri e fut identifikimin e Klientit
3. Sistemi e aplikon kredinë deri në çmimin = 0, dhe e redukton kredinë e mbetur.

6a. Klienti thotë se ka dashtë me pagu në kesh por nuk ka mjaft kesh:

1. Kasieri kërkon mënyrë alternative të pagesës

   1a. Klienti i tregon Kasierit me anulu shitjen. Kasieri e anulon shitjen.

7a. Pagesa me kesh:

1. Kasieri e fut sasinë e parave të ofruara.
2. Sistemi e paraqet balansin, dhe e liron fjokën e kasës.
3. Kasieri e depoziton keshin e futur dhe ia kthen balansin në kesh klientit.
4. Sistemi e regjistron pagesën kesh.

7b. Pagesa me kredi:

1. Klienti i fut informatat e llogarisë së kreditit.
2. Sistemi e shfaq pagesën e vet për verifikim.
3. Kasieri konfirmon.

   3a. Kasieri e anulon hapin e pagesës:

      1. Sistemi kthehet në mënyrën "futja e artikujve"

4. Sistemi e dërgon kërkesën për autorizim të pagesës te një Sistem i Shërbimit të Autorizimit të Pagesës, dhe kërkon aprovim të pagesës.

   4a. Sistemi detekton gabim për me bashkëpunu me sistemin e jashtëm:

      1. Sistemi e sinjalizon gabimin te Kasieri.

2. Kasieri i thotë Klientit për pagesë alternative.

5. Sistemi e pranon aprovimin e pagesës, e sinjalizon aprovimin te Kasieri, dhe e liron fjokën e kasës (për me futë faturën e nënshkruar të pagesës me kredi).

5a. Sistemi pranon refuzim të pagesës:

1. Sistemi e sinjalizon refuzimin te klienti

2. Kasieri i thotë Klientit për pagesë alternative.

5b. Kalon koha për pritjen e përgjigjjes

1. Sistemi e sinjalizon kalimin e kohës

2. Kasieri mund të provojë prapë, ose t'i thotë klientit për pagesë alternative

6. Sistemi e regjistron pagesën e kredisë, që e përfshin aprovimin e pagesës.

7. Sistemi e prezenton mekanizmin e futjes së nënshkrimit të kredisë.

8. Kasieri e lut klientin për nënshkrim të pagesës me kredi. Klienti e jep nënshkrimin.

9. Nëse nënshkrimi në letër, Kasieri e vendos faturën në fjokë dhe e mbyll.

7c. Pagesa me çek...

7d. Pagesa me debi...

7e. Kasieri e anulon hapin e pagesës.

1. Sistemi kthehet në mënyrën "futja e artikujve".

7f. Klienti i paraqet kuponat:

1. Para trajtimit të pagesës, Kasieri e regjistron secilin kupon dhe Sistemi e redukton çmimin si duhet. Sistemi i regjistron kuponat e përdorur për arsye të kontabilitetit.

1a. Kuponi i futur nuk është për asnjë artikull të blerë:

1. Sistemi sinjalizon gabim te Kasieri.

9a. Ka rabat (zbritje) të produkteve:

1. Sistemi i paraqet format e zbritjes dhe faturat e zbritjes për secilin artikull me rabat.

9b. Klienti kërkon faturë të dhuratës (pa çmime):

1. Kasieri kërkon faturë të dhuratës dhe sistemi e paraqet atë.

9c. Printeri pa letër.

1. Nëse sistemi mund ta detektojë gabimin, do ta sinjalizojë problemin.

2. Kasieri e vendos letrën.

3. Kasieri kërkon edhe një faturë.

**Kërkesat speciale:**

- Ndërfaqe e përdoruesit për Ekran Prekës në një monitor të madh të rrafshtë të panelit. Teksti duhet të jetë i dukshëm nga 1 metër.
- Përgjigjje e autorizimit të kredisë brenda 30 sekondave në 90% të kohës.
- Disi, duam rikëndellje të shpejtë kur qasja në shërbimet në distancë si ç'është sistemi i inventarit, dështon.
- Ndërkombëtarizimi i gjuhës në tekstin e shfaqur.
- Rregulla të futshme të biznesit të jenë të futshme në hapat 3 dhe 7.

**Lista e Teknologjisë dhe e Variacioneve të Shënimeve:**

*a. Mbishkrimi i menaxherit futet duke e kalu një kartelë të mbishkrimit përmes një lexuesi të kartelave, ose duke e futë një kod të autorizimit përmes tastierës.

3a. Identifikuesi i artikujve i futur nga barkod laser skeneri (nëse bar kodi është prezent) ose tastiera.

3b. Identifikues i sistemit mund të jetë cilado skemë e kodimit UPC, EAN; JAN, apo SKU.

7a. Informatat e llogarisë së kredisë të futura nga lexuesi i kartelës apo tastiera.

7b. Nënshkrimi i pagesës së kredisë i kapur në faturë letër. Por brenda dy viteve, parashikojmë se shumë klientë do të duan kapje të nënshkrimit digjital.

**Frekuenca e Ngjarjes:** mund të jetë pothuajse e vazhdueshme.

**Çështjet e hapura:**

- Cilat janë variacionet e ligjeve të taksave?
- Hulumtoje çështjen e rikëndelljes së shërbimit në distancë.
- Çfarë specifikimi nevojitet për biznese të ndryshme?
- A duhet një kasier me e marrë fjokën e kasës kur shkyçen?
- A mundet ta përdorë direkt klienti lexuesin e kartelës apo duhet me bë kasieri?

Ky rast i përdorimit është ilustrues e jo i detalizuar (edhe pse është i bazuar në kërkesat e një sistemi real POS – i zhvilluar me dizajn OO në Jaba). Megjithatë, këtu ka mjaft detale dhe kompleksitet për me ofru një sens realistik se një rast i përdorimit "plotësisht i veshur" (fully dressed) mundet me regjistru shumë detale të kërkesave. Ky shembull do të shërbejë edhe si model për shumë probleme të rasteve të përdorimit.

## Rasti i përdorimit RP1: Luaje lojën Monopoly

(kapitulli 6)

**Shtrirja:** Aplikacioni Monopoly

**Niveli:** qëllimi i përdoruesit

**Aktori kryesor:** Vështruesi

**Aksionarët dhe Interesat:**

- Vështruesi: Dëshiron që ta vështrojë lehtë daljen e simulimit të lojës

**Skenari kryesor i suksesit:**

1. Vështruesi e kërkon një inicializim të ri të lojës, i jep numrat e lojtarëve
2. Vështruesi fillon play
3. Sistemi e shfaq gjurmën e lojës për lëvizjen e lojtarit tjetër (shih rregullat e domenit, dhe "gjurmën e lojës" në fjalor për detale të gjurmës).

*Përsërite hapin 3 derisa të ketë fitues apo deri sa Vështruesi e anulon.*

**Zgjerimet**:

*a. Në çdo kohë, Sistemi dështon:

(Për me përkrahë rikëndelljen, Sistemi shkruan në ditar pas çdo lëvizjeje të kryer)

1. Vështruesi e ristarton sistemin
2. Sistemi e detekton dështimin e kaluar, e rikonstrukton gjendjen, dhe kërkon për të vazhduar
3. Vështruesi zgjedh me vazhdu (nga rendi i lojtarit të fundit që e ka kry).

**Kërkesat speciale:**

- Ofro të dy mënyrat: edhe gjurmë grafike edhe tekstuale.

## Specifikim plotësues (suplementar) – Shembulli NextGen

(kapitulli 7)

**Historia e Rishikimeve**

| Versioni | Data | Përshkrimi | Autori |
|---|---|---|---|
| Drafti i zanafillës | 10 janar 2031 | Drafti i parë. M'u gdhendë kryesisht gjatë shtjellimit (elaborimit). | Craig Larman |
| | | | |

**Hyrje**

Ky dokument është depo e të gjitha kërkesave të NextGen që nuk janë përfshirë në rastet e përdorimit.

**Funksionaliteti**

(Funksionaliteti i përbashkët nëpër shumë raste të përdorimit)

**Shkruarja në Ditar dhe Trajtimi i Gabimeve**

Shkruaji në ditar (log) të gjitha gabimet në një depo persistente (të qëndrueshme)

**Rregullat e kyçshme (të integrueshme, pluggable)**

Në pika të ndryshme të skenarit të disa rasteve të përdorimit (që do të definohen) përkrahe aftësinë me e specifiku (customize) funksionalitetin e sistemit me një grup të rregullave arbitrare që ekzekutohen në atë pikë ose ngjarje.

**Siguria**

Të gjitha përdorimet kërkojnë autentikim të përdoruesit

**Përdorshmëria**

***Faktorët njerëzorë***

Klienti do të jetë në gjendje ta shohë një shfaqje të POS në një monitor të madh. Prandaj:

- Teksti duhet të jetë i dukshëm nga 1 metër
- Shmangiu ngjyrave që shoqërohen me format më të shpeshta të verbërisë së ngjyrave.

Përpunimi i shpejtë, i lehtë, dhe pa gabime janë supreme në përpunimin e shitjeve, meqë bleresi don të shkojë shpejt, ose e perceptojnë përjetimin e blerjes (dhe shitësin) si më pak pozitive.

Kasieri shpesh e shikon klientin apo artikujt, jo ekranin e kompjuterit. Prandaj, sinjalet dhe alarmet duhet të njoftohen me zë e jo vetëm përmes grafikës.

**Besueshmëria**

***Rikëndellshmëria***

Nëse ka dështim në përdorimin e shërbimeve të jashtme (autorizuesi i pagesës, sistemi i kontabilitetit, ...) mundohu me i zgjidhë me një zgjidhje lokale (p.sh. vendos dhe dërgo) në mënyrë që ende të mund të kryhet shitja. Këtu nevojitet shumë më shumë analizë...

**Performansa**

Si u përmend te faktorët njerëzorë, blerësit duan ta bëjnë përpunimin e shitjeve shumë shpejt. Një qafëshishe është autorizimi i jashtëm i pagesës. Qëllimi ynë: autorizimi për më pak se 1 minutë, 90% të kohës.

**Përkrahshmëria**

***Adaptabiliteti***

Klientët e ndryshëm të POSit NextGen kanë nevoja unike të rregullave të biznesit dhe të përpunimit gjatë përpunimit të një shitjeje. Prandaj, në disa pika të definuara të skenarit (për shembull, kur niset një shitje, kur shtohet një artikull i ri) rregulla e futshme do të aktivizohet.

***Konfigurueshmëria***

Klientët e ndryshëm dëshirojnë konfigurime të ndryshme të rrjetës për sistemet e veta POS, siç kanë klientët e trashë versus të hollë, shtresat fizike dy-shtresore versus N-shtresore, etj. Pastaj, ata e duan mundësinë e ndryshimit të këtyre konfiguracioneve, për m'i reflektu nevojat e tyre të ndryshueshme të biznesit dhe performansës. Prandaj, sistemi do të jetë disi i konfigurueshëm për

m'i reflektu këto nevoja. Në këtë fushë nevojitet shumë më shumë analizë për m'i zbulu zonat dhe shkallën e fleksibilitetit, dhe përpjekjet për me e arritë atë.

**Kufizimet e implementimit**

Udhëheqja e NextGen insiston në zgjidhje me teknologji të Java's, duke parashiku se kjo do ta përmirësojë bartjen dhe përkrahshmërinë afat-gjatë, si shtojcë të lehtësisë së zhvillimit.

**Komponentat e blera**

- Llogaritësi i taksave. Duhet të përkrahë llogaritës së integrueshëm për shtete të ndryshme.

**Komponentat e Lira (free) me Kod të Hapur**

Në përgjithësi, e rekomandojmë makzimizimin e komponentave të lira me kod të hapur të teknologjisë Java në këtë projekt.

Edhe pse është heret të dizajnohen dhe zgjedhen definitivisht komponentat, sugjerojmë si në vijim si kandidatë potencialë:

- JLog korniza për kyçje
- ...

**Interfejsat**

***Hardueri dhe Interfejsat e Rëndësishëm***

- Monitori me ekran prekës (ky perceptohet nga sistemet operative si monitor i rregullt, dhe gjestet e prekjes si ngjarje të mausit)
- Skener laserik i barkodit (këta normalisht lidhen me tastierë speciale, dhe hyrja e skenuar perceptohet në softuer si taste të trusura)
- Printeri për fatura
- Lexuesi i kartelave debi/kredi
- Lexuesi i nënshkrimit (por jo në lëshimin 1)

***Interfejsat Softuerikë***

Për shumicën e sistemeve të jashtme bashkëpunuese (llogaritësi i taksave, kontabiliteti, inventari,...) na duhet të jemi të aftë me integru sisteme të ndryshme, dhe prandaj interfejsa të ndryshëm.

**Rregullat e Domenit (Biznesit) që janë Specifike për Aplikacionin**

(Shih dokumentin e veçantë Rregullat e Biznesit për rregullat e përgjithshme)

| ID | Rregulla | Ndërrueshmëria | Burimi |
|---|---|---|---|
| **RREGULLA1** | Rregullat e zbritjes së blerësit. Shembuj:<br>Nëpunësi: 20% më lirë.<br>Klient i preferuar: 10% më lirë.<br>Senior: 15% më lirë. | E lartë.<br>Secili shitës me pakicë përdor rregulla të ndryshme. | Politika e shitësit. |
| **RREGULLA2** | Rregullat e shitjes (në nivel të transaksionit). Zbatohet në totalin para-tatimit. Shembuj:<br>10% më lirë nëse totali mbi 100$<br>5% më lirë çdo të Hënë.<br>10% më lirë secila shitje nga 10am në 3pm sot.<br>Tofu 50% më lirë nga 9am-10am sot. | E lartë.<br>Secili shitës përdor rregulla të ndryshme, dhe ato mund të ndryshojnë në baza ditore ose brenda orës. | Politika e shitësit. |
| **RREGULLA3** | Rregullat e zbritjes për produkt (artikull i rreshtit).<br>Shembuj:<br>10% më lirë traktorët këtë javë.<br>Bleji dy vegjeburgerë, merre 1 falas. | E lartë.<br>Secili shitës përdor rregulla të ndryshme, dhe ato mund të ndryshojnë në baza ditore ose brenda orës. | Politika e shitësit. |

**Çështjet ligjore**

I rekomandojmë disa komponenta me kod të hapur nëse mund të zgjidhen kufizimet e tyre të licensimit për me leju rishitjen e produkteve që përfshijnë softuer me kod të hapur.

Të gjitha rregullat duhet, sipas ligjit, të zbatohen gjatë shitjeve. Vëreni se këto mund të ndryshojnë shpesh.

**Informata në Domenet e Interesit**

***Caktimi i çmimeve***

Pos rregullave të caktimit të çmimit të përshkruara në seksionin e rregullave të domenit, vëreni se produktet kanë çmim origjinal, dhe opsionalisht një çmim të përhershëm të zbritjes. Çmimi i produktit (para zbritjeve të mëtutjeshme) është çmimi permanent i zbritjes, nëse ekziston. Organizatat e mirëmbajnë çmimin origjinal edhe nëse ka çmim të përhershëm të zbritjes, për arsye të kontabilitetit dhe të taksimit.

***Trajtimi i pagesave debi dhe kredi***

Kur një pagesë elektronike debi apo kredi aprovohet nga një shërbim i autorizimit të pagesës, ata janë përgjegjës për pagesën e shitësit, jo blerësi. Si pasojë, për secilën pagesë, shitësi duhet t'i regjistrojë paratë që i ka borxh në llogaritë e tyre të pranueshme, nga shërbimi i autorizimit. Zakonisht, çdo mbrëmje, shërbimi i autorizimit do të kryejë transfer elektronik të fondeve në llogarinë e shitësit për borxhin total ditor, minus një tarifë (e vogël) për transaksion që e kërkon si pagesë shërbimi.

***Taksat e shitjes***

Llogaritjet e taksave të shitjes mund të jenë shumë komplekse, dhe ndërrohen rregullisht në përgjigjje të legjislacionit në të gjitha nivelet e qeverisjes. Prandaj, delegimi i llogarive të taksimit një softueri llogaritës palë e tretë (prej të cilëve disa janë në dispozicion) është i këshillueshëm. Tatimin mund të jetë borxh për trupat e qytetit, regjionit, shtetit, dhe atij nacional. Disa artikuj mund të jenë të liruar nga taksa pa kualifikim, ose të liruar varësisht nga blerësi apo pranuesi cak (për shembull, një fermer apo një fëmijë).

***Identifikuesit e artikujve: UPCtë, EANtë, SKUtë, Bar Kodet, dhe Lexuesit e Barkodeve.***

NextGen POS nevojitet të përkrahë skema të ndryshme të identifikimit të artikuve. UPCtë (Universal Product Codes), EANs (European Article Numbering) dhe SKUtë (Stock Keeping Units) janë tre sisteme më të shpeshta të identifikimit për produktet që janë shitur. Japanese Article Numbers (JANët) janë një lloj i versionit EAN.

SKUtë janë identifikues plotësisht arbitrarë që definohen nga shitësi.

Mirëpo, UPCtë dhe EANët kanë një komponentë të standardeve dhe rregullatore. Shih www.adams1.com/pub/russadam/upccode.html për një përmbledhje të mirë. Po ashtu, shih www.uc-council.org dhe www.ean-int.org.

## Dokument i vizionit - Shembulli NextGen: Vizioni (i Pjesshëm)

(kapitulli 7)

**Historia e rishikimit**

| Versioni | Data | Përshkrimi | Autor |
|---|---|---|---|
| drafti i zanafillës | 10 janar 2031 | Drafti i parë. Do të gdhendet gjatë shtjellimit (elaboration) | Craig Larman |
| | | | |

**Hyrje**

E vizionojmë një aplikacion pikë-e-shitjes (PESH) të gjeneratës së ardhshme, NextGen POS, me fleksibilitetin me përkrahë rregulla të ndryshme të biznesit të klientit, shumë mekanizma terminalë dhe të ndërfaqeve të përdoruesve, dhe integrimin me shumë sisteme përkrahëse si palë të treta.

*Analiza në këtë shembull është ilustrative, por imagjinare.*

**Pozicionimi**

***Oportuniteti i Biznesit***

Produktet ekzistuese POS nuk janë të adaptueshme për biznesin e klientit, në termat e rregullave të ndryshueshme të biznesit dhe dizajnet e ndryshueshme të rrjetave (për shembull klient i trashë apo jo, arkitektura 2, 3, apo 4-shtresore). Pastaj, ato nuk shkallëzohen mirë derisa numri i terminaleve dhe biznesi rriten. Dhe, asnjë nuk mund të punojnë ose në mënyrën on-line ose off-line, duke u adaptu në mënyrë dinamike varësisht nga dështimet. Asnjëri nuk integrohet lehtë me shumë sisteme palë-të-treta. Asnjë nuk lejon teknologji të reja të terminalëve siç janë PDAtë mobile. Ka pakënaqësi të tregut me ktë gjendje jofleksibile të çështjeve, dhe kërkesë për një POS që e rregullon këtë.

***Deklarata e Problemit***

Sistemet tradicionale janë jofleksibile, jotolerante ndaj gabimeve, dhe vështirë të integrueshme me sistemet e palëve të treta. Kjo çon në probleme në përpunimin në kohë të shitjeve, duke kriju procese të avansuara që nuk përputhen me softuerin, dhe shënimet e sakta dhe në kohë të kontabilitetit dhe të inventarit për me përkrahë matjen dhe planifikimin, ndërmjet shqetësimeve tjera. Kjo ndikon në kasierët, menaxherët e shitjes, administratorët e sistemeve, dhe menaxhmentin e korporatës.

***Deklarata e pozicionit të produktit***

Përmbledhje e shkurtër se për kë është sistemi, veçoritë e tij të shquara, dhe çka e dallon nga konkurenca.

***Alternativat dhe Gara...***

***Përshkrimet e aksionarëve***

*Kuptoni se kush janë lojtarët, dhe problemet e tyre.*

***Demografitë e tregut...***

***Përmbledhje e Aksionarëve (Jo-Përdorues)...***

***Përmbledhje e Përdoruesve...***

***Qëllimet kryesore të nivelit të lartë dhe problemet e aksionarëve***

Një punëtori (workshop) një-ditore e kërkesave me ekspertët e temës dhe aksionarët tjerë, dhe analizat në disa tregje të shitjeve çuan në identifikimin e qëllimeve dhe problemeve vijuese kryesore:

*Konsolido hyrjen nga Lista e Aktorëve dhe e Qëllimeve, dhe seksioni Interesat e Aksionarëve i rasteve të përdorimit.*

| **Qëllimi i nivelit të lartë** | **Prioriteti** | **Problemet dhe shqetësimet** | **Zgjidhjet aktuale** |
|---|---|---|---|
| Përpunim i shpejtë, i fuqishëm dhe i integruar i shitjeve | I lartë | Shpejtësia e zvogëluar derisa rritet ngarkesa<br>Humbja e mundësisë së përpunimit të shitjeve nëse komponentat dështojnë.<br>Mungesa e informatave aktuale dhe të sakta nga kontabiliteti dhe sistemet tjera si pasojë e jo-integrimit me sistemet ekzistuese të kontabilitetit, inventarit, dhe HR. Çon në vështirësi në matje dhe planifikim.<br>Pamundësia me i specifiku rregullat e biznesit për kërkesat unike të biznesit.<br>Vështirësia me shtu terminal të ri apo tipe të ndërfaqe së përdoruesit (për shembull PDA mobile). | Produktet ekzistuese POS ofrojnë përpunim themelor të shitjes, por nuk i adresohen këtyre problemeve. |
| . . . | . . . | . . . | . . . |

***Qëllimet e nivelit të përdoruesit***

Përdoruesit (dhe sistemet e jashtme) kërkojnë një sistem që i plotëson këto qëllime:

*Kjo mund të jetë Lista e Qëllimeve-Aktorëve e krijuar gjatë modelimit të rasteve të përdorimit, apo përmbledhje më e shkurtër.*

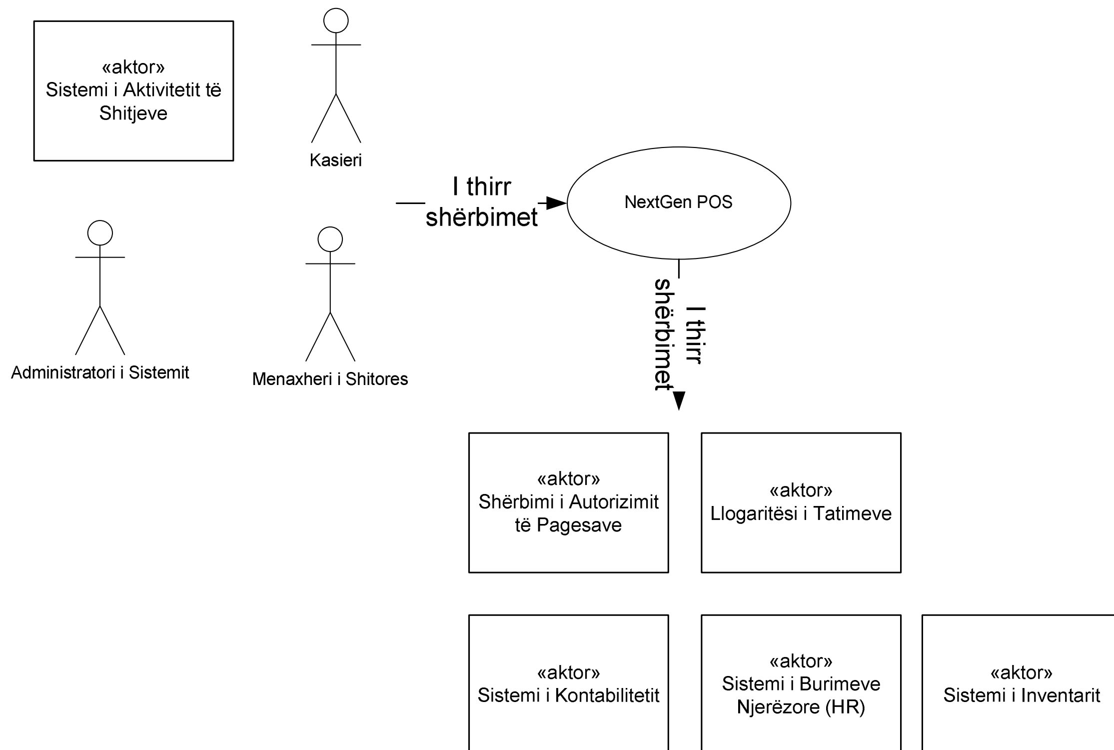
- Kasieri: përpuno shitjet, trajto kthimet, paret mrena, paret jashtë
- Administratori i sistemit: menaxho përdoruesit, sigurinë, dhe tabelat e sistemit
- Menaxheri: lshoje, nale
- Sistemi i aktivitetit të shitjeve: Analizo shënimet e shitjes

***Ambienti i përdoruesit…***

**Përmbledhje e Produktit**

***Perspektiva e Produktit***

POSi NextGen zakonisht do të rrijë në shitore; nëse përdoren terminalët mobilë, ata do të jenë në afërsi të afërt me rrjetën e shitorës, ose brenda ose afër jashtë. Do të ofrojë shërbime për përdoruesit, dhe bashkëpunojë me sistemet tjera, siç është e cekur në Figurën 1.




*Përmbledhur nga Diagrami i Rasteve të Përdorimit. Diagramet e kontekstit vijnë në formate të ndryshme me detale ndryshuese, por të gjitha i tregojnë aktorët kryesorë që kanë të bëjnë me sistemin.*

***Përmbledhja e Benefiteve***

| **Veçoria Përkrahëse** | **Përfitimi i Aksionarit** |
|---|---|
| Funksionaliteti, sistemi do t'i ofrojë të gjitha shërbimet e zakonshme që i kërkon një organizatë e shitjes, duke i përfshirë regjistrimin e shitjes, autorizimin e pagesës, trajtimin e kthimit, etj. | Shërbime të automatizuara, të shpejta të Pikës-së-Shitjes. |
| Detektimi automatik i dështimeve, kalimi në përpunimin offline për shërbimet e padisponueshme | Përpunimi i vazhduar kur komponentat e jashtme dështojnë. |
| Rregullat e integrueshme të biznesit në skenarë të ndryshëm gjatë përpunimit të shitjeve. | Konfigurim fleksibilil i logjikës së biznesit. |
| Transaksione në kohë reale me sisteme të palëve të treta, duke i përdorë protokolet standarde industriale. | Shitje me kohë, të sakta, informata të kontabilitetit dhe të inventarit, për me përkrahë matjen dhe planifikimin |
| … | … |

*Ngjashëm me listën Qëllimet-Aktorët, kjo tabelë i lidh qëllimet, përfitimet dhe zgjidhjet, por në nivel më të lartë që nuk ka të bëjë vetëm me rastet e përdorimit. Ajo e përmbledh vlerën dhe cilësitë dalluese të produktit.*

***Supozimet dhe varësitë…***

***Kostoja dhe caktimi i çmimeve…***

***Licensimi dhe Instalimi…***

**Përmbledhja e Veçorive të Sistemit**

*Siç diskutohet më poshtë, veçoritë e sistemit janë një format i shkurtër për me përmbledhë funksionalitetin.*

- Regjistrimi i shitjeve
- Autorizimi i pagesës (kredi, debi, çek)
- Administrimi sistemor për përdorues, siguri, kod dhe tabela të konstanave, etj.
- Procesimi automatik i shitjeve ofline kur komponentat e jashtme dështojnë
- Transaksionet në kohë reale, duke u bazu në standarde industriale, me sisteme të palëve të treta, duke e përfshi inventarin, kontabilitetin, resurset humane, llogaritësit e taksave, dhe shërbimet e autorizimit të pagesës
- Definicioni dhe ekzekutimi i rregullave të specifikueshme të futshme në pika të caktuara të përbashkëta në skenarë të përpunimit
- …

**Kërkesat dhe kufizimet tjera**

Duke i përfshirë kufizimet e dizajnit, përdorshmërinë, besueshmërinë, performansën, përkrahshmërinë, kufizimet e dizajnit, dokumentacionin, paketimin, etj: Shih Specifikacionin plotësues dhe rastet e përdorimit.

## Fjalor i projektit

(kapitulli 7)

**Shembulli NextGen: Fjalori (i pjesshëm)**

**Historia e rishikimit**

| Versioni | Data | Përshkrimi | Autori |
|---|---|---|---|
| Drafti i zanafillës | 10 janar 2031 | Drafti i parë. Do të gdhendet gjatë përpunimit. | Craig Larman |
| | | | |

**Definicionet**

| Termi | Definicioni dhe informata | Formati | Rregullat e validimit | Sinonimet/Aliaset |
|---|---|---|---|---|
| Artikull | Produkt ose shërbim për shitje | | | |
| Autorizim i pagesës | Validim nga një shërbim i jashtëm i autorizimit që do ta bëjnë ose do ta garantojnë pagesën ndaj shitësit | | | |
| Kërkesa e autorizimit të pagesës | Kompozit i elementeve që dërgohen elektronikisht në një shërbim të autorizimit, zakonisht një varg i karakterëve. Elementet përfshijnë: ID e shitores, numri i llogarisë së klientit, sasia dhe vula e kohës. | | | |
| UPC | Kodi numerik që e identifikon një produkt. Zakonisht simbolizohet me një barkod të vendosur në produkte. Shih www.uc-council.org për detale të formatit dhe validim. | Kod 12-shifror me disa nënpjesë | Shifra 12 është shifër e kontrollimit | Universal Product Code |
| … | … | | | |

## Rregullat e domenit - Shembulli NextGen

(kapitulli 7)

**Rregullat e domenit**

Historia e rishikimit

| Versioni | Data | Përshkrimi | Autori |
|---|---|---|---|
| Drafti i zanafillës | 10 Janar 2031 | Drafti i parë. Do të gdhendet së pari gjatë elaborimit | Craig Larman |
| | | | |

**Lista e rregullave**

(Shih po ashtu Rregullat specifike për Aplikacionin në Specifikimin Plotësues)

| ID | Rregulla | Ndërrueshmëria | Burimi |
|---|---|---|---|
| RREGULLA1 | Kërkohet nënshkrimi për pagesat kreditore | "Nënshkrimi" i blerësit do të vazhdojë të kërkohet, por brenda 2 viteve shumica e klientëve tonë duan regjistrim të nënshkrimit në një pajisje digjitale të regjistrimit, dhe brenda 5 viteve presim të ketë kërkesë për përkrahje të "nënshkrimit" me kod të ri digjital unik që përkrahet nga ligji. | Politika e thuajse të gjitha kompanive të autorizimit të kredive. |
| RREGULLA2 | Rregullat e taksimit. Shitjet kërkojnë taksa të shtuara. Shih statuset e qeverisë për detalet aktuale. | E lartë. Ligjet e taksave ndërrojnë çdo vit, në të gjitha nivelet e qeverisë. | ligji |
| RREGULLA3 | Ndryshimet e pagesës së kredisë mund të paguhen vetëm si kredi në llogarinë e kredisë të blerësit, jo si kesh. | E ulët | Politika e kompanisë së autorizimit të kredisë |

## Kërkesat e iteracionit 1

(kapitulli 8)

**NextGen POS**

Kërkesat për iteracionin e parë të aplikacionit NextGen POS janë si në vijim:

- Implemento një skenar themelor, kryesor të rastit të përdorimit Përpuno Shitjen: futja e artikujve dhe pranimi i pagesës kesh.
- Implemento një rast të përdorimit Fillo sipas nevojës për t'i përkrahë nevojat e inicializimit për iterimin.
- Asgjë ekstra ose komplekse nuk trajtohet, vetëm një skenar i thjeshtë i rrugës së lumtur, dhe dizajni dhe implementimi për me përkrahë atë.
- Nuk ka bashkëpunim me shërbimet e jashtme, siç është llogaritësi i tatimeve apo baza e produktit.
- Nuk aplikohen rregulla komplekse të çmimimit.
- Dizajni dhe implementimi i NP (UI) përkrahëse, bazës e kështu me radhë, po ashtu do të bëhej, por nuk është e mbuluar në ndonjë detal.

**Monopoly**

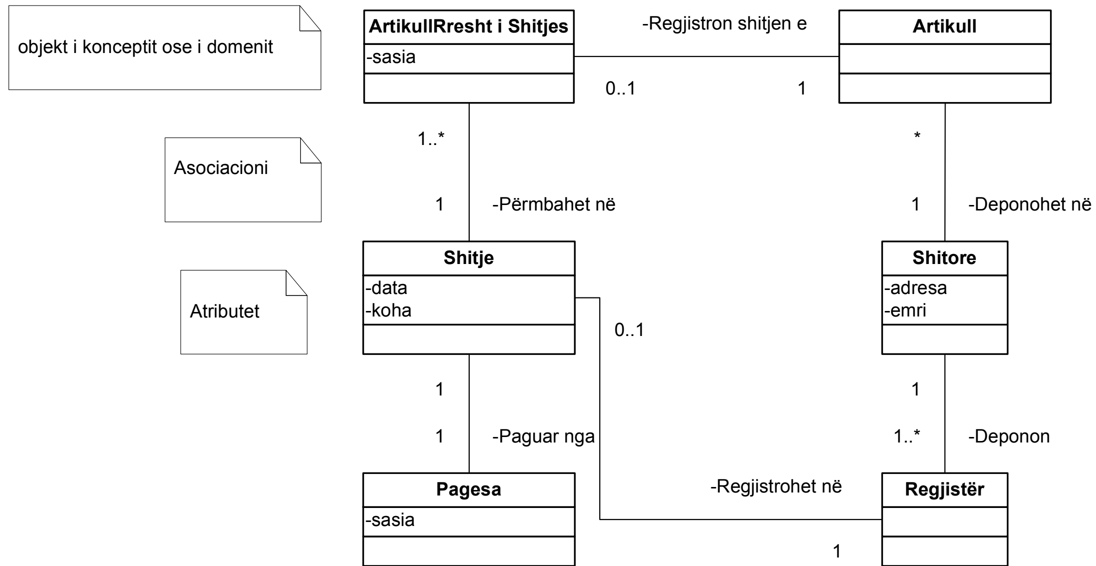
Kërkesat për iteracionin e parë të aplikacionit Monopoly janë si vijon:

- Implemento një skenar themelor, bazik të rastit të përdorimit Luaje Lojën Monopoly: lojtarët duke lëvizur nëpër katrorët e fushës.
- Implemento një rast të përdorimit Fillo sipas nevojës për me i përkrahë nevojat e inicializimit të iteracionit.
- Dy deri në tetë lojtarë mund të luajnë.
- Loja luhet si seri e rundeve. Gjatë një rundi, secili lojtar e merr radhën një herë. Në secilën radhë, një lojtar e shtyn figurën e tij në drejtim të akrepave përreth fushës në numër të katrorëve baraz me shumën e rrotulluar në dy zare gjashtë-faqëshe.
- Luaje lojën për vetëm 20 runde.
- Pasi të jenë rrokullisur zaret, emri i lojtarit dhe rrokullisja shfaqen. Kur lojtari lëviz dhe vendoset në një kuti, emri i lojtarit dhe emri i kutisë në të cilën është ndalur shfaqen.
- Në iteracionin-1 nuk ka para, fitues apo humbës, as prona për të blerë apo qira me pagu, dhe nuk ka kuti speciale të asnjë lloji.
- Secila kuti e ka një emër. Secili lojtar e fillon lojën me figurën e tij të vendosur në kutinë "Nisu." Emrat e kutive do të jenë Nisu, Kutia 1, Kutia 2, ... Kutia 39
- Nise lojën si simulim që nuk kërkon hyrje të përdoruesit, pos numrit të lojtarëve.

Iterimet pasuese do të ndërtohen në këto baza.

## Model i Domenit

(kapitulli 9)




## Klasat konceptuale

(kapitulli 9)

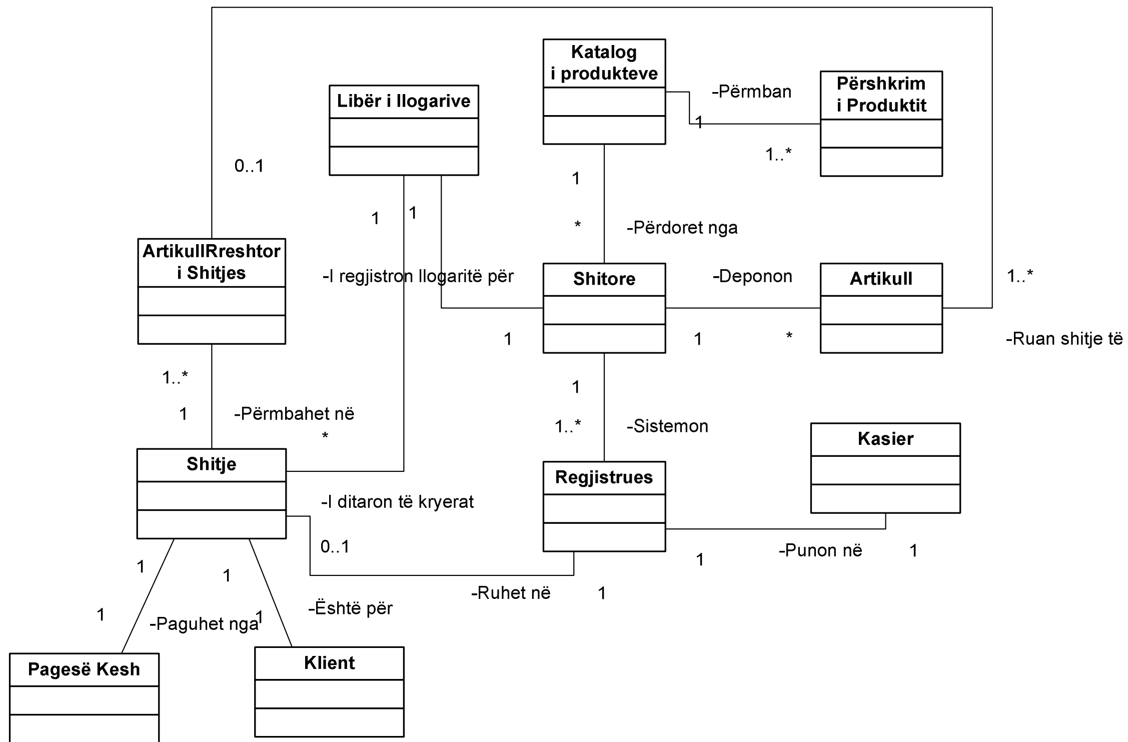
## Lista e asociacioneve më të shpeshta

(kapitulli 9)

| Kategoria | Shembuj |
|---|---|
| A është transaksion i lidhur me një transaksion tjetër B | PagesëKesh – Shitje<br>Anulim – Rezervim |
| A është artikull rreshtor i një transaksioni B | ArtikullRreshtorIShitjes – Shitje |
| A është produkt apo shërbim për një tranaksion (apo artikull të rreshtit) B | Artikull – ArtikullRreshtorIShitjes<br>Fluturim – Rezervim |
| A është rol i lidhur me një transaksion B | Klienti – Pagesa<br>Udhëtari – Bileta |
| A është pjesë fizike apo logjike e B | Tërheqësi – Regjistri<br>Fusha – Tabela<br>Ulëse – Aeroplan |
| A përmbahet fizikisht apo logjikisht në B | Regjistër – Shitore, Artikull – Raft, Fushë – Tabelë, Udhëtar – Aeroplan |
| A është përshkrim për B | PërshkrimIProduktit – Artikull<br>PërshkrimIFluturimit – Fluturim |
| A dihet/shkruhet në ditar/regjistrohet/raportohet/kapet në B | Shitje – Regjistër<br>Figurë – Fushë<br>Rezervim – ManifestIFluturimit |
| A është anëtar i B | Kasier – Shitore<br>Lojtar – LojëMonopol<br>Pilot – VijëAjrore |
| A është nënnjësi organizative e B | Departamenti – Shitorja<br>Mirëmbajtje - VijëAjrore |
| A e përdor ose e menaxhon ose e posedon B | Kasier – Regjistër<br>Lojtar – Figurë<br>Pilot – Aeroplan |
| A është afër B | ArtikullRreshtorIShitjes – ArtikullRreshtorIShitjes<br>Fushë – Fushë<br>Qytet-Qytet |

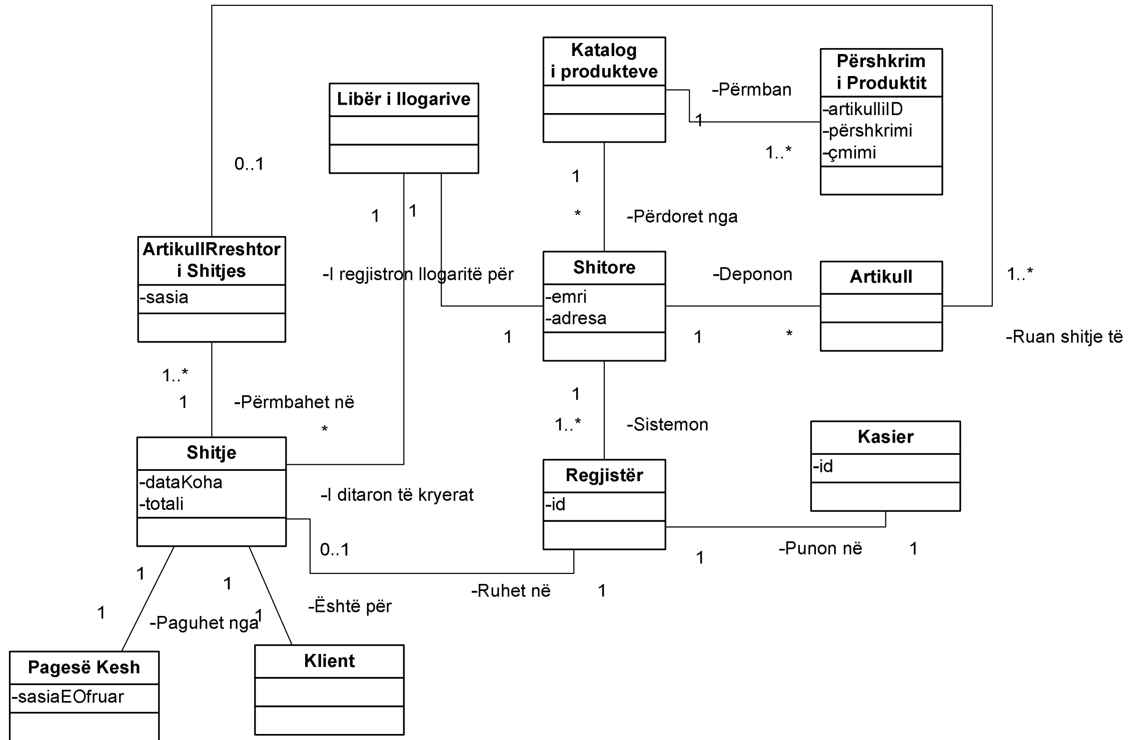
## Modeli i pjesshëm i domenit për NextGen POS

(kapitulli 9)



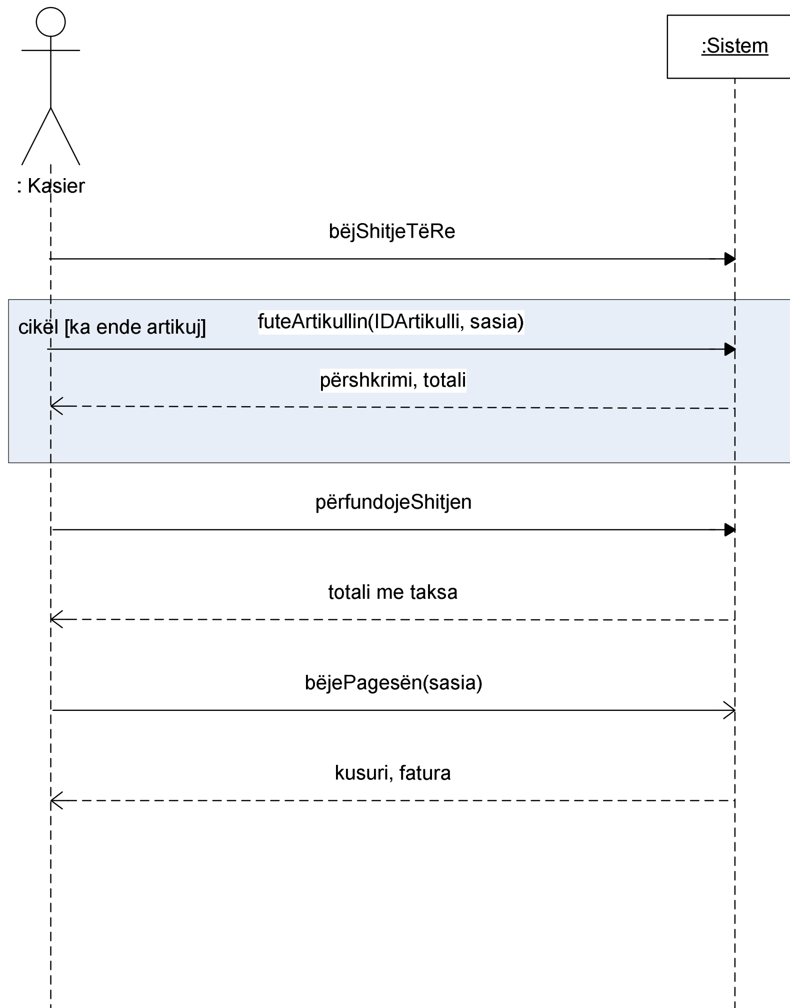
## Atributet - Modeli i pjesshëm i domenit për NextGen POS

(kapitulli 9)



## Diagram Sekuencial i Sistemit NextGen

(kapitulli 10)



## Kontrata të operacioneve

(kapitulli 11)

**Kontrata K01: bëjShitjeTëRe**

| **Operacioni:** | bëjShitjeTëRe() |
|---|---|
| **Ndër – referencat:** | Rastet e Përdorimit: Përpunoje Shitjen |
| **Parakushtet:** | Asnjë |
| **Paskushtet:** | - Është kriju një instancë Shitje sh<br>- Sh është asociu me një Regjistër<br>- Atributet e sh janë inicializu |

**Kontrata K02: regjistroArtikullin**

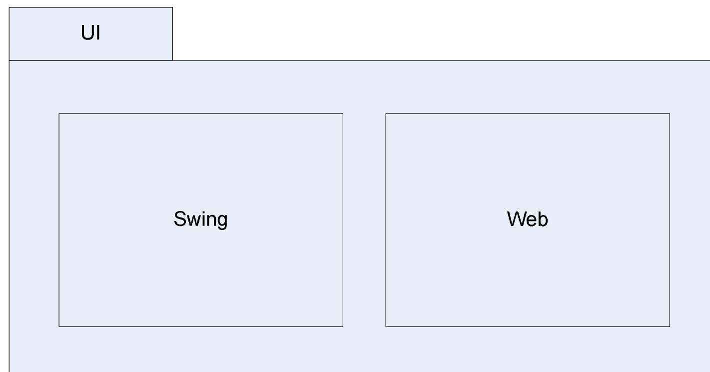
| **Veprimi:** | regjistroArtikullin(ArtikulliID: ArtikullID, sasia: integer) |
|---|---|
| **Ndër – referencat:** | Rastet e Përdorimit: Përpunoje Shitjen |
| **Parakushtet:** | Është një shitje në proces |
| **Paskushtet:** | - Është kriju një instancë ars e tipit ArtikullRreshtIShitjes (krijimi i instancës)<br>- Ars është asociu me Shitjen aktuale (asociacioni u formua)<br>- Ars.sasia është bë sasia (ndryshimi i atributit)<br>- Ars është shoqëru me një PërshkrimIProduktit, duke u bazu në përputhjen e artikullitID (asociacioni i formuar) |

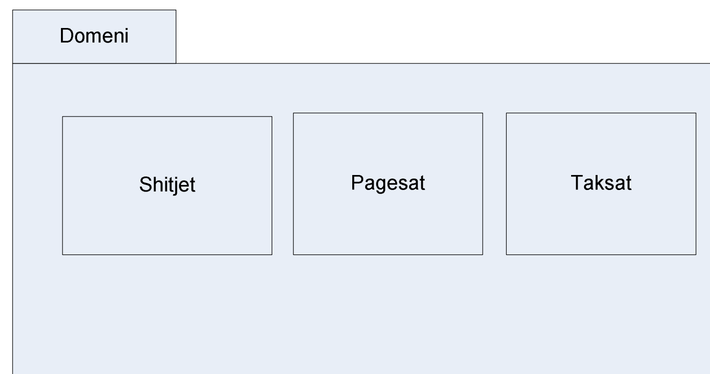
**Kontrata K03: mbaroShitjen**

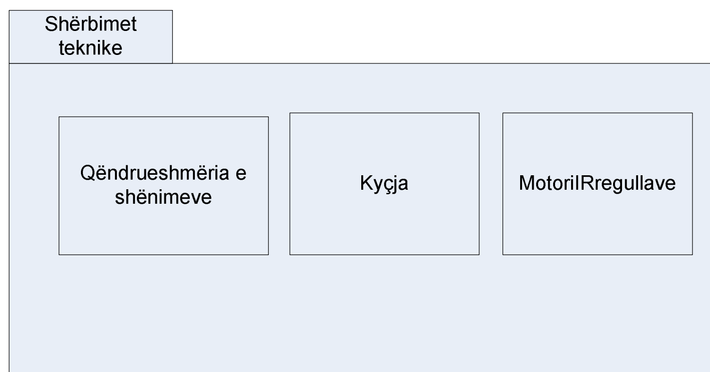
| **Operacioni:** | mbaroShitjen() |
|---|---|
| **Ndër – referencat:** | Rastet e Përdorimit: Përpuno Shitjen |
| **Parakushtet:** | Është një shitje në proces |
| **Paskushtet:** | - Është kriju një instancë Pagesë p<br>P.sasiaEOfruar është bë sasia<br>- P është shoqëru me Shitjen aktuale<br>- Shitja aktuale është shoqëru me Shitoren |

## Arkitektura logjike – shtresat

(kapitulli 13)

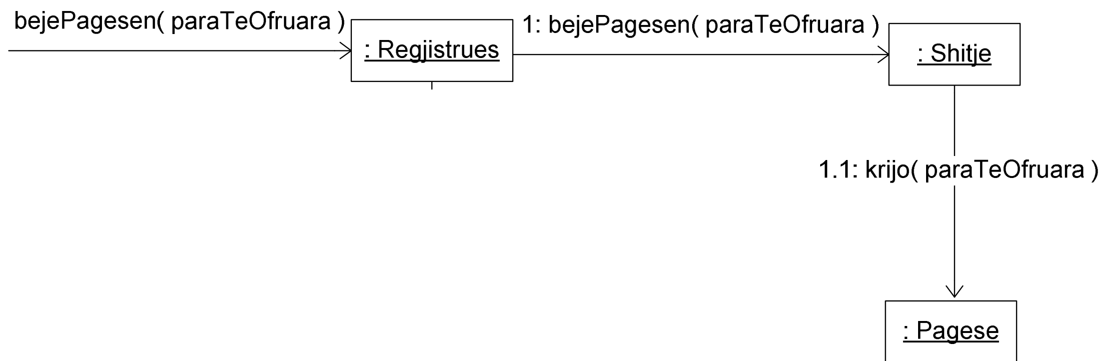


## Diagram sekuencial

(kapitulli 15)




## Shembull i kodit – rritja e ndryshores

(kapitulli 15)

```
public class Shitje
{
	private Liste<ArtikullRreshtIShitjes> artikujtRreshta =
		new ListeVarg<ArtikullRreshtIShitjes>();

	public TëHolla merreTotalin()
	{
		TëHolla totali = new TëHolla();
		TëHolla nentotali = null;

		for ( ArtikullRreshtIShitjes artikulliRresht : artikujtRreshta)
		{
			nentotali = artikulliRresht.merreNentotalin();
			totali.shtoje(nentotali);
		}

		return totali;
	}

	// ...
}
```

## Thirrje asinkronike

(kapitulli 15)

```
public class NisesIOres
{
      public void niseOren()
      {
            Thread t = new Thread( new Ore() );
            t.start();   // thirrja asinkronike e metodes 'run' ne Ore
            System.runFinalization(); // shembull i mesazhit pasues
      }

      // ...

}

// objektet duhet me implementu interfejsin Runnable
// ne Java per m'u perdore ne thread'a te rinj

public class Ore implements Runnable
{
      public void run()
      {
            while (true) // sille perhere ne thread'in (fijen) e vet
            {
                  // ...
            }
      }
      // ...
}
```

## Disa kontrata me diagrame sekuenciale

(kapitulli 18)

**Kontrata K01: bejShitjeTeRe**

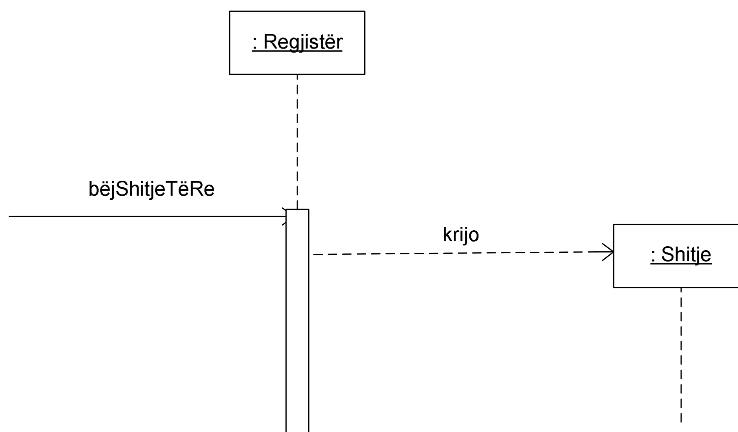
Operacioni: bějShitjeTëRe()

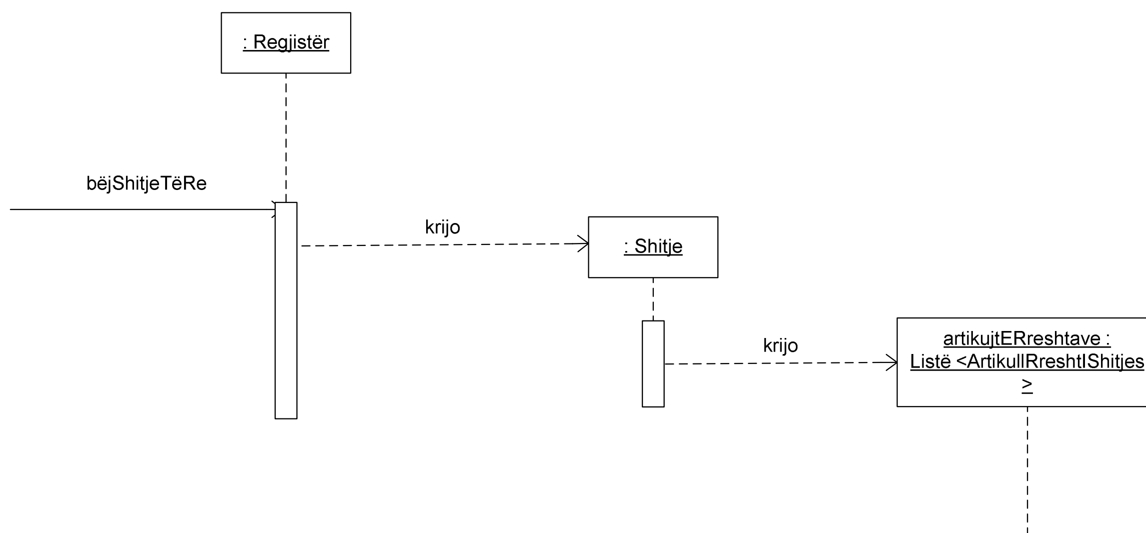
Ndër-referencat: Rastet e Përdorimit: Përpuno Shitjen

Parakushtet: asnjë

Paskushtet:

- Një instancë Shitje s është kriju (krijimi i instancës)
- s është shoqëru me Regjistrin (asociacioni është formu)
- Atributet e s janë inicializu.




: Regjistër
bëjShitjeTëRe
krijo
: Shitje
krijo
artikujtERreshtave :
Listë <ArtikullRreshtIShitjes
>

**Kontrata K02: futArtikull**

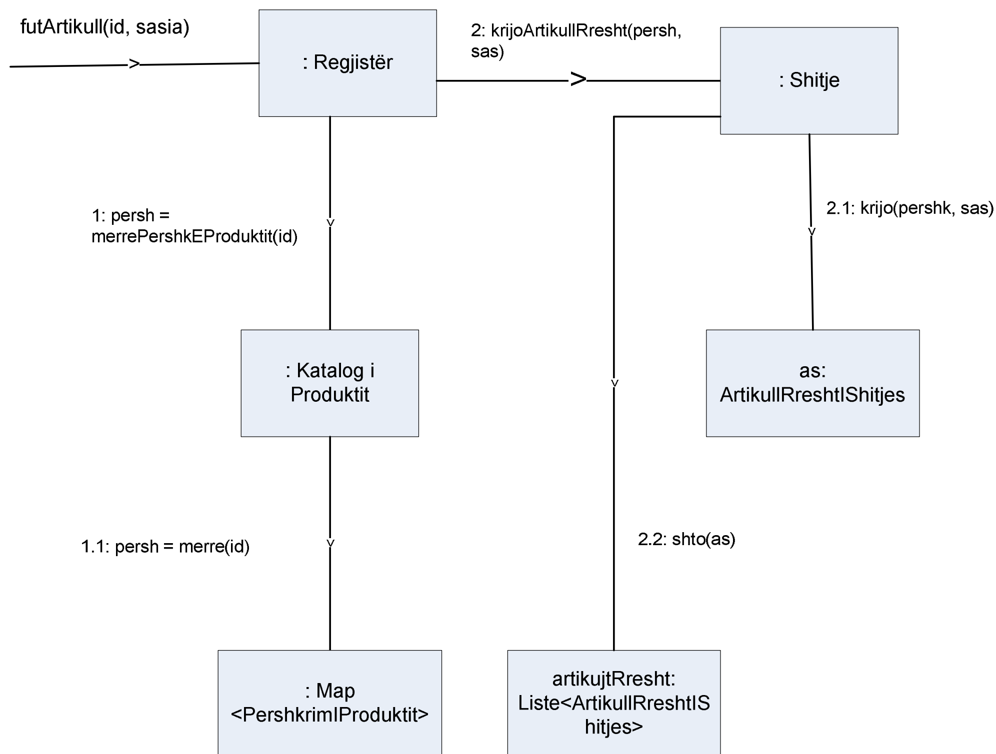
Operacioni: futArtikull(artikullID : ArtikulliID, sasia : integer)

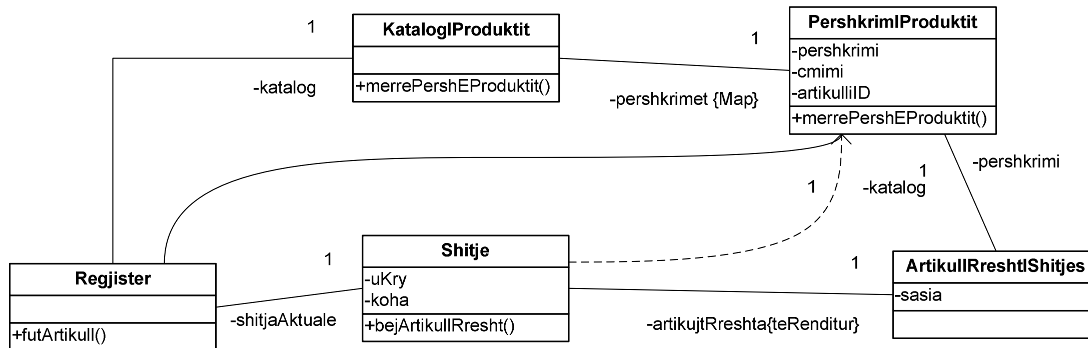
Ndër-referencat: Rastet e Përdorimit: Përpuno Shitjen

Parakushtet: Është një shitje rrugës

Paskushtet:

- Një instancë ArtikullRreshtIShitjes ars është kriju (krijimi i instancës)
- Ars është shoqëru me shitjen aktuale (asociacioni është formu)
- Ars.sasia u bë sasia (ndryshimi i atributeve)
- Ars është shoqëru me një PërshkrimIProduktit, duke u bazu në përputhjen në Id të Artikullit (asociacioni është formu).




KatalogIProduktit
+merrePershEProduktit()
PershkrimIProduktit
-pershkrimi
-cmimi
-artikulliID
+merrePershEProduktit()
1
-katalog
1
-pershkrimet {Map}
-pershkrimi
1
-katalog
1
Regjister
+futArtikull()
1
-shitjaAktuale
Shitje
-uKry
-koha
+bejArtikullRresht()
1
-artikujtRreshta{teRenditur}
ArtikullRreshtIShitjes
-sasia

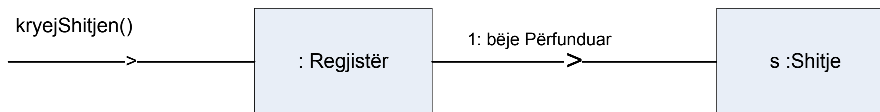
**Kontrata K03: kryejShitjen**

Operacioni: kryejShitjen()

Ndër-referencat: Rastet e Përdorimit: Përpuno Shitjen

Parakushtet: Është një shitje rrugës

Paskushtet: Shitja.ështëKry u bë True (ndryshimi i atributit)




| Informata e nevojitur për Totalin e Shitjes | Eksperti i Informatës |
|---|---|
| PërshkrimiIProduktit.çmimi | PërshkrimIProduktit |
| ArtikullRreshtIShitjes.sasia | ArtikullRreshtIShitjes |
| Të gjithë ArtikujtRreshtaTëShitjes në Shitjen aktuale | Shitja |

**Kontrata K04: kryejPagesën**

Operacioni: kryejPagesën(sasia: Para)

Ndër-referencat: Rastet e Përdorimit: Përpuno Shitjen

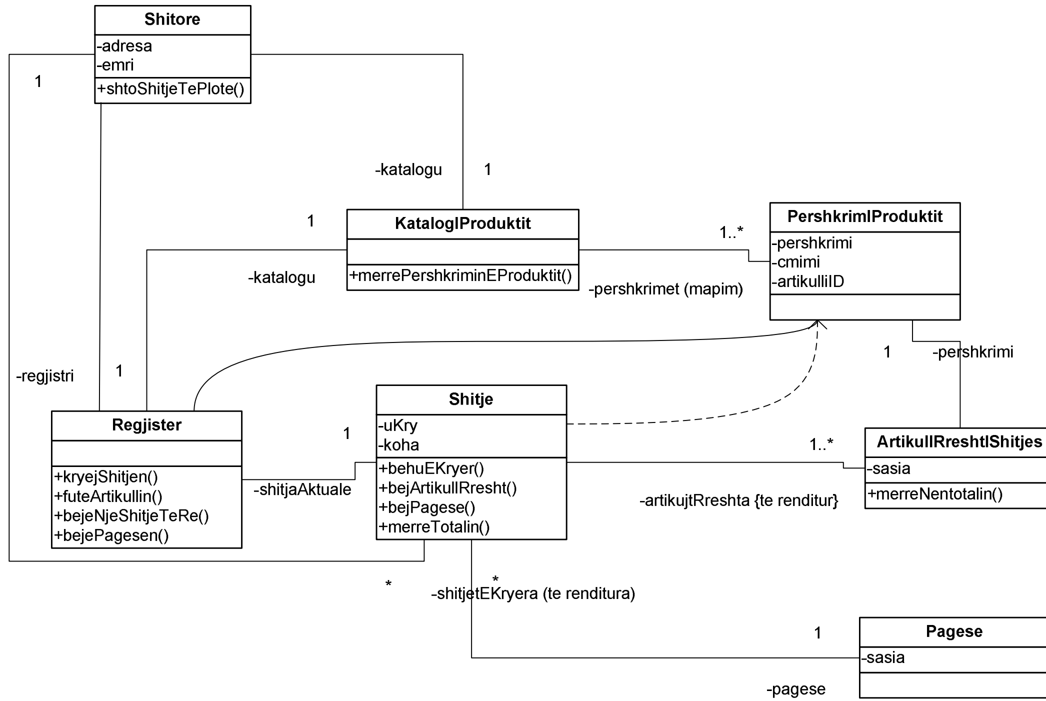
Parakushtet: Është një shitje rrugës

Paskushtet:

- Një instancë Pagesë p është kriju
- p.sasiaEOfruar është bë sasia
- p është shoqëru me shitjen aktuale
- shitja aktuale është asociu me Shitore (për me shtu në ditarin historik të shitjeve të kryera)

## Diagram i Dizajnit të Klasave

(kapitulli 18)



## Implementim në kod – Hyrje në zgjidhjen NextGen POS

(kapitulli 20)

```
package com.foo.nextgen.domen;

public class Pagese
{
	private Money sasia;

	public Pagese(Money parateEOfruara) { sasia = parateEOfruara; }
	public Money merreSasine() { return sasia; }
}


public class KatalogIProdukteve
{
	private Map<ArtikullID, PershkrimIProduktit>
		pershkrimet = new HashMap()<ArtikullID, PershkrimIProduktit>;

	public KatalogIProdukteve()
	{
		// shenime shembuj
		ArtikullID id1 = new ArtikullID( 100 );
		ArtikullID id2 = new ArtikullID( 200 );
		Money cmimi = new Money( 3 );

		PershkrimIProduktit persh;
		persh = new PershkrimIProduktit( id1, cmimi, "produkti 1" );
		pershkrimet.put( id1, persh );
		persh = new PershkrimIProduktit( id2, cmimi, "produkti 2" );
		pershkrimet.put( id2, persh );
	}

	public PershkrimIProduktit merrePershkriminEProduktit(ArtikullID id)
	{
		return pershkrimet.get( id );
	}
}


public class Regjistrues
{
	private KatalogIProdukteve katalogu;
	private Shitje shitjaAktuale;

	public Regjistrues( KatalogIProdukteve katalogu )
	{
		this.katalogu = katalogu;
	}

	public void kryejShitjen()
	{
		shitjaAktuale.behuEKryer();
	}
```

```
	public void futeArtikullin( ArtikullID id, int sasia )
	{
		PershkrimIProduktit persh =
			katalogu.merrePershkriminEProduktit( id );
		shitjaAktuale.bejArtikullRresht( persh, sasia );
	}

	public void bejShitjeTeRe()
	{
		shitjaAktuale = new Shitje();
	}

	public void bejePagesen( Money parateEOfruara )
	{
		shitjaAktuale.bejePagesen( parateEOfruara );
	}
}



public class PershkrimIProduktit
{
	private ArtikulliID id;
	private Money cmimi;
	private String pershkrimi;

	public PershkrimIProduktit
		( ArtikullID id, Money cmimi, String pershkrimi )
	{
		this.id = id;
		this.cmimi = cmimi;
		this.pershkrimi = pershkrimi;
	}

	public ArtikullID merreArtikullinID() { return id; }

	public Money merreCmimin() { return cmimi; }

	public String merrePershkrimin() { return pershkrimi; }
}


public class Shitje
{
	private List<ArtikullRreshtIShitjes> artikujtRreshta =
		new ArrayList()<ArtikullRreshtIShitjes>;
	private Date data = new Date();
	private eshteKryer = false;
	private Pagese pagesa;

	private Money merreBalansin()
	{
		return pagesa.merreSasine().minus( merreTotalin() );
	}

	public void behuEKryer() { eshteKryer = true; }

	public boolean eshteKryer() { return eshteKryer; }
```

```
	public void bejArtikullRresht
		( PershkrimIProduktit persh, int sasia )
	{
		artikujtRreshta.add(new ArtikullRreshtIShitjes(persh, sasia));
	}

	public Money merreTotalin()
	{
		Money totali = new Money();
		Money nentotali = null;

		for (ArtikullRreshtIShitjes artikulliRresht : artikujtRreshta)
		{
			nentotali = artikulliRresht.merreNentotalin();
			totali.add( nentotali );
		}
		return totali;
	}

	public void bejePagesen( Money parateEOfruara )
	{
		pagesa = new Pagese( parateEOfruara );
	}
}


public class ArtikullRreshtIShitjes
{
	private int sasia;
	private PershkrimIProduktit pershkrimi;

	public ArtikullRreshtIShitjes(PershkrimIProduktit persh, int sasia)
	{
		this.pershkrimi = persh;
		this.sasia = sasia;
	}

	public Money merreNentotalin()
	{
		return pershkrimi.merreCmimin().times( sasia );
	}
}

public class Shitore
{
	private KatalogIProdukteve katalogu = new KatalogIProdukteve();
	private Regjistrues regjistruesi = new Regjistrues( katalogu );

	public Regjistrues merreRegjistruesin() { return regjistruesi; }
}
```

## Implementim në kod – Hyrje në zgjidhjen Monopoly

(kapitulli 20)

```
package com.foo.monopoly.domen;

public class Fushe
{
	private String emri;
	private Fushe fushaTjeter;
	private int indeksi;

	public Fusha( String emri, int indeksi )
	{
		this.emri = emri;
		this.indeksi = indeksi;
	}

	public void caktoFushenTjeter(Fushe f)
	{
		this.fushaTjeter = f;
	}

	public Fushe merreFushenTjeter()
	{
		return fushaTjeter;
	}

	public String merreEmrin()
	{
		return emri;
	}

	public int merreIndeksin()
	{
		return indeksi;
	}
}


public class Figure
{
	private Fushe lokacioni;

	public Figure( Fushe lokacioni )
	{	this.lokacioni = lokacioni; }

	public Fushe merreLokacionin()
	{
		return lokacioni;
	}
}

public class Zar
{
	public static final int MAX = 6;
	private int vleraEFytyres;
```

```
	public Zar()
	{
		sillu();
	}

	public void sillu()
	{
		vleraEFytyres = (int) ( ( Math.random( ) * MAX ) + 1 );
	}

	public int merreVlerenEFytyres( )
	{
		return vleraEFytyres;
	}
}



public class Tabele
{
	private static final int MADHESIA = 40;
	private List fushat = new ArrayList(MADHESIA);

	public Tabela()
	{
		ndertoFushat();
		lidhiFushat();
	}

	public Fushe merreFushen(Fushe fillimi, int distanca)
	{
		int indeksiIFundit =
			(fillimi.merreIndeksin() + distanca) % MADHESIA;

		return (Fushe) fushat.get(indeksiIFundit);
	}

	public Fushe merreFushenFillestare()
	{
		return (Fushe) fushat.get(0);
	}

	private void ndertoFushat()
	{
		for (int i = 1; i <= MADHESIA; i++)
		{
			nderto(i);
		}
	}

	private void nderto(int i)
	{
		Fushe s = new Fushe("Fusha " + i, i - 1);
		fushat.add(s);
	}
```

```
	private void lidhiFushat()
	{
		for (int i = 0; i < ( MADHESIA - 1); i++ )
		{
			lidhe(i);
		}

		Fushe i_pari = (Fushe) fushat.get(0);
		Fushe i_fundit = (Fushe) fushat.get(MADHESIA - 1);
		i_fundit.caktoFushenTjeter(i_pari);
	}

	private void lidhe(int i)
	{
		Fushe aktualja = (Fushe) fushat.get(i);
		Fushe tjetra = (Fushe) fushat.get(i + 1);
		aktualja.caktoFushenTjeter(tjetra);
	}
}


public class Lojtar
{
	private String emri;
	private Figure figura;
	private Tabele tabela;
	private Zar[] zaret;

	public Lojtar(String emri, Zar[] zaret, Tabele tabela)
	{
		this.emri = emri;
		this.zari = zari;
		this.tabela = tabela;
		figura = new Figure(tabela.merreFushenFillestare());
	}

	public void merreRendin()
	{
		// sille zarin
		int totaliISilljeve = 0;
		for (int i = 0; i < zaret.length; i++)
		{
			zari[i].sillu();
			totaliISilljeve += zari[i].merreVlerenEFytyres();
		}

		Fushe lokIRi = tabela.merreFushen(
			figura.merreLokacionin, totaliISilljeve);
		figura.caktoLokacionin(lokIRi);
	}

	public Fushe merreLokacionin()
	{
		return figura.merreLokacionin();
	}
```

```
	public String merreEmrin()
	{
		return emri;
	}
}


public class LojaMonopol
{
	private static final int TOTALI_I_RUNDEVE = 20;
	private static final int TOTALI_I_LOJTAREVE = 2;
	private List lojtaret = new ArrayList( TOTALI_I_LOJTAREVE );
	private Tabele tabela = new Tabele( );
	private Zar[] zaret = { new Zar(), new Zar() };

	public LojaMonopol( )
	{
		Lojtar l;
		l = new Lojtar("Kale", zaret, tabela );
		lojtaret.add( l );
		l = new Lojtar("Kerr", zaret, tabela );
		lojtaret.add( l );
	}

	public void luajeLojen( )
	{
		for ( int i = 0; i < TOTALI_I_RUNDEVE; i++ )
		{
			luajeRundin();
		}
	}

	public List merriLojtaret( )
	{
		return lojtaret;
	}

	private void luajeRundin( )
	{
		for (Iterator iter = lojtaret.iterator( ); iter.hasNext( ); )
		{
			Lojtar lojtari = (Lojtar) iter.next();
			lojtari.merreRendin();
		}
	}
}
```

# Fjalor i publikimit

**Collaboration diagram** - diagram i bashkëpunimit (kolaborues) - diagram që i tregon ndërveprimet e organizuara rreth strukturës së një modeli, duke i përdorë ose klasifikuesit dhe asociacionet, ose instancat dhe lidhjet. Për dallim nga diagrami sekuencial, një diagram kolaborues i tregon relacionet ndërmjet instancave. Diagramet sekuenciale dhe kolaboruese (të bashkëpunimit) shprehin informata të ngjashme, por i shfaqin ato në mënyra të ndryshme. Shih SSD.

**CRC card** - Class-Responsibility-Collaborator card - kartelat Përgjegjësitë dhe Bashkëpunëtorët e Klasës janë vegël e analizës që përdoret në dizajnin e softuerit të orientuar kah objektet. Ato janë propozu nga Ward Cunningham and Kent Beck. Ato zakonisht përdoren gjatë përcaktimit të parë se cilat klasa nevojiten dhe si do të veprojnë ato ndërmjet vete.

Kartelat CRC zakonisht krijohen nga kartelat indeks në të cilat shkruhen:

1. Emri i klasës
2. Klasat e saj Mbi dhe Nën (nëse aplikohet)
3. Përgjegjësitë e klasës
4. Emrat e klasëve tjera me të cilat do të bashkëpunojë klasa për m'i plotësu përgjegjësitë e veta.
5. Autori

**diagram i bashkëpunimit (kolaborues)** - shih collaboration diagram

**diagrami sekuencial** - shih SSD

**DCD - Design Class Diagram** - Diagrami i Dizajnit të Klasave i tregon definicionet e klasave të softuerit. Ai bazohet në diagramin e bashkëpunimit (collaboration diagram). Dukshmëria e atributeve tregohet për lidhjet e përhershme. Klasat tregohen të listuara me atributet dhe metodat e veta të thjeshta.

**GRASP** - General Responsibility Assignment Software Patterns (ose nganjëherë Principles) - Paternat (Modelet, ose nganjëherë Principet) e Përgjithshme Softuerike të Ndarjes së Përgjegjësive. Përdoret në dizajnin e orientuar kah objektet, dhe jep udhëzime për m'iu nda përgjegjësi klasave dhe objekteve.

**GUI** - Graphical User Interface - Ndërfaqja Grafike e Përdoruesit

**GoF**- Gang of Four - Banda e Katërshes

**JDBC** - Java DataBase Connectivity - Lidhshmëria e Javës me Baza të Shënimeve - interfejs i programimit që i lejon aplikacionet Java m'iu qasë një baze përmes gjuhës SQL. Meqë interpretuesit e Javës (Makinat Virtuale të Javës, JVM) janë në dispozicion për të gjitha platformat e mëdha klient, kjo lejon m'u shkru aplikacion me baza i pavarur nga platforma. Më 1996, JDBC ka qenë zgjerimi i parë në platformën Java. JDBC është homologu Java i ODBC të Microsoft.

**LRG** - Low Representational Gap - Zbraztësia e Ulët e Reprezentimit - Përdor klasa të softuerit që inspirohen nga domeni për me përmirësu kuptueshmërinë, duke përdorë emra dhe terma nga domeni i problemit.

**Modeli i Dizajnit** - është model i objekteve që e përshkruan realizimin e rasteve të përdorimit, dhe shërben si abstraksion i modelit të implementimit dhe i kodit të tij burimor. Modeli i dizajnit përdoret si e dhënë hyrëse esenciale për aktivitetetet në implementim dhe testim.

**Modeli i Domenit** - i përmbledh tipet më të rëndësishme të objekteve në kontekst të domenit. Objektet e domenit i përfaqësojnë entitetet që ekzistojnëa po ngjarjet që ndodhin në ambientin në të cilin punon sistemi. Modeli i domenit është nënpjesë e modelit të objekteve të biznesit.

**OO** - Object Oriented - i Orientuar kah Objektet. Shih OOA/D

**OOA** - Shih OOA/D

**OOA/D** - Object Oriented Analysis and Design - Analiza dhe Dizajni i Orientuar kah Objektet. OOAD është qasje e inxhinierimit të softuerit që e modelon një sistem si grup të objekteve ndërvepruese. Secili objekt e përfaqëson një entitet të interesit në sistemin që modelohet, dhe karakterizohet nga klasa e vet, gjendja e vet (elementet shënime), dhe sjellja e vet. Mund të krijohen modele të ndryshme për me tregu strukturën statike, sjelljen dinamike, dhe shpërndarjen gjatë ekzekutimit të këtyre objekteve bashkëpunuese. Ka disa notacione të ndryshme për m'i paraqitë këto modele, siç është UML.

- Analiza e orientuar kah objektet (OOA) shikon në domenin e problemit, me qëllim për me prodhu një model konceptual të informatave që ekzistojnë në zonën që po analizohet. Modelet e analizës nuk i marrin në konsiderim kufizimet e implementimit që munden me ekzistu, si konkurenca, shpërndarja, persistenca, apo se si do të ndërtohet sistemi. Kufizimet e implementimit trajtohen gjatë dizajnit të orientuar kah objektet (OOD). Analiza bëhet para Dizajnit.
- Dizajni i orientuar kah objektet (OOD) e transformon modelin konceptual të prodhuar në analizën e orientuar kah objektet për m'i trajtu kufizimet që imponohen nga arkitektura e zgjedhur dhe çfarëto kufizime jo-funksionale - teknologjike apo ambientore, siç është gjerësia e brezit të transaksioneve, koha e përgjigjjes, platforma e ekzekutimit, ambienti i zhvillimit apo gjuha programuese.

**OOD** - Object Oriented Design - Dizajni i Orientuar kah Objektet - Shih OOA/D

**POS** - Point of Sale. Pikë e shitjes. Një biznes apo një vend ku mund të blehet një produkt apo një shërbim.

**POSA** - Pattern- Oriented Software Architecture - Arkitekturë e Softuerit e Orientuar kah Paternat

**RDB** - Relational Database - Bazë relacionale e shënimeve

**RDD** - Responsibility-Driven Design, Dizajn i Udhëhequr nga Përgjegjësitë

**RPC** - Remote Procedure Call - Thirrje e procedurës nga distanca

**Sequence Diagram** - shih SSD

**SQL** - Structured Query Language - Gjuhë e Strukturuar për Kërkesa (Pyetësorë)

**SSD** - System Sequence Diagram - Diagram Sekuencial i Sistemit - Diagram që i tregon ndërveprimet e objekteve të aranzhuara në sekuencë kohore. Në veçandi, i tregon objektet që marrin pjesë në ndërveprim dhe renditjen (sekuencën) e mesazheve të shkëmbyera. Për dallim nga diagrami i bashkëpunimit (collaboration diagram), diagrami sekuencial i përfshin sekuencat e kohës por nuk i përfshin relacionet e objekteve. Një diagram sekuencial mund të ekzistojë në formë gjenerike (i përshkruan të gjithë skenarët e mundshëm) dhe në formë instancë (e përshkruan një skenar aktual). Diagramet sekuenciale dhe kolaboruese (të bashkëpunimit) shprehin informata të ngjashme, por i shfaqin ato në mënyra të ndryshme. Shih collaboration diagram.

**UI** - User Interface - Ndërfaqe e Përdoruesit

**UML** - Unified Modeling Language - Gjuha e Unifikuar e Modelimit

**UP** - Unified Process - Procesi i Unifikuar

# Përmbledhje e GRASP

**General Responsibility Assignment Software Patterns or Principles**

**Paternat apo Principet e Përgjithshme e Përgjithshme Softuerike të Ndarjes së Përgjegjësive**

| Paterni/Principi | Përshkrimi |
|---|---|
| **Eksperti i Informatave** | Një princip i përgjithshëm i dizajnit të objekteve dhe i ndarjes së përgjegjësive? |
| | Ndaja përgjegjësinë ekspertit të informatave – klasës që i ka informatat e nevojshme me plotësu përgjegjësinë. |
| **Krijuesi** | Kush krijon? (Vëreni se Fabrika është zgjidhje e shpeshtë alternative.)<br><br>Ngaja klasës B përgjegjësinë për me kriju një instancë të klasës A nëse njëra nga këto është e vërtetë: |
| | 1. B e përmban A | 4. B e regjistron A |
| | 2. B e agregon A | 5. B e përdor ngushtësisht A |
| | 3. B i ka shënimet inicializuese për A |
| **Kontrolluesi** | Cili objekt i pari përtej shtresës UI e pranon dhe e koordinon ("e kontrollon") një operacion të sistemit?<br><br>Ngaja përgjegjësinë një objekti që e përfaqëson njërin nga këto zgjedhje:<br><br>1. E përfaqëson "sistemin" e përgjithshëm, një "objekt rrënjë", një pajisje brenda të cilës ekzekutohet softueri, apo një nënsistem madhor (këto janë të gjitha variacione të një kontrolluesi fasadë).<br>2. E përfaqëson një skenar të rastit të përdorimit brenda të cilit ndodh operacioni i sistemit (kontrollues i rastit të përdorimit apo i sesionit) |
| **Çiftimi i Ulët (vlerësues)** | Si me zvogëlu ndikimin e ndryshimit?<br><br>Ndaji përgjegjësitë ashtu që çiftimi (i panevojshëm) të mbetet i vogël.<br><br>Përdore këtë princip për m'i vlerësu alternativat. |
| **Kohezioni i Lartë (vlerësues)** | Si m'i mbajtë objektet të fokusuara, të kuptueshme, dhe të menaxhueshme, dhe si efekt anësor, me përkrahë Çiftimin e Ulët?<br><br>Ndaji përgjegjësitë ashtu që kohezioni të mbetet i lartë. Përdore këtë për m'i vlerësu alternativat. |

| Paterni/Principi | Përshkrimi |
| --- | --- |
| **Polimorfizmi (Shumëformësia)** | Kush është përgjegjës kur sjellja ndryshon sipas tipit?<br><br>Kur alternativat apo sjelljet e afërta ndryshojnë sipas tipit (klasës), ndaja përgjegjësinë për sjelljen – duke i përdorë operacionet polimorfike – tipeve për të cilat ndryshon sjellja. |
| **Fabrikimi i Pastër** | Kush është përgjegjës kur je i pashpresë, dhe nuk don me shkelë kohezionin e lartë dhe çiftimin e ulët?<br><br>Ngaja një grup shumë koheziv të përgjegjësive një klase artificiale appo përshtatshme "të sjelljes" që nuk e përfaqëson një koncept të domenit të problemit – diçka të trilluar, për me përkrahë kohezionin e lartë, çiftimin e ulët, dhe ripërdorimin. |
| **Indireksioni** | Si m'i nda përgjegjësitë për me iu shmangë çiftimit të drejtpërdrejtë?<br><br>Ndaja përgjegjësinë një objekti të ndërmjetëm për me ndërmjetësu ndërmjet komponentave apo serviseve tjera, ashtu që ato nuk janë të çiftuara drejtpërdrejt. |
| **Variacionet e Mbrojtura** | Si me ua nda përgjegjësitë objekteve, nënsistemeve, dhe sistemeve ashtu që ndryshimet apo jostabiliteti në këto elemente të mos kenë ndikim të padëshiruar në elementet tjera?<br><br>Identifikoji pikat e ndryshimit apo jostabilitetit të parashikuar; ndaji përgjegjësitë për me kriju një "interfejs" stabil përreth tyre. |