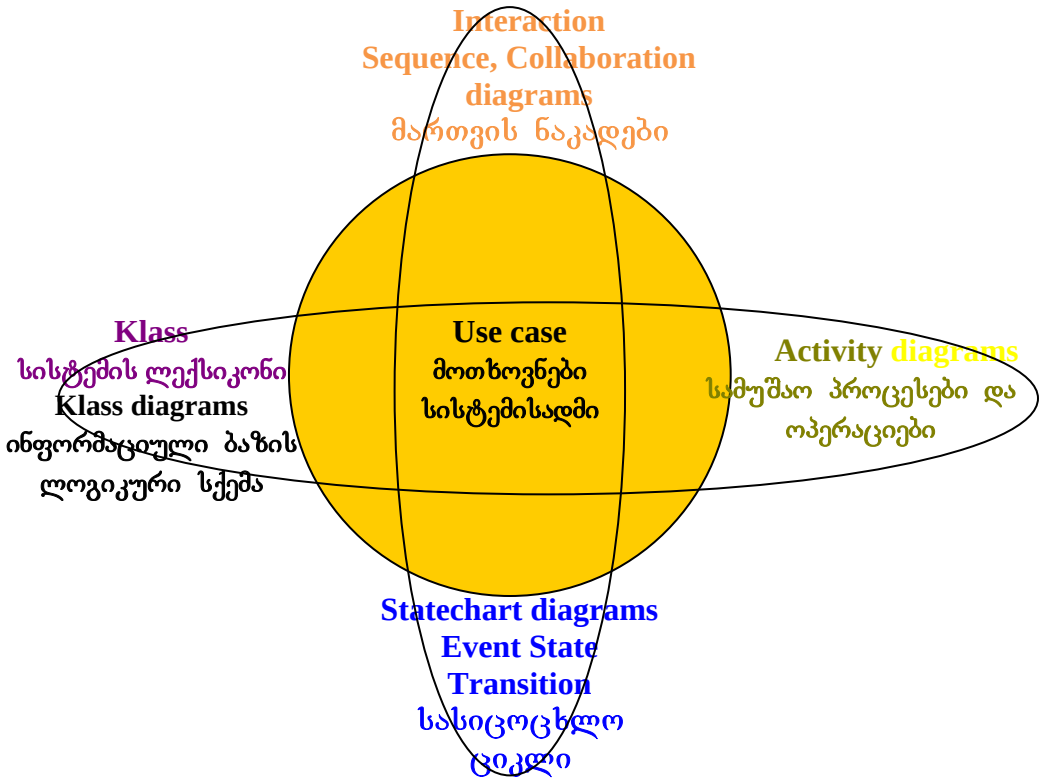
თ. სუხიაშვილი

მოდელირების უნიფიცირებული ენა(UML2) და პროგრამული სისტემის დამუშავების უნიფიცირებული პროცესი(UP)

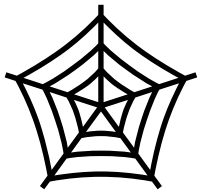


“ტექნიკური უნივერსიტეტი”

საქართველოს ტექნიკური უნივერსიტეტი

თეიმურაზ სუხიაშვილი

მოდელირების უნიფიცირებული ენა(UML2) და პროგრამული სისტემის დამუშავების უნიფიცირებული პროცესი(UP)



დამტკიცებულია სტუ-ს სამეცნიერო-ტექნიკური საბჭოს მიერ

თბილისი 2020

უაკ 681.3.06.

მოცემულია პროგრამული სისტემის დამუშავების მეთოდიკა ობიექტ – ორიენტირებული მიდგომით, მოდელირების უნივერსალური ენისა (UML2) და უნიფიცირებული პროცესის სინთეზის(UP) საფუძველზე.

წიგნი მოიცავს UML2-ს ზუსტ და ლაკონურ მიმოხილვას ობიექტ - ორიენტირებული ანალიზისა და დაპროექტების თვალთახედვით. მკაფიოდ და გასაგებათ დახასიათებულია UML2-ს პრაქტიკული გამოყენება უნიფიცირებული პროცესის ანალიზისა და დაპროექტების ეტაპებზე. ახსნილია მოდელირების მნიშვნელობა პროგრამული უზრუნველყოფის დამუშავების ციკლში, რაც დამპროექტებელს საშუალებას მისცემს განსაზღვროს როგორ და როდის გამოიყენონ UML2, რათა იპოვონ ოპტიმალური გადაწყვეტა პროექტისათვის.

განკუთვნილია ინფორმატიკისა და მართვის სისტემების ფაკულტეტის სტუდენტების, მაგისტრანტების, დოქტორანტების და სპეციალისტებისათვის.

რეცენზენტი: პროფ. გ. სურგულაძე

გამომცემლობა « ტექნიკური უნივერსიტეტი » 2020

ს ა რ ჩ ე ვ ი

თავი 1. პროგრამული სისტემის დამუშავება მოდელირების უნიფიცირებული ენის(UML2) გამოყენებით

თავი 2. სისტემისადმი მოთხოვნების დადგენა

თავი 3. ანალიზი

თავი 4. დაპროექტება

თავი 5. სისტემის რეალიზება

თავი 1
პროგრამული სისტემის დამუშავება მოდელირების უნიფიცირებული ენის(UML) გამოყენებით

ორგანიზაცია, რომელიც დაკავებულია პროგრამული უზრუნველყოფის წარმოებით იქნება წარმატებული მხოლოდ იმ შემთხვევაში, თუ მის მიერ გამოშვებული პროდუქცია გამოირჩევა მაღალი ხარისხით და დამუშავებულია მომხმარებლების მოთხოვნათა შესაბამისად. გარდა ამისა, იქნება სტაბილურად წარმატებული, თუ იგი ამას ახდენს ყველა არსებული ადამიანური და მატერიალური რესურსების მაქსიმალურად სრული და ეფექტური გამოყენების საფუძველზე.

ეფექტური პროგრამის დასამუშავებლად, რომელიც შეესაბამება თავის დასაშვებ დანიშნულებას, საჭიროა სისტემატურად შევხვდეთ და ვიმუშაოთ მომხმარებლებთან იმისათვის, რომ დავადგინოთ რეალური მოთხოვნები სისტემისადმი. ხარისხიანი პროგრამული უზრუნველყოფის დასამუშავებლად აუცილებელია მტკიცე არქიტექტურული საფუძვლის დამუშავება, რომელიც ღია იქნება ნებისმიერი შესაძლო სრულყოფისათვის. პროგრამული პროდუქტის დროული და ეფექტური დამუშავებისათვის უნდა მოვიწვიოთ სამუშაო ძალა, ამოვირჩიოთ სწორი ინსტრუმენტები და განვსაზღვროთ სწორი მიმართულება. დასახული ამოცანის გადასაწყვეტად საჭიროა, რომ დამუშავების პროცესი კარგათ იყოს მოფიქრებული და ადაპტირებული იყოს ცვალებად მოთხოვნებთან.

ცენტრალურ ელემენტს მოღვაწეობისა, რომელსაც მივყავართ მაღალი ხარისხის პროგრამული უზრუნველყოფის შექმნასთან, წარმოადგენს მოდელირება. მოდელი საშუალებას გვაძლევს მოვახდინოთ სისტემის სასურველი სტრუქტურის და ქცევის თვალნათლივ დემონსტრირება. ის ასევე აუცილებელია სისტემის არქიტექტურის ვიზუალიზაციისა და მართვისათვის. მოდელი გვეხმარება აგრეთვე შესაქმნელი სისტემის უკეთ გაგებაში და ხელმეორედ გამოყენების შესაძლებლობაში. ბოლოს მოდელი საჭიროა რისკების მინიმიზირებისათვის.

1. მოდელირების უნიფიცირებული ენა(UML)

ობიექტ - ორიენტირებული მიდგომა პროგრამული უზრუნველყოფის დასამუშავებლად ამჟამად ყველაზე მიღებულია უბრალოდ იმიტომ, რომ მან დაანახა თავისი სარგებლიანობა ნებისმიერი განზომილებისა და სირთულის სისტემების აგებისას სხვადასხვა სფეროებში.

გარდა ამისა თანამედროვე დაპროგრამების ენების უმრავლესობა, ინსტრუმენტალური საშუალებები და ოპერაციული სისტემები წარმოადგენენ ამა თუ იმ ზომით ობიექტ- ორიენტირებულებს.

თუ ჩვენ მივიღებთ ობიექტ- ორიენტირებულ მიდგომას, მაშინ უნდა განისაზღვროს როგორი სტრუქტურა უნდა გააჩნდეს პროგრამულ სისტემას ობიექტ- ორიენტირებული არქიტექტურიდან.

სისტემის გაგებისათვის აუცილებელია მოდელირება. ამასთან მარტო ერთი მოდელი არასდროს არ არის საკმარისი. ყოველი სისტემის გაგებისათვის, საჭიროა დიდი რაოდენობის ურთიერთ დაკავშირებული მოდელების დამუშავება. პროგრამულ სისტემებთან მიმართებაში ეს ნიშნავს, რომ საჭიროა ენა, რომლის მეშვეობითაც შესაძლებელი იქნება სხვადასხვა კუთხით ავღწეროთ სისტემის არქიტექტურა, მისი დამუშავების ციკლის განმავლობაში.

ამჟამად სტანდარტულ ინსტრუმენტს პროგრამული უზრუნველყოფის „მონახაზის" შესაქმნელად წარმოადგენს მოდელირების უნიფიცირებული ენა - Unified Modeling Language (**UML**). მისი მეშვეობით შესაძლებელია პროგრამული სისტემების არტეფაქტების ვიზუალირება, სპეციფიცირება, კონსტრუირება და დოკუმენტირება.

UML შეიძლება გამოვიყენოთ ნებისმიერი სისტემის მოდელირებისათვის დაწყებული ინფორმაციული სისტემებიდან დამთავრებული განზოგადებული **WEB-** წინადადებით. იგი ძალიან გამოხატული ენაა. რომელიც საშუალებას იძლევა განვიხილოთ სისტემა ყველა მხრიდან. მიუხედავად გამოხატვის ფართო შესაძლებლობისა ეს ენა ადვილი გასაგები და გამოსაყენებელია.

ენა შესდგება ანბანისა და წესებისაგან, რომელთა მეშვეობით შესაძლებელია მასში შემავალი სიტყვების კომბინირება და მივიღოთ აზრობრივი კონსტრუქციები.

UML არ არის ვიზუალური დაპროგრამების ენა, მაგრამ მისი მეშვეობით შექმნილი მოდელები, შეიძლება უშუალოდ გადაყვანილ იქნას დაპროგრამების სხვადასხვა ენებზე. სხვა სიტყვებით რომ ვთქვათ **UML** მოდელი შეიძლება გამოვსახოთ ისეთ ენებზე როგორიც არის **JAVA, C++, Visual Basik** და რელაციურ მონაცემთა ბაზის ცხრილებზე ან ობიქტ – ორიენტირებული მონაცემთა ბაზის მდგრად ობიექტებზე. ის მცნებები, რომლებიც უფრო ხელსაყრელია გადავცეთ გრაფიკულად, ასეთივე სახით წარმოიდგინებიან **UML**-ში, ხოლო ის მცნებები, რომლებიც უფრო ხელსაყრელია ავღწეროთ ტექსტური სახით, გამოისახებიან დაპროგრამების ენების მეშვეობით.

მოდელის ასეთი ასახვა დაპროგრამების ენებზე საშუალებას გვაძლევს განვახორციელოთ პირდაპირი პროექტირება - **UML** მოდელიდან კონკრეტული ენის

კოდების გენერაცია. შეიძლება გადაწყდეს უკუამოცანაც- მოვახდინოთ მოდელის რეკონსტრუირება არსებული რეალიზაციიდან.

დაპროგრამების ენებზე პირდაპირ ასახვასთან ერთად **UML**-ის გამომხატველობისა და ერთმნიშვნელობის მეშვეობით საშუალება გვეძლევა შევასრულოთ მოდელი, მოვახდინოთ სისტემის ქცევის იმიტირება და ვაკონტროლოთ არსებული სისტემები.

UML ენა განსაზღვრულია პირველ რიგში პროგრამული სისტემების დასამუშავებლად სხვადასხვა სფეროში. მაგრამ მისი გამოყენების სფერო არ შემოიზღუდება მხოლოდ პროგრამული უზრუნველყოფის მოდელირებით. მისი გამომხატველობა საშუალებას გვაძლევს მოვახდინოთ იურიდიულ სისტემებში დოკუმენტბრუნვის, სხვადასხვა მომსახურების სისტემების სტრუქტურისა და ფუნქციონირების მოდელირებისათვის, განახორციელოს აპარატული საშუალებების პროექტირება.

UML-ის შეცნობა იწყება სტრუქტურის განხილვით. ეს სტრუქტურა მოიცავს:

- **სამშენებლო ბლოკებს** – ძირითადი ელემენტები, UML მოდელის დიაგრამები და მიმართებები;
- **საერთო მექანიზმები** - გარკვეული მიზნების მიღწევის საერთო **UML**-გზები**.**
- **არქიტექტურა** – სისტემის არქიტექტურის **UML** წარმოდგენა.

1.1. UML-ის სამშენებლო ბლოკები

UML-ის ანბანი მოიცავს სამ სამშენებლო ბლოკს:

- **არსები** - აბსტრაქციები, რომლებიც წარმოადგენენ მოდელის ძირითად ელემენტებს;
- **მიმართებები -** აკავშირებენ სხვადასხვა არსებს:
- **დიაგრამები** - ახდენენ ჩვენთვის საინტერესო არსთა ერთობლიობის დაჯგუფებას..

<u>არსები.</u> საპრობლემო სფეროს ასახვისათვის **UML**-ში ძირითადათ გამოიყენება სტრუქტურული, ქცევის, დაჯგუფებისა და ანოტაციური არსები, რომლებიც წარმოადგენენ ენის ძირითად ობიექტ - ორიენტირებულ ბლოკებს. მათი მეშვეობით შესაძლებელია შევქმნათ კორექტული მოდელები.

სტრუქტურული არსები ეს **UML** ენაზე მოდელში არსებითი სახელებია. როგორც წესი, ისინი წარმოადგენენ მოდელის სტატიკურ ნაწილებს, რომლებიც შეესაბამებიან სისტემის კონცეპტუალურ და ფიზიკურ ელემენტებს.

არსებობს სტრუქტურული არსების შვიდი ნაირსახეობა:

- კლასი (**CLASS**) – ეს ობიექტთა ერთობლიობის აღწერაა საერთო ატრიბუტებით, დამოკიდებულებებით და სემანტიკით.

- ინტერფეისი (**Interface**) – ეს ოპერაციების ერთობლიობაა, რომლებიც განაპირობებენ მომსახურების ერთობლიობას, რომელსაც წარმოადგენს კლასი ან კომპონენტი. ინტერფეისს შეუძლია წარმოადგინოს კლასის ან კომპონენტის ყოფაქცევა მთლიანად ან ნაწილობრივ. იგი განსაზღვრავს ოპერაციის მხოლოდ სპეციფიკაციებს(სიგნატურას), მაგრამ არასდროს მათ რეალიზაციებს.
- კოოპერაცია (**Collaboracion**) განსაზღვრავს ურთიერთქმედებას, იგი წარმოადგენს როლებისა და სხვა ელემენტების ერთობლიობას, რომლებიც მუშაობენ რა ერთად ქმნიან გარკვეულ კოოპერაციულ ეფექტს. იგი არ დაიყვანება ელემენტთა უბრალო ჯამამდე. კოოპერაციას აქვს როგორც სტრუქტურული, ასევე ქცევითი ასპექტი, ერთი და იგივე კლასი შეიძლება მონაწილეობდეს რამოდენიმე კოოპერაციაში. მაშასადამე, ისინი წარმოადგენენ ქცევის სახეობების რეალიზაციას.
- პრეცენდენტი (**Use case**) - ეს სისტემის მიერ შესასრულებელი მოქმედებათა თანმიმდევრობის აღწერაა, რომელიც ქმნის ხილულ შედეგს და რომელიც მნიშვნელოვანია რომელიმე განსაზღვრული აქტიორისათვის(**Aktor**), პრეცენდენტი გამოიყენება ქცევის არსებების სტრუქტურირებისათვის. პრეცენდენტები რეალიზდებიან კოოპერაციის მეშვეობით. სამი შემდეგი არსება აქტიური კლასები, კომპონენტები და კვანძები კლასების მსგავსია, ისინი აღწერენ ობიექტების ერთობლიობას საერთო ატრიბუტებით, ოპერაციებით, დამოკიდებულებებით და სემანტიკით.
- აქტიური კლასი (**Active class**) უწოდებენ კლასს, რომლის ობიექტები ჩართულნი არიან ერთ ან რამოდენიმე პროცესში, ამიტომ შეიძლება მოახდინონ მმართველი ზემოქმედების ინიციირება.
- კომპონენტი (**Component**) სისტემის ფიზიკური ნაწილია, რომელიც შეესაბამება ინტერფეისების გარკვეულ ნაკრებს და უზრუნველყოფს მის რეალიზაციას(მაგ. ფაილები, ბიბლიოთეკები, გვერდები, ცხრილები). კომპონენტი წარმოადგენს ლოგიკური ელემენტების ფიზიკურ წარმოდგენას, როგორიც არის კლასები, ინტერფეისები და კოოპერაციები.
- კვანძი (**Node**) - ეს არის რეალური (ფიზიკური) სისტემის ელემენტი, რომელიც არსებობს პროგრამული კომპლექსის ფუნქციონირებისას და წარმოადგენს გამოთვლით რესურს, გარკვეული მეხსიერებით და დამუშავების შესაძლებლობით. კომპონენტების ერთობლიობა შეიძლება განლაგდეს კვანძში, ასევე გადაადგილდეს ერთი კვანძიდან მეორეში.

ქცევითი არსები წარმოადგენენ **UML** მოდელის დინამიურ შემადგენელს. ეს ენის ზმნებია, ისინი აღწერენ მოდელის ქცევას დროში და სივრცეში. ქცევითი არსებია:

- ურთიერთქმედება (**Interaction**) – ეს არის ქცევა, რომლის არსი მდგომარეობს შეტყობინებების გაცვლაში ობიექტებს შორის კონკრეტული კონტექსის ფარგლებში გარკვეული მიზნების მისაღწევად. ურთიერთქმედების საშუალებით შესაძლებელია ავღწეროთ როგორც ცალკეული ოპერაცია, ისე ობიექტთა ერთობლიობის ქცევა.
- ავტომატი (**State machine**) – ეს ქცევის ალგორითმია, რომელიც განსაზღვრავს მდგომარეობათა თანმიმდევრობას, რომლებზედაც ობიექტები ან ურთიერთქმედება გადადის მთელი სასიცოცხლო ციკლის განმავლობაში სხვადასხვა მოვლენების საპასუხოდ. ავტომატის მეშვეობით შესაძლებელია ავღწეროთ ცალკეული კლასის ან კლასის კოოპერაციის ქცევა. ავტომატთან დაკავშირებულია რიგი სხვა ელემენტებისა: მდგომარეობები, გადასვლები (ერთი მდგომარეობიდან მეორეში), მოვლენები (არსები, რომლებიც გადასვლების ინიციალიზაციას ახდენენ) მოქმედებათა სახეები (რეაქცია გადასვლაზე). სემანტიკურათ ქცევითი არსების ეს ორივე ელემენტი ხშირათ არიან დაკავშირებული სხვადასხვა სტრუქტურულ ელემენტთან, პირველ რიგში – კლასებთან, კოოპერაციებთან და ობიექტებთან.

  გარდა მოყვანილი არსებისა **UML** ენაში გამოიყენება:

  დაჯგუფების არსები – პაკეტები, მოდელის სემანტიკურად დაკავშირებული ელემენტების დაჯგუფებისათვის ერთიანი მთელი მოდულის შესაქმნელად. პაკეტი წარმოადგენს ელემენტების ჯგუფებში ორგანიზაციის უნივერსალურ მექანიზმს. პაკეტში შესაძლებელია მოვათავსოთ სტრუქტურული, ქცევითი და სხვა დაჯგუფების არსები.

  ანოტაციური არსები – შენიშვნები, რომლებიც შესაძლებელია დაემატოს მოდელს სპეციალური ინფორმაციის ჩასაწერად. ანოტაციური არსები ეს **UML** მოდელის განმარტებითი ნაწილია. მას მიეკუთვნება კომენტარიები დამატებითი აღწერისათვის, მოდელის ნებისმიერი ელემენტის განმარტების ან შენიშვნისათვის. ძირითადათ გამოიყენება მხოლოდ ერთი ანოტაციური ელემენტი – შენიშვნა(Note).

<u>მიმართებები.</u> **UML** ენაში განსაზღვრულია ოთხი ტიპის მიმართება:

- დამოკიდებულება;
- ასოციაცია;
- განზოგადება;
- რეალიზაცია.

ეს მიმართებები წარმოადგენენ ძირითად დამაკავშირებელ სამშენებლო ბლოკებს **UML**-ში და გამოიყენება კორექტული მოდელების შესაქმნელად.

დამოკიდებულება (Dependency) – ეს სემანტიკური კავშირია ორ არსებას შორის. რომლის დროსაც ერთერთის ცვლილებას, დამოუკიდებელის, შეიძლება გავლენა მოახდინოს მეორეს, დამოკიდებულის, სემანტიკაზე.

ასოციაცია (Association) - სტრუქტურული მიმართებაა, რომლებიც აღწერენ კავშირების ერთობლიობას. კავშირი – ეს ობიექტებს შორის შეერთებაა. ასოციაციის ნაირსახეობას წარმოადგენს აგრეგირება (ასე ეძახიან სტრუქტურულ მიმართებას მთელსა და მის ნაწილს შორის) და კომპოზიცია აგრეგირების უფრო მკაცრი(შეზღუდული) ფორმა.

განზოგადება (Generalization) – ეს არის მიმართება როდესაც ობიექტი(შვილობილი) შეიძლება წარმოვადგინოთ განზოგადებული ობიექტის(მშობელის) მაგიერ. მაშასადამე, შვილობილი მემკვიდრეობით ღებულობს თავისი მშობლის სტრუქტურასა და ქცევას.

რეალიზაცია (Realization) – ეს სემანტიკური მიმართებაა კლასიფიკატორებს შორის, რომლის დროსაც კლასიფიკატორი განსაზღვრავს კონტრაქტს, ხოლო მეორე გარანტიას იძლევა მის შესრულებაზე. რეალიზაციიც მიმასრთება გვხვდებაორ შემთხვევაში: პირველ რიგში ინტერფეისსა და მის მარეალიზებელ კლასებს ან კომპონენტებს შორის, მეორეს მხრივ, პრეცენდენტებსა და მათ მარეალიზებელ კოოპერაციებს შორის.

გარდა ამ ოთხი ტიპის მიმართებისა არსებობს აგრეთვე ვარიაციები მაგალითად, დაზუსტება(**Refinement**), ტრასსირება(**Trace**), ჩართვა(Include) და გაფართოება(Extent) - დამოკიდებულებისათვის.

<u>დიაგრამები.</u> დიაგრამა **UML**-ში ეს არის გრაფიკული წარმოდგენა ელემენტების ერთობლიობისა, რომელიც გამოისახება შეკრული გრაფით, რომლის წვეროებია არსები და წიბოები კი მიმართებები.

UML-ში გამოყოფენ ცხრა ტიპის დიაგრამებს, რომლებიც გამოიყენებიან სისტემის როგორც სტატიკური, ისე დინამიური ასპექტების ვიზუალირებისათვის:

კლასების დიაგრამაზე გამოისახება კლასები, ობიექტები, ინტერფეისები, კოოპერაციები და მათ შორის მიმართებები. ობიექტ- ორიენტირებული სისტემების მოდელირებისას კლასების დიაგრამები გამოიყენება ყველაზე ხშირად. პროექტირების თვალსაზრისით კლასების დიაგრამა შეესაბამება სისტემის სტატიკურ სახეს.

ობიექტების დიაგრამაზე წარმოიდგინება ობიექტები და მიმართებები მათ შორის. ობიექტების დიაგრამაც სისტემის სტატიკურ სახეს ასახავს.

პრეცენდენტების დიაგრამაზე წარმოდგენილია პრეცენდენტები, აქტიორები და მიმართებები მათ შორის. პრეცენდენტების გამოყენების თვალსაზრისით მოცემული დიაგრამაც ასახავს სისტემის სტატიკურ სახეს.

მიმდევრობისა და კომუნიკაციის დიაგრამები წარმოადგენენ ურთიერთქმედების დიაგრამების კერძო სახეს. მათზე წარმოიდგინება ობიექტებს შორის კავშირები და შეტყობინებები, რომლებითაც ხდება ინფორმაციის მიმოცვლა მოცემულ ობიექტებს შორის. ურთიერთქმედების დიაგრამები ასახავენ სისტემის დინამიკურ სახეს. ამასთან მიმდევრობის დიაგრამები გამოხატავენ შეტყობინებათა დროის მიხედვით მოწესრიგებას, ხოლო კოოპერაციის დიაგრამები – სტრუქტურულ ორგანიზაციას შეტყობინებათა მიმოცვლაში მონაწილე ობიექტებს შორის.

მდგომარეობათა დიაგრამაზე (Statechart diagrams) წარმოიდგინება ავტომატი, რომელიც მოიცავს მდგომარეობებს და გადასვლებს მათ შორის. ისინი ყურადღებას ამახვილებენ ობიექტის ქცევაზე, რომელიც დამოკიდებულია მოვლენათა თანმიმდევრობაზე. მდგომარეობათა დიაგრამები ასახავენ სისტემის დინამიკურ სახეს.

მოღვაწეობის დიაგრამაზე გამოისახება სისტემის შიგნით მართვის ნაკადის გადასვლები ერთი მოქმედებიდან მეორეზე. მოღვაწეობის დიაგრამები ასახავენ სისტემის დინამიკურ სახეს.

კომპონენტების დიაგრამაზე წარმოიდგინება კომპონენტების ორგანიზაცია და მათ შორის არსებული დამოკიდებულებები. რეალიზაციის თვალსაზრისით კომპონენტების დიაგრამა მიეკუთვნება სისტემის სტატიკურ სახეს.

განლაგების დიაგრამაზე წარმოიდგინება სისტემის დასამუშავებელი კვანძების კონფიგურაცია და მათში განლაგებული კომპონენტები.

დიაგრამები შესაძლებელია დაიყოს სისტემის სტატიკური სტრუქტურის ამსახველ და დინამიური სტრუქტურის ამსახველ დიაგრამებად. სტატიკური მოდელი აფიქსირებს არსებს და სტრუქტურულ მიმართებებს მათ შორის, ხოლო დინამიური მოდელი გამოსახავს, თუ როგორ ურთიერთქმედებენ არსები პროგრამული სისტემის საჭირო ქცევის გენერირებისათვის.

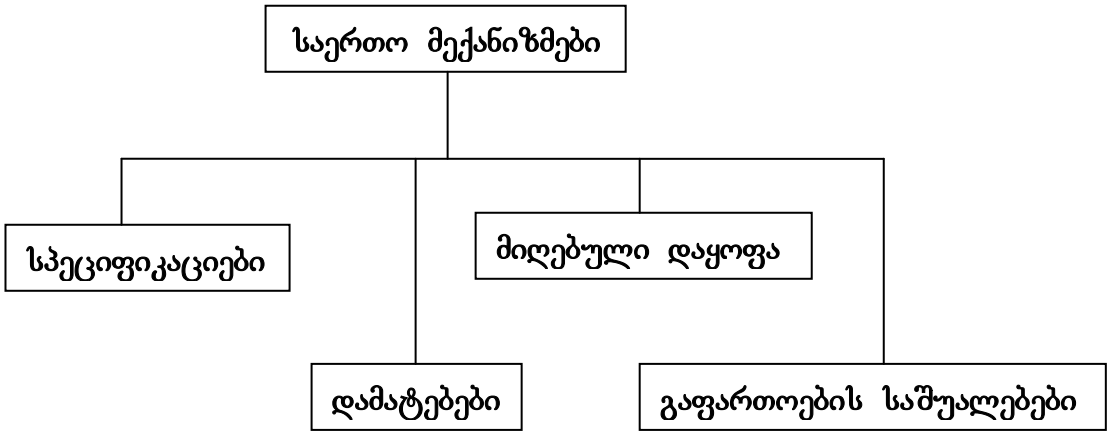
- დიაგრამების შექმნის განსაზღვრული თანმიმდევრობა არ არსებობს, თუმცა ჩვეულებრივ იწყებენ პრეცედენტების დიაგრამით სისტემის საპრობლემო სფეროს განსაზღვრისათვის. როგორც წესი, მუშაობა მიმდინარეობს ერთდროულად რამოდენიმე დიაგრამაზე, რომელთაგან თითოეული ზუსტდება დასამუშავებელი პროგრამული სისტემის შესახებ უფრო დეტალური ინფორმაციის გამოვლენის შესაბამისად. მაშასადამე დიგრამები წარმოადგენენ როგორც მოდელის წარმოდგენის, ისე მოდელში ინფორმაციის შეტანის ძირითად მექანიზმს.

1.2. UML-ის საერთო მექანიზმები

UML-ში არის ოთხი საერთო მექანიზმი, რომლებიც თანმიმდევრულად გამოიყენებიან მოდელირების მთელი ენისათვის. ისინი ასახავენ ობიექტთა მოდელირებისადმი მიდგომის ოთხ სტრატეგიას(ნახ.1.1.1.), რომლებიც **UML**-ში სხვადასხვა კონტექსტში მრავალჯერ გამოიყენებიან.

<u>სპეციფიკაციები.</u> სპეციფიკაციები – ეს **UML** მოდელის არსია. ისინი უზრუნველყოფენ მოდელის სემანტიკურ უკანა(დაფარულ) მხარეს.

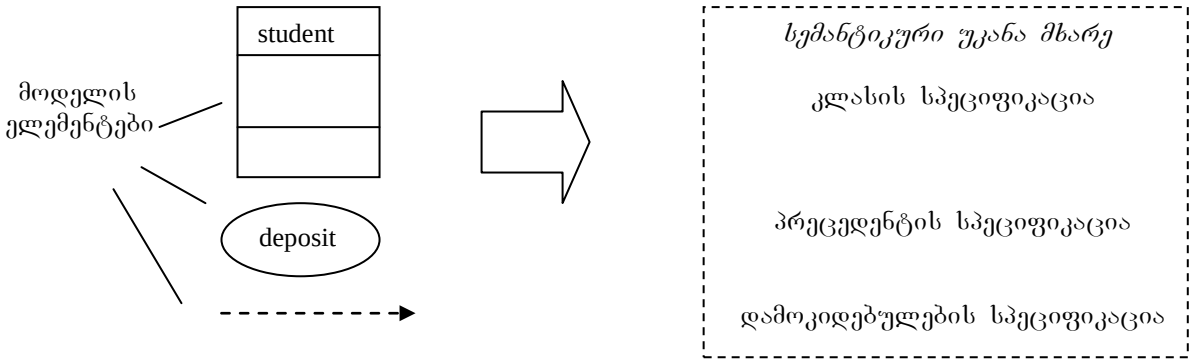
UML-ის მოდელებს აქვთ, უკიდურეს შემთხვევაში, ორი განზომილება: გრაფიკული, რომლებიც საშუალებას მოგვცემენ მოვახდინოთ მოდელის ვიზუალირება დიაგრამების მეშვეობით, და ტექსტური, რომელიც შესდგება მოდელის სხვადასხვა ელემენტების სპეციფიკაციისაგან(ნახ.1.1.2.). სპეციფიკაცია – ეს ელემენტთა სემანტიკის ტექსტური აღწერაა.



ნახ.1.1.1.

კლასი, მაგალითად შესაძლებელია გამოვსახოთ გრაფიკულად, მიუთითებთ რა ატრიბუტებსა და ოპერაციებს, მაგრამ ეს წარმოდგენა არაფერს არ გვამცნობს კლასის სემანტიკაზე. სწორედ სპეციფიკაციები ახდენენ სემანტიკის ფორმირებას, რაც აერთიანებს და აზრობრივად ავსებს მოდელს. მოდელის ელემენტების სემანტიკა ფიქსირდება სპეციფიკაციებში.

სპეციფიკაციების ნაკრები – ეს მოდელის არსია. სპეციფიკაციები ახდენენ *სემანტიკური უკანა გეგმის* ფორმირებას, რომელიც აერთიანებს მოდელს და ავსებს მას აზრობრივად. სხვადასხვა დიაგრამები – ეს მხოლოდ წარმოდგენა ან ამ გეგმის ვიზუალური პროექციაა. დიაგრამები უზრუნველყოფენ სემანტიკური უკანა მხარის გეგმის წარმოდგენას.



ნახ.1.1.2.

მოდელების დამუშავება **UML-ი**ს მეშვეობით იწყება გრაფიკული მოდელიდან, რომელიც საშუალებას გვაძლევს მოვახდინოთ სისტემის ვიზუალირება, ხოლო შემდეგ მისი განვითარების მიხედვით ამატებენ უკანა მხარეს, სულ უფრო და უფრო მეტ სემანტიკას. ამავე დროს მოდელი ითვლება სასარგებლოდ ან სრულად, თუ მოდელის სემანტიკა მოცემულია სემანტიკურ უკანა გეგმაში.

<u>დამატებები</u>. UML2-ში მოდელის ყოველი ელემენტი აღინიშნება უბრალო სიმბოლოთი, რომელსაც შესაძლებელია დავამატოთ, რიგი დამატებები, რომლებიც მოახდენენ სპეციფიკაციის ელემენტების ასპექტების ვიზუალირებას. ამ მექანიზმის მეშვეობით დიაგრამაზე ხილული ინფორმაცია შესაძლებელია წარმოვადგინოთ შესაბამისი კონკრეტული მოთხოვნებით.

დიაგრამაზე დამატებები მოდელის ელემენტებზე გამოიყენება იმისათვის, რომ ხაზი გაესვას მნიშვნელოვან დეტალებს.

დაწყებისას შესაძლებელია შეიქმნას მაღალდონიანი დიაგრამები, რომლებიც იყენებენ მხოლოდ ძირითად სიმბოლოებს ერთი ან ორი დამატებებით. დროთა განმავლობაში დიაგრამა ზუსტდება სულ უფრო მეტი და მეტი დამატებებით მანამადე, სანამ არ არ იქნება საკმარისად კონკრეტული-დეტალური.

მთავარია დავიმახსოვროთ, რომ ნებისმიერი დიაგრამა ეს მხოლოდ მოდელის წარმოდგენაა, ამოტომ აუცილებელია უჩვენოთ მხოლოდ ის დამატებები, რომლებიც ხაზს უსმევენ მოდელის მნიშვნელოვან მახასიათებლებს და ქმნიან მოდელს უფრო გასაგებს.

<u>მიღებული დაყოფა</u>. მიღებული დაყოფა აღწერს სამყაროს წარმოდგენის კონკრეტულ საშუალებებს. **UML-**ში არსებობს ორი მიღებული დაყოფა: კლასიფიკატორი/ეგზემპლიარი და ინტერფეისი/რეალიზაცია.

კლასიფიკატორი და ეგზემპლიარი. UML-ში იგულისხმება, რომ შესაძლებელია არსებობდეს აბსტრაქტული მცნება არსების ტიპის(მაგალითად, სტუდენტი) და ცალკეული კონკრეტული ეგზემპლიარები ამ აბსტრაქციისა(“ჩემი სპეციალობის სტუდენტი”).

აბსტრაქტული მცნება არსების ტიპის – ეს კლასიფიკატორია, ხოლო კონკრეტული არსები – ეგზემპლიარები.

UML-ში ეგზემპლიარი ჩვეულებრივ წარმოიდგინება იმავე პიქტოგრამით, როგორითაც შესაბამისი კლასიფიკატორი, მაგრამ ეგზემპლიარებისათვის დასახელება პიქტოგრამაზე ხაზგასმულია. **UML2**-ში სულ 33 კლასიფიკატორია (მაგალითად, აქტიორი, კლასი, კომპონენტი, ინტერფეისი, კვანძი, პრეცედენტი), რომლებიც განხილული იქნება შემდეგ თავებში.

ინტერფეისი და რეალიზაცია. ძირითადი იდეა ამ მცნებებისა იმაშია, რომ გამოვყოთ ის, რაც ასრულებს მოქმედებას(ინტერფეისი), იმისგან, როგორ კეთდება ეს(რეალიზაცია). მაგალითად, მანქანის მართვისას მძღოლი ურთიერთქმედებს მარტივ და მკაფიოდ განსაზღვრულ ინტერფეისთან. სხვადასხვა მანქანაზე ეს ინტერფეისი რეალიზებულია სხვადასხვანაირად. ინტერფეისი განსაზღვრავს კონტრაქტს, რომლის მხარდაჭერას უზრუნველყოფენ კონკრეტული რეალიზაციები.

<u>UML-ის გაფართოების მექანიზმები</u>. **UML** მოდელირების გაფართოებადი ენაა. დამმუშავებლებს ესმოდათ, რომ უბრალოდ შეუძლებელია შეიქმნას მთლიანად უნივერსალური მოდელირების ენა, რომელიც დააკმაყოფილებდა ყველა თანამედროვე მოთხოვნებს და კიდევ იმათ, რომლებიც შესაძლებელია წარმოიქმნას უახლოეს პერიოდში. ამიტომ UML შეიცავს გაფართოების სამ უბრალო მექანიზმს – შეზღუდვები, სტერეოტიპები და მონიშნული მნიშვნელობა.

შეზღუდვები – აფართოვებს ელემენტის სემანტიკას, უზრუნველყოფს შესაძლებლობით დაუმატოს ახალი წესები. შეზღუდვა ეს ტექსტის სტრიქონია, მოთავსებული ფიგურულ ფრჩხილებში({}), რომელიც განსაზღვრავს გარკვეულ პირობას ან წესს მოდელის ელემენტისათვის, რომელიც უნდა რჩებოდეს ჭეშმარიტი. განსაზღვრავს ობიექტური შეზღუდვების ენას (OCL), როგორც სტანდარტულ გაფართოებას.

სტერეოტიპები – უზრუნველყოფს შესაძლებლობით განისაზღვროს ახალი ელემენტები მოდელის არსებულების საფუძველზე. სტერეოტიპის სემანტიკას განვსაზღვრავთ ჩვენ დამოუკიდებლად.

სტერეოტიპი წარმოადგენს არსებული მოდელის ელემენტის სახეობას, რომელსაც აქვს იგივე ფორმა, მაგრამ სხვა დანიშნულება. შესაბამისად, სტერეოტიპები საშუალებას იძლევიან შევქმნათ მოდელის ახალი ელემენტები არსებულების საფუძველზე. ამისათვის ახალი ელემენტის სახელს ემატება სტერეოტიპის სახელი ფრჩხილებში. სტერეოტიპების რიცხვი ყოველი ელემენტის მოდელისათვის შესაძლებელია იცვლებოდეს ნულიდან გარკვეულ მნიშვნელობამდე.

რამდენადაც სტერეოტიპებს შემოაქვთ მოდელის ახალი ელემენტები სხვა დანიშნულებით, უნდა განსაზრვრული იყოს ამ ელემენტების სემანტიკა. ამისათვის მოდელების დამმუშავებელთა უმრავლესობა უთითებენ შენიშვნას ან მიმართვას გარე დოკუმენტზე, რომელშიც აღიწერება სტერეოტიპი.

სტერეოტიპები შესაძლებელია გამოვსახოთ დასახელებით მოთავსებული ფრჩხილებში ან პიქტოგრამით. მაგალითად, კლასი Ticket შესაძლებელია გამოისახოს:

მონიშნული მნიშვნელობა – წარმოგვიდგენს ელემენტის სპეციფიკაციის გაფართოების საშუალებას, უზრუნველყოფს შესაძლებლობით დაუმატოთ მასში ახალი სპეციალური ინფორმაცია.

UML-ში თვისება – ეს ნებისმიერი მნიშვნელობაა, რომელიც მიმაგრებულია მოდელის ელემენტს. ელემენტების უმრავლესობას აქვთ დიდი რაოდენობის წინასწარ განსაზღვრული თვისებები. ზოგიერთი მათგანი შესაძლებელია გამოისახოს დიაგრამაზე, სხვები წარმოადგენენ მოდელის უკანა მხარის ნაწილის სემანტიკურ ნაწილს.

მონიშნული მნიშვნელობების დახმარებით UML საშუალებას იძლევა დაუმატოთ მოდელის ელემენტებში საკუთარი თვისებები. მონიშნული მნიშვნელობა – ეს სამომხმარებლო სიტყვაა, რომელსაც შესაძლებელია მივამაგროთ მნიშვნელობა. მონიშნულ მნიშვნელობას აქვს შემდეგი სინტაქსისი:

{ნიშნული1 = მნიშვნელობა1, ნიშნული2 = მნიშვნელობა2, . . ნიშნულიN = მნიშვნელობაN}. მოყვანილ თანმიმდევრობას უწოდებენ ნიშნულების ცხრილს.

1.3. პროგრამული სისტემის არქიტექტურის UML წარმოდგენა

არქიტექტურა ეს არსებითი გადაწყვეტილებების ერთობლიობაა, რომელიც ეხება:

- პროგრამული სისტემის ორგანიზაციას;
- სტრუქტურული ელემენტების ამორჩევას, რომელიც შეადგენს სისტემას და მათ ინტერფეისებს;
- ამ ელემენტების ქცევას, რომელიც სპეციფიცირებულია სხვა ელემენტებთან კოოპერაციაში;

- ამ სტრუქტურული და ქცევითი ელემენტებისაგან სულ უფრო მსხვილი ქვესისტემების შექმნას;
- არქიტექტურულ სტილს, რომელიც წარმართავს და განსაზღვრავს სისტემის მთელ ორგანიზაციას: სტატიკურ და დინამიკურ ელემენტებს, მათ ინტერფეისებს, კოოპერაციებს და მათი გამოყენების საშუალებებს.

პროგრამული სისტემის არქიტექტურა მოიცავს არა მარტო მის სტრუქტურულ და ქცევით ასპექტებს, არამედ გამოყენებას, ფუნქციონირებას, წარმადობას, მოქნილობას, განმეორებითი გამოყენების შესახლებლობას, სისრულეს, ეკონომიურ და ტექნოლოგიურ შეზღუდვებს და კომპრომისებს, ასევე ესთეთიკურ საკითხებს.

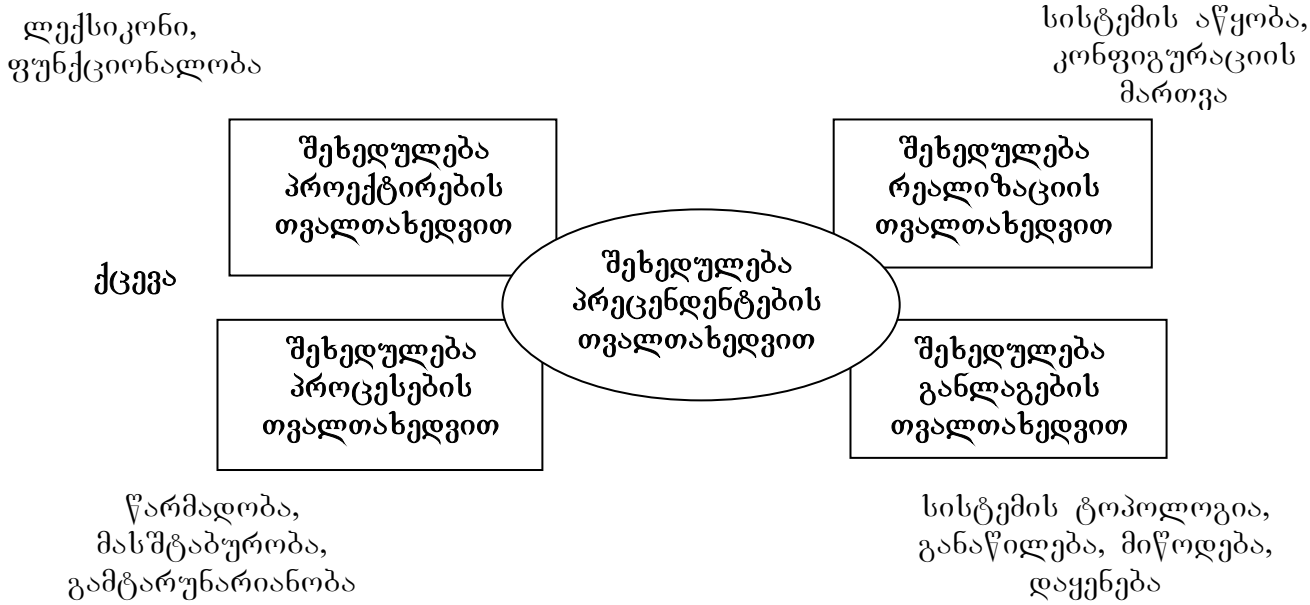
ყველას ვისაც კავშირი აქვს დასამუშავებელ სისტემასთან – მომხმარებლები, ანალიტიკოსები, დამმუშავებლები, მატესტირებლები, ტექნიკოსები და პროექტის მენეჯერები გააჩნიათ საკუთარი ინტერესები და ყოველი მათგანი უყურებს შესაქმნელ სისტემას თავისებურად მისი სასიცოცხლო ციკლის სხვადასხვა მომენტში. ეს მიგვანიშნებს იმ ფაქტზე, რომ სისტემის სრულყოფილი აღწერისა და შესწავლისათვის აუცილებელია საპრობლემო სფეროს განხილვა სხვადასხვა თვალსაზრისით.

პროგრამული სისტემის არქიტექტურა ყველაზე ოპტიმალურად შესაძლებელია აღწერილი იყოს ხუთი ურთიერთ დაკავშირებული სახით ან წარმოდგენით, თითოეული მათგანი წარმოადგენს სისტემის ორგანიზაციის და სტრუქტურის ერთერთ შესაძლო პროექციას და ყურადღებას ამახვილებს მისი ფუნქციონირების განსაზღვრულ ასპექტზე(ნახ.1.3.1.). ყოველი მათგანი გულისხმობს სტრუქტურულ და ქცევით მოდელირებას (სტატიკური და დინამიკური არსებების მოდელირება).

შეხედულება პრეცენდენტების თვალსაზრისით მოიცავს პრეცენდენტებს, რომლებიც აღწერენ სისტემის ქცევას, მომხმარებლის დაკვირვებით. ეს სახე არ წარმოგვიდგენს პროგრამის ჭეშმარიტ ორგანიზაციას, არამედ იმ მამოძრავებელ ძალებს, რომლებისგანაც დამოკიდებულია სისტემური არქიტექტურის ფორმირება. ***შეხედულება დაპროექტების თვალსაზრისით*** მოიცავს კლასებს, ინტერფეისებსა და კოოპერაციებს, რომლებიც ახდენენ ამოცანის და მისი გადაწყვეტის ლექსიკონის ფორმირებას. ეს შეხედულება პირველ რიგში აყენებს ფუნქციონალურ მოთხოვნებს, რომელიც წაეყენება სისტემას, ან იმ მომსახურებას, რომელიც მან უნდა წარუდგინოს მომხმარებლებს.

შეხედულება პროცესების თვალსაზრისით მოიცავს პროცესებსა და ძაფებს, რომლებიც ახდენენ პარალელიზმისა და სინქრონიზაციის ფორმირებას სისტემაში. ეს სახე უმთავრესად აღწერს სისტემის წარმადობას, მასშტაბურობას და გამტარუნარიანობას.

შეხედულება რეალიზაციის თვალსაზრისით მოიცავს კომპონენტებსა და ფაილებს, რომლებიც გამოიყენებიან საბოლაო პროგრამული პროდუქტის აწყობისა და გამოშვებისათვის. ეს სახე პირველ რიგში განკუთვნილია სისტემის კონფიგურაციის ვერსიების მართვისათვის, რომლებიც შედგენილია დამოუკიდებელი კომპონენტებისა და ფაილებისაგან.



ნახ.1.3.1.

შეხედულება განლაგების თვალსაზრისით მოიცავს კვანძებს, რომლებიც ახდენენ სისტემის აპარატული საშუალებების ტოპოლოგიის ფორმირებას, რომელზედაც იგი არის შესრულებული. პირველ რიგში იგი დაკავშირებულია სისტემის ფიზიკური შემადგენელი ნაწილების განაწილებასთან, შეკვეთასა და დაყენებასთან.

თითოეული ამ ჩამონათვალიდან, შეიძლება ჩაითვალოს სავსებით დამოუკიდებელ ასპექტად, ისე რომ, პირები რომლებსაც კავშირი აქვთ სისტემის დამუშავებასთან, შეუძლიათ ყურადღება გაამახვილონ არქიტექტურის მხოლოდ იმ ასპექტების შესწავლაზე, რომლებიც უშუალოდ მათ ეხებიან. მაგრამ არ უნდა დავივიწყოთ, რომ ეს სახეები ურთიერთქმედებენ ერთმანეთში.

ამრიგად, **სახე ან წარმოდგენა (View)** - ეს მოდელია, რომელიც განიხილება გარკვეული თვალსაზრისით: მათში გამოსახულია ერთი არსები და გამოტოვებულია სხვები, რომლებსაც მოცემული თვალსაზრისით ინტერესი არ გააჩნიათ.

იმისათვის, რომ გამოვსახოთ სისტემა რომელიმე თვალთახედვით გამოიყენება დიაგრამები. ამჟამად განსაზღვრულია ცხრა ტიპის დიაგრამა, რომელთა კომბინირება შესაძლებელია საჭირო წარმოდგენის მისაღებათ. სისტემის სტატიკური ნაწილების განხილვისას გამოიყენება კლასების, ობიექტების, კომპონენტების და განლაგების

დიაგრამები, ხოლო სისტემის დინამიურ ნაწილებთან სამუშაოდ გამოიყენება პრეცენდენტების, მიმდევრობის, კოოპერაციის, მდგომარეობის და მოღვაწეობის დიაგრამები.

2. პროგრამული სისტემის წარმოების უნიფიცირებული პროცესი

პროგრამული უზრუნველყოფის წარმოების პროცესი (Software engineering Process, SEP) ასევე ცნობილი როგორც პროგრამული უზრუნველყოფის დამუშავების პროცესი (Software Development Process), განსაზღვრავს ვინ, რა, როდის და როგორ პროგრამული უზრუნველყოფის(პუ) დამუშავებაში. SEP - ეს არის პროცესი, რომელშიც მომხმარებლის მოთხოვნები გარდაიქმნებიან პროგრამულ უზრუნველყოფაში და მას უწოდებენ უნიფიცირებულ პროცესს – UP.

ამჟამად UML-ი პროექტის ხილული ნაწილია, ხოლო UP – პროცესია, UML-ი სტანდარტიზირებულია, ხოლო UP არა. ამიტომ არ არსებობს სტანდარტული SEP UML-ისთვის.

UP ეფუძნება იმ პროცესების გამოკვლევებს, რომლებსაც აწარმოებენ Ericsson-ის, Rational- ის და სხვა წამყვანი კომპანიები. შესაბამისად, უნიფიცირებულ პროცესი, რომელიც დამუშავებული იქნა კომპანია Rational- ის მიერ (RUP) ეს კომერციული პროდუქტია, რომელიც აფართოვებს UP-ს. მიუხედავად ამისა, ყოველი ახალი პროექტის დამუშავებისას იგი უნდა მიუსადაგოდ გარკვეულ ორგანიზაციებს და კონკრეტულ პროექტს. რაც იმას ნიშნავს, რომ პროგრამული უზრუნველყოფის დამუშავების პროექტები განსხვავდებიან და მათი გაერთიანება „ერთი ზომით“ პროგრამული უზრუნველყოფის წარმოების პროცესის (SEP) შემთხვევაში არ მუშაობს.

საერთოდ კი შეიძლება ითქვას, რომ UP ბაზირდება სამ აქსიომაზე:

- მოთხოვნებითა და რისკით მართვადობა;
- არქიტექტურულ-ცენტრისტული;
- იტერაციული და ინკრემენტული.

ცხადია, მომხმარებელთა **მოთხოვნები,** რომლებსაც ვაყალიბებთ პრეცედენტების სახით, უნდა გათვალისწინებული იქნას დამუშავების მთელ პროცესში. მასთან ერთად მმართველი მექანიზმი უნდა იყოს – რისკი. ხარისხის კონტროლი წარმოადგენს პროცესის განუყოფელ ნაწილს, მოიცავს ყველა სახის და მონაწილის სამუშაოებს. ამასთან გამოიყენება შედარების ობიექტური კრიტერიუმები და მეთოდები. რისკის მართვა ჩართულია პროცესში, ისე რომ შესაძლო დაბრკოლებები პროექტის წარმატებით

დასრულების გზაზე გამოვლინდება და გამოირიცხება ადრეულ ეტაპებზე, როდესაც რეაგირებისათვის საკმარისი დროა. შესაბამისად, პრეცედენტები წარმოადგენენ ძირითად არტეფაქტს, რომლის საფუძველზეც დგინდება სისტემის სასურველი ქცევა, მოწმდება და დასტურდება არჩეული არქიტექტურის სისწორე, ხდება ტესტირება და ხორციელდება ურთიერთქმედება პროექტის მონაწილეებს შორის.

პროცესს უწოდებენ **არქიტექტურაზე დაფუძნებულს**(Architecture-centric), როდესაც სისტემური არქიტექტურა წარმოადგენს გადამწყვეტ ფაქტორს კონცეფციის დამუშავებისას, შესაქმნელი სისტემის კონსტრუირების, მართვისა და განვითარებისას. UP ითვალისწინებს სისტემის საიმედო არქიტექტურის შექმნას, რამდენადაც არქიტექტურა აღწერს სისტემის კომპონენტებათ დაყოფის სტრატეგიულ ასპექტებს. ამავე დროს ამ კომპონენტების ურთიერთქმედებას და განლაგებას აპარატულ მოწყობილობებზე. ცხადია, რომ მხოლოდ ხარისხიანად დამუშავებული არქიტექტურა უზრუნველუოფს მუშაობის უნარის მქონე სისტემის შექმნას.

ბოლოს, UP არის **იტერაციული და ინკრემენტული**. UP-ს იტერაციული ასპექტი ნიშნავს, რომ პროექტი იყოფა უფრო მცირე ქვეპროექტებათ(იტერაციებათ), რომლებიც უზრუნველყოფენ სისტემის ფუნქციალობას ნაწილნაწილ. იგი ითვალისწინებს სისტემის შესრულებადი ვერსიების ნაკადის მართვას.

იმისათვის, რომ გავიგოთ UP, საჭიროა გავიგოთ იტერაციები. საქმე იმაშია, რომ ადამიანი პატარა პრობლემებს წყვეტს უფრო ადვილად, ვიდრე დიდს. ამიტომ ყოფენ პროგრამული უზრუნველყოფის დიდ პროექტს რამოდენიმე მცირე „მინი პროექტებათ“, რომელთა მართვა ადვილია. ყოველი ასეთი „მინი პროექტი“ კი არის იტერაცია. ყოველი იტერაცია მოიცავს პროგრამული უზრუნველყოფის დამუშავების ყველა ელემენტებს:

- დაგეგმვა
- ანალიზი და დაპროექტება
- აგება
- ინტეგრაცია და ტესტირება
- ვერსია შიდა ან გარე გამოყენებისათვის.

ყოველი იტერაცია ქმნის ბაზურ ვერსიას, რომელიც მოიცავს მიზნობრევი სისტემის ნაწილობრივად დამთავრებულ ვერსიას. მიმდევრობითი იტერაციების შედეგად ბაზური ვერსია იზრდება მანამდე, სანამ არ შეიქმნება სისტემის საბოლაო სრული ვარიანტი. განსხვავება ორ მომდევნო ბაზურ ვერსიას შორის არის ინკრემენტი. ინკრემენტული პროცესი გულისხმობს სისტემური არქიტექტურის მუდმივ გაფართოებას ახალი ვერსიების გამოშვებით, ამასთან ყოველი შემდეგი ვერსია უფრო სრულყოფილია წინმდებარესთან

შედარებით. როცესი, რომელიც ერთდროულად იტერაციული და ინკრემენტულია უწოდებენ რისკით მართვადს(Risk-driven), რამდენადაც ამ დროს ყოველ ახალ ვერსიაში ყურადღება ექცევა ფაქტორების გამოვლენას, რომლებიც ქმნიან რისკს პროექტის წარმატებით შესრულებისათვის, და დაიყვანონ ისინი მინიმუმამდე.

მაშასადამე, UP-ს მიზანია ნაბიჯ-ნაბიჯ ავაგოთ სისტემის არქიტექტურა. UP-ში ხუთი ძირითადი სამუშაო ნაკადია:

მოთხოვნების განსაზღვრა – მონაცემების შეგროვება იმის შესახებ, თუ რა უნდა აკეთოს სისტემამ;

ანალიზი – მოთხოვნების დაზუსტება და სტრუქტურირება;

დაპროექტება – სისტემის არქიტექტურაში მოთხოვნების რეალიზება;

რეალიზება – პროგრამული უზრუნველყოფის აგება;

ტესტირება – მოწმდება, პასუხობს თუ არა რეალიზაცია წარმოდგენილ მოთხოვნებს.

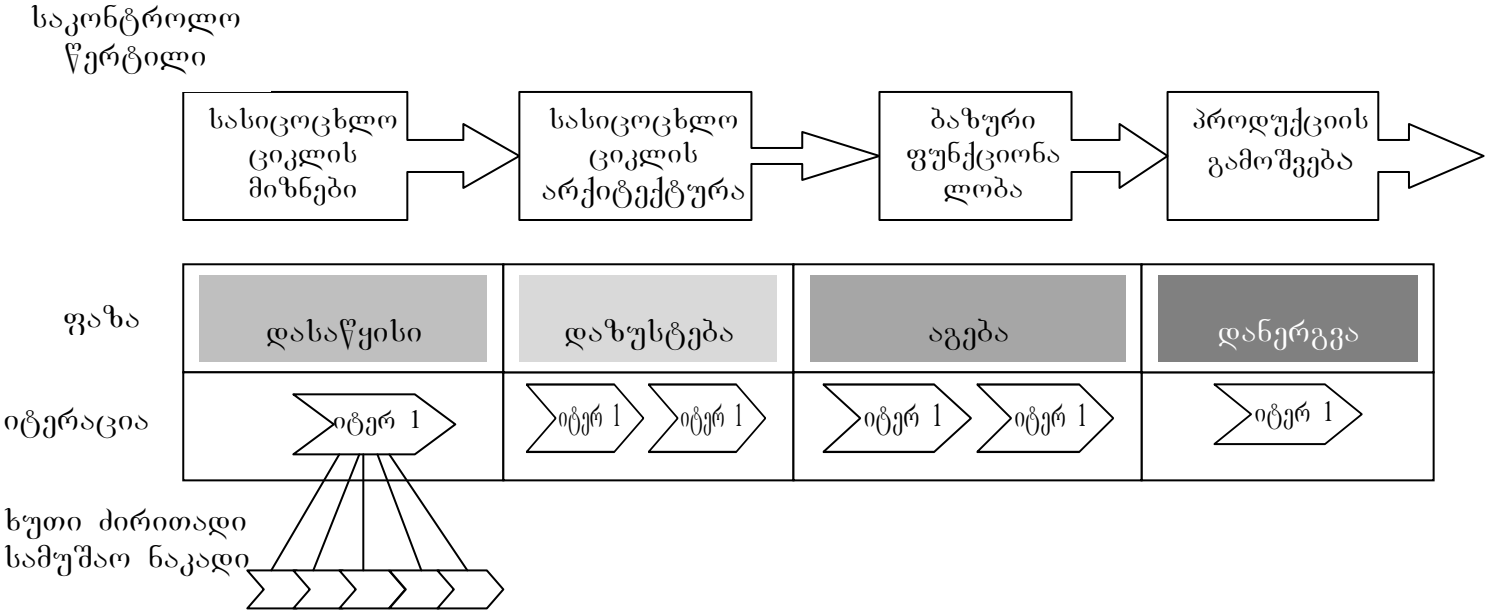
თითოეულ იტერაციაში შეიძლება გვქონდეს ხუთივე მართვის ნაკადი, მაგრამ იტერაციის პროექტის სასიცოცხლო ციკლში მისი ადგილმდებარეობის მიხედვით ყურადღება გამახვილდეს რომელიმე ერთ სამუშაო პროცესზე.

პროექტის დაყოფა იტერაციების სერიებათ საშუალებას გვაძლევს უფრო მოქნილად მიუდგეთ მის დაგეგმვას. ველაზე მარტივი მიდგომა – იტერაციათა თანმიმდევრობების მოწესრიგებაა დროში, რომლის დროსაც ყოველი შემდგომი იტერაცია წარმოადგენს წინამორბედის შედეგს. მაგრამ ხშირად იტერაციები შესაძლებელია განვალაგოთ პარალელურად, ეს განსაკუთრებით ეხება იტერაციებს, რომელთა არტეფაქტები ურთიერთ დამოკიდებულია და მოითხოვს არქიტექტურაზე და მოდელირებაზე დაფუძნებულ მიდგომას პროგრამული უზრუნველყოფის დამუშავებისადმი. პარალელური იტერაციების უპირატესობა –დამუშავების შედარებით მცირე დროა.

პრეცედენტებით მართული, არქიტექტურაზე დაფუძნებული, იტერაციული და ინკრემენტული პროცესი შესაძლებელია დაიყოს ფაზებათ. **ფაზა** – ეს დროის მონაკვეთია პროცესის ორ მნიშვნელოვან საყრდენ წერტილებს შორის, რომელშიც უნდა მიღწეულ იქნას მკაფიოდ გამოხატული მიზნები, მომზადებული იქნას ესა თუ ის არტეფაქტები და მიღებული იქნას გადაწყვეტილება- საჭიროა შემდეგ ფაზაზე გადასვლა თუ არა.

როგორც ნახ.2.1-დან ჩანს UP-ში პროექტის სასიცოცხლო ციკლი იყოფა ოთხ ფაზად, რომელთაგან თვითეულს აქვს თავისი საკონტროლო წერტილები:

1. დასაწყისი(**Inception**) – სასიცოცხლო ციკლის მიზნები.
2. გამოკვლევა(**Elaboration**) - სასიცოცხლო ციკლის არქიტექტურა.
3. აგება(**Construction**) – ბაზური ფუნქციონალობა.
4. დანერგვა(**Transition**) – პროდუქციის გამოშვება.



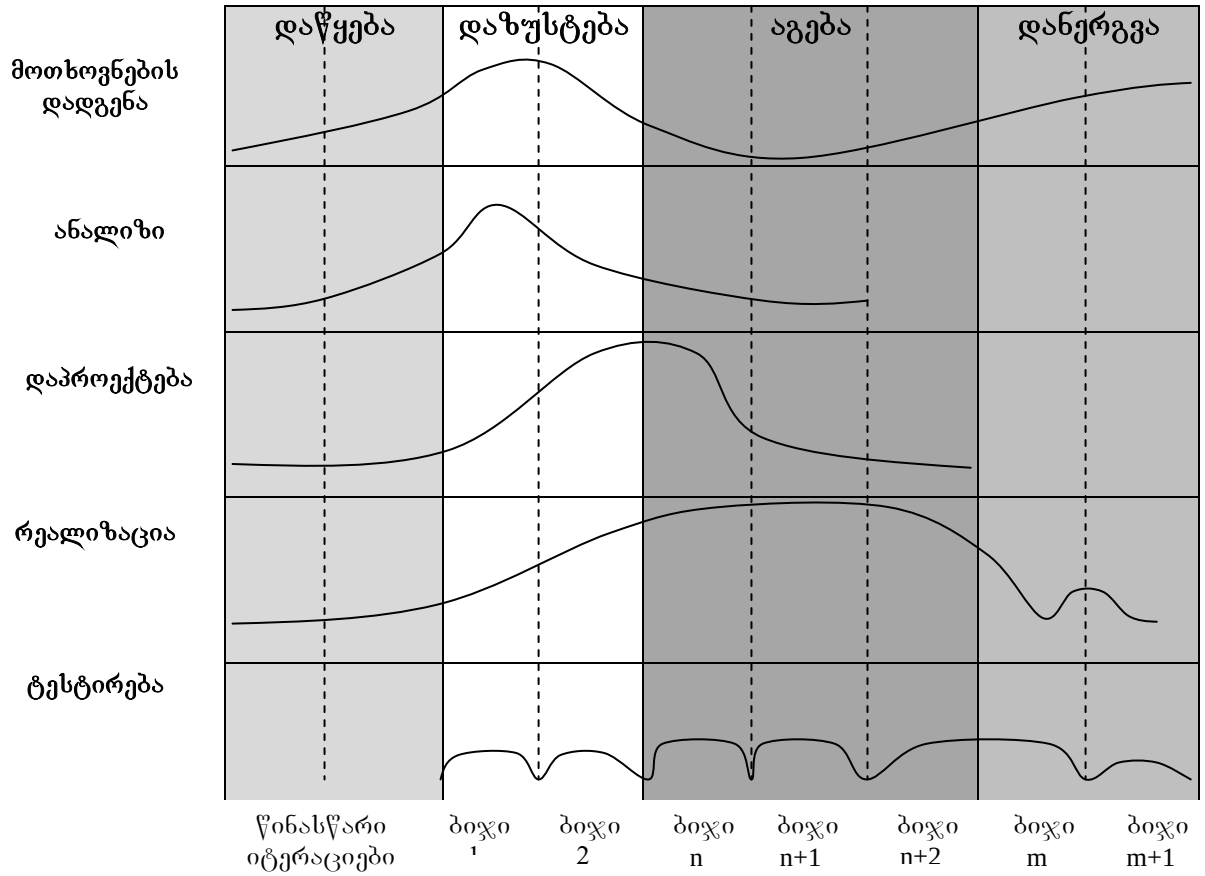
ნახ.2.1.

ყოველ ფაზაში შეიძლება გვქონდეს ერთი ან მეტი იტერაცია, ყოველ იტერაციაში სრულდება ხუთი ძირითადი და ნებისმიერი რაოდენობის დამატებითი სამუშაო ნაკადები. იტერაციების ზუსტი რაოდენობა ფაზაში დამოკიდებულია პროექტის განზომილებაზე, მაგრამ ყოველი იტერაცია უნდა გრძელდებოდეს არაუმეტეს ორი-სამი თვისა.

ნახ.2.2.-ზე მოყვანილია UP-ს მუშაობის პრინციპი. ზემოთ მოყვანილია ფაზები. მარცხენა კიდურა სვეტში – ხუთი ძირითადი სამუშაო ნაკადი. ქვევით გამოსახულია იტერაციები. კლაკნილი ხაზები უჩვენებენ მუშაობის შეფარდებით მოცულობას, რომელიც სრულდება ხუთივე ძირითად სამუშაო ნაკადში პროექტის ფაზებზე გასვლისას. მუშაობის მოცულობა, რომელიც სრულდება ყოველ ძირითად მუშა ნაკადში იცვლება ფაზების მიხედვით.

როგორც ნახაზიდან ჩანს ფაზაში **დაწყება** სამუშაოს დიდი ნაწილი ეთმობა მოთხოვნების განსაზღვრას და ანალიზს. ფაზაში **დაზუსტება** ძირითადი აქცენტი გადატანილია მოთხოვნებზე, ანალიზზე და პროექტირებაზე. ცხადია, რომ ფაზაში **აგება** ძირითადი ყურადღება მიმართულია პროექტირებასა და რეალიზაციაზე. ბოლოს, ფაზაში **დანერგვა** მთავარი ხდება რეალიზაცია და ტესტირება.

UP-ს განსაკუთრებული თავისებურება იმაშია, რომ იგი ორიენტირებულია მიზნებზე, და არა არტეფაქტების შექმნაზე. ყოველი ფაზა მთავრდება საკონტროლო წერტილით, შედგენილი პირობების ნაკრებისაგან, რომლებიც უნდა დაკმაყოფილდნენ, ეს პირობები შეიძლება შეიცავდნენ ან არ შეიცავდნენ, პროექტის კონკრეტული მოთხოვნებისაგან დამოკიდებულებით, მიწოდებისათვის მზადყმყოფი პროდუქტის შექმნას.



ნახ.2.2.

ფაზა **დასაწყისი** ახდენს პროექტის ინიციირებას. მოცემულ ფაზაში ძირითადი ყურადღება გამახვილებულია მოთხოვნების განსაზღვრასა და ანალიზზე. ტესტირება ჩვეულებრივ არ გამოიყენება. მოცემულ სტადიაზე განისაზღვრება სისტემის მიზნები და პროექტის საზღვრები. მიზნების ანალიზი მოიცავს წარმატების კრიტერიუმის გამომუშავებას, აუცილებელ რესურსებს და გეგმის შედგენას, რომელშიც გამოხატულია ძირითადი საყრდენი წერტილები. ყოველი საყრდენი წერტილი ადგენს გარკვეულ მიზნებს, რომლებიც უნდა შერულდეს რათა ჩაითვალოს, რომ საკონტროლო წერტილი გავლილია. კერძოდ, ზოგიერთი მიზნები შესაძლებელია შედგებოდეს გარკვეული არტეფაქტების წარმოებაში. ფაზის **დასაწყისი** საკონტროლო წერტილს წარმოადგენს სასიცოცხლო ციკლის მიზნები. პირობები, რომლებიც უნდა შესრულდნენ, რომ ეს საკონტროლო წერტილი ჩაითვალოს გავლილად, მოყვანილია ცხრილში 2.1.

ცხრ.2.1

მიღების პირობები	წარმოდგენილი არტეფაქტები
დაინტერესებულმა პირებმა შეათანხმეს პროექტის მიზნები	საერთო აღწერა, რომელიც განსაზღვრავს ძირითად მოთხოვნებს, პროექტის მახასიათებლებს და შეზღუდვებს
დაინტერესებულმა პირებმა განსაზღვრეს და შეათანხმეს სისტემის საპრობლემო სფერო	პრეცედენტების საწყისი მოდელი(შესრულებული მხოლოდ 10-20%-ით)
დაინტერესებულმა პირებმა განსაზღვრეს და შეათანხმეს სისტემის ძირითადი მოთხოვნები	პროექტის ტერმინები
დაინტერესებულმა პირებმა განსაზღვრეს და შეათანხმეს დანახარჯები და მუშაობის გეგმა	მუშაობის საწყისი გეგმა
პროექტის ხელმძღვანელმა ჩამოაყალიბა პროექტის ეკონომიკური დასაბუთება	ეკონომიკური დასაბუთება
პროექტის ხელმძღვანელმა ჩაატარა რისკების შეფასება	რისკების შეფასების დოკუმენტი
არქიტექტურა გათვალისწინებულია ზოგადათ	დოკუმენტი საწყისი არქიტექტურით

საწყისი ფაზის დასასრულს კიდევ ერთხელ ყურადღებით შეისწავლება პროექტის სასიცოცხლო ციკლი და მიიღება გადაწყვეტილება, ღირს თუ არა დავიწყოთ ფართო მასშტაბიანი დამუშავება.

.ცხრ.2.2

მიღების პირობები	წარმოდგენილი არტეფაქტები
შეიქმნა არქიტექტურის მოქნილი საიმედო ბაზური ვერსია	არქიტექტურის შესრულებადი ბაზური ვერსია
არქიტექტურის შესრულებადი ბაზური ვერსია გვიჩვენებს, რომ მნიშვნელოვანი რისკები გატვალისწინებული და გამოვლენილია	სტატიკური UML- მოდელი დინამიური UML- მოდელი პრეცედენტების UML- მოდელი
როექტის ეკონომიკური დასაბუთება გადაიხედა და შეთანხმებულია ყველა დაინტერესებული მხარის მიერ	პროექტის განახლებული ეკონომიკური დასაბუთება
შექმნილია პროექტის საკმაოდ დეტალური გეგმა, რამაც საშუალება მოგვცა ჩამოგვეყალიბებინა რეალური მოთხოვნა დროის, ფულის და რესურსების დანახარჯებზე შემდეგ ფაზებში	პროექტის განახლებული გეგმა
ჩატარდა ეკონომიკური დასაბუთების შემოწმება პროექტის გეგმის მიხედვით	პროექტის ეკონომიკური დასაბუთება
დაინტერესებულმა მხარეებმა მიაღწიეს შეთანხმებას პროექტის გაგრძელების შესახებ	ხელმოწერილი დოკუმენტი

დაზუსტება(გამოკვლევა). მოცემულ ეტაპზე უნდა გაანალიზდეს საგნობრივი სფერო, დავამუშაოთ სისტემის არქიტექტურული საფუძვლები, შევადგინოთ პროექტის გეგმა და გამოვრიცხოთ ყველაზე საშიში რისკები. გამოვავლინოთ პრეცედენტები, რომლებიც შეადგენენ ფუნქციონალური მოთხოვნების 80%-ს, შეიქმნას ფაზა **აგების** დეტალური გეგმა, ჩამოყალიბდეს წინადადებები, რომლებიც მოიცავენ რესურსებს, დროს, მოწყობილობებს, შტატს და ღირებულებას. შესაბამისად, ძირითადი ყურადღება დაზუსტების ფაზაში მიმართულია მოთხოვნების, ანალიზისა და დაპროექტების სამუშაო ნაკადების განსაზღვრაზე. რეალიზაცია მნიშვნელობას იძენს ფაზის დასასრულს არქიტექტურის ბაზური ვერსიის შექმნისას.

ფაზის **დაზუსტება** საკონტროლო წერტილს წარმოადგენს სასიცოცხლო ციკლის არქიტექტურა. პირობები, რომლებიც უნდა შესრულდნენ, რომ ეს საკონტროლო წერტილი ჩაითვალოს გავლილად, მოყვანილია ცხრილში 2.2

აგების ფაზაში თანდათან ან იტერაციულად მუშავდება პროდუქტი, რომელიც შესაძლებელი იქნება დაინერგოს. მოცემულ ეტაპზე აღიწერება დარჩენილი მოთხოვნები

და მიღების კრიტერიუმები, მთავრდება დამუშავება და პროგრამული კომპლექსის ტესტირება.

აგების ფაზის მიზანია – მოთხოვნების განსაზღვრის, ანალიზის და დაპროექტების დასრულება და განვავითაროთ არქიტექტურის ბაზური ვერსია, რომელიც შეიქმნა ფაზაში დაზუსტება, დამთავრებულ სისტემაში. მთავარი ყურადღება ამ ფაზაში დათმობილი აქვს რეალიზების მართვის ნაკადს. ყოველი ახალი ინკრემენტი ეწყობა წინამორბედს, ამიტომ, ტესტირება ხდება ძალიან მნიშვნელოვანი, ამასთან აუცილებელია როგორც ცალკეული ელემენტების, ისე ერთობლივი ტესტირება. შესაბამისად, ჩვენ შეგვიძლია მოკლედ დავახასიათოდ სამუშაოები, რომლებიც სრულდება ყოველ სამუშაო ნაკადში:

- მოთხოვნების განსაზღვრა – ყველა გაუთვალისწინებელი მოთხოვნის გამოვლენა;
- ანალიზი – ანალიტიკური მოდელის დამთავრება;
- დაპროექტება – დასაპროექტებელი სისტემის მოდელის დასრულება;
- რეალიზაცია – ბაზური ფუნკციონალობის შექმნა;
- ტესტირება – ბაზური ფუნქციონალობის ტესტირება.

საკონტროლო წერტილი ფაზისათვის აგება – ბაზური ფუნქციონალობაა. მოცემული საკონტროლო წერტილის მიღების პირობები მოყვანილია ცხრილში 2.3.

აგების ფაზის დასასრულს მიიღება გადაწყვეტილება დანერგვისათვის პროგრამების და მომხმარებლების მზადყოფნის შესახებ.

დანერგვის ფაზაში პროგრამული უზრუნველყოფა გადაეცემა მომხმარებელს. ამ დროს ხშირად წარმოიშვება დამატებითი დამუშავების მოთხოვნის საკითხები სისტემის გაწყობასთან დაკავშირებით, შეცდომების გასასწორებლად, რომლებიც მანამდე არ იყო შემჩნეული და რიგი ფუნქციების საბოლაოდ გასაფორმებლად, რომელთა რეალიზებაც გადადებული იყო. დანერგვა მიმართულია დასრულებული სისტემის განლაგებისათვის მომხმარებელთა სფეროში. შესაბამისად საჭიროა მომხმარებელთა საიტების მომზადება ახალი პროგრამული უზრუნველყოფისათვის და მათი სამუშაო მდგომარეობაში მოყვანა.

ცხრ. 2.3.

მიღების პირობები	წარმოდგენილი არტეფაქტები
პროგრამული პროდუქტი საკმაოდ სტაბილური და ხარისხიანია მომხმარებლებს შორის გავრცელებისათვის დაინტერესებული პირები შეთანხმდნენ და მზად არიან შეიტანონ პროგრამული პროდუქტი თავის სფეროში. განთანხმება რეალური დანახარჯებისა მოსალოდნელთან მისაღებია.	პროგრამული პროდუქტი. UML მოდელი. სატესტო კომპლექტი. სახელმძღვანელო მომხმარებლისათვის. მოცემული ვერსიის აღწერა. პროექტის გეგმა

ძირითადი ყურადღება მიმართულია რეალიზაციისა და ტესტირების სამუშაო ნაკადებს. უნდა მივისწრაფოდეთ იმისკენ, რომ დანერგვის ფაზაში სამუშაო ნაკადებმა მოთხოვნების დადგენა და ანალიზი პრაქტიკულად არ იმოქმედონ. წინააღმდეგ შემთხვევაში პროექტში ყველაფერი კარგათ არ არის. შესაბამისად, სამუშაოები ნაწილდება ასე:

- მოთხოვნების განსაზღვრა – არ ტარდება;
- ანალიზი – არ ტარდება;
- დაპროექტება – კონსტრუქციის შეცვლა ბეტა-ტესტირებისას გამოვლენილი პრობლემებისას;
- რეალიზაცია – პროგრამული უზრუნველყოფის დაყენება მომხმარებლის საიტისათვის და პრობლემების შესწორება, რომლებიც არ იყვნენ გამოვლენილი ბეტა-ტესტირებისას;
- ტესტირება – ბეტა-ტესტირება და მისაღები გამოცდები მომხმარებელთა საიტებზე.

საკონტროლო წერტილი ფაზისათვის დანერგვა – პროდუქციის გამოშვებაა. მოცემული საკონტროლო წერტილის მიღების პირობები მოყვანილია ცხრილში 2.4.

დანერგვის ფაზის ბოლოს დგინდება მიღწეულია თუ არა პროექტის მიზნები და საჭიროა თუ არა დავიწყოთ დამუშავების ახალი ციკლი.

ცხრ.2.4.

მიღების პირობები	წარმოდგენილი არტეფაქტები
ბეტა-ტესტირება დამთავრებულია, აუცილებელი ცვლილებები გაკეთდა და მომხმარებლები ეთანხმებიან იმას, რომ სისტემა წარმატებით განლაგდა. მომხმარებლები აქტიურად იყენებენ პროდუქტს.	პროგრამული პროდუქტი.
პროდუქციის მხარდამჭერი სტრატეგიები შეთანხმებულია მომხმარებლებთან და რეალიზებულია.	მომხმარებლის მხარდამჭერი გეგმა მომხმარებელთა განახლებული სახელმძღვანელო.

თავი 2
სისტემისადმი მოთხოვნების დადგენა
2.1. მოთხოვნის ცნების განსაზღვრა

მოთხოვნა(**Requirement**) – ეს პროექტის თავისებურება, თვისება ან სისტემის ქცევაა. მოთხოვნების დადგენისას ჩვენ გარკვეულ წილად აღვწერთ კონტრაქტების პირობებს, რომელიც დაიდება სისტემასა და მის გარეთ მყოფ არსებს შორის, რომელშიც დეკლარირდება თუ რა უნდა აკეთოს სისტემამ. ამასთან ჩვენ არ გვაინტერესებს თუ როგორ შეასრულებს სისტემა დასახულ ამოცანას, არამედ ის თუ რას გააკეთებს იგი. კარგათ დაპროექტებულმა სისტემამ მთლიანად უნდა შეასრულოს ყველა მოთხოვნები, ამასთან უნდა აკეთოს ეს საიმედოთ.

არსებობს მოთხოვნების ორი ძირითადი ტიპი:

1. ფუნქციონალური მოთხოვნები – როგორ ქცევას უნდა გვთავაზობდეს სისტემა;
2. არაფუნქციონალური მოთხოვნები – განსაკუთრებული თვისება ან შეზღუდვა, რომელიც დაიდება სისტემაზე.

მოთხოვნები – ეს ყველა სისტემისათვის საფუძველია. UP-ს გააჩნია ფორმალური მიდგომა მოთხოვნების განსაზღვრისათვის, რომლებიც დაფუძნებულია პრეცედენტების მოდელზე. მოთხოვნების მოდელი შესაძლებელია შეიქმნას ტექსტურ რედაქტორში ან მოთხოვნათა დამუშავების სპეციალიზებულ ინსტუმენტალურ საშუალებებში, მაგ. ReguisitePro ან DOORS.

მოთხოვნათა ფორმულირების ფორმატი შემდეგია:

უნიკალური იდენტიფიკატორი გასაღებური სიტყვა

<id> <სისტემა> chall <მოქმედება>

სისტემის დასახელება მოქმედება, რომელიც უნდა გამოვლენილი იქნას

მოთხოვნათა ჩაწერისათვის გამოიყენება მტკიცება < chall> (უნდა). ყოველ მოთხოვნას აქვს უნიკალური იდენტიფიკატორი (ჩვეულებრივ ეს რიცხვია), გასაღებური სიტყვა(chall უნდა) და მოქმედების აღწერა. მას ძირითადათ მიმართავენ მოთხოვნის სინტაქსური გარჩევის გამარტივებისათვის მოთხოვნათა მართვის ინსტრუმენტებისათვის.

ფუნქციონალური მოთხოვნები – ეს არის ის, რაც უნდა სისტემამ გააკეთოს. მაგ. ბანკომატისათვის ATM, შესაძლებელია განვსაზღვროთ შემდეგი ფუნქციონალური მოთხოვნები:

- სისტემა ATM < chall> (უნდა) ამოწმებდეს ბანკომატში ჩადებული ბარათის მართებულობას;
- სისტემა ATM < chall> (უნდა) ამოწმებდეს მომხმარებლის შეყვანილი PIN კოდის სისწორეს;
- სისტემა ATM < chall> (უნდა) გამოსცემდეს თითო ATM ბარათზე არა უმეტეს 250 დოლარს დღე-ღამეში.

არაფუნქციონალური მოთხოვნები – ეს შეზღუდვებია, რომლებიც დაიდება სისტემაზე. ATM სისტემას შესაძლებელია გააჩნდეს შემდეგი არაფუნქციონალური მოთხოვნები:

- სისტემა ATM < chall> (უნდა) ცვლიდეს ინფორმაციას ბანკთან 256-თანრიგიანი კოდირების გამოყენებით;
- სისტემა ATM < chall> (უნდა) დაწერილი იქნას C++ ენაზე;
- სისტემა ATM < chall> (უნდა) ამოწმებდეს ATM ბარათის ვარგისიანობას სამი წამის განმავლობაში;
- სისტემა ATM < chall> (უნდა) ამოწმებდეს PIN კოდის სამართლიანობას სამი წამის განმავლობაში;

ადვილი შესამჩნევია, რომ არაფუნქციონალური მოთხოვნები ადებენ შეზღუდვებს იმაზე, თუ როგორ იქნება რეალიზებული სისტემა.

მოთხოვნების ორგანიზება. ინსტრუმენტალური საშუალებების გამოყენებისას შესაძლებელია მოთხოვნების ორგანიზება **ტიპების იერარქიაში**, რომელიც შესაძლებელია გამოყენებულ იქნას მოთხოვნათა კლასიფიკაციისას. მოთხოვნათა ტიპები ძირითადათ გამოიყენებიან შედარებით მცირე ჯგუფების შესადგენად არასტრუქტურირებული დიდი რაოდენობის მოთხოვნებიდან. ასეთი ჯგუფების მართვა უფრო ეფექტურია. ბაზური დაყოფა ფუნქციონალურ და არა ფუნქციონალურ მოთხოვნებათ, საკმაოდ მარტივია, მაგრამ შესაძლებელია შემდგომი დაყოფა და შემოვიტანოთ მოთხოვნათა კატეგორიები:

ფუნქციონალური მოთხოვნები

- კლიენტები
- პროდუქტები
- შეკვეთები
- მომარაგების წყაროები
- გადასახადები

არაფუნქციონალური მოთხოვნები

- წარმადობა
- წარმოების სიმძლავრე
- არსებობა
- სტანდარტებთან შესაბამისობა
- უშიშროება

მოთხოვნათა კონკრეტული ტიპები დამოკიდებულია შესაქმნელი პროგრამული უზრუნველყოფის ტიპზე. ეს განსაკუთრებით ეხება ფუნქციონალურ მოთხოვნებს.

მოთხოვნათა ატრიბუტები. ყოველ მოთხოვნას შესაძლებელია გააჩნდეს რიგი ატრიბუტებისა, რომლებიც აფიქსირებენ დამატებით ინფორმაციას მოთხოვნებზე.

მოთხოვნის ყოველ ატრიბუტს აქვს აღწერითი სახელი და მნიშვნელობა. მაგალითად, მოთხოვნას შესაძლებელია გააჩნდეს ატრიბუტი dueDate(გადახდის თარიღი), რომლის მნიშვნელობაა მოცემული მოთხოვნის შესრულების თარიღი. ასევე მოთხოვნას შესაძლებელია გააჩნდეს ატრიბუტი source(წყარო), რომლის მნიშვნელობა გამომდინარეობს აღწერიდან, საიდან წარმოიშვა მოცემული მოთხოვნა. გამოყენებული ატრიბუტების კონკრეტული ნაკრები დამოკიდებულია პროექტის ბუნებაზე და შესაძლებელია იცვლებოდეს მოთხოვნის ტიპის მიხედვით. ყველაზე გავრცელებულ ატრიბუტს წარმოადგენს priority(პრიორიტეტი). მისი მნიშვნელობა განსაზღვრავს მოთხოვნის პრიორიტეტს. ჩვეულებრივ პრიორიტეტის დანიშვნისათვის გამოიყენება კრიტერიების ნაკრები MoSCoW, რომელიც მოყვანილია ცხრილში 2.1.

მოთხოვნათა ატრიბუტების უფრო სრულ ნაკრებს განსაზღვრავს RUP. თუ რომელი გამოვიყენოთ – MoSCoW, RUP ან რომელიმე სხვა მოთხოვნათა ატრიბუტების ნაკრები დამოკიდებულია კონკრეტულ პროექტზე. უნდა ამოვირჩიოთ მხოლოდ ის ატრიბუტები, რომელიც სასარგებლოა პროექტისათვის. მოთხოვნათა ატრიბუტების რაოდენობა უნდა იყოს მინიმალური, ამით პროექტი მხოლოდ მოიგებს.

ცხრ.2.1.

ატრიბუტი პრიორიტეტი მნიშვნელობა სემანტიკა

Must have (ვალდებული ქონდეს).	აუცილებელი მოთხოვნები, რომლებიც წარმოადგენენ ფუნდამენტალურს სისტემისადმი.
Should have (უნდა ქონდეს).	მნიშვნელოვანი მოთხოვნები, რომლებიც შესაძლებელია გამოტოვებულ იქნას
Could have (შესაძლებელია ქონოდა).	ნამდვილად არა აუცილებელი მოთხოვნები (რეალიზდებიან, თუ ამისათვის არის დრო)
Want to have (უნდოდა ქონოდა).	მოთხოვნები, რომლებმაც შესაძლებელია დაიცადონ სისტემის შემდეგი ვერსიისათვის

თუ გამოიყენება სქემა MoSCoW, ყოველ მოთხოვნას აქვს ატრიბუტი priority, რომელსაც შეუძლია მიიღოს ერთერთი მნიშვნელობა: M, S, C ან W. მოთხოვნათა გამომუშავების ინსტრუმენტალური საშუალებები ჩვეულებრივ უზრუნველყოფენ მიმართვას მოთხოვნების მოდელთან ატრიბუტის მნიშვნელობით. მაშასადამე, საშუალება გვეძლევა მაგალითად, მოვახდინოთ მაქსიმალური პრიორიტეტის (**M**ust have) მქონე ყველა მოთხოვნების გენერირება.

მოთხოვნების ძიება. მოთხოვნები დგინდება სამოდელო სისტემიდან და ამ კონტექსში შედიან:

- სისტემის უშუალო მომხმარებლები;
- დაინტერესებული პირები (ხელმძღვანელები, მომსახურების სპეციალისტები, დამყენებლები);
- სხვა სისტემები, რომლებთანაც ურთიერთქმედებს მოცემული სისტემა;
- აპარატული მოწყობილობები, რომლებთანაც ურთიერთქმედებს მოცემული სისტემა;
- ტექნიკური შეზღუდვები;
- კომერციული მიზნები.

როგორც წესი, მოთხოვნების გამომუშავება იწყება დოკუმენტიდან, რომლითაც აღიწერება, თუ რის გაკეთებას აპირებს სისტემა და რა მომსახურებას წარუდგენს იგი დაინტერესებულ პირებს. ამ დოკუმენტის დანიშნულებაა – გამოიკვეთოს სისტემის მთავარი მიზნები დაინტერესებული პირების თვალთახედვით. საერთო აღწერა დგება სისტემური ანალიტიკოსის მიერ UP-ს ფაზაში **დაწყება**. სისტემის საერთო აღწერის შემდეგ იწყება მოთხოვნების კონკრეტიზირება ცალკეული მომხმარებლებისა და შეზღუდვების გათლისწინებით.

მოთხოვნათა გამოვლენის სხვადასხვა საშუალება არსებობს, კერძოთ პირადი საუბრები, ინტერვიუ, გამოკითხვის ანკეტები და სემინარები. ველა შემთხვევაში ჩვენ ვცდილობთ მივიღოთ ზუსტი სურათი – მოდელი მათი მოქმედების სფეროს შესახებ. ასეთი სურათი იქმნება სამი პროცესით(ნოამ ხომსკის თეორიით): გაშვება – გამოტოვება(deletion), დამახინჯება(distortion) და განზოგადება (generalization). ეს პროცესები აუცილებელია რამდენადაც, ჩვენ არ ვფლობთ შემეცნების მექანიზმს, რომელსაც აქვს უნარი დააფიქსიროს ჩვენი მოღვაწეობის სფეროს ყოველი დეტალი რაღაც წარმოსახვითში. ამიტომ უნდა ვიყოთ გმომგონებლები. ჩვენ ვახდენთ "ამორჩევას" შესაძლო ინფორმაციის უდიდესი მასივიდან სამი ფილტრის გამოყენებით:

- გამოტოვება – ინფორმაცია იფილტრება;
- დამახინჯება – ინფორმაცია იცვლება ურთიერთ დაკავშირებული გამოგონებისა და წარმოდგენის მექანიზმებით;
- განზოგადება – ინფორმაცია განზოგადდება წესებში, რწმენასა და ჭეშმარიტებისა და სიცრუის შესახებ მცნებებში.

ეს ფილტრები ქმნიან ბუნებრივ ენას. ამის შესახებ მნიშვნელოვანია ვიცოდეთ მოთხოვნების დადგენისა და ანალიზისას.

მოთხოვნათა დადგენის ყველაზე სწორ საშუალებას წარმოადგენს დაინტერესებულ პირებთან ინტერვიუს ჩატარება. ჩვეულებრივ უფრო სრული ინფორმაცია შესაძლებელია მივიღოთ ერთი ერთზე ინტერვიუს დროს.

სასარგებლოა ანკეტების გამოყენებაც. მაგრამ, მხოლოდ ინტერვიუს შემდეგ, რადგან არ აღმოვჩნდეთ ისეთ სიტუაციაში, როდესაც არ დადგენილა თუ რა კითხვები დაისვას. ანკეტები სასარგებლოა მხოლოდ დამატებით ინტერვიუზე. ინტერვიუდან შესაძლებელია გამოიყოს ძირითადი შეკითხვები, გავაერთიანოთ ისინი ანკეტაში, ხოლო შემდეგ გავავრცელოდ ფართო აუდიტორიაზე. ეს საშუალებას მოგვცემს შევამოწმოთ, სწორათ გვესმის თუ არა მოთხოვნა.

მოთხოვნათა დადგენის ეფექტურ საშუალებას წარმოადგენს სემინარი. ძირითადი მოთხოვნების დასადგენად ტარდება სემინარი და დაინტერესებულ პირებს იწვევენ მასში

მონაწილეობის მისაღებათ სემინარში უნდა მონაწილეობდნენ პროექტის ხელმძღვანელი, მოთხოვნების დამმუშავებელი, ძირითადი დაინტერესებული პირები და მოცემული საპრობლემო სფეროს სპეციალისტები. სემინარზე გამოთქმული ყველა იდეა ფიქსირდება, მაგრამ მის შესახებ არ კამათობენ. შეხვედრის შემდეგ შედეგები ანალიზდება და ყალიბდება მოთხოვნების სახით,

მოთხოვნების დადგენა – ეს იტერაციული პროცესია, რომლის დროსაც მოთხოვნები ვლინდება დაინტერესებულ პირთა საჭირობების გაგებისა და დაზუსტების მიხედვით. ამიტომ, შესაძლებელია დაგვჭირდეს რამოდენიმე სემინარის ორგანიზება.

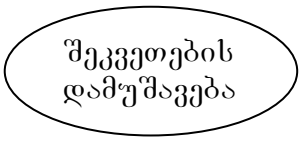
მაშასადამე, მოთხოვნები შეიძლება გამოვსახოთ სხვადასხვანაირად, არასტრუქტურირებული ტექსტიდან დაწყებული, დამთავრებული ფორმალურ ენაზე წარმოდგენით. ფუნქციონალური მოთხოვნების უმეტესობა შეიძლება გამოვსახოთ გამოყენების პრეცენდენტების სახით, რაშიც გვეხმარება პრეცენდენტების დიაგრამები.

2.2. მოთხოვნების მოდელირება

მოთხოვნების მოდელირება – ეს მოთხოვნების გამომუშავების ფორმაა, დამატებითი საშუალებაა მოთხოვნების გამოვლენისა და დოკუმენტირების. მოთხოვნების მოდელირება მოიცავს აქტიორებისა და პრეცედენტების გამოვლენას.

პრეცენდენტს (**Use case**) უწოდებენ გარკვეული თანმიმდევრობის მოქმედებათა სიმრავლის აღწერას, რომელსაც ასრულებს სისტემა იმისათვის, რომ აქტიორს შეეძლოს მიიღოს განსაზღვრული შედეგი.

ყოველ პრეცედენტს უნდა გააჩნდეს სახელი, რომელიც განასხვავებს მას სხვა პრეცედენტებისაგან. გრაფიკულად პრეცედენტი გამოისახება ელიფსის სახით,

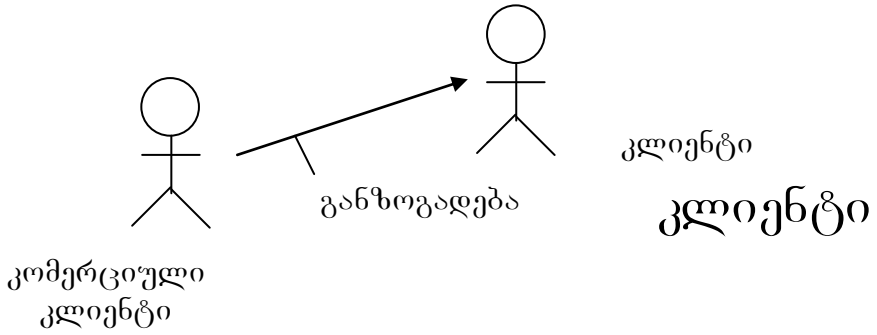


რომლის შიგნით იწერება დასახელება. პრეცედენტის სახელი წარმოადგენს ტექსტურ სტრიქონს

სისტემის აგებისას პირველ რიგში უნდა განისაზღვროს მისი საზღვრები ანუ განვსაზღვროთ, რა არის სისტემის ნაწილი(იმყოფება სისტემის საზღვრების შიგნით) და რა იმყოფება სისტემის გარეთ(მისი საზღვრების გარეთ). UML-ში სისტემის საზღვარს უწოდებენ კონტექსტს. იგი გამოისახება სწორკუთხედით სისტემის დასახელებით. პრეცედენტების მოდელირების დასწყისში გვაქვს მხოლოდ მიახლოებითი წარმოდგენა თუ

სად არის სისტემის საზღვარი. აქტიორებისა და პრეცედენტების გამოვლენასთან ერთად სისტემის კონტექსტი იძენს სულ უფრო მკვეთრ საზღვრებს.

აქტიორი წარმოადგენს დამაკავშირებელი როლების სიმრავლეს, რომელსაც პრეცენდენტების მომხმარებლები ასრულებენ მათთან ურთიერთობაში. ჩვეულებრივ აქტიორი წარმოადგენს როლს, რომელსაც თამაშობს პიროვნება, აპარატული მოწყობილობა ან სხვა სისტემა. აქტიორები გამოისახებიან შემდეგი სახით



აქტიორების იდენტიფიკაციისათვის საჭიროა გავცეთ პასუხი კითხვებს: ვინ ან რა იყენებს სისტემას და რანაირი როლები სრულდება აქტიორების მიერ სისტემასთან ურთიერთქმედებისას.

პრეცედენტების იდენტიფიკაცია უფრო კარგია დავიწყოთ აქტიორთა ცხრილიდან, ხოლო შემდეგ განვიხილოთ, თუ ყოველი აქტიორი როგორ აპირებს სისტემის გამოყენებას. ასეთი სტრატეგიით შესაძლებელია მივიღოთ პოტენციური პრეცედენტები.

ცალკეული პრეცედენტის თვალსაზრისით არსებობს ორი ტიპის აქტიორი:

- მთავარი აქტიორი – აქტიორები, რომლებიც ახდენენ პრეცედენტის ინიციირებას;
- მეორეხარისხოვანი აქტიორები – აქტიორები, რომლებიც ურთიერთქმედებენ პრეცედენტთან მისი ინიციაციის შემდეგ.

აქტიორები პრეცედენტებს უკავშირდებიან მხოლოდ ასოციაციური კავშირით. ასოციაცია აქტიორსა და პრეცედენტს შორის გვიჩვენებს, რომ ისინი ურთიერთობენ, შესაძლოა უგზავნიან ან ღებულობენ ერთმანეთისაგან შეტყობინებებს. გარდა ამისა, აქტიორებს შესაძლებელია გააჩნდეთ მსგავსი ქცევა, რაც გამოხატულობას პოვებს დიაგრამაზე გადამკვეთი კავშირების არსებობით აქტიორებსა და პრეცედენტებს შორის. ეს მიუთითებს საერთო ქცევის არსებობას, რომელიც შესაძლებელია გამოვიტანოთ და წარმოვადგინოთ უფრო ზოგად აქტიორად. შესაბამისად ჩვენ ვქმნით აბსტრაქტულ აქტიორს, რომელიც ურთიერთქმედებს პრეცედენტებთან. მაგრამ, აქვე უნდა აღინიშნოს, რომ აქტიორი-მშობელი განზოგადებაში ყოველთვის არ არის აბსტრაქტული. ეს შესაძლებელია იყოს კონკრეტული როლი, რომელსაც ასრულებს ადამიანი ან სისტემა. აქტიორი შვილობილი შესაძლებელია გამოყენებულ იქნას ყველგან, სადაც

მოსალოდნელია აქტიორი-მშობელი. აქტიორი შვილობილები მემკვიდრეობით იძენენ როლს და პრეცედენტებთან მიმართებებს აქტიორი-მშობელისაგან. აქტიორი შვილობილი შესაძლებელია გამოყენებულ იქნას აქტიორი-მშობელის მაგიერ.

პრეცენდენტი აღწერს თუ რას აკეთებს სისტემა, მაგრამ არ განსაზღვრავს თუ როგორ აკეთებს იგი ამას. მოდელირების პროცესში ყოველთვის მნიშვნელოვანია გამოვყოთ შიდა და გარე წარმოდგენა. პრეცენდენტის აღწერაში სასურველია მიეთითოს, თუ როგორ და როდის იწყება და მთავრდება პრეცენდენტი, როდის ურთიერთქმედებს აქტიორებთან და რომელ ობიექტებთან იცვლებიან ინფორმაციით. როგორც წესი, სამუშაოს დაწყებისას მოვლენათა ნაკადებს აღწერენ ტექსტის სახით. სისტემისადმი მოთხოვნების დაზუსტების შესაბამისად გადადიან გრაფიკულ გამოსახვაზე ურთიერთქმედების დიაგრამის მეშვეობით.

პროექტის ლექსიკონი. პროექტის ლექსიკონი – ერთერთი მნიშვნელოვანი არტეფაქტია პროექტის. მოღვაწეობის ნებისმიერ სფეროს აქვს საკუთარი უნიკალური ენა, ამიტომ მოთხოვნათა გამომუშავებისა და ანალიზის პროცესის ძირითადი მიზანია ამ ენის გაგება და ფიქსაცია. იგი უზრუნველყოფს საქმიანი ტერმინებითა და განსაზღვრებით. UML არ ადგენს რაიმე სტანდარტებს პროექტის ლექსიკონისათვის. იგი უნდა იყოს მაქსიმალურად მარტივი და მოკლე. შესაძლებელია გამოვიყენოთ სორტირების ფორმატი სიტყვებისა და გამოსახულებებისა ალფაბიტის მიხედვით.

პრეცენდენდები და სცენარები. ყოველ პრეცენდენტს შეესაბამება მოვლენათა სიმრავლე, რომლებიც განაპირობებენ მისი განხორციელების სხვადასხვა თანმიმდევრობას, ანუ სცენარს. შესაბამისად პრეცენდენტი აღიწერება არა ერთი, არამედ თანამიმდევრობათა სიმრავლით, რამდენადაც პრეცენდენტის ჩვენთვის საინტერესო ყველა დეტალის გამოხატვა ერთი თანმიმდევრობით შეუძლებელია. ამიტომ, სასურველია გამოვყოთ მთავარი(ძირითადი) მოვლენათა ნაკადი ალტერნატიულებისაგან. მაგალითად, პრეცენდენტს "მუშაკის დაქირავება" გააჩნია განხორციელების სხვადასხვა ვარიანტები:

- გადმოვიბიროთ სხვა კომპანიიდან;
- გადმოვიყვანოთ რომელიმე სხვა განყოფილებიდან;
- დავიქირავოთ ახალი მუშაკი.

ყოველი ვარიანტი გამოისახება თავისი თანმიმდევრობით, ანუ სცენარით. შესაბამისად თითოეული სცენარი წარმოადგენს ერთ შესაძლო ვარიანტს მოცემულ მოვლენათა ნაკადში. სცენარი – ეს მოქმედებათა გარკვეული თანმიმდევრობაა, რომელიც წარმოადგენს სისტემის ქცევას. სცენარები ისეთივე დამოკიდებულებაში არიან პრეცენდენტებთან, როგორითაც ეგზემპლიარები კლასებთან, ე. ი. სცენარი – ეს პრეცენდენტის ეგზემპლიარია.

შედარებით რთული სისტემა შეიცავს რამოდენიმე ათეულ პრეცენდენტს, რომელთაგან თითოეული შეიძლება გაიშალოს რამოდენიმე ათეულ სცენარში. ყოველი პრეცენდენტისათვის შეიძლება გამოვყოთ ძირითადი სცენარები, რომელიც აღწერს ძირითად თანმიმდევრობას, და დამხმარე, რომლებიც აღწერენ ალტერნატიულ თანმიმდევრობებს.

ძირითადი ნაკადი არეგისტრირებს პრეცედენტების ეტაპებს, რომლებიც გამოსახავენ ”იდეალურ” სიტუაციას, როდესაც ყველაფერი მიდის, როგორც მოსალოდნელია და გვინდა, ანუ არ წარმოიშვება შეცდომები, გადახრები, ან წყვეტები.

პრეცედენტის აღწერა წარმოადგენს სპეციფიკაციას, რომელიც შედგება შემდეგი პუნქტებისაგან:

- დასახელება;
- მოკლე დახასიათება;
- მიზნები და შედეგები (მოქმედი პირის თვალთახედვით);
- სცენარების აღწერა (ძირითადის და ალტერნატიულის);
- სპეციალური მოთხოვნები (შეზღუდვები დროში ან სხვა რესურსებში);
- გაფართოება (კერძო შემთხვევები);
- კავშირები სხვა პრეცედენტებთან;
- მოღვაწეობის დიაგრამები (სცენარების თვალსაჩინო აღწერისათვის – აუცილებლობის შემთხვევაში).

მაგალითისათვის, მოვიყვანოთ რამოდენიმე პრეცედენტის(“სისტემაში შესვლა”, “კურსებზე დარეგისტრირება”} სპეციფიკაცია.

დასახელება:
“სისტემაში შესვლა”
მოკლე აღწერა:
პრეცედენტი აღწერს მომხმარებლის შესვლას სისტემაში.

მოვლენათა ძირითადი ნაკადი:

პრეცედენტი იწყებს შესრულებას, როდესაც მომხმარებელს სურს სისტემაში შესვლა სასურველი სამუშაოს შესასრულებლად.

1. სისტემა ჩაეკითხება მომხმარებლის სახელს და პაროლს.
2. მომხმარებელს შეყავს სახელი და პაროლი.
3. სისტემა ადასტურებს სახელს და პაროლს, რომლის შემდეგ იხსნება სისტემასთან მიმართვის შესაძლებლობა.

ალტერნატიული ნაკადები:

არასწორი დასახელება/პაროლი:

თუ ძირითადი ნაკადის შესრულებისას აღმოჩნდება, რომ მომხმარებელმა შეიტანა არასწორად სახელი/პაროლი, სისტემას გამოყავს შეტყობინება შეცდომის შესახებ. მომხმარებელს შეუძლია დაბრუნდეს ძირითადი ნაკადის დასაწყისთან ან უარი თქვას სისტემაში შესვლაზე, ამით პრეცედენტის შესრულება მთავრდება.

დასახელება:
კურსებზე დარეგისტრირება.
მოკლე აღწერა:
მოცემული პრეცედენტი საშუალებას აძლევს სტუდენტს დარეგისტრირდეს შემოთავაზებულ კურსებზე მიმდინარე სემესტრში.
ძირითადი სცენარი:

1. სტუდენტი მიდის დეკანატის მოსამსახურესთან და გადასცემს მას შევსებულ ფორმას კურსებზე რეგისტრაციისათვის.
2. დეკანატის მოსამსახურე ადასტურებს ფორმის შევსების სისწორეს.
3. დეკანატის მოსამსახურე ადასტურებს, რომ სტუდენტმა შეასრულა წინასწარი მოთხოვნები ყოველი არჩეული კურსისათვის (გარკვეული კურსების გავლა), ასევე თავისუფალი ადგილების არსებობას.
4. დეკანატის მოსამსახურეს შეყავს სტუდენტი, მის მიერ არჩეულ, ყოველი კურსის სიაში.
5. დეკანატის მოსამსახურე ავსებს სტუდენტის გრაფიკს კურსებზე მიმდინარე სემესტრში და გადასცემს მას სტუდენტს.

ალტერნატიული სცენარი:
2ა. არასწორად არის შევსებული რეგისტრაციის ფორმა.
დეკანატის მოსამსახურე უბრუნებს სტუდენტს ფორმას შეცდომების გასწორებისათვის.
3ა. არ არის შესრულებული წინასწარი მოთხოვნები ან კურსი შევსებულია.
თუ დეკანატის მოსამსახურე აღმოაჩენს, რომ სტუდენტს არ შეუსრულებია აუცილებელი წინასწარი მოთხოვნები ან მის მიერ არჩეული კურსი შევსებულია (უკვე ჩაწერილია 10 სტუდენტი), მაშინ იგი სთავაზობს სტუდენტს შეიცვალოს თავისი არჩევანი ან წაიღოს ფორმა და დაუბრუნდეს მას მოგვიანებით.

ყოველ პრეცედენტს გააჩნია ძირითადი ნაკადი და უამრავი ალტერნატიული ნაკადები. ალტერნატიული ნაკადი შესაძლებელია იწყებოდეს დროის ნებისმიერ მომენტში და გამოიყენება იმის მოდელირებისათვის, თუ რა შეიძლება მოხდეს ძირითადი ნაკადის ნებისმიერ წერტილში. იმისათვის, რომ გამოვავლინოთ ალტერნატიული ნაკადი, უნდა ყურადღებით იქნას შესწავლილი ძირითადი ნაკადი. ძირითადი ნაკადის ყოველ ბიჯზე აუცილებელია ვეძებოთ:

- შესაძლო ალტერნატივები ძირითად ნაკადს;
- შეცდომები, რომლებიც შესაძლებელია აღიძვრეს ძირითად ნაკადში;

- გამორიცხვები, რომლებიც შესაძლებელია მოხდეს ძირითადი ნაკადის ნებისმიერ წერტილში.

ალტერნატიული ნაკადების რაოდენობა უნდა დაყვანილი იქნას მინიმუმამდე. ამისათვის არსებობს ორი სტრატეგია:

- ამოვირჩიოთ ყველაზე მთავარი ალტერნატიული ნაკადები.
- მსგავსი ალტერნატიული ნაკადები ჩამოვაყალიბოთ როგორც ერთი ნიმუშის სახით, თუ საჭირო არის მიუთითოდ შენიშვნები, რომლითაც აიხსნება თუ რით განსხვავდებიან სხვა ნაკადები ნიმუშისაგან.

პრეცენდენტების ორგანიზება. პრეცენდენტების ორგანიზებისათვის მათ აერთიანებენ პაკეტებში ან განსაზღვრავენ მათ შორის განზოგადების, ჩართვისა და გაფართოების კავშირებს.

განზოგადება ნიშნავს, რომ პრეცენდენტი შვილობილი მემკვიდრეობით იძენს თავისი მშობლის ქცევას და სემანტიკას, შეუძლია ჩაენაცვლოს მას ან შეავსოს მისი ქცევა. პრეცედენტების განზოგადება გამოიყენება, თუ გვაქვს ერთი ან მეტი პრეცედენტი, რომლებიც ჩვეულებრივ წარმოადგენენ უფრო ზოგადი პრეცედენტების სპეციალიზაციებს. მას იყენებენ იმ შემთხვევაში, როდესაც ისინი ამარტივებენ პრეცედენტების მოდელს. პრეცედენტების განზოგადებას გამოაქვს ქცევა, რომლებიც საერთოა ერთი ან მეტი პრეცედენტისათვის, მშობელ პრეცედენტში. პრეცედენტების განზოგადებაში შვილობილი პრეცედენტები წარმოადგენენ მათი მშობლების უფრო სპეციალიზირებულ ფორმებს. შვილობილებს შეუძლიათ:

- მემკვიდრეობით მიიღონ მშობელი პრეცედენტის შესაძლებლობები;
- შემოიტანონ ახალი შესაძლებლობები;
- შეცვალონ მემკვიდრეობით მიღებული შესაძლებლობები.

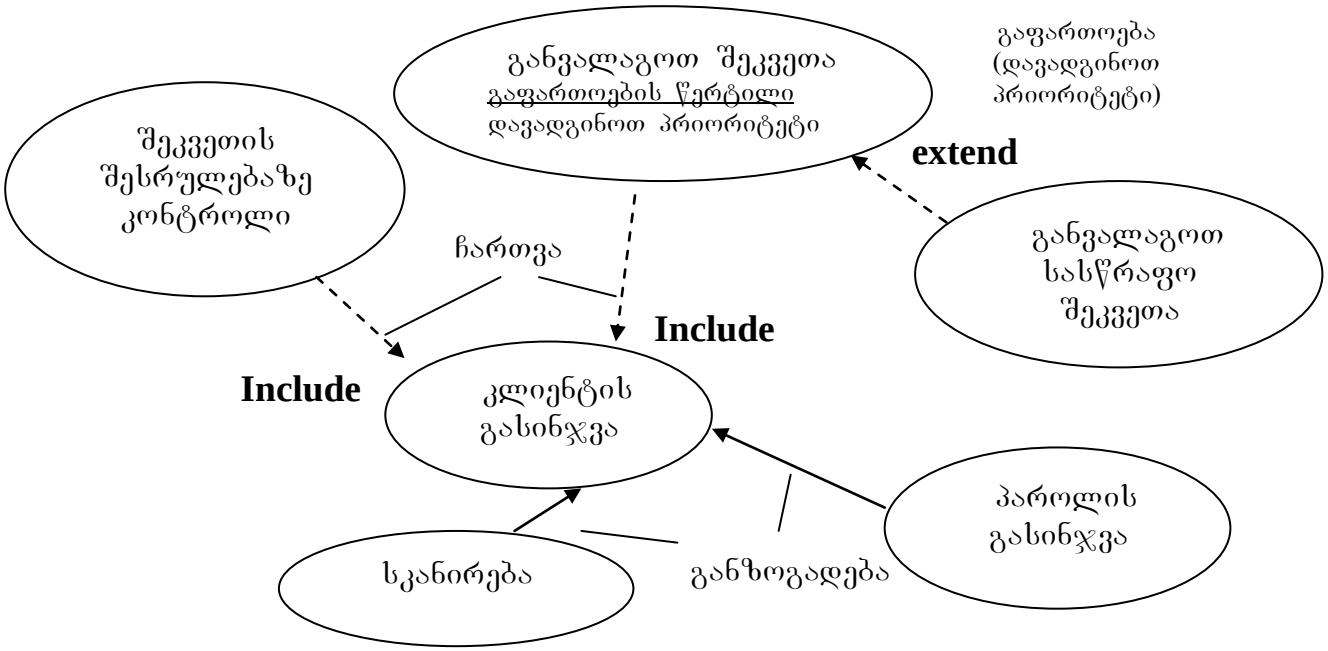
შვილობილი პრეცედენტი ავტომატურად იძენს თავისი მშობლის ყველა შესაძლებლობებს.

ჩართვის მიმართება პრეცენდენტებს შორის ნიშნავს, რომ ბაზური პრეცენდენტის გარკვეულ წერტილში თავმოყრილია მეორე პრეცენდენტის ქცევა. ჩართვადი პრეცენდენტი არასდროს არ არსებობს ავტონომიურად, არამედ განიხილება როგორც მომცველი პრეცენდენტის ნაწილი. შეიძლება ჩაითვალოს, რომ ბაზური პრეცენდენტი ითავსებს ჩართულების თვისებებს.

გაფართოების მიმართება გულისხმობს, რომ ბაზური პრეცენდენტი მოიცავს სხვა პრეცენდენტის ქცევას. ბაზური პრეცენდენტი შეიძლება იყოს ავტონომიურად, მაგრამ გარკვეულ პირობებში მისი ქცევა შეიძლება გაფართოვდეს მეორეს ხარჯზე.

მაგალითად, საბანკო სისტემაში შესაძლებელია არსებობა პრეცედენტის ***კლიენტის გასინჯვა***, რომელიც პასუხს აგებს კლიენტის პიროვნების შემოწმებაზე. აღნიშნული პრეცედენტი შესაძლებელია აღვწეროთ ნახ.2.2.1. - ზე მოყვანილი სახით.

როგორც მოყვანილი მაგალითებიდან ჩანს განზოგადება პრეცედენტებს შორის გამოისახება ისრიანი ხაზით, ჩართვის მიმართება გამოისახება ისრიანი წყვეტილი ხაზით სტერეოტიპით **include**, ხოლო გაფართოების მიმართება იგივე სახით სტერეოტიპით **extend**.



ნახ.2.2.1.

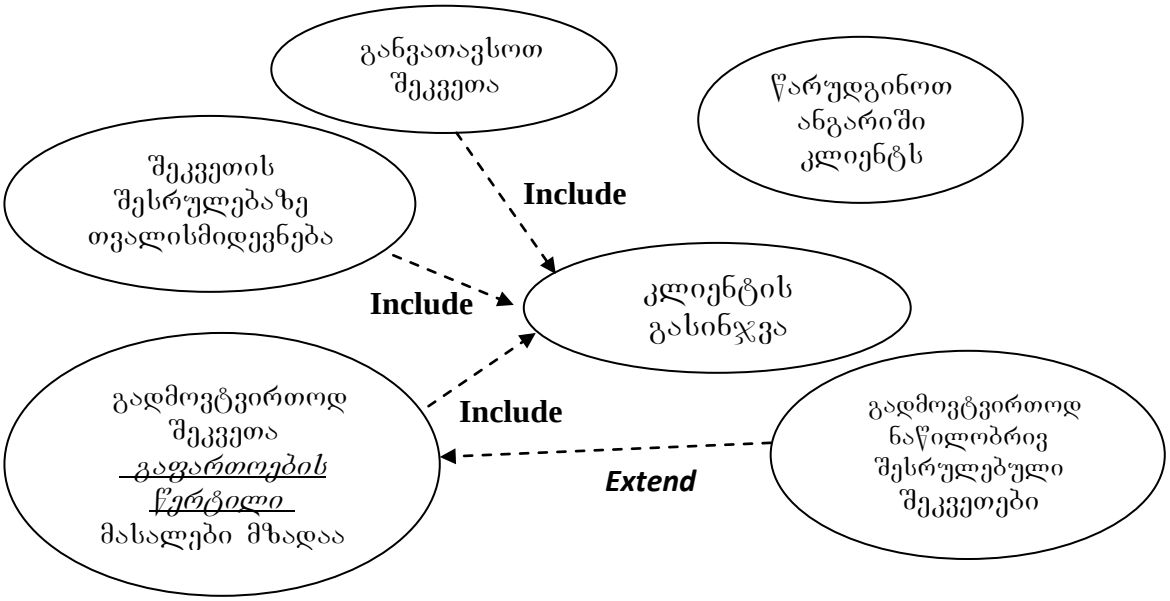
პრეცენდენდები წარმოადგენენ კლასიფიკატორებს და შეიძლება ქონდეთ ატრიბუტები და ოპერაციები. ატრიბუტები შეიძლება ჩაითვალოს ობიექტებათ პრეცენდენდების შიგნით, რომლებიც საჭიროა მისი გარე ქცევის აღწერისათვის, ხოლო ოპერაციები - სისტემის მოქმედებათ, რომლებიც საჭიროა მოვლენათა ნაკადის აღწერისათვის. ეს ობიექტები და ოპერაციები დაშვებულია ჩაირთოს ურთიერთქმედების დიაგრამებში, პრეცენდენტების ქცევის აღწერისათვის. ისევე როგორც სხვა კლასიფიკატორებს, პრეცედენტებს შესაძლებელია დაუკავშიროთ ავტომატები.

ამრიგად, პრეცენდენდები შეიძლება გამოვიყენოთ სისტემის ან კლასის მოდელირებისათვის. ელემენტის ქცევის მოდელირება ხდება შემდეგნაირად:

1. მოვახდინოთ მოცემულ ელემენტთან ურთიერთქმედებაში მყოფი აქტიორების იდენტიფიცირება.
2. მოვახდინოთ აქტიორების ორგანიზება, გამოვყოთ რა საერთო და სპეციალიზებული როლები.

3. ყოველი აქტიორისათვის განვიხილოთ ელემენტებთან მისი ურთიერთქმედების ძირითადი გზები. განვიხილოთ აგრეთვე ისეთი ურთიერთქმედებები, რომლებიც ცვლიან ელემენტის მდგომარეობებს.
4. განვიხილოთ აქტიორების ელემენტებთან ურთიერთქმედების ალტერნატიული საშუალებები.
5. მოვახდინოთ გამოვლენილი ქცევის ორგანიზება პრეცენდენტების სახით. გამოვიყენოთ რა ჩართვისა და გაფართოების მიმართებები საერთო და განსაკუთრებული ქცევის გამოყოფისათვის.

მაგალითად, საცალო ვაჭრობის სისტემა უნდა ურთიერთქმედებდეს კლიენტებთან, რომლებიც განალაგებენ შეკვეთებს და თვალი უნდა მიადევნონ მათ მოძრაობას. სისტემა გამოსცემს შესრულებულ შეკვეთებს და წარუდგენს ანგარიშებს კლიენტებს. ასეთი სისტემის მოდელირება შესაძლებელია პრეცენდენტების გამოცხადებით, რომელიც მოყვანილია ნახ.2.2.2. –ზე.



ნახ.2.2.2.

მოყვანილ მაგალითებში შესაძლებელია გამოვყოთ საერთო ქცევა (***კლიენტის გასინჯვა***) და ვარიაციები (***გადმოვტვირთოდ ნაწილობრივ შესრულებული შეკვეთები***). თითოეული მათგანისათვის უნდა ჩავრთოდ ქცევის სპეციფიკაცია ტექსტის სახით, ავტომატით ან ურთიერთქმედებით.

2.3. პრეცენდენტების დიაგრამა

პრეცენდენტების ან გამოყენებით შემთხვევათა დიაგრამას(Use case) უწოდებენ დიაგრამას, რომელზედაც ნაჩვენებია პრეცენდენტებისა და აქტიორების ერთობლიობა და აგრეთვე მიმართებები მათ შორის.

პრეცენდენტების დიაგრამა ჩვეულებრივ მოიცავს თავისში:

- პრეცენდენტებს;
- აქტიორებს;
- დამოკიდებულების, განზოგადების და ასოციაციის მიმართებებს.

პრეცენდენტების დიაგრამები გამოიყენება სისტემის სტატიკური სახის მოდელირებისათვის პრეცენდენტების თვალთახედვიდან. ამისათვის გამოყენებით შემთხვევათა დიაგრამები ძირითადათ ორი საშუალებით გამოიყენებიან:

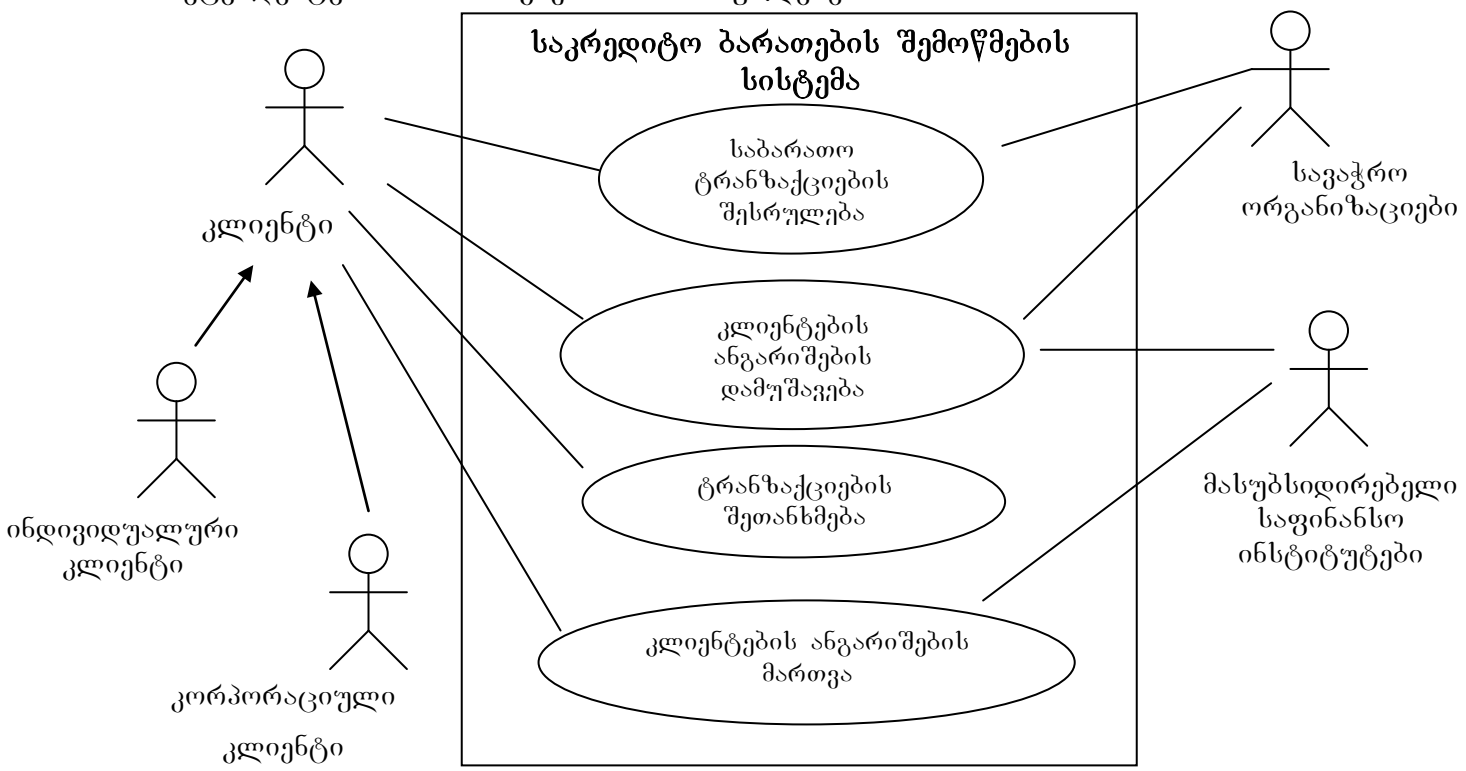
- ***სისტემის კონტექსტის მოდელირებისათვის***, რომელიც გულისხმობს რომ ჩვენ სისტემას შემოვავლებთ წარმოსახვით ხაზს და გამოვავლენთ აქტიორებს, რომლებიც იმყოფებიან ამ ხაზის იქით და ურთიერთქმედებენ სისტემასთან. პრეცენდენტების დიაგრამა ამ ეტაპზე საჭიროა აქტიორებისა და მათი როლების სემანტიკის იდენტიფიცირებისათვის.
- ***სისტემისადმი მოთხოვნილებების მოდელირებისათვის***. სისტემისადმი მოთხოვნილებების მოდელირება მიუთითებს, თუ რას უნდა აკეთებდეს სისტემა გარე მეთვალყურის თვალსაზრისით, იმისგან დამოუკიდებლად თუ როგორ უნდა აკეთებდეს იგი ამას. პრეცენდენტების დიაგრამა აქ საჭიროა სისტემის სასურველი ქცევის სპეციფიცირებისათვის. ისინი საშუალებას იძლევიან განვიხილოთ სისტემა როგორც "შავი ყუთი'. თქვენ ხედავთ ყველაფერს მის გარეთ, აკვირდებით მის რეაქციას მოვლენაზე, მაგრამ არაფერი იცით მის შინაგან მოწყობაზე.

სისტემის კონტექსტი. ნებისმიერი სისტემა თავის შიგნით შეიცავს გარკვეულ არსებს, მაშინ როდესაც სხვა არსები რჩებიან მის გარეთ. არსები სისტემის შიგნით პასუხს აგებენ ქცევის რეალიზებაზე, რომელსაც ელოდებიან არსები გარედან. არსები, რომლებიც იმყოფებიან სისტემის გარეთ და ურთიერთქმედებენ მასთან, შეადგენენ მის კონტექტს. მაშასადამე კონტექსტს უწოდებენ სისტემის გარემოცვას.

UML საშუალებას იძლევა მოვახდინოთ კონტექსის მოდელირება პრეცენდენტების დიაგრამის მეშვეობით, რომელშიც ყურადღება მიმართულია სისტემის გარემომცველ აქტიორებზე.. მნიშვნელოვანია სწორად განვსაზღვროთ აქტიორები, რამდენადაც ეს საშუალებას იძლევა ავღწეროთ არსების კლასები, რომლებიც ურთიერთქმედებენ სისტემასთან.

სისტემის კონტექსტის მოდელირება შედგება შემდეგი ბიჯებისაგან:

1. მოვახდინოთ სისტემის გარემომცველი აქტიორების იდენტიფიცირება. ამისათვის უნდა დადგინდეს ის ჯგუფები, რომლებისთვისაც სისტემის მონაწილეობაა საჭირო თავიანთი ამოცანების გადასაწყვეტად.
2. მოვახდინოთ მსგავსი აქტიორების ორგანიზება განზოგადებისა და სპეციალიზაციის მიმართებების გამოყენებით.
3. შემოვიტანოთ სტერეოტიპი ყოველი აქტიორისათვის თუ ეს გაადვილებს გაგებას.
4. მოათავსეთ აქტიორები პრეცენდენტების დიაგრამაზე და განსაზღვრეთ სისტემის პრეცენდენტებთან მათი კავშირის საშუალებები.



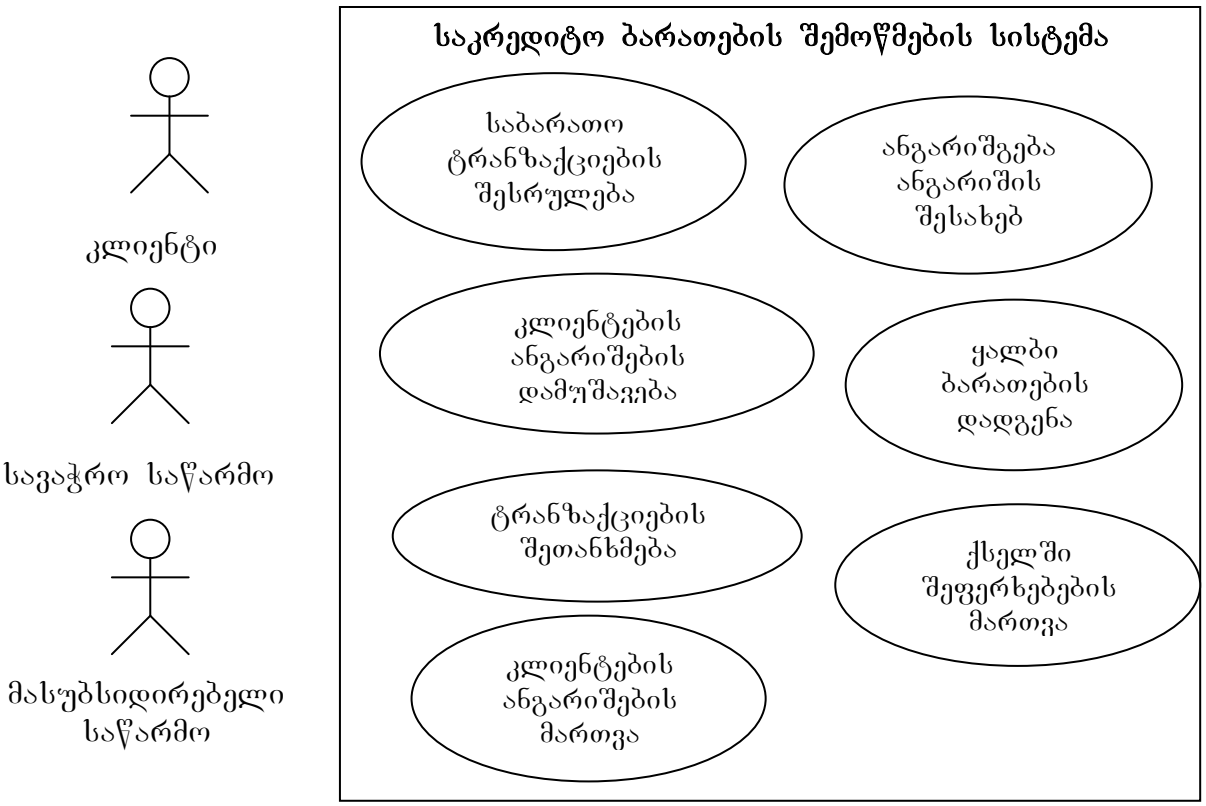
ნახ.2.3.1.

მაგალითად, ნახ.23.1. -ზე მოყვანილია სისტემის კონტექსტი, რომელიც მუშაობს საკრედიტო ბარათებთან, სადაც ძირითადი ყურადღება ექცევა მის გარემომცველ აქტიორებს. პირველ რიგში ეს კლიენტებია ორი სახის(ინდივიდუალური კლიენტი და კორპორაციული კლიენტი), რომლებიც შეესაბამებიან როლებს, რომლებსაც თამაშობენ ადამიანები სისტემასთან ურთიერთქმედებისას. ამ კონტექსში ნაჩვენებია ის აქტიორები, რომლებიც წარმოადგენენ სხვა ორგანიზაციებს, ისეთი როგორიც არის *სავაჭრო საწარმოები* (მათთან მყიდველები აწარმოებენ საბარათო ტრანზაქციებს, იძენს რა საგნებს ან მომსახურებას) და *მასუბსიდირებელი ფინანსური ინსტიტუტები* (ასრულებენ საკლირინგო პალატის როლს საბარათო ანგარიშებისათვის). რეალურ სამყაროში ბოლო ორი აქტიორი თვითონ იქნებიან პროგრამული სისტემები.

სისტემის მოთხოვნების მოდელირება ხდება შემდეგი სახით:

1. დავადგინოთ სისტემის კონტექსტი, მოვახდენთ რა გარემომცველი აქტიორების იდენტიფიცირებას.
2. ყოველი აქტიორისათვის განვიხილოთ ქცევა, რომელსაც ის ელოდება ან მოითხოვს სისტემისაგან.
3. დავასახელოთ ეს საერთო ვარიანტები როგორც პრეცედენტები.
4. გამოვყოთ საერთო ქცევა ახალ პრეცედენტებში, რომელიც გამოიყენება სხვების მიერ; გამოვყოთ ქცევის ვარიაციები ახალ პრეცენდენტებში, რომლებიც აფართოვებენ მოვლენათა ძირითად ნაკადს.
5. მოვახდინოთ პრეცენდენტების დიაგრამაზე ამ პრეცენდენტების, აქტიორებისა და მათ შორის მიმართებების მოდელირება.
6. დავამატოდ პრეცენდენტებს შენიშვნები, რომლებიც აღწერენ არაფუნქციონალურ მოთხოვნებს.

ნახ.2.3.2. –ზე მოყვანილი ნახაზი აფართოვებს წინამორბედს. მართლია კავშირები აქტიორებსა და პრეცედენტებს შორის მასზე არ არის გამოსახული, მაგრამ გამოსახულია დამატებით პრეცედენტები, რომლებიც აღწერენ სისტემის ქცევის მნიშვნელოვან ელემენტებს. ეს დიაგრამა იმით არის მოხერხებული, რომ საშუალებას იძლევა მომხმარებლებმა და დამმუშავებლებმა ერთობლივად ჩამოაყალიბონ სისტემისადმი ფუნქციონალური მოთხოვნები.



ნახ.2.3.2.

მაგალითად, პრეცედენტი ***ყალბი ბარათების დადგენა*** აღწერს ქცევას, რომელიც მნიშვნელოვანია როგორც ***სავაჭრო ორგანიზაციებისათვის***, ისე ***მასუბსიდირებელი ფინანსური ინსტიტუტებისათვის***. მეორე პრეცედენტი – ***ანგარიშგება ანგარიშის შესახებ*** – ასევე აღწერს ქცევას, რომელიც მოითხოვება სისტემისაგან სხვა ორგანიზაციების მიერ თავიანთ კონტექსტში. მოთხოვნები, რომლებიც მოდელირდება პრეცედენტით ***ქსელში შეფერხებების მართვა*** განსხვავდება დანარჩენებისაგან, რამდენადაც წარმოადგენს სისტემის დამხმარე ქცევას, რომელიც აუცილებელია მისი საიმედო და უწყვეტი ფუნქციონირებისათვის.

ძირითადათ პრეცედენტები აფიქსირებენ ფუნქციონალურ მოთხოვნებს, ამიტომ არაეფექტურებია ისეთი სისტემისათვის, რომლებშიც დომინირებს არაფუნქციონალური მოთხოვნები. შესაბამისად, მათი გამოყენება კარგია სისტემის ფუნქციონალობის განსაზღვრისათვის. ისინი ცუდათ ეთანადებიან - უხდებიან სისტემის შეზღუდვების გამოვლენას.

თავი 3
ანალიზი

უნიფიცირებული პროცესის მნიშვნელოვანი ეტაპია ანალიტიკური მოდელის შექმნა, რაც ქმნის ბაზას ობიექტ-ორიენტირებული ანალიზისა და შემდგომი უფრო დეტალური კვლევისათვის.

ანალიტიკურ მოდელირებას აქვს სტრატეგიულად დიდი მნიშვნელობა, რამდენადაც ამ ეტაპზე იქმნება მცდელობა მოვახდინოთ სისტემის ძირითადი ქცევის მოდელირება. ძირითადი ანალიტიკური სამუშაო იწყება ფაზის **დასაწყისი**ს ბოლოს, მოთხოვნების დადგენასთან ერთად ანალიზი წარმოადგენს ფაზის **დაზუსტება** მთავარ ამოცანას.

ფაზაში **დაზუსტება** ძირითადი ძალები მიმართულია მოდელების შექმნაზე, რომელიც ასახავენ სისტემის სასურველ ქცევას. აქტიურად ანალიზი იკვეთება მოთხოვნების განსაზღვრისას. ეს ორი მოღვაწეობა ხშირად მიმდინარეობს თანმიმდევრულად. ჩვეულებრივ მოთხოვნების დადგენასთან ერთად საჭიროა ჩავატაროთ მოთხოვნების ანალიზი, რათა გავხადოთ ისინი უფრო გასაგები და გამოვავლინოთ ყველა ხარვეზი.

3.1. ანალიზის სამუშაო ნაკადი

ობიექტ-ორიენტირებული ანალიზის თვალსაზრისით ანალიზის სამუშაო ნაკადის მიზანია – ანალიტიკური მოდელის შექმნა. მოცემული მოდელი ფოკუსირდება იმაზე, თუ რა უნდა აკეთოს სისტემამ. ხოლო, როგორ უნდა აკეთოს ეს სისტემამ ხდება დაპროექტების სამუშაო ნაკადისას.

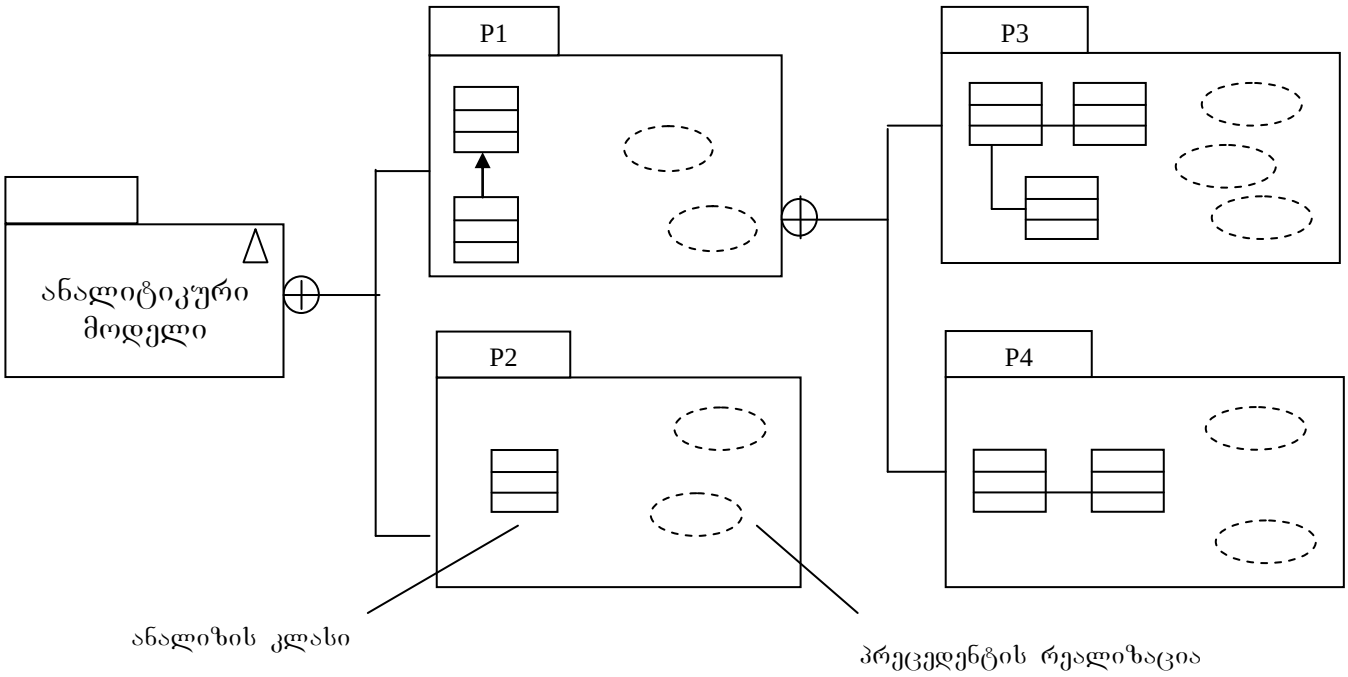
საზღვარი ანალიზსა და დაპროექტებას შორის საკმაოდ გაურკვეველია, ყველაფერი დამოკიდებულია ანალიტიკოსზე.

ანალიზის სამუშაო ნაკადში იქმნება ორი არტეფაქტი:

- ანალიზის კლასები – მნიშვნელოვანი მცნებები ბიზნეს – სფეროდან;
- პრეცედენტების რეალიზაცია – ახდენს იმის ილუსტრაციას, თუ ანალიზის კლასების ეგზემპლიარებს როგორ შეუძლიათ ურთიერთქმედება სისტემის ქცევის რეალიზებისათვის, რომელიც აღწერილია პრეცედენტებით.

UML-ის მეშვეობით შესაძლებლობა გვეძლევა დამოუკიდებლად შევქმნათ ანალიტიკური მოდელი. ანალიტიკური მოდელის მეტამოდელი მოყვანილია ნახ,3.1.-ზე. ანალიტიკურ მოდელს წარმოადგენენ პაკეტის სახით, სამკუთხედით ზედა მარჯვენა ბოლოში. ეს პაკეტი შეიცავს ერთ ან რამოდენიმე ანალიზის პაკეტს. ნახაზზე ნაჩვენებია მხოლოდ ანალიზის ოთხი პაკეტი, მაგრამ ანალიტიკური მოდელები შეიძლება შეიცავდნენ

უფრო მეტ პაკეტებს. თითოეულ პაკეტში, თავის მხრივ, შესაძლებელია ჩართული იყოს ანალიზის პაკეტი.



ნახ.3.1.

მაშასადამე, ანალიზის დანიშნულებაა მოდელების შექმნა, რომლებიც ახდენენ მოთხოვნების გამოვლენას – ანალიტიკურ მოდელირებას, რომელსაც აქვს სტრატეგიული მნიშვნელობა. ანალიზის ძირითადი სამუშაოები ტარდება ფაზის **დაწყება** დასასრულსა და ფაზაში **დაზუსტება.**

ანალიტიკური მოდელი იქმნება ენაზე, რომელიც შეესაბამება საპრობლემო სფეროს და ახდენს მოცემული საპრობლემო სფეროს მოდელირებას. საშუალო სიდიდის სისტემისათვის ანალიტიკური მოდელი შეიცავს 50-100 ანალიზის კლასს, რომელშიც შედის მხოლოდ ის კლასები, რომლებიც ახდენენ საპრობლემო სფეროს ლექსიკონის მოდელირებას.

3.2. ობიექტები და კლასები

კლასები - ეს ყველაზე მნიშვნელოვანი სამშენებლო ბლოკია ნებისმიერი ობიექტ-ორიენტირებული სისტემისათვის. მათი მეშვეობით აღიწერება პროგრამული, აპარატული ან სუფთა კონცეპტუალური არსები.

სისტემის მოდელირება გულისხმობს არსების იდენტიფიცირებას, რომლებიც მნიშვნელოვანია ამა თუ იმ თვალსაზრისით.

UML-ში ყველა ეს არსები მოდელირდება როგორც კლასები. კლასი ეს არსების აბსტრაქციაა, იგი წარმოადგენს არა ინდივიდუალურ ობიექტს, არამედ არსთა ერთობლიობას. აქედან გამომდინარე ჩვენ შეგვიძლია ჩავთვალოთ, რომ "კედელი" არის ობიექტების კლასი, საერთო თვისებებით, როგორიც არის სიმაღლე, სიგანე, სიგრძე, სისქე, მზიდი კედელია თუ არა და ა. შ. ამასთან ცალკეული კედლები განიხილება როგორც კლასი "კედელი"-ს ცალკეული ეგზემპლიარები.

მაშასადამე, ***კლასს*** (**CLASS**) უწოდებენ ობიექტთა ერთობლიობის აღწერას საერთო ატრიბუტებით, ოპერაციებით, მიმართებებით და სემანტიკით.

ყოველ კლასს უნდა გააჩნდეს **სახელი**, რომელიც განასხვავებს მას სხვა კლასებისაგან. კლასის სახელი - ეს ტექსტური სტრიქონია. სახელის გარდა კლასის დახასიათებისათვის გამოიყენება ატრიბუტები, ოპერაციები და მოვალეობები.

ატრიბუტი - კლასის დასახელებული თვისებაა, რომელიც შეიცავს მნიშვნელობათა სიმრავლეს, რომელსაც ღებულობენ ამ თვისების ეგზემპლიარები. კლასს შეიძლება ქონდეს ატრიბუტების ნებისმიერი რაოდენობა ან საერთოდ არ ქონდეს. ატრიბუტი წარმოადგენს სამოდელო არსების გარკვეულ თვისებას, რომელიც საერთოა მოცემული კლასის ყველა ობიექტისათვის. მაგ. კლიენტის მოდელირებისას შეიძლება მიუთითოთ გვარი, სახელი, მისამართი, ტელეფონი და დაბადების თარიღი. მაშასადამე, ატრიბუტი წარმოადგენს ობიექტის მონაცემების ან მისი მდგომარეობის აბსტრაქციას. დროის ყოველ მომენტში ობიექტის ნებისმიერ ატრიბუტს გააჩნია სავსებით გარკვეული მნიშვნელობა. ატრიბუტის სახელი, ისევე როგორც კლასის, შეიძლება იყოს ნებისმიერი ტექსტური სტრიქონი.

ოპერაციას უწოდებენ მომსახურების რეალიზაციას, რომელიც შეიძლება მოვითხოვოთ მოცემული კლასის ნებისმიერი ობიექტისაგან ქცევაზე ზემოქმედებისათვის. სხვა სიტყვებით რომ ვთქვათ, ოპერაცია – ეს აბსტრაქციაა იმისა, თუ რისი გაკეთება შეიძლება ობიექტზე. კლასს შეიძლება ქონდეს ოპერაციების ნებისმიერი რაოდენობა ან საერთოდ არ ქონდეს. ოპერაციის სახელი, ისევე როგორც კლასის, შეიძლება იყოს ნებისმიერი ტექსტური სტრიქონი. ოპერაციის აღწერისას შეიძლება მიუთითოთ პარამეტრები, მათი ტიპი და მნიშვნელობები, ფუნქციების შემთხვევაში – დასაბრუნებელი მნიშვნელობის ტიპი.

კლასის **მოვალეობები** - ეს თავისებურად კონტრაქტია, რომელსაც იგი უნდა დაემორჩილოს. განსაზღვრავთ რა კლასს, თქვენ უთითებთ, რომ ყველა მის ობიექტებს აქვთ ერთგვაროვანი მდგომარეობა და ქცევა. შესაბამისი ატრიბუტები და ოპერაციები წარმოადგენენ სწორედ იმ თვისებებს, რომლის მეშვეობითაც სრულდება კლასის

მოვალეობები. მაგალითად, კლასი კედელი პასუხს აგებს ინფორმაციაზე სიმაღლის, სიგანისა და სისქის შესახებ და ა.შ.

კლასის გრაფიკული გამოსახვისას მიუთითებენ როგორც სახელს, ისე ატრიბუტებსა და ოპერაციებს. მოვალეობებს გამოსახავენ კლასის ქვედა ნაწილის სპეციალურ განყოფილებაში. ისინი შეიძლება მიუთითოთ აგრეთვე შენიშვნებში. კლასები ობიექტ-ორიენტირებული სისტემების ყველაზე მნიშვნელოვანი სამშენებლო ბლოკებია. მაგრამ კლასები წარმოადგენენ მხოლოდ ერთ ნაირსახეობას UML-ის უფრო რთული სამშენებლო ბლოკის - კლასიფიკატორის.

პროგრამული სისტემის შექმნისას, მუშაობის საწყის ეტაპზე საკმარისია ითქვას, რომ თქვენ დაუშვათ იყენებთ კლასს ***კლიენტი,*** რომელსაც ეკისრება გარკვეული მოვალეობა. არქიტექტურის დაზუსტების შესაბამისად უნდა დაზუსტდეს კლასის სტრუქტურა(ატრიბუტები, ოპერაციები), რომლებითაც უზრუნველყოფილი იქნება ამ მოვალეობების შესრულება. ბოლოს, მოდელის შესრულებად სისტემად გარდაქმნისას, უნდა დადგინდეს ისეთი დეტალები, როგორიც არის ცალკეული ატრიბუტებისა და ოპერაციების ხედვა, კლასის პარალელიზმის სემანტიკა, ასევე კლასის მიერ რეალიზებადი ინტერფეისები.

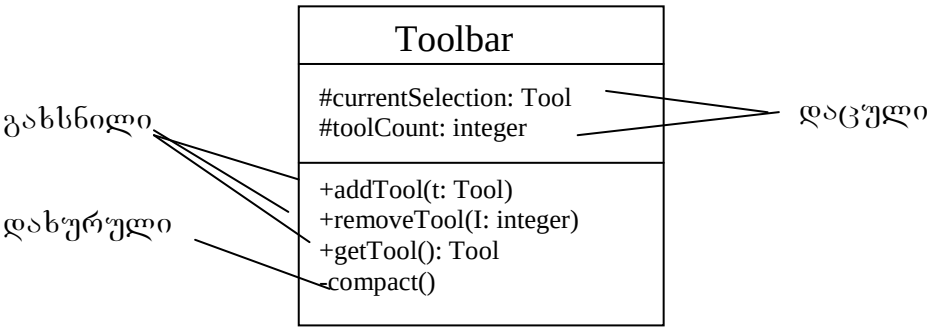
მექანიზმი, რომელიც აღწერს სტრუქტურულ და ქცევით თვისებებს უწოდებენ კლასიფიკატორს. კლასიფიკატორების რიცხვს მიეკუთვნება კლასები, ინტერფეისები, მონაცემთა ტიპები, სიგნალები, კომპონენტები, კვანძები, პრეცედენტები და ქვესისტემები. თითოეულ მათგანს გააჩნია ეგზემპლიარების ნებისმიერი რაოდენობა, მაშინ როდესაც UML-ის ზოგიერთ არსს ეგზემპლიარები არ გააჩნია(მაგ. პაკეტები, განზოგადების მიმართება). ამიტომ, მოდელირების ის ელემენტები, რომლებსაც შესაძლებელია ქონდეთ ეგზემპლიარები უწოდებენ კლასიფიკატორებს.

კლასიფიკატორების ყველაზე მნიშვნელოვანი სახეობაა კლასები, რომლებიც ხასიათდებიან დიდი რაოდენობის დამატებითი თვისებებით, გარდა იმ მარტივებისა, რომელიც განხილული იყო წინა თავებში. ამ თვისებებს მიეკუთვნება ხედვა, მოქმედების არე, ჯერადობა და სხვა.

ხედვა. ერთერთი დეტალი, რომელიც საკმაოდ მნიშვნელოვანია ატრიბუტებისა და ოპერაციებისათვის ეს არის მათი ხედვა. თვისების ხედვა მიუთითებს შეიძლება თუ არა იგი გამოყენებულ იქნას სხვა კლასიფიკატორების მიერ. ბუნებრივია, ეს გულისხმობს თვით კლასიფიკატორის ხედვას. ერთ კლასიფიკატორს შეუძლია "შეხედოს" მეორეს, თუ იგი იმყოფება პირველის მოქმედების სფეროში და მათ შორის არსებობს პირდაპირი ან უშუალო მიმართება. UML – ში არსებობს ხედვის სამი დონე:

- **public** (გახსნილი) – ნებისმიერი გარე კლასი, რომელიც "ხედავს" მოცემულს შეუძლია ისარგებლოს მისი გახსნილი თვისებებით. აღინიშნება ნიშნით +(პლიუსი) ატრიბუტის ან ოპერაციის წინ.
- **proteqted** (დაცული) – ნებისმიერ შვილობილს მოცემული კლასისა შეუძლია ისარგებლოს მისი დაცული თვისებებით. აღინიშნება №(დიეზ) ნიშნით.
- **private** (დახურული) – მხოლოდ მოცემულ კლასს შეუძლია ისარგებლოს დახურული თვისებებით. აღინიშნება სიმბოლოთი –(მინუსი).

კლასის თვისებების ხედვას განსაზღვრავენ იმისათვის, რომ დამალონ მისი რეალიზაციის დეტალები და უჩვენოთ მხოლოდ ის თავისებურებანი, რომლებიც აუცილებელია მოვალეობების განსახორციელებლად. სწორედ ამაში მდგომარეობს ინფორმაციის დახურვის მიზეზი, ურომლისოდ შეუძლებელია საიმედო და მოქნილი სისტემის შექმნა. თუ ხედვის სიმბოლო არ არის მითითებული, ჩვეულებრივ იგულისხმება, რომ თვისება არის გახსნილი.

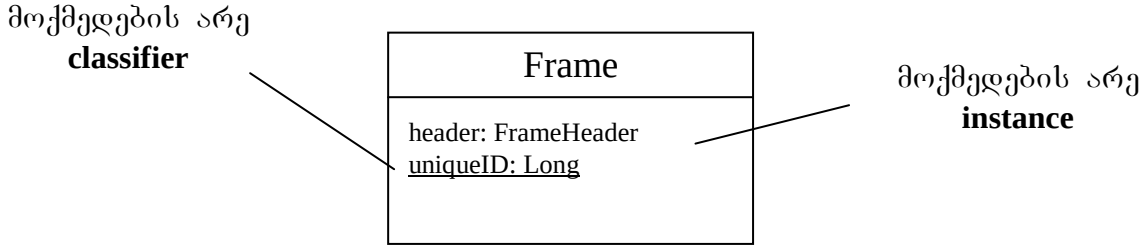


მოქმედების არე. კლასიფიკატორის ატრიბუტების და ოპერაციების კიდევ ერთ მნიშვნელოვან მახასიათებელს წარმოადგენს მოქმედების არე. რომელიმე თვისებისათვის მოქმედების არის მითითებით აღნიშნავენ, გამოავლენს თუ არა იგი თავის თავს სხვა სახით კლასის ყოველ ეგზემპლიარში, თუ თვისების ერთი და იმავე მნიშვნელობა ერთობლივად გამოიყენება ყველა ეგზემპლიარის მიერ. UML – ში განსაზღვრულია მოქმედებათა არის ორი სახე:

- instance (ეგზემპლიარი) – კლასის ყოველ ეგზემპლიარს აქვს მოცემული თვისების საკუთარი მნიშვნელობა.
- classifier (კლასიფიკატორი) – კლასის ყველა ეგზემპლიარები ერთობლივად იყენებენ მოცემული თვისების საერთო მნიშვნელობას.

როგორც წესი, სამოდელო კლასების თვისებებს აქვთ მოქმედების სფერო ეგზემპლიარი. თვისებები მოქმედების არით კლასიფიკატორი ყველაზე ხშირად გამოიყენება დაკეტილი ატრიბუტების აღწერისათვის.

ქვემოთ მოყვანილ ნახაზზე თვისება, რომელსაც აქვს მოქმედების არე classifier ხაზგასმულია, ხოლო თუ ხაზგასმული არ არის იგულისხმება მოქმედების არე instance**.**



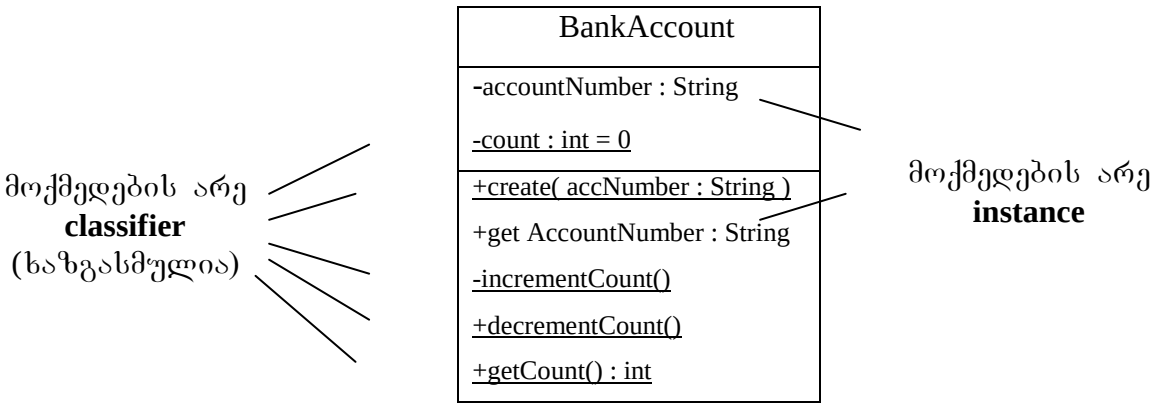
ატრიბუტები და ოპერაციები, რომელთა მოქმედების არე არის instance (ეგზემპლიარი), ეკუთვნიან ან სრულდებიან გარკვეული ობიექტით, ხოლო ატრიბუტები და ოპერაციები, რომელთა მოქმედების არე არის კლასი classifier (კლასიფიკატორი), ეკუთვნიან ან სრულდებიან მოცემული კლასის ყველა ობიექტებში.

ოპერაციის კლასის სხვა თვისებასთან მიმართვის შესაძლებლობა განისაზღვრება ოპერაციების მოქმედების არით და თვისების მოქმედების არით, რომელთანაც ცდილობენ მიიღონ მიმართვის უფლება.

ოპერაციები, რომელთა მოქმედების არე ეგზემპლიარია, შეუძლიათ მიმართვის ორგანიზება სხვა ატრიბუტებთან ან ოპერაციებთან ასეთივე მოქმედების არით, ასევე ყველა ატრიბუტებსა და ოპერაციებთან, რომელთა მოქმედების არე კლასია.

ოპერაციები, რომელთა მოქმედების არე კლასია, შეუძლიათ მიმართვის ორგანიზება მხოლოდ ატრიბუტებთან ან ოპერაციებთან მოქმედების არით კლასი, ოპერაციებს, რომელთა მოქმედების არე კლასია არ აქვთ მიმართვის შესაძლებლობა ოპერაციებთან მოქმედების არით ეგზემპლიარი, რადგან:

- შეიძლება ჯერ არ არის შექმნილი კლასის არც ერთი ეგზემპლიარი;
- თუ კლასის ეგზემპლიარები არსებობენ, უცნობია, რომელი იმათგანი გამოვიყენოთ.



ობიექტების შექმნა და მოსპობა. კონსტრუქტორები – სპეციალური ოპერაციებია, რომლებიც ქმნიან ახალ ობიექტებს, მათი მოქმედების არე არის კლასი. ისინი რომ ყოფილიყვნენ ეგზემპლიარი, შეუძლებელი იქნებოდა კლასის პირველი ეგზემპლიარის შექმნა.

კონსტრუქტორები დაპროექტების ზრუნვის საგანია. ჩვეულებრივ მათ არ უჩვენებენ ანალიტიკურ მედელებში.

დაპროგრამების სხვადასხვა ენებში მოქმედებს კონსტრუქტორების დასახელების სხვადასხვა სტანდარტები. უნივერსალური მიდგომაა კონსტრუქტორს უწოდონ create( ) (შეიქმნას). მაგრამ ენები Java, C# და C+ ითხოვენ, რომ კონსტრუქტორის დასახელება ემთხვეოდეს კლასის დასახელებას.

კლასებს შესაძლებელია გააჩნდეს რამოდენიმე კონსტრუქტორი ერთნაირი სახელებით, მაგრამ სხვადასხვა პარამეტრების ნაკრებით. კონსტრუქტორს პარამეტრების გარეშე უწოდებენ დუმილით გამოყენებულ კონსტრუქტორს. კონსტრუქტორის პარამეტრები გამოიყენებიან ატრიბუტების მნიშვნელობების ინიციალიზაციისათვის ობიექტების შექმნის მომენტში.

ნახ.2.-ზე მოყვანილია მარტივი მაგალითი კლასისა BankAccount(საბანკო ანგარიში). ობიექტის BankAccount ყოველი შექმნისას პარამეტრის სახით მის კონსტრუქტორში უნდა გადაიცეს String ტიპის მნიშვნელობა. ეს სტრიქონი გამოიყენება ატრიბუტის accountNumber(ანგარიშის ნომერი) შესაქმნელად. მაგალითად, მნიშვნელობა ”XYZ001002”. ის ფაქტი, რომ კლასი BankAccount აუცილებელია პარამეტრი მეტყველებს იმაზე, რომ BankAccount ობიექტის შექმნა არ შეიძლება ამ პარამეტრის დაუდგენლად. ეს გარანტირებულია ყოველი ობიექტისათვის BankAccount ატრიბუტის accountNumber მიცემით ობიექტის შექმნის მომენტში.

ობიექტის მოსპობა არც თუ ისე მარტივი ოპერაციაა როგორც მისი შექმნა. დესტრუქტორები სპეციალური ოპერაციაებია, რომლებიც “ამყარებენ წესრიგს” ობიექტების მოსპობისას.

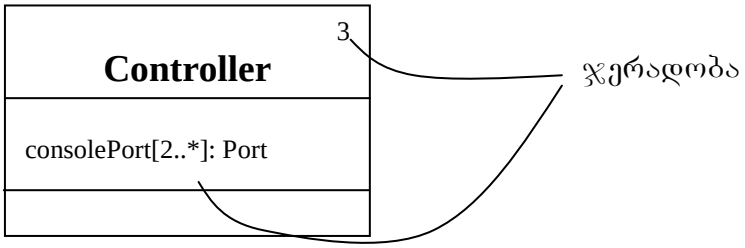
სხვადასხვა ობიექტ-ორიენტირებული ენებში ობიექტების მოსპობის სემანტიკა სხვადასხვაა. საერთოდ კი უმეტესობა ენებში დესტრუქტორები ავტომატურად გამოიძახება ობიექტის მოსპობის მომენტში. მაგალითად, C++ - ში მოსპობის ოპერაცია *გარანტირებულად* გამოიძახება ობიექტის მოსპობის მომენტში.

Java – შიც არის ანალოგიური შესაძლებლობა: ყოველ კლასს აქვს ოპერაცია დასახელებით finalize(), რომელიც გამოიძახება ობიექტის ობიექტის საბოლაოდ მოსპობისას. მაგრამ თვით პროგრამა აშკარად არ სპობს ობიექტებს. ამით დაკავებულია

ნაგვის ავტომატურად ამკრეფი. თქვენთვის ცნობილია, რომ finalize() გამოძახებულ იქნება, მაგრამ, არ იცით, ეს როდის მოხდება.

ჯერადობა. კლასებთან მუშაობისას იგულისხმება, რომ შეიძლება არსებობდეს მისი ნებისმიერი რაოდენობის ეგზემპლიარები. თუ რასაკვირველია ის არ არის აბსტრაქტული კლასი, რომელსაც საერთოდ არ გააჩნია ეგზემპლიარები, თუმცა მის შვილობილებს შეიძლება გააჩნდეთ ნებისმიერი რაოდენობით.

კლასის ეგზემპლიარების რაოდენობას უწოდებენ მის ჯერადობას. კლასის ჯერადობა მიეთითება გამოსახულებით ზედა მარჯვენა კუთხეში. ჯერადობა გამოიყენება ატრიბუტების მიმართაც. ატრიბუტის ჯერადობა იწერება გამოსახულების სახით კვადრატულ ფრჩხილებში და განთავსდება ატრიბუტის დასახელების შემდეგ.



3.3. ანალიზის კლასები და მათი გამოვლენა

ანალიზის კლასების წარმოადგენენ UP-ს "პრეცედენტების ანალიზის" მოღვაწეობის შედეგს. მათი მეშვეობით ხდება საპრობლემო სფეროს მნიშვნელოვანი ასპექტების მოდელირება.

ანალიზის კლასები წარმოადგენენ "მაღალდონიან" ატრიბუტების ნაკრებს, რომლებიც *შესაძლოა* იმყოფებოდნენ საპროექტო კლასებში. შეიძლება ითქვას, რომ ანალიზის კლასები შეიცავენ საპროექტო კლასების მოსალოდნელ ატრიბუტებს. ანალიზის კლასის მინიმალური ფორმა ასეთია:

- დასახელება – აუცილებელია და უნდა ასახავდეს მის დანიშნულებას.
- ატრიბუტები – ატრიბუტების დასახელება აუცილებელია, მხოლოდ მნიშვნელოვანი მოსალოდნელი ატრიბუტების. ატრიბუტების ტიპები არ არის აუცილებელი.
- ოპერაციები – ანალიზში ოპერაციები შეიძლება იყვნენ კლასის მოვალეობების მიახლოებითი ფორმულირებიდან გამომდინარე. პარამეტრები და ოპერაციის დასაბრუნებელი ტიპები მიეთითება მხოლოდ იმ შემთხვევაში, თუ ისინი მნიშვნელოლოვანია მოდელის გაგებისათვის.

- ხედვა – ჩვეულებრივ არ მიეთითება.
- სტერეოტიპები – შეიძლება მიეთითოს იმ შემთხვევაში, თუ ისინი აუმჯობესებენ მოდელს.
- მონიშნული მნიშვნელობები - შეიძლება მიეთითოს იმ შემთხვევაში, თუ ისინი აუმჯობესებენ მოდელს.

ანალიზის კლასის მაგალითი მოყვანილია ნახ.2.-ზე. ანალიზის კლასის ძირითადი დანიშნულება იმაშია, რომ გამოვავლინოთ აბსტრაქციის არსი, ხოლო რეალიზაციის დეტალები რჩება დაპროექტების ეტაპისათვის.

კლასების გამოვლენა. კლასების მეშვეობით ჩვეულებრივ გამოხატავენ აბსტრაქციებს, რომლებიც გამოიყოფა გადასაწყვეტი ამოცანიდან და გამოიყენება მის გადასაწყვეტად. ასეთი აბსტრაქციები წარმოადგენენ ჩვენი სისტემის ლექსიკონს ე.ი. წარმოადგენენ არსებს, რომლებიც მნიშვნელოვანია მომხმარებლებისა და დამმუშავებლისათვის.

სისტემის ლექსიკონის მოდელირება მოიცავს შემდეგ ეტაპებს:

- განვსაზღვროთ რომელ ელემენტებს იყენებენ მომხმარებლები და დამმუშავებლები ამოცანის აღწერისა და მისი გადაწყვეტისათვის.
- სწორი აბსტრაქციების მოძებნისათვის გამოვიყენოთ პრეცენდენტების ანალიზი.
- ყოველი აბსტრაქციისათვის დავადგინოთ მისი შესაბამისი მოვალეობების სიმრავლე.
- დავამუშავოთ ატრიბუტები და ოპერაციები, რომლებიც აუცილებელია კლასების მიერ თავიანთი მოვალეობის შესასრულებლათ.

აბსტრაქციების გამოყოფის ყველაზე მარტივი საშუალებაა ანალიზი არსებითი სახელი/ზმნა. იგი ითვალისწინებს ტექსტის ანალიზს – არსებითი სახელები და სახელობითი ჯგუფები მიუთითებენ კლასებზე და ატრიბუტებზე, ხოლო ზმნები და ზმნური ჯგუფები მოვალეობებსა და ოპერაციებზე. მაგ. საცალო ვაჭრობის სისტემისათვის გვექნება:

- არსებითი სახელი – კლიენტი, შეკვეთა, საქონელი.
- დასახელებითი ჯგუფები – მისამართი, რაოდენობა, ფასი.
- ზმნები – მიღება, განლაგება.
- ზმნური ჯგუფები – შემოწმდეს შეკვეთების შესრულება.

სისტემის ასეთი ლექსიკონის შექმნის შემდეგ ხდება შემავალი არსების მოვალეობების განაწილება. პრინციპში კლასის მოვალეობების რაოდენობა შეიძლება იყოს ნებისმიერი, მაგრამ კარგათ სტრუქტურირებულ კლასს აქვს სულ მცირე ერთი მოვალეობა. მეორეს მხრივ, მათი რაოდენობა არ უნდა იყოს ძალიან დიდი. მოდელის

დაზუსტებისას კლასის მოვალეობები გარდაიქმნებიან ატრიბუტებისა და ოპერაციების ერთობლიობაში, რომლებმაც უნდა უზრუნველყონ მათი შესრულება.

მაგ. საცალო ვაჭრობის სისტემისათვის გვექნება შემდეგი კლასები:

შეკვეთა
საქონელი რაოდენობა

კლიენტი
სახელი მისამართი ტელეფონი

საქონელი
იდენტიფიკატორი დასახელება ფასი ადგილმდებარეობა

უნდა მოვერიდოთ ძალიან დიდი ან ძალიან მცირე კლასების შექმნას. ყოველი კლასი კარგათ უნდა აკეთებდეს ერთ გარკვეულ მოვალეობას. მოვალეობების განაწილების მოდელირება სისტემაში მოიცავს შემდეგ ეტაპებს:

- მოახდინეთ იმ კლასების იდენტიფიცირება, რომლებიც ერთობლივად აგებენ პასუხს გარკვეულ მოქმედებაზე.
- განსაზღვრეთ ყოველი კლასის მოვალეობა.
- შეხედეთ მიღებულ კლასებს როგორც ერთ მთლიანს, გამოყავით მათგან ისინი, რომლებსაც ძალიან ბევრი მოვალეობები აქვთ და დაყავით შედარებით მცირე კლასებათ – პირიქით, პატარა კლასები ელემენტარული მოვალეობებით, გააერთიანეთ უფრო დიდში.
- გადაანაწილეთ მოვალეობები ისე, რომ ყოველი აბსტრაქცია გახდეს ავტონომიური.
- განიხილეთ თუ როგორ კოოპერირებენ კლასები ერთმანეთთან და გადაანაწილეთ მოვალეობები ისეთი ანგარიშით, რომ არც ერთი კლასი კოოპერაციის ჩარჩოში არ აკეთებდეს ძალიან ბევრს ან ძალიან ცოტას.

მაგალითისათვის მოვიყვანოთ მოვალეობების განაწილება კლასებს შორის მოდელი, ხედი, კონტროლერი.

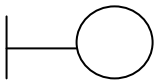
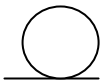
ხედი
მოვალეობები • გამოვსახოთ მოდელი ეკრანზე • ვმართოთ მზერის ველის ზომის ცვლილება და გადაადგილება

მოდელი
მოვალეობები • მოდელის მდგომარეობების მართვა

კონტროლერი
მოვალეობები • მოვახდინოთ მოდელის ცვლილებების სინქრონიზირება

კლასების გამოვლენის დამატებითი საშუალებაა სტერეოტიპების მეთოდი. ამ მეთოდის არსი იმაშია, რომ ანალიზის პროცესში განიხილება ანალიზის კლასების სამი ტიპი: სტერეოტიპით “boundary”(მოსაზღვრე), “control”(მართვის) და “entity”(არსი). სტერეოტიპებით, რომლებიც მოყვანილია ცხრ.1., შესაძლებელია აღნიშნული იქნას ანალიზის კლასების სამივე ტიპი.

ცხრ.1.

“boundary”		კლასი, რომელიც წარმოადგენს შუამავალს სისტემასა და მის გარემოცვას შორის
“control”		კლასი, რომელშიც ინკაფსულირებულია პრეცედენტისათვის დამახასიათებელი ქცევა
“entity”		კლასი, რომელიც გამოიყენება მუდმივი ინფორმაციის მოდელირებისათვის

“boundary” ტიპის კლასები ძირითადათ არსებობენ სისტემის საზღვარზე და ურთიერთობენ გარე აქტიორებთან. ასეთი კლასები შესაძლებელია გამოვავლინოთ სისტემის კონტექსტის განხილვისას და გავარკვიოთ, თუ რომელი კლასები არიან შუამავლები სისტემის კონტექსტსა და მის გარემოცვას შორის. ასეთი შეიძლება იყოს:

- მომხმარებელთა ინტერფეისების კლასები – კლასები, რომლებიც აკავშირებენ სისტემას და ადამიანებს.
- სისტემური ინტერფეისების კლასები – კლასები, რომლებიც აკავშირებენ სისტემას სხვა სისტემასთან.
- აპარატული ინტერფეისების კლასები – კლასები, რომლებიც აკავშირებენ სისტემას გარე მოწყობილობებთან, მაგალითად მიმღებთან.

ყოველი კავშირი აქტიორსა და პრეცედენტს შორის მოდელში უნდა წარმოდგენილი იქნას სისტემის გარკვეული ობიექტით. ეს ობიექტები – ეგზემპლიარებია მოსაზღვრე კლასების.

“control” ტიპის კლასები წარმოადგენენ მმართველს – მათი ეგზემპლიარები ახდენენ სისტემის ქცევის კოორდინაციას, რომელიც შეესაბამება ერთ ან რამოდენიმე პრეცედენტს.

მმართველი კლასები გამოვლინდებიან სისტემის ქცევის განხილვისას, რომელიც აღწერილია პრეცედენტებით. მარტივი ქცევა ხშირად შესაძლებელია გადავანაწილოთ მოსაზღვრე ან კლას-არსებზე. მაგრამ უფრო რთული ქცევისას, როგორიც არის *შეკვეთების დამუშავება* გამოვიყენოთ მმართველი კლასი. ხშირად მმართველი კლასის აღნიშვნისას მათ დასახელებას ამატებენ სიტყვებს Maneger ან Controller.

“entity” ტიპის კლასები ახდენენ მარტივი ინფორმაციის მოდელირებას, რომელიც ძირითადათ შედგება მნიშვნელობების მიღებასა და მინიჭებაში. კლასები, რომლებიც წარმოადგენენ მუდმივ ინფორმაციას, მაგალითად მისამართები (კლასი Address) და ადამიანები(კლასი Person), არიან კლასი-არსები.

კლასი – არსები მონაწილეობენ სხვადასხვა პრეცედენტებში, მათთან ოპერირებს მმართველი კლასი, ღებულობს ან აწვდის ინფორმაციას მოსაზღვრე კლასებს. ისინი გამოსახავენ სისტემის მონაცემების ლოგიკურ სტრუქტურას.

კლასების მოდელირებისას უნდა დავიმახსოვროთ, რომ ყოველ კლასს უნდა შეესაბამებოდეს გარკვეული არსება და კონცეპტუალური აბსტრაქცია იმ სფეროდან, რომელთანაც საქმე აქვს მომხმარებელს ან დამმუშავებელს. კარგათ სტრუქტურირებულ კლასს აქვს შემდეგი თვისებები:

- წარმოადგენს გარკვეული მცნებების მკვეთრად გამოხატულ აბსტრაქციას საპრობლემო სფეროს ლექსიკონიდან;
- შეიცავს გარკვეულ მოვალეობების ნაკრებს და ასრულებს ყოველ მათგანს;
- უზრუნველყოფს მკვეთრ გამიჯვნას აბსტრაქციების სპეციფიკაციებსა და მათ რეალიზაციას შორის;
- გასაგები და მარტივია, მაგრამ ამავე დროს დასაშვებია გაფართოებისა და ადაპტაციის შესაძლებლობა.

  კლასების გამოხატვისას დაცული უნდა იქნას შემდეგი წესები:

  - უჩვენეთ მხოლოდ ის თვისებები, რომლებიც მნიშვნელოვანია აბსტრაქციების გაგებისათვის მოცემულ კონტექსში.
  - დაყავით ატრიბუტების და ოპერაციების გრძელი ცხრილები ჯგუფებათ მათი კატეგორიების შესაბამისად;
  - უჩვენეთ ურთიერთდაკავშირებული კლასები ერთ და იმავე დიაგრამაზე.

3.4. მიმართებები

აბსტრაქციების აგებისას დავრწმუნდით, რომ კლასები იშვიათად არსებობენ ავტონომიურად. როგორც წესი ისინი სხვადასხვა საშუალებით ურთიერთქმედებენ ერთმანეთთან. ეს იმას ნიშნავს, რომ სისტემის მოდელირებისას თქვენ არა მარტო უნდა მოახდინოთ არსების იდენტიფიცირება, არამედ უნდა აღწეროთ თუ როგორ დამოკიდებულებაში არიან ერთმანეთთან. არსებობს მიმართებების სამი სახე, რომლებიც განსაკუთრებით მნიშვნელოვანია ობიექტ - ორიეტირებული მოდელირებისათვის:

დამოკიდებულება, რომლითაც აღიწერება კლასებს შორის არსებული გამოყენების მიმართებები.

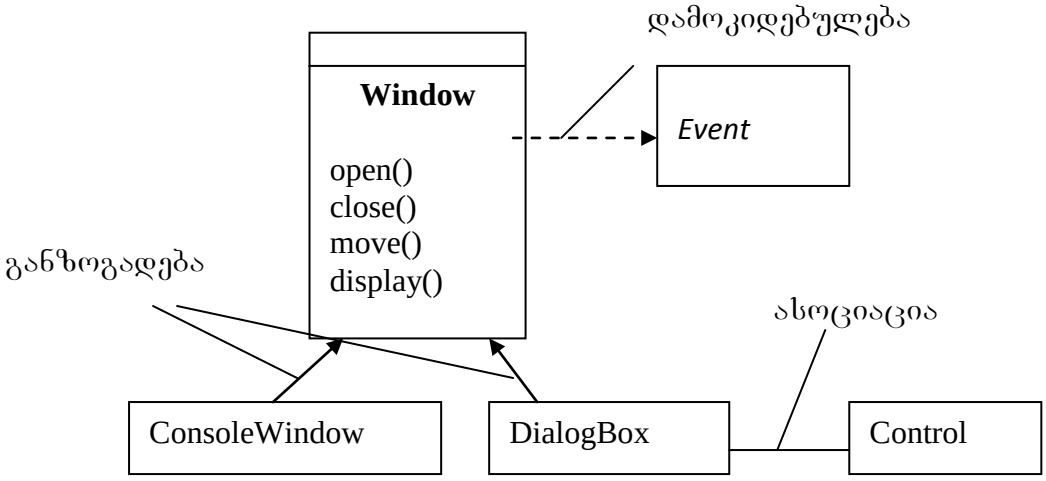
განზოგადება, რომლებიც აკავშირებენ განზოგადებულ კლასებს სპეციალიზირებულთან.

ასოციაცია, რომლებიც აღწერენ ობიექტებს შორის სტრუქტურულ კავშირებს.

ყოველი მათგანი აბსტრაქციების სხვადასხვა სახით კომბინირების საშუალებას იძლევა.

UML ენაში საშუალებები, რომლებითაც ელემენტები უკავშირდებიან ერთმანეთს, მოდელირდებიან მიმართებების სახით. ეს სამი მიმართება მოიცავს ელემენტების ურთიერთქმედების საშუალებების უდიდეს ნაწილს, რაც კარგად გამოისახება ობიექტებს შორის კავშირების საშუალებებით, რომლებიც მიღებულია დაპროგრამების ობიექტ-ორიენტირებულ ენებში.

თითოეული დასახელებული მიმართებისათვის არსებობს გრაფიკული გამოსახვა, რომელიც ნაჩვენებია ქვემოთ მოყვანილ მაგალითზე. ეს ნოტაცია საშუალებას გვაძლევს მოვახდინოთ სამოდელირო მიმართებების ვიზუალირება გამოყენებელი დაპროგრამების ენისაგან დამოუკიდებლად, ისე რომ ხაზი გაესვას მათ ყველაზე მნიშვნელოვან შემადგენლებს: სახელს, დამაკავშირებელ არსებს და თვისებებს.



3.4.1. დამოკიდებულება

დამოკიდებულება ეს არის გამოყენების მიმართება, რომლის მიხედვითაც ერთი ელემენტის სპეციფიკაციების ცვლილებას (მაგ. Event), შეიძლება გავლენა იქონიოს მეორე ელემენტზე, რომელიც მას იყენებს (მოცემულ შემთხვევაში კლასი Window), ამასთან პირიქით არ არის აუცილებელი.

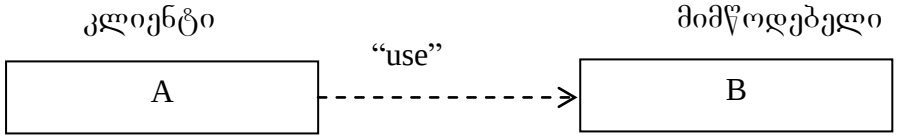
ყველაზე ხშირად დამოკიდებულებას იყენებენ კლასებთან მუშაობისას, იმისათვის რომ გამოვსახოთ ოპერაციათა სიგნატურაში ის ფაქტი, რომ ერთი კლასი იყენებს მეორეს არგუმენტის სახით ე.ი. ერთი კლასში მომხდარი ცვლილება გამოისახება მეორეს მუშაობაზე. მაგალითად, გათბობის მილები დამოკიდებულია გამათბობლისგან, რომელიც აცხელებს წყალს.

UML2-ში განსაზღვრულია სამი ტიპის დამოკიდებულება – გამოყენების(Usage), აბსტრაქცია(abstraqction), ხელმისაწვდომი(permission).

გამოყენების თვალსაზრისით არსებობს დამოკიდებულების მიმართების ხუთი სახეობა(სტერეოტიპი):

დამოკიდებულება „use“, გვიჩვენებს, რომ კლიენტი გარკვეული საშუალებით იყენებს მიმწოდებელს. ნახ.1.-ზე მოყვანილია ორი კლასი A და B, რომელთა შორის დამყარებულია მიმართება დამოკიდებულება „use“. ეს დამოკიდებულება გენერირდება ნებისმიერი შემდეგი სიტუაციებისას:

1. კლასი A ოპერაციას ესაჭიროება B კლასის პარამეტრი.
2. A კლასის ოპერაცია აბრუნებს B კლასის მნიშვნელობას.



3. A კლასის ოპერაცია სადღაც თავისი რეალიზაციაში იყებს B კლასის ობიექტს, მაგრამ არა ატრიბუტის სახით.

მართალია ერთი დამოკიდებულება „use“ შესაძლებელია გამოყენებულ იქნას სამივე შემთხვევისათვის, არის სხვა, უფრო სპეციალიზირებული დამოკიდებულების სტერეოტიპები, რომლებიც ასევე არის შესაძლებელი გამოყენებულ იქნან. პირველი და მეორე შემთხვევაში შესაძლებელია გამოვიყენოთ სტერეოტიპი „parameter“, ხოლო მესამე სიტუაციაში დამოკიდებულება „call“.

დამოკიდებულება „call“ (გამოძახება) მყარდება ოპერაციებს შორის – ოპერაცია კლიენტი იძახებს ოპერაცია – მიმწოდებელს. დამოკიდებულების ეს ტიპი გამოიყენება არც ისე ხშირად, რადგან გამოიყენება დეტალიზაციის უფრო დაბალ დონეზე, ვიდრე დამმუშავებელთა უმრავლესობა ქმნის UML-მოდელს.

დამოკიდებულება „parameter“ გამოიყენება იმ შემთხვევაში, როდესაც მიმწოდებელი წარმოადგენს პარამეტრს კლიენტის ოპერაციისათვის.

დამოკიდებულება „send“ გამოიყენება იმ შემთხვევაში, როდესაც კლიენტი - ეს ოპერაციაა, რომელიც აგზავნის მიმწოდებელს(რომელიც უნდა იყოს სიგნალი) რომელიმე გაურკვეველი მიზნისთვის. მას განიხილავენ როგორც კლასის განსაკუთრებულ სახეს, რომელიც გამოიყენება კლიენტსა და მიზანს შორის მონაცემების გადაცემისათვის.

დამოკიდებულება „send“ კიდევ გამოიყენება იმ შემთხვევაში, როდესაც კლიენტი – მიმწოდებლის ეგზემპლიარია.

აბსტრაქციის დამოკიდებულებებით ახდენენ დამოკიდებულების მოდელირებას არსებს შორის, რომლებიც იმყოფებიან აბსტრაქციის სხვადასხვა დონეებზე. აბსტრაქციის დამოკიდებულების ოთხი სახე არსებობს: „trace“, „substitute“, „refine“ და „derive“.

დამოკიდებულება „trace“ გამოიყენება იმ შემთხვევაში, როდესაც კლიენტი და მიმწოდებელი წარმოადგენენ ერთ მცნებას, მაგრამ იმყოფებიან სხვადასხვა მოდელებში. მისგან განსხვავებით დამოკიდებულება „refine“ (დაზუსტება) შესაძლებელია გამოყენებულ იქნას ერთი და იმავე მოდელების მიმართ.

დამოკიდებულება „substitute“ (ჩავანაცვლოთ) გვიჩვენებს, რომ კლიენტმა შესრულებისას შესაძლებელია ჩაანაცვლოს მიმწოდებელი. ჩანაცვლება ეფუძნება კლიენტისა და მიმწოდებლის კონტრაქტებისა და ინტერფეისების ერთობას, ე.ი. ისინი წარმოადგენდნენ სერვისების ერთ და იგივე ნაკრებს.

დამოკიდებულება „derive“ გამოიყენება იმ შემთხვევაში, როდესაც აუცილებელია ნათლად მიუთითოდ ერთი არსების მიღება როგორც წარმოებული მეორესგან.

ხელმისაწვდომობის დამოკიდებულებებით გამოხატავენ ერთი არსების ხელმისაწვდომობის შესაძლებლობას მეორესთან. არსებობს ხელმისაწვდომობის დამოკიდებულების სამი სახე: „access“. „import“ და „permit“.

დამოკიდებულება „access“ გამოიყენება პაკეტებს შორის. იგი საშუალებას აძლევს ერთ პაკეტს მიწვდეს მეორე პაკეტის გახსნილი შედგენილობით.

დამოკიდებულება „import“ კონცეპტუალურად ანალოგიურია „access“, განსხვავებით იმისა, რომ მიმწოდებლის სახელების სივრცე ერთიანდება კლიენტის სახელების სივრცესთან. ეს საშუალებას აძლევს კლიენტის ელემენტებს მოახდინონ წვდომა მიმწოდებლების ელემენტებთან პაკეტის დასახელების მიუთითებლად ელემენტის დასახელებაში.

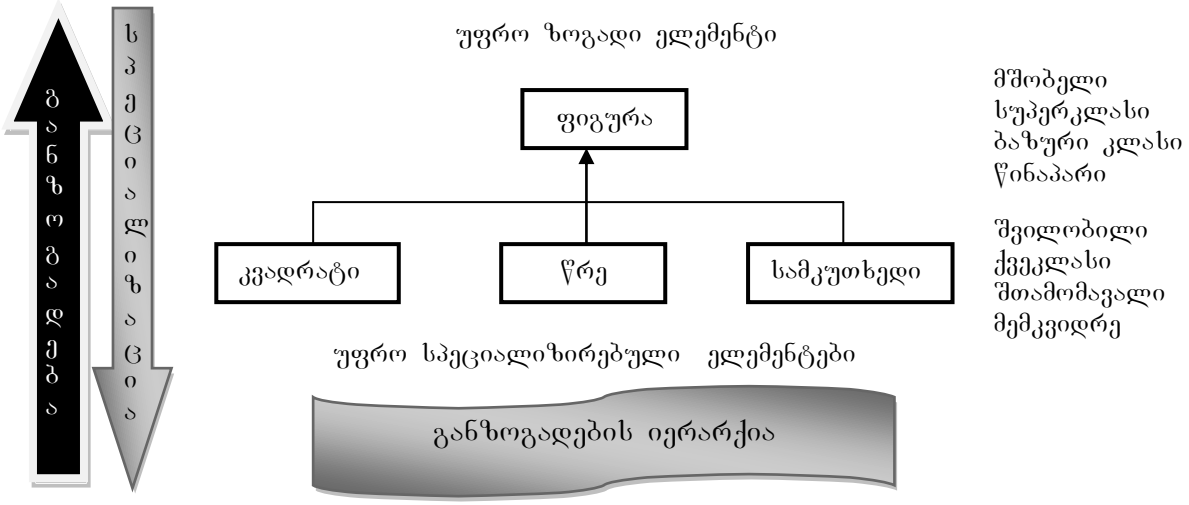
დამოკიდებულება „permit“ (ნება დაერთოს) უზრუნველყოფს ინკაფსულაციის მართვადი დარღვევის შესაძლებლობას. კლიენტურ ელემენტს აქვს წვდომა მიმწოდებელ ელემენტთან უკანასკნელის ხედვის გამოცხადების გარეშე.

3.4.2 განზოგადება

განზოგადება ეს მიმართებაა საერთო არსებასა და მის კონკრეტულ გამოვლინებას შორის. მას კიდევ უწოდებენ მიმართებას "წარმოადგენს", იმის გამო რომ ერთი არსება უფრო ზოგადი წარმოადგენს მეორეს კერძო შემთხვევას. განზოგადება ნიშნავს იმას, რომ შვილობილი ობიექტი შეიძლება გამოყენებულ იქნას ყველგან, სადაც გვხვდება მშობელი კლასის ობიექტები, მაგრამ არა პირიქით. სხვა სიტყვებით რომ ვთქვათ შვილობილი შეიძლება დავაყენოთ მშობელის მაგიერ. ამასთან ის მემკვიდრეობით ღებულობს მშობელის თვისებებს, კერძოთ მის ატრიბუტებსა და ოპერაციებს. ხშირათ შვილობილს გააჩნია თავისი საკუთარი ატრიბუტები და ოპერაციები, გარდა იმისა რაც გააჩნია მშობელს. შვილობილი ღებულობს ოპერაციებს იმავე შემადგენლობით მშობლისაგან.

კლასს, რომელსაც არ გააჩნია მშობელი, მაგრამ აქვს შვილობილი უწოდებენ ბაზურს ან ფუძისეულს (ფესვს), ხოლო კლასს რომელსაც არ აქვს შვილობილი-ფოთლოვანს (leaf).

ნახ.3.2. – ზე მოყვანილია განზოგადების გამოყენება კლასებისათვის.



ნახ.3.2.

განზოგადების იერარქია იქმნება უფრო სპეციალიზირებული არსების განზოგადებით და უფრო ზოგადი არსების სპეციალიზაციით. ამ კლასების ატრიბუტებისა და ოპერაციების განხილვისას ჩანს, რომ ასეთი იერარქიის მიღება შესაძლებელია ორი საშუალებით: ან სპეციალიზაციის პროცესით, ან განზოგადების პროცესით. სპეციალიზაციისას თავიდან განისაზღვრება საერთო კონცეფცია **ფიგურა**, ხოლო შემდეგ ხდება მისი სპეციალიზაცია კონკრეტულ ფიგურამდე. განზოგადებისას გამომჟღავნდება უფრო სპეციალიზირებულები **კვადრატი, წრე** და **სამკუთხედი**, ხოლო შემდეგ დგინდება რომ მათ აქვთ საერთო მახასიათებლები, რომლებიც შესაძლებელია გამოიყოს უფრო ზოგადში. ქვეკლასები მემკვიდრეობით იძენენ თავისი სუპერკლასის თვისებებს.

მაგრამ, ხშირად ოპერაციები, რომლებსაც ქვეკლასები ღებულობენ მემკვიდრეობით მათ არ ერგება. საჭირო ხდება სუპერკლასის ქცევის შეცვლა(რედინაცია) ქვეკლასებში.

ასევე, ხშირად შეუძლებელია სუპერკლასის ოპერაციის რეალიზება ქვეკლასში - ოპერაცია აბსტრაქტულია. კლასები ერთი ან რამოდენიმე აბსტრაქტული ოპერაციით წარმოადგენს აბსტრაქტულს, მათ არა აქვთ ეგზემპლიარები. UML-ში ამისათვის ოპერაციის და კლასის დასახელება იწერება დახრილი შრიფტით.

მემკვიდრეობის მიმართების მოდელირება წარმოებს შემდეგი თანმიმდევრობით:

1. ვიპოვოთ ატრიბუები, ოპერაციები და მოვლენები, რომლებიც საერთოა ორი ან მეტი კლასისათვის.
2. გამოიტანეთ ეს ელემენტები საერთო კლასში, რათა შევქმნათ ახალი კლასი. ამასთან ყურადღება მივაქციოთ იმას, რომ დონეები არ აღმოჩნდეს ძალიან ბევრი.
3. მიუთითოთ მოდელში, რომ შედარებით სპეციალიზებული კლასები მემკვიდრეობით ღებულობენ უფრო ზოგადის თვისებებს, რისთვისაც გამოვიყენოთ განზოგადების მიმართება მიმართული შვილობილიდან მის მშობელზე.

3.4.3. პოლიმორფიზმი

პოლიმორფიზმი ნიშნავს „მრავალ ფორმას“. პოლიმორფული ოპერაცია – ეს ოპერაციაა, რომელსაც აქვს მრავალი რეალიზაცია. მოყვანილ მაგალითში, კლასს ფიგურა(*Shape*) გააჩნია აბსტრაქტული ოპერაციები დავხაზოთ *draw*() და გამოვთვალოთ ფართი *getArea*(), რომლებსაც აქვთ ორი სხვადასხვა რეალიზაცია კლასებში კვადრატი(*Square*) და წრე(*Circle*). ამ ოპერაციებს აქვთ „მრავალი ფორმა“, შეასაბამისად ისინი პოლიმორფულია. კონკრეტული ქვეკლასი უნდა ახდენდეს აბსტრაქტული ოპერაციების რეალიზებას, რომლებსაც ის იღებს მემკვიდრეობით. ცხადია, რომ ოპერაციების *draw*() და *getArea*() რეალიზაციები კლასებისათვის *Square* და *Circle* იქნება სხდასხვა. ოპერაცია *draw*() კლასი *Square* ობიექტებისათვის დახაზავს კვადრატებს, ხოლო კლასი *Circle* ობიექტებისათვის – წრეებს. ასევე, ოპერაციის *getArea*() რეალიზაციაც იქნება სხვადასხვა. კვადრატისათვის ის დააბრუნებს width*height, ხოლო წრისათვის – პ*რ2. ამაშია პოლიმორფიზმის არსი: სხვადასხვა კლასის ობიექტებს აქვთოპერაციები *ერთნაირი* სიგნატურიტ, მაგრამ *სხვადასხვა* რეალიზაციით.

ინკაფსულაცია, მემკვიდრეობითობა და პოლიმორფიზმი – ეს სამი ძირითადი თვისებაა ობიექტ-ორიენტირებული სისტემების. პოლიმორფიზმი საშუალებას გვაძლევს

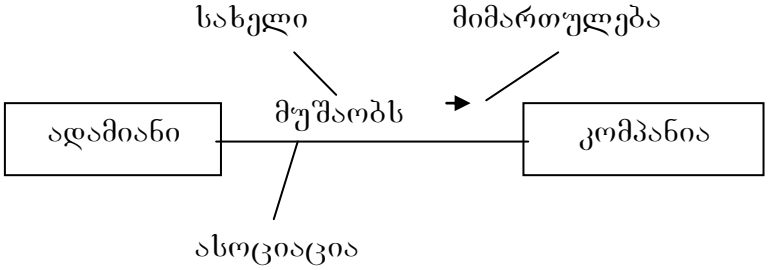
დამუშავდეს უფრო მარტივი სისტემები, რომელთა შეცვლა უფრო ადვილია, რადგან სხვადასხვა ობიექტების ინტერპრეტაცია მათში ხდება ერთნაირად. იგი საშუალებას იძლევა სხვადასხვა კლასის ობიექტებს გაუგზავნოთ *ერთნაირი* შეტყობინება და მივიღოთ შესაბამისი პასუხი. თუ, კლასი *Square* ობიექტებს გაუგზავნით შეტყობინებას *draw*(), ისინი გამოხაზავენ კვადრატებს, მაგრამ თუ იგივე შეტყობინებას გაუგზავნით კლასი *Circle* ობიექტებს, ისინი გამოხაზავენ წრეს. ობიექტები ჩანან გონიერებათ.

3.4.4. ასოციაცია

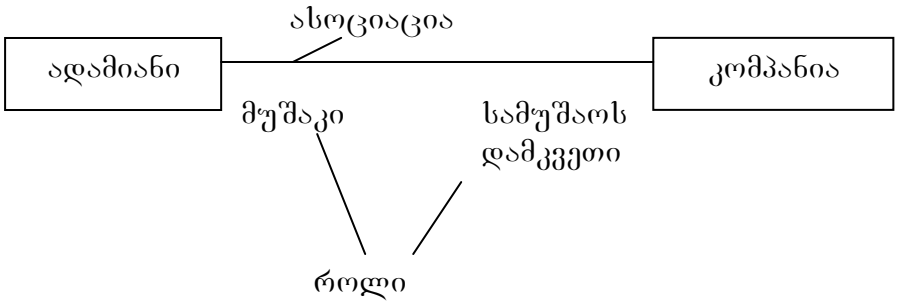
ასოციაციას უწოდებენ სტრუქტურულ მიმართებას, რომელიც გვიჩვენებს, რომ ერთი ტიპის ობიექტები გარკვეულად დაკავშირებული არიან მეორე ტიპის ობიექტებთან. დასაშვებია შემთხვევა, როდესაც ასოციაციის ორივე ბოლო ერთი და იმავე კლასს ეკუთვნის. ეს ნიშნავს, რომ კლასის რომელიმე ობიექტზე დასაშვებია დაუკავშირდეს სხვა ობიექტები იმავე კლასიდან. ასოციაციას, რომელიც აკავშირებს ორ კლასს უწოდებენ ბინარულს. შესაძლებელია, თუმცა იშვიათია, შეიქმნას ასოციაცია, რომელიც აკავშირებს ერთდროულად რამოდენიმე კლასს, მათ უწოდებენ n- არულს.

არსებობს ოთხი დამატება, რომელიც გამოიყენება ასოციაციასთან მიმართებაში.

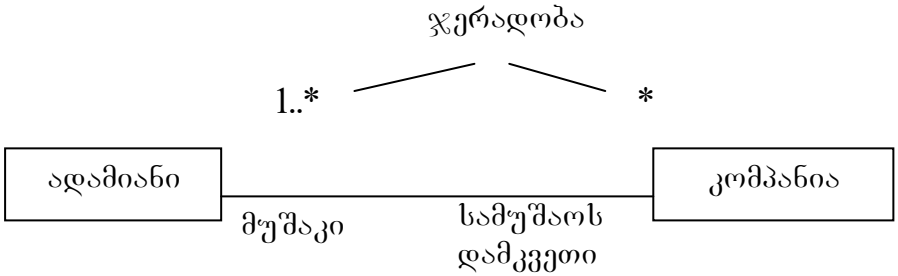
დასახელება. ასოციაციას შეიძლება მიენიჭოს სახელი და მისი კითხვის მიმართულება.



როლი. კლასი, რომელიც მონაწილეობს ასოციაციაში, აკისრია მასში გარკვეული როლი. შეგვიძლია მიუთითოთ ეს როლი.



ჯერადობა. ხშირათ მოდელირებისას საჭიროა მიეთითოს, რამდენი ობიექტი შეიძლება დაუკავშირდეს ასოციაციის ერთი ეგზემპლიარით. ამ რიცხვს უწოდებენ ასოციაციის როლის ჯერადობას. მას მიუთითებენ მნიშვნელობათა დიაპაზონის სახით. მაგ. ერთი (1), ერთი ან ნოლი (0..1), "ბევრი" (0..*), ერთი ან მეტი (1..*). შესაძლებელია მიეთითოს გარკვეული რიცხვი- (3). ქვემოთ მოყვანილი მაგალითიდან გამომდინარეობს, რომ ერთი ან მეტი მუშაკი მუშაობს ნებისმიერ (*) კომპანიაში.



აგრეგირება. უბრალო ასოციაცია ორ კლასს შორის გამოსახავს სტრუქტურულ მიმართებას თანაბარ არსებს შორის, როდესაც ორივე კლასი იმყოფება ერთ კონცეპტუალურ დონეზე და არც ერთი არ არის უფრო მნიშვნელოვანი მეორესთან შედარებით. მაგრამ ამას გარდა გვხვდება შემთხვევები, როდესაც საჭიროა მოდელირება მიმართებისა "ნაწილი/მთელი". ერთერთ კლასს აქვს მეტი რანგი(მთელი) და შედგება რამოდენიმე უფრო მცირე რანგის ნაწილებისაგან. ასეთი ტიპის მიმართებას უწოდებენ აგრეგირებას. გრაფიკულად მას გამოხატავენ რომბით.

აგრეგაციის სემანტიკასთან დაკავშირებით აღსანიშნავია ორი მნიშვნელოვანი თვისება:

- აგრეგაცია ტრანზიტიულია რაც ნიშნავს, რომ თუ C არის B-ს ნაწილი, და B თავის მხრივ A-ს ნაწილი, მაშინ C ასევე არის A-ს ნაწილი.
- აგრეგაცია არის ასიმეტრიული რაც ნიშნავს, რომ ობიექტი არასდროს არ შეიძლება იყოს თავისი თავის ნაწილი.

აგრეგაციის უფრო მკაცრი ფორმაა – კომპოზიცია. მას აქვს მსგავსი სემანტიკა.

კომპოზიცია. უბრალო აგრეგირება ეს კონცეპტუალური მიმართებაა, რომლითაც შესაძლებლობა გვეძლევა განვასხვავოთ "მთელი" მისი "ნაწილისაგან", მაგრამ არ ცვლის ასოციაციის ნავიგაციის არსს მთელ და მის ნაწილებს შორის, არ ადებს არავითარ შეზღუდვებს მთელის და ნაწილის შესაბამისობაზე მთელ სიცოცხლის დროში – არა აქვს

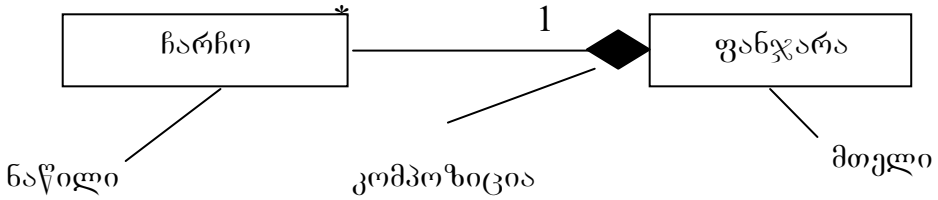
დამოუკიდებელი სიცოცხლე მთელის გარეშე. კომპოზიტურს უწოდებენ აგრეგირების ფორმას ფლობის მკვეთრად გამოხატული მიმართებით, ამასთან სიცოცხლის დრო მთელისა და ნაწილის ერთმანეთს ემთხვევა. ეს ნიშნავს, რომ კომპოზიტური აგრეგირებისას ობიექტი დროის ნებისმიერ მომენტში შესაძლებელია ეკუთვნოდეს მხოლოდ ერთ კომპოზიტს. მაგალითად, კლასი ***ჩარჩო*** ეკუთვნის მხოლოდ ერთ კლასს ***ფანჯარა***, მაშინ როდესაც უბრალო აგრეგირების დროს "ნაწილი" შესაძლებელია ეკუთვნოდეს ერთდროულად რამოდენიმე "მთელს". დაუშვათ ობიექტი ***კედელი*** შესაძლებელია ეკუთვნოდეს რამოდენიმე ობიექტს ***ოთახი.***

ამას გარდა, კომპოზიტური აგრეგირებისას მთელი პასუხს აგებს მის ნაწილებზე, მართავს რა მათ შექმნასა და მოსპობას. იქმნება რა კლასი ***ჩარჩო*** ის უნდა დაუკავშიროთ მის მომცავ ***ფანჯარას.*** როდესაც ობიექტი ***ფანჯარა*** ისპობა, მან თავის მხრივ უნდა მოსპოს მისი კუთვნილი ობიექტები ***ფანჯარა.***

როგორც ნაჩვენებია ნახ.2-ზე კომპოზიცია არის ასოციაციის კერძო შემთხვევა და აღინიშნება შეფერადებული რომბით მის ბოლოზე მთელის მხრიდან.

სტრუქტურული მიმართებების მოდელირება წარმოებს შემდეგნაირად:

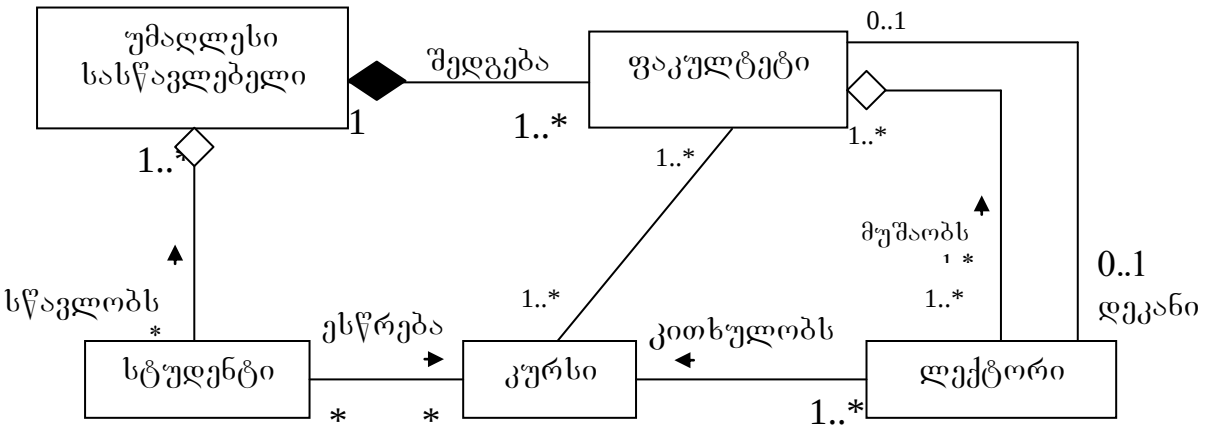
- განსაზღვრეთ ასოციაცია კლასების ყოველი წყვილისათვის. ეს იქნება შეხედულება ასოციაციაზე მონაცემების თვალსაზრისით.



- თუ ერთი კლასის ობიექტები უნდა ურთიერთქმედებდნენ სხვა კლასის ობიექტებთან განსხვავებულად, ვიდრე ეს ოპერაციის პარამეტრებით არის განსაზღვრული, მიზანშეწონილია განისაზღვროს ამ კლასებს შორის ასოციაცია. ეს არის შეხედულება ასოციაციაზე ქცევის თვალსაზრისით.
- ყოველი განსაზღვრული ასოციაციისათვის მიუთითეთ ჯერადობა და როლების დასახელება.
- თუ ასოციაციის ერთერთი კლასი სტრუქტურულად ან ორგანიზაციულად წარმოადგენს მთელს ასოციაციის მეორე ბოლოში მყოფი კლასის მიმართ, მონიშნეთ ასეთი ასოციაცია როგორც აგრეგირება.

მოყვანილი მსჯელობის ვიზუალირებისათვის ქვემოთ მოყვანილია მაგალითი.

კლასებს ***სტუდენტი*** და ***კურსი*** არსებობს ასოციაცია, რომელიც გვიჩვენებს, რომ სტუდენტი ესწრება კურსებს. ყოველ სტუდენტს შეუძლია დაესწროს კურსების ნებისმიერ რაოდენობას, და ყოველ კურსზე შესაძლებელია მოდიოდეს სტუდენტების ნებისმიერი რაოდენობა.



ანლოგიურად კლასებს ***კურსი*** და ***ლექტორი*** განსაზღვრულია ასოციაცია, რომელიც გვიჩვენებს, რომ ლექტორი კითხულობს კურსს. ყოველი კურსისავის უნდა არსებობდეს ერთი ლექტორი მაინც, და ყოველ ლექტორს შეუძლია წარმართოს კურსების ნებისმიერი რაოდენობა (მათ შორის არც ერთი).

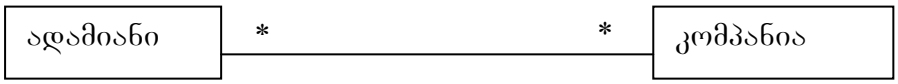
მიმართება კლასებს ***უმაღლესი სასწავლებელი*** და კლასებს ***სტუდენტი*** და ***ფაკულტეტი*** მცირედ განსხვავდებიან ერთმანეთისაგან, თუმცა ორივე წარმოადგენს აგრეგირების მიმართებას. უმაღლეს სასწავლებელში შესაძლებელია იყოს სტუდენტების ნებისმიერი რაოდენობა (ნოლის ჩათვლით), და ყოველი სტუდენტი შესაძლებელია სწავლობდეს ერთ ან რამოდენიმე უმაღლეს სასწავლებელში; უმაღლეს სასწავლებელი შესაძლებელია შედგებოდეს ერთი ან რამოდენიმე ფაკულტეტისაგან, მაგრამ ყოველი ფაკულტეტი ეკუთვნის მხოლოდ ერთ უმაღლეს სასწავლებელს. მიმართებას კლასებს შორის ***უმაღლესი სასწავლებელი*** და ***ფაკულტეტი*** უწოდებენ კომპოზიციურ აგრეგირებას.

ნახაზიდან ასევე ჩანს, რომ კლასებს ***ფაკულტეტი*** და ***ლექტორი*** დგინდება ორი ასოციაცია. ერთი მათგანი გვიჩვენებს, რომ ყოვლი ლექტორი მუშაობს ერთ ან რამოდენიმე ფაკულტეტზე, და ყოველ ფაკულტეტზე უნდა იყოს სულ მცირე ერთი ლექტორი. მეორე ასოციაცია გვიჩვენებს, რომ ყოველ ფაკულტეტს მართავს მხოლოდ ერთი ლექტორი – დეკანი. ამ მოდელის თანახმად, ლექტორი შესაძლებელია იყოს მხოლოდ ერთი ფაკულტეტის დეკანი, ამასთან ზოგიერთი ლექტორი არ არის დეკანი.

ასოციაცია -კლასები. ასოციაცია კლასები – ეს ასოციაციაა და ამავე დროს წარმოადგენს კლასს. კლასი ასოციაცია ნიშნავს, რომ დროის ნებისმიერ მომენტში ნებისმიერ ორ ობიექტს შორის შესაძლებელია არსებობდეს მხოლოდ ერთი კავშირი.

განვიხილოთ მაგალითი, რომელიც მოყვანილია ნახ.3.-ზე. მოყვანილი მოდელიდან ჩანს, რომ:

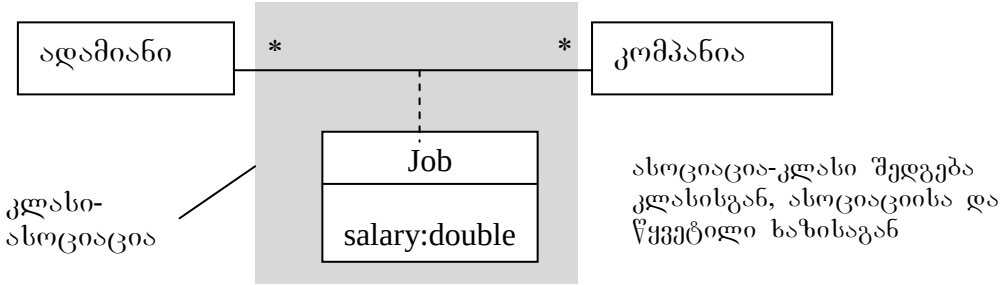
- ყოველ ადამიანს შეუძლია იმუშაოს რამოდენიმე კომპანიაში;
- ყოველ კომპანიას შეუძლია დაიქირავოს რამოდენიმე ადამიანი.



მაგრამ, თუ გვინდა დავამატოთ ბიზნეს-წესი – ადამიანი ღებულობს ხელფასს ყოველ კომპანიაში, რომელმაც იგი დაიქირავა. დგება პრობლემა – სად უნდა ჩაიწეროს ეს ხელფასი.

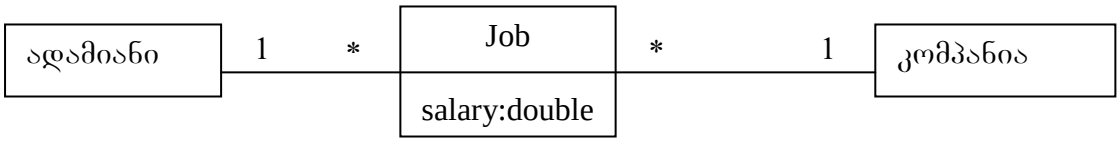
შეუძლებელია ხელფასი გავხადოთ კლასი ადამიანის ატრიბუტათ, რადგან ყოველი ადამიანი შესაძლებელია მუშაობდეს მრავალ კომპანიაში და ყოველ მათგანში ღებულობდეს სხვადასხვა ხელფასს. ანალოგიურად შეუძლებელია ხელფასი გავხადოთ კლასი კომპანიის ატრიბუტათ, რადგან კომპანიის ყოველ ეგზემპლიარს შეუძლია დაიქირავოს რამოდენიმე ადამიანი და ყოველი მათგანი ღებულობდეს სხვადასხვა ხელფასს.

ასეთი სიტუაციის მოდელირებისათვის UML –ში გამოიყენება ასოციაცია-კლასი. იგი არა მარტო აერთებს ორ კლასს, როგორც ჩვეულებრივი ასოციაცია. იგი ამავე დროს განსაზღვრავს მახასიათებელთა ნაკრებს, რომელიც ეკუთვნის თვით ასოციაციას. ყოველ ასოციაცია-კლასს შესაძლებელია ქონდეს ატრიბუტები, ოპერაციები და სხვა ასოციაციები.



კლასი-ასოციაციის ეგზემპლიარები – ეს ჩვეულებრივ კავშირებია, რომლებსაც აქვთ ატრიბუტები და ოპერაციები. კლასი-ასოციაციის გამოყენება მოცემულ შემთხვევაში ნიშნავს, რომ მოცემული ობიექტისათვის ადამიანი და კომპანია შესაძლებელია არსებობდეს ერთი ობიექტი Job(თანამდებობა), ანუ ყოველ ადამიანს შეუძლია დაიკავოს მხოლოდ ერთი Job(თანამდებობა) მოცემულ კომპანიაში. მაგრამ ჩვენ კვლავ გვჭირდება შევინახოთ ხელფასი ყოველი ჯგუფისათვის კომპანია/თანამდებობა/ადამიანი. ამიტომ ახდენენ მიმართების მატერიალიზებას(აკეთებენ რეალურს), წარმოადგენენ მას

ჩვეულებრივი კლასის სახით. ნახ.4.-ზე Job(თანამდებობა) წარმოადგენს ჩვეულებრივ კლასს და ყოველ ადამიანს შესაძლებელია ქონდეს რამოდენიმე Job(თანამდებობა), ყოველი Job(თანამდებობა) ყოველ კონკრეტულ კომპანიაში.



მაშასადამე, მატერიალიზებული მიმართება საშუალებას იძლევა დროის ყოველ კონკრეტულ მომენტში ნებისმიერ ორ ობიექტს შორის არსებობდეს ერთზე მეტი კავშირი. კლასი-ასოციაციები შესაძლებელია გამოყენებულ იქნას მხოლოდ იმ შემთხვევაში, როდესაც ყოველ კავშირს აქვს უნიკალური ინდივიდუალურობა. კავშირის ინდივიდუალურობა კი განისაზღვრება მის ბოლოებზე არსებული ობიექტების ინდივიდუალურობით.

3.5. კლასების და ობიექტების დიაგრამა

კლასების დიაგრამა ობიექტ - ორიენტირებული სისტემების მოდელირებისას გვხვდება ყველაზე ხშირად. ასეთ დიაგრამებზე უჩვენებენ კლასებს, ინტერფეისებს და მიმართებებს მათ შორის.

კლასების დიაგრამა გამოიყენება სისტემის სტატიკური სახით მოდელირებისათვის პროექტირების თვალსაზრისით. კლასების დიაგრამა წარმოადგენს საფუძველს ორი შემდეგი სახის დიაგრამისათვის – კომპონენტების და განლაგების.

პროგრამული უზრუნველყოფის შესაქმნელად მიუხედავად პროგრამების აზრობრივი ბუნებისა **UML** საშუალებას გვაძლევს შევქმნათ საჭირო სამშენებლო ბლოკები. კლასების დიაგრამა გამოიყენება სწორედ ამ სამშენებლო ბლოკების და მათ შორის მიმართებების სტატიკური ასპექტების ვიზუალირებისათვის.

კლასების დიაგრამა ძირითადათ გამოიყენება შემდეგი მიზნებისათვის:

- **სისტემის ლექსიკონის მოდელირებისათვის**. იგი გულისხმობს იმას, თუ რომელი აბსტრაქცია წარმოადგენს სისტემის ნაწილს და რომელი არა. კლასების დიაგრამით ჩვენ შეგვიძლია განვსაზღვროთ ეს აბსტრაქციები და მათი მოვალეობები.

- **უბრალო კოოპერაციების მოდელირებისათვის**. კოოპერაცია ეს კლასების, ინტერფეისების და სხვა ელემენტების ერთობლიობაა, რომლებიც მუშაობენ ერთობლივად გარკვეული კოოპერაციული ქცევის უზრუნველსაყოფად, უფრო მნიშვნელოვანის ვიდრე მათი ელემენტების ჯამი. მაგ. განაწილებულ სისტემებში ტრანზაქციების სემანტიკის მოდელირებისას, ვერ გავიგებთ მიმდინარე პროცესებს, მხოლოდ ერთი კლასით, რადგან შესაბამისი სემანტიკა უზრუნველყოფილი ხდება ერთობლივად მომუშავე კლასებით. მოდელირების ეს საშულება განხილული იქნება კოოპერაციების დახასიათებისას.
- **მონაცემთა ბაზის ლოგიკური სქემის მოდელირებისათვის**. ლოგიკური სქემა შეიძლება წარმოვადგინოთ როგორც მონაცემთა ბაზის კონცეპტუალური პროექტის ნახაზი.

სამოდელო სისტემების დიდი ნაწილი შეიცავენ მდგრად ობიექტებს, ესე იგი ისეთებს, რომლებიც შესაძლებელია შევინახოთ მონაცემთა ბაზაში და შემდეგ საჭიროებისამებრ გამოვიძახოთ. ამისათვის ყველაზე ხშირად იყენებენ რელაციურ, ობიექტ- ორიენტირებულ ან ჰიბრიდულ ობიექტ-რელაციურ მონაცემთა ბაზებს. მონაცემთა ბაზების ლოგიკური პროექტირებისათვის ხშირად იყენებენ დიაგრამებს "არსება-კავშირი" (**E-R** დიაგრამები). მაგრამ თუ კლასიკურ **E-R** დიაგრამებზე ძირითადი ყურადღება გამახვილებულია მხოლოდ მონაცემებზე, კლასების დიგრამით შესაძლებელია ქცევის მოდელირებაც.

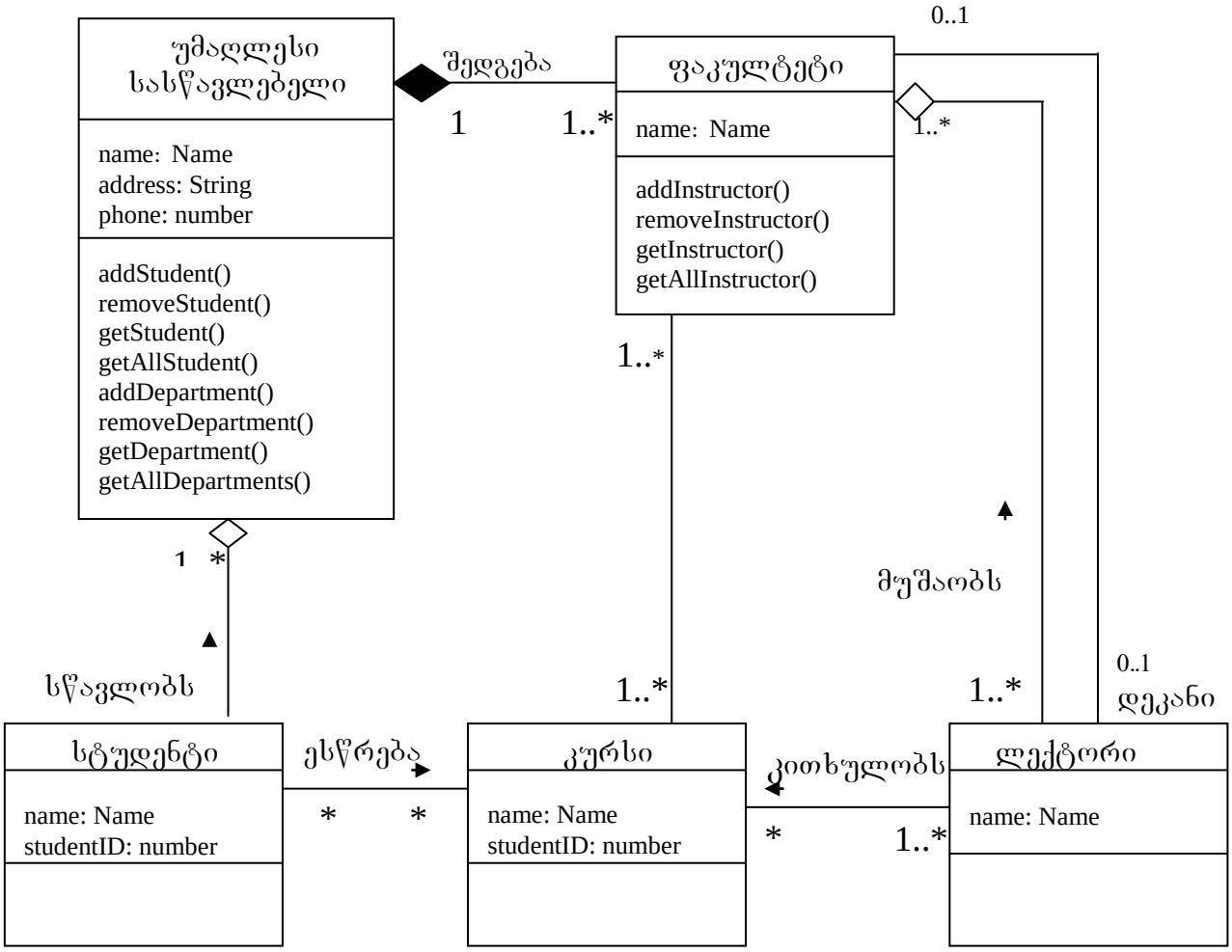
ლოგიკური სქემის მოდელირება წარმოებს შემდეგნაირად:

- მოვახდინოთ მოდელში შემავალი კლასების იდენტიფიცირება, რომელთა მდგომარეობაც უნდა შენახულ იქნას.
- შევქმნათ კლასების დიგრამა და დავახასიათოთ ისინი როგორც მდგრადი, სტანდარტული მონიშნული მნიშვნელობით **persistent** (მდგრადი).
- გავხსნათ კლასების სტრუქტურული თავისებურებანი. ეს ნიშნავს, რომ დეტალურად უნდა მიუთითოთ ატრიბუტები და განსაკუთრებული ყურადღება მივაქციოთ ასოციაციას და მათ ჯერადობას.
- მოვძებნოთ ტიპიური სტრუქტურული სახეები, რომლებიც ართულებენ ფიზიკურ მონაცემთა ბაზის პროექტირებას, მაგ. ციკლური ასოციაციები, ასოციაცია ერთი ერთზე და **n**-არული ასოციაციები. საჭიროების შემთხვევაში შევქმნათ შუალედური აბსტრაქციები ლოგიკური სქემის გამარტივებისათვის.
- განვიხილოთ ამ კლასების ქცევები, გავხსნათ ოპერაციები, რომლებიც მნიშვნელოვანია მონაცემებთან მიმართვისათვის და მათი მთლიანობის დასაცავად.

- ეცადეთ გამოიყენოთ ინსტრუმენტალური საშუალებები, რომლებიც საშუალებას მოგვცემენ გარდავქმნათ ლოგიკური პროექტი ფიზიკურში.

ნახაზზე ქვევით მოყვანილია კლასების ერთობლიობა აღებული უმაღლესი სასწავლებლის საინფორმაციო სისტემიდან. ეს ნახაზი ფაქტიურად წარმოადგენს მანამდე მოყვანილი კლასების დიაგრამის გაფართოებას და შეიცავს საკმაო რაოდენობის დეტალებს ფიზიკური მონაცემთა ბაზის კონსტრუირებისათვის.

ყველა კლასი დიაგრამაზე მონიშნულია როგორც მდგრადი (**persistent**), ესე იგი მათი ეგზემპლიარები უნდა იყვნენ მონაცემთა ბაზაში. მოყვანილია აგრეთვე კლასის ატრიბუტები.



ორი კლასი (უმაღლესი სასწავლებელი და ფაკულტეტი) შეიცავენ რამოდენიმე ოპერაციას, რომლებიც უზრუნველყოფენ მათზე მანიპულირებას. ეს ოპერაციები ჩართული იქნა მათი მნიშვნელობის გამო მონაცემთა მთლიანობის დაცვის თვალსაზრისით. ცხადია, არსებობენ სხვა ოპერაციებიც, რომლებიც სასურველი იყო განგვეხილა მსგავსი კლასების მოდელირებისას, მაგალითად დაკვეთა სტუდენტის კურსზე ჩარიცხვისათვის საჭირო წინასწარი ცოდნის შესახებ. მაგრამ ეს, უფრო ბიზნეს-წესებია, ვიდრე ოპერაციები

მონაცემთა მთლიანობის დასაცავად, ამიტომ უკეთესი იქნება ისინი აბსტრაქციის უფრო მაღალ დონეზე მოვათავსოთ.

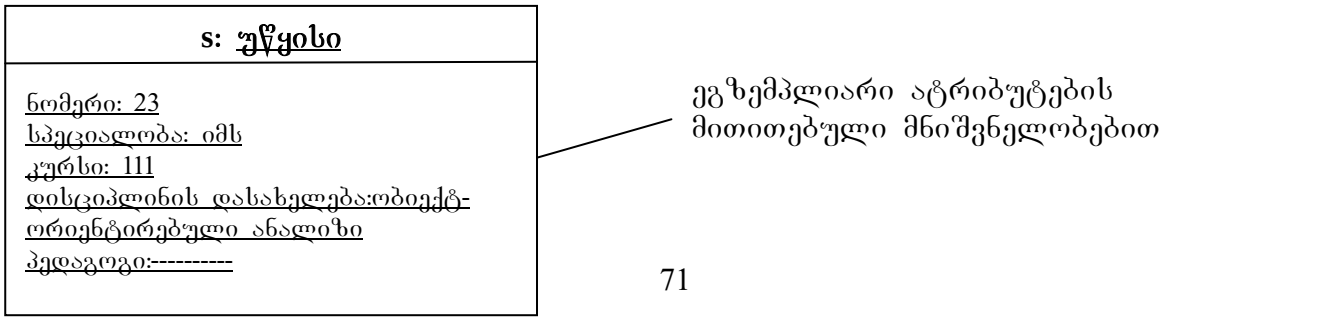
ობიექტების დიაგრამა. UML-ში არსებობს ფუნდამენტალური განსხვავება არსის ეგზემპლიარსა და აბსტრაქციას შორის. პირველი – ეს მხოლოდ აბსტრაქციაა, რომელიც აღწერს არსის გარკვეულ ტიპს სხვადასხვა თვისებებით, მაშინ როდესაც მეორე წარმოადგენს კონკრეტულ ეგზემპლიარს ამ აბსტრაქციის, რომელიც არსებობს რეალურ სამყაროში და მის ყოველ თვისებას აქვს რეალური მნიშვნელობა.

აბსტრაქცია აღწერს საგნის იდეალურ არსს, ეგზემპლიარი – მის კონკრეტულ მატერიალიზაციას. ერთ აბსტრაქციას შეიძლება გააჩნდეს რამოდენიმე ეგზემპლიარი. მოცემული ეგზემპლიარისათვის ყოველთვის არსებობს აბსტრაქცია, რომელიც განსაზღვრავს საერთო მახასიათებლებს ყველა მსგავსი ეგზემპლიარებისათვის.

ეგზემპლიარს (Instance) უწოდებენ აბსტრაქციის კონკრეტულ მატერიალიზაციას, რომლის მიმართ შეიძლება გამოყენებულ იქნას ოპერაციები და რომელსაც შეუძლია შეინახოს მათი შედეგები. ცნებები "ეგზემპლიარი" და "ობიექტი" პრაქტიკულად სინონიმებია. ეგზემპლიარს გამოხატავენ ხასგასმული სახელით.

ჩვეულებრივ ობიექტს უწოდებენ კლასის კონკრეტულ მატერიალიზაციას. ობიექტები – ეს კლასების ეგზემპლიარებია. ობიექტი არა მარტო იკავებს ადგილს რეალურ სამყაროში, მათზე შესაძლებელია ასევე მანიპულირება. ოპერაციები, რომლებიც სრულდება ობიექტებზე, გამოცხადდება მის აბსტრაქციაში. ეს მიუთითებს იმაზე, რომ ყოველი ობიექტი ხასიათდება მდგომარეობით.

ობიექტის მდგომარეობას უწოდებენ მისი ყველა თვისებების ერთობლიობას და მათ მიმდინარე მნიშვნელობებს. თვისებათა რიცხვში შედის ობიექტთა ატრიბუტები. მაშასადამე, ობიექტის მდგომარეობა დინამიურია და მისი ვიზუალიზაციისას ფაქტიურად აღიწერება მისი მდგომარეობა დროის მოცემულ მომენტში და სივრცის მოცემულ წერტილში. ვასრულებთ რა ობიექტზე ოპერაციას, ფაქტიურად ვცვლით მის მდგომარეობას. მაგრამ, აქვე უნდა აღინიშნოს, რომ ობიექტის გამოკითხვისას მდგომარეობა არ იცვლება. ობიექტის მდგომარეობათა შეცვლა შეიძლება გამოვსახოთ ურთიერთქმედების დიაგრამაზე, დავხაზავთ რა მას რამოდენიმეჯერ ან შესაძლებელია გამოვიყენოთ მოცემული პროცესის აღწერისათვის ავტომატი.



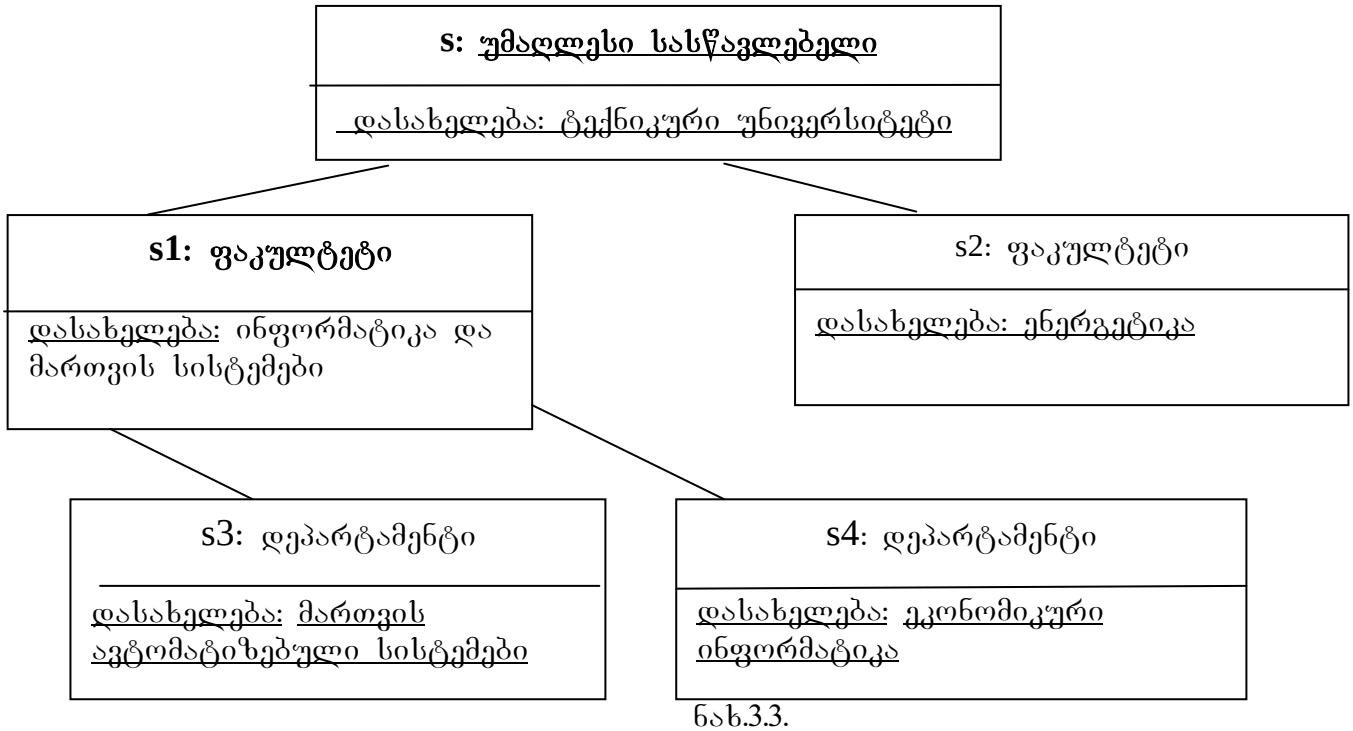
ობიექტების დიაგრამა საშუალებას იძლევა მოვახდინოთ იმ არსების ეგზემპლიართა მოდელირება, რომლებიც არსებობენ კლასების დიაგრამაზე. ობიექტების დიაგრამაზე ნაჩვენებია ობიექტების სიმრავლე და მათ შორის მიმართებები დროის რომელიღაც მომენტში, როგორც ეს ნაჩვენებია ნახ.3.3.-ზე.

ობიექტების დიაგრამა გამოიყენება სისტემის სტატიკური სახით მოდელირებისათვის პროექტირებისა და პროცესების თვალთახედვით. ობიექტების დიაგრამით ახდენენ ობიექტების სტრუქტურის მოდელირებას.

ობიექტთა სტრუქტურის მოდელირება გულისხმობს სისტემის ობიექტთა "სურათის" მიღებას დროის მოცემულ მომენტში. ობიექტების დიაგრამა წარმოადგენს დინამიური სცენარის სტატიკურ საფუძველს, რომელიც აღიწერება ურთიერთქმედების დიაგრამით.

ობიექტური სტრუქტურების მოდელირება ხორციელდება შემდეგნაირად:

1. მოახდინეთ იმ მექანიზმის იდენტიფიცირება, რომლის მოდელირებასაც აპირებთ. მექანიზმი წარმოადგენს გარკვეულ ფუნქციას ან სამოდელო სისტემის ქცევის ნაწილს, რომელშიც მონაწილეობს კლასები, ინტერფეისები და სხვა არსები.



ნახ.3.3.

2. ყოველი მექანიზმისათვის მოვახდინოთ კლასების, ინტერფეისების და კოოპერაციაში მონაწილე სხვა ელემენტების და მათ შორის მიმართებების იდენტიფიცირება.

3. განვიხილოთ მექანიზმის მუშაობის ერთ ერთი სცენარი. დავაფიქსიროთ ეს სცენარი დროის გარკვეული მომენტისათვის და გამოსახეთ ყველა ობიექტები, რომლებიც მონაწილეობენ მექანიზმში.
4. მიუთითეთ ყოველი ობიექტის მდგომარეობა და ატრიბუტების მნიშვნელობები, თუ ეს საჭიროა სცენარის გაგებისათვის.
5. მიუთითეთ აგრეთვე კავშირები ამ ობიექტებს შორის, რომლებიც წარმოადგენენ არსებული ასოციაციების ეგზემპლიარებს.

მაგალითისათვის ნახ.3.4.-ზე მოყვანილია ობიექტების ერთობლიობა, აღებული უმაღლეს სასწავლებელში სესიის ჩატარების სისტემიდან.

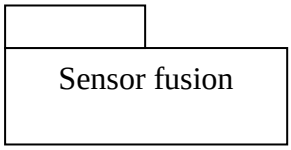
როგორც ნახაზიდან ჩანს, ერთი ობიექტი შეესაბამება უწყისს (s, კლასი უწყისის ეგზემპლიარი), რომლის მდგომარეობა აკმაყოფილებს ჩატარებული საგამოცდო სესიის შედეგებს. ეს ობიექტი დაკავშირებულია ეგზემპლიართან s1 კლასისა ინდივიდუალური დატვირთვის უწყისი. თავის მხრივ ობიექტი s1 დაკავშირებულია ობიექტთან r დეპარტამენტის დატვირთვა. თავის მხრივ ობიექტი r დაკავშირებულია ობიექტთან r1 დეპარტამენტის ჯგუფური გეგმა და ეგზემპლიართან r2 კლასისა ფაკულტეტის სასწავლო გეგმა, რომელთა საფუძველზე დგება s: უწყისის მონაცემები. აღნიშნული მონაცემები შესაძლებელია მიუთითოდ ურთიერთქმედების დიაგრამაზე (იხ. თავი 3).

ნახ. 3.4.

3.6. პაკეტები

დიდი სისტემების პროექტირება განსაზღვრავს პოტენციალურად დიდი რაოდენობის კლასებს, ინტერფეისებს, კვანძებს, კომპონენტებს, დიაგრამებს და სხვა ელემენტებს. მათი გაგების გამარტივებისათვის უნდა მოვახდინოთ ამ არსების ორგანიზება უფრო დიდ ბლოკებში, რომელსაც UML-ში უწოდებენ პაკეტებს(Packages).

პაკეტი – ეს მოდელის ელემენტების ორგანიზების საშუალებაა უფრო დიდ ბლოკში, რომელთა მანიპულირება შემდგომში შესაძლებელია როგორც ერთიანი მთლიანის. შესაძლებლობა გვეძლევა ვმართოდ პაკეტში შემავალი არსების ხედვა, ისე, რომ ზოგიერთი იქნება გარედან ხედვადი, ხოლო სხვები – არა. ამის, გარდა პაკეტების საშუალებით შესაძლებელია წარმოვადგინოთ სისტემის არქიტექტურის სხვადასხვა სახე. კარგათ დაპროექტებული პაკეტი აჯგუფებს სემანტიკურად ახლოს მყოფ ელემენტებს, რომლებსაც აქვთ ტენდენცია იცვლებოდნენ ერთად. გრაფიკულად პაკეტი წარმოიდგინება შემდეგი სახით

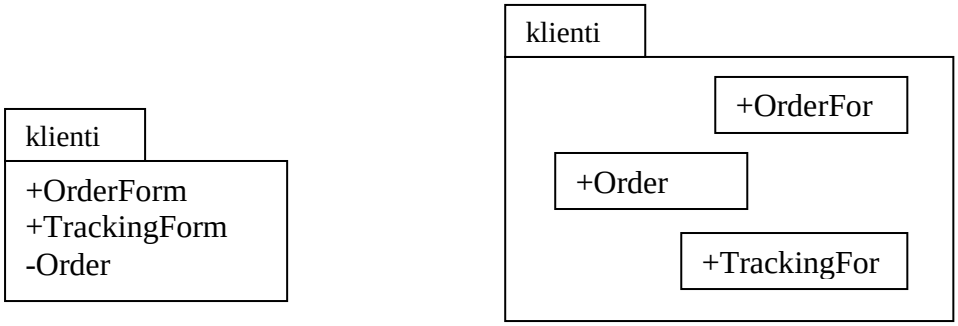


ყოველ პაკეტს უნდა გააჩნდეს დასახელება, რომელიც განასხვავებს მას სხვა პაკეტებისაგან. პაკეტის დასახელება – ტექსტური სტრიქონია.

პაკეტის შემადგენელი ელემენტები. პაკეტი როგორც ავღნიშნეთ შესაძლებელია შეიცავდეს სხვა ელემენტებს. ეს ნიშნავს, რომ ისინი გამოცხადებული უნდა იქნენ პაკეტის შიგნით. თუ პაკეტი იხურება, ისპობა მასზე მიკუთვნილი ელემენტები. ამასთან ელემენტი შესაძლებელია ეკუთვნოდეს მხოლოდ ერთ პაკეტს და უნდა ქონდეს თავისი უნიკალური სახელი. დაუშვებელია ერთი და იგივე ელემენტების არსებობა ერთნაირი დასახელებით, მაგრამ დაშვებულია სხვადასხვა ელემენტის არსებობა ერთი დასახელებით.

პაკეტებს, შესაძლებელია ეკუთვნოდეს სხვა პაკეტები, რაც საშუალებას იძლევა შევქმნათ მოდელის იერარქიული დეკომპოზიცია. მაგალითად, შესაძლებელია გვქონდეს კლასი Camera, რომელიც ეკუთვნის პაკეტს Vision, რომელიც თავის მხრივ არის პაკეტის Sensors შემადგენელი. შედგენილი დასახელება ამ კლასის იქნება Sensors::Vision::Camera. მაგრამ უნდა მოვერიდოთ პაკეტების ძალიან ღრმა შემცველობას – ორი-სამი დონე წარმოადგენს ზღვარს.

პაკეტის შემცველობა შესაძლებელია წარმოვადგინოთ გრაფიკულად ან ტექსტური სახით:

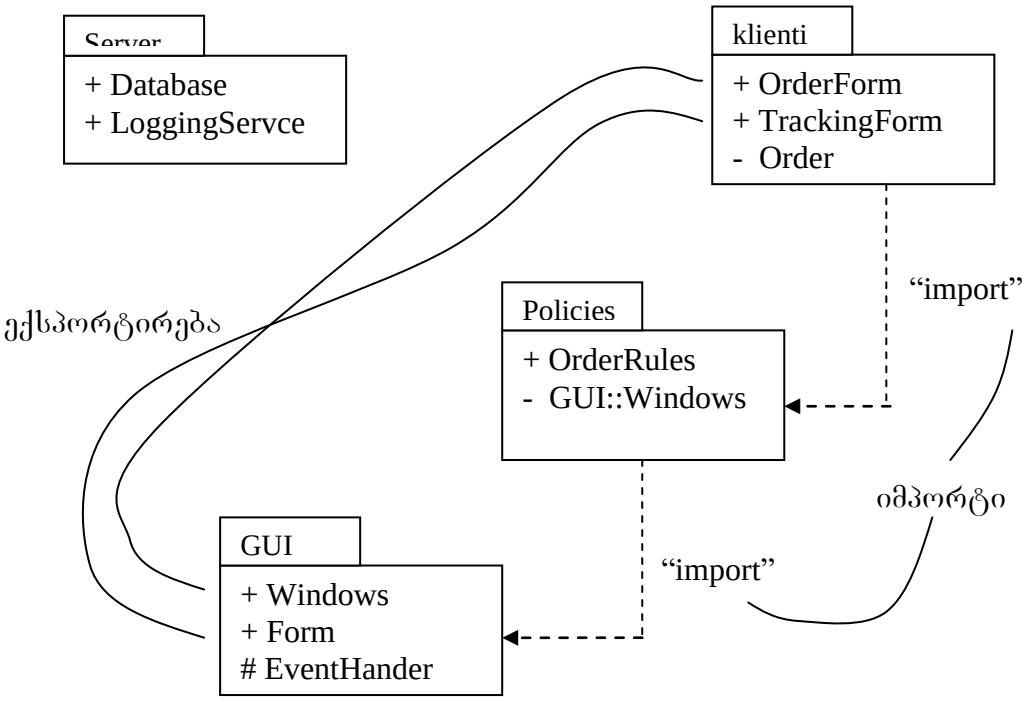


ხედვა. პაკეტის ელემენტების ხედვის კონტროლი შესაძლებელია ისევე, როგორც კლასის ოპერაციებისა და ატრიბუტების ხედვა. დუმილით ასეთი ელემენტები გახსნილია, ანუ ხედვადია ყველა ელემენტებისათვის ნებისმიერ პაკეტში, რომელიც ახდენს მის იმპორტირებას. დაცული ელემენტები ხედვადია მხოლოდ შვილობილებისათვის, ხოლო დახურული თავისი პაკეტის გარეთ საერთოდ არახედვადია.

პაკეტის შემადგენელი ელემენტების ხედვა აღინიშნება ხედვის სიმბოლოთი ამ ელემენტის დასახელების წინ. გახსნილი ელემენტებისათვის გამოიყენება სიმბოლო +(პლიუსი). პაკეტის ყველა გახსნილი ნაწილები შეადგენენ მის ინტერფეისს. ისევე როგორც კლასებისათვის დაცული ელემენტებისათვის გამოიყენება სიმბოლო #(დიეზი), ხოლო დახურულისათვის უმატებენ სიმბოლოს –(მინუსი). ბოლოს, დაცული ელემენტები ხედვადი იქნება მხოლოდ პაკეტებისათვის, რომლებიც მემკვიდრეობით არის მოცემულისათვის, ხოლო დახურული საერთოდ არახედვადია პაკეტის გარეთ, რომელშიც იგი გამოცხადებულია.

იმპორტი და ექსპორტი. დაუშვათ ჩვენ გვაქვს კლასი A ერთ პაკეტში, ხოლოკლასი B მეორეში და ორივე პაკეტი თანაბარია. ამასთან, A-ც და B-ც გამოცხადებულია ღიათ თავიანთ პაკეტებში. თავისუფალი მიმართვა ერთერთის მეორესთან არ არის შესაძლებელი, რადგან პაკეტის საზღვრები გაუმჭირვალეა. მაგრამ, თუ ერთი პაკეტი, რომელიც შეიცავს პაკეტს A, ახდენს B კლასის შემცველი პაკეტის იმპორტირებას, მაშინ A შეძლებს “შეხედოს” B-ს. მაგრამ B კი მაინც ვერ “შეხედავს” A-ს. იმპორტი საშუალებას აძლევს ერთი პაკეტის ელემენტებს მოახდინონ ერთმხვრივი მიმართვა მეორეს ელემენტებთან. UML-ში იმპორტის მიმართება მოდელირდება როგორც დამოკიდებულება სტერეოტიპით Import. სემანტიკურად მოფიქრებულ ბლოკებში აბსტრაქციების დაჯგუფებით და იმპორტის მეშვეობით მათთან მიმართვის კონტროლით, ჩვენ საშუალება გვეძლევა ვმართოდ რთული სისტემა, რომელიც ითვლის რამოდენიმე ათეულ აბსტრაქციას.

პაკეტის ღია ელემენტებს უწოდებენ ექსპორტირებულებს. ასე მაგალითად, ნახ.2.6.1. –ზე პაკეტი GUI ექსპორტირებას უწევს ორ კლასს – Windows და Form.



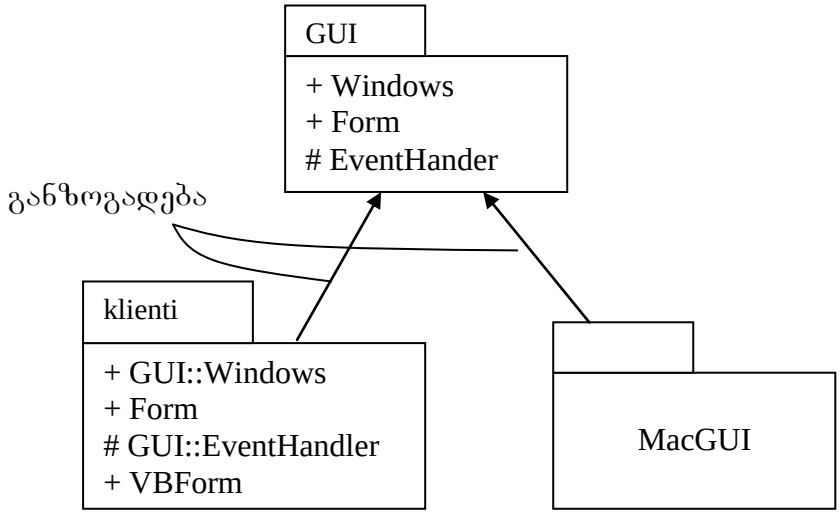
ნახ.3.5. იმპორტი და ექსპორტი

კლასი EventHandler პაკეტის დაცული ნაწილია. ექსპორტირებული ელემენტები იქნებიან ხედვადი მხოლოდ იმ პაკეტებისათვის, რომლებიც ახდენენ მოცემულის იმპორტირებას. როგორც ნახაზიდან ჩანს Policies ახდენს GUI-ს იმპორტირებას. მაშასადამე GUI::Window და GUI::Form იქნებიან მისგან ხედვადი. მაგრამ კლასი GUI::EventHandler არ იქნება ხედვადი, რადგან არის დაცული. მეორეს მხრივ, პაკეტი Server არ ახდენს GUI-ს იმპორტირებას, ამიტომ არა აქვთ უფლება GUI ელემენტებთან მიმართვისა. იმავე მიზეზით GUI-ს არ აქვს უფლება Server პაკეტის შემცველობასთან მიმართვისა. დამოკიდებულება იმპორტი არ არის ტრანზიტიული. მოცემულ მაგალითში პაკეტი Client ახდენს Policies იმპორტირებას, ხოლო Policies – GUI-ს, მაგრამ ეს არ ნიშნავს, რომ Clienti ახდენს GUI-ს იმპორტირებას. მაშასადამე პაკეტიდან Client პაკეტი Policies ექსპორტირებულ ნაწილებთან მიმართვა დაშვებულია, ხოლო GUI-ს ექსპორტირებულ ნაწილებთან არა.

განზოგადება. პაკეტებს შორის განსაზღვრულია ორი ტიპის მიმართება: იმპორტზე დამოკიდებულების, რომელიც გამოიყენება პაკეტში ელემენტების იმპორტისათვის, რომლებიც ექსპორტირებულები არიან სხვა პაკეტებით და განზოგადების, რომელიც გამოიყენება კლასთა ოჯახის სპეციფიცირებისათვის.

მაგალითად, როგორც ჩანს ნახ. 3.6.-დან GUI ექსპორტირებას უწევს ორ კლასს – Windows და Form. კლასი EventHandler პაკეტის დაცული ნაწილია. რსებობს პაკეტ GUI ორი სპეციალიზაცია WindowsGUI და MacGUI. ისინი მემკვიდრეობით იღებენ თავისი მშობლის ღია და დაცულ ელემენტებს. როგორც კლასების შემთხვევაში, პაკეტებს შესაძლებლობა

აქვთ ჩაანაცვლონ მშობლის ელემენტები ან დაამატონ ახალი. მაგალითად, პაკეტი WindowsGUI შეიცავს კლასებს GUI::Window და GUI::EventHandler. არდა ამისა, მასში გადაწერილია კლასი Form და დამატებულია ახალი კლასი VBForm.



ნახ.3.6. პაკეტებს შორის განზოგადება

განზოგადებაში მონაწილე პაკეტები მიყვებიან ჩართვის იმავე პრინციპს, რასაც კლასები. სპეციალიზირებული პაკეტები (როგორიც არის WindowsGUI) შესაძლებელია გამოყენებულ იქნან ყველგან, სადაც დასაშვებია მათი მშობლების გამოყენება (GUI-ს მსგავსად).

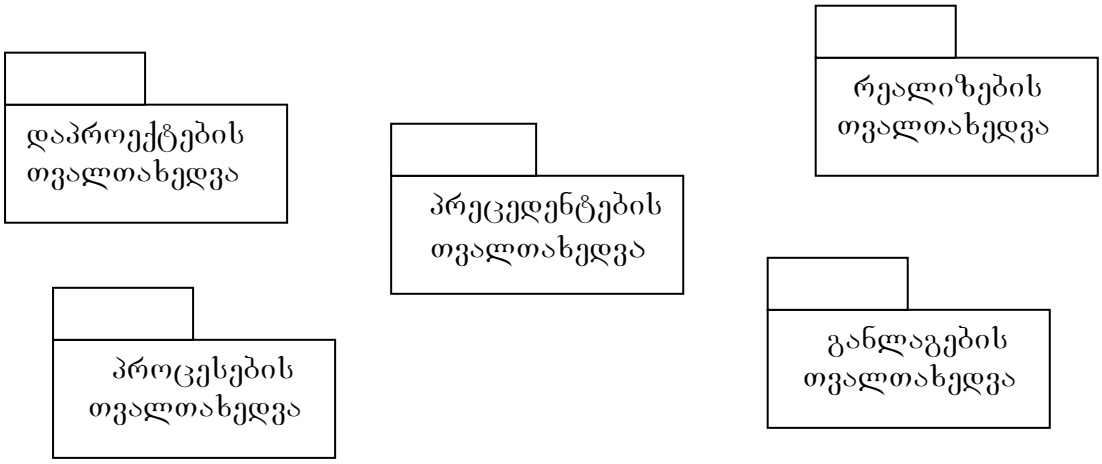
გამოყენების ტიპიური ხერხები. ველაზე ხშირად პაკეტებს იყენებენ მოდელირების ელემენტების ორგანიზებისათვის ჯგუფებში, რომელთანაც შემდეგ შესაძლებელია მუშაობა როგორც ერთიანი მთელის. მარტივი სისტემის შექმნისას, შესაძლებელია პაკეტები საერთოდ არ დაგვჭირდეს, რადგა ყველა ჩვენი აბსტრაქციები შესაძლებელია მოთავსდეს ერთ პაკეტში. აგრამ უფრო რთული სისტემების დამუშავებისას აღმოვაჩენთ, რომ ბევრი კლასები, კომპონენტები, კვაძები, ინტერფეისები და დიაგრამებიც კიბუნებრივად იყოფიან ჯგუფებათ. შწორედ ამ ჯგუფება ვამოდელირებთ ჯგუფებში.

ყველაზე ხშირად პაკეტების მეშვეობით ერთი ტიპის ელემენტებს აერთიანებენ ჯგუფებში. მაგალითად, კლასები და მათი მიმართებები სისტემის წარმოდგენიდან დაპროექტების თვალთახედვით შესაძლებელია გავაერთიანოდ შესაძლებელია დავყოთ რამოდენიმე პაკეტად და ვაკონტროლოთ მასთან მიმართვა იმპორტზე დამოკიდებულებით.

პაკეტები შესაძლებელია გამოყენებულ იქნან აგრეთვე სხვადასხვა ტიპის ელემენტების ორგანიზებისათვის. ეს განსაკუთრებით ეხება ისეთ შემთხვევას, როდესაც სისტემა მუშავდება რამოდენიმე კოლექტივის მიერ, რომლებიც განლაგებულნი არიან სხვადასხვა ადგილზე. ამ შემთხვევაში პაკეტები გამოიყენებიან კონფიგურაციის მართვისათვის. განვალაგებთ რა მათში ყველა კლასებს და დიაგრამებს, სხვადასხვა

კოლექტივის წევრებს დამოუკიდებლად შეეძლებათ ამოიღონ და მოათავსოთ უკან სასურველი ელემენტი.

პაკეტების გამოყენება მონათესავე ელემენტების დაჯგუფებისათვის საკმაოდ მნიშვნელოვანია – მის გარეშე რთული სისტემების დამუშავება შეუძლებელია. მაგრამ პროგრამული სისტემების არქიტექტურული სახეების განხილვისას იქმნება მოთხოვნა უფრო რთული ბლოკების შექმნისა. კერძოთ, არქიტექტურული სახეებიც შესაძლებელია პაკეტების მეშვეობით დავამოდელიროთ. ამისათვის პირველ რიგში უნდა დადგინდეს თუ რომელი სახეებია ჩვენი სისტემისათვის მნიშვნელოვანი. ჩვეულებრივ ეს სახეებია პრეცედენტების, დაპროექტების, პროცესების რეალიზაციისა და განლაგების თვალთახედვა. ასეთი დაჯგუფებისათვის, შესაბამის პაკეტებში მოვათავსებთ ელემენტებს და დიაგრამებს, რომლებიც აუცილებელია ყოველი ცალკეული სახეობის ვიზუალირებისა და სპეციფიცირებისათვის.



ნახ.3.7. არქიტექტურული სახეობების მოდელირება

აუცილებლობის შემთხვევაში ყოველი სახეობის ელემენტები დავაჯგუფოთ უფრო მცირე პაკეტებათ. ელემენტებს შორის სხვადასხვა სახეობიდან, ცხადია იარსებებს დამოკიდებულების მიმართება, ამიტომ ზოგადათ უნდა გავხსნათ ყველა სახეობები სისტემის ზედა დონეზე ყველა დანარჩენი სახეობებისათვის იმავე დონეზე. მაგალითისათვის, ნახ.2.6.3-ზე მოყვანილია ზედა დონის კანონიკური დეკომპოზიცია, რომელიც გამოიყენება ყველაზე რთული სისტემებისათვისაც.

3.7. პრეცენდენტების რეალიზება

როგორც აღნიშნული იყო ობიექტ-ორიენტირებული ანალიზის თვალსაზრისით ანალიზის სამუშაო ნაკადის მიზანია – ანალიტიკური მოდელის შექმნა(იხ.4.4.). კლასების ანალიტიკური მოდელი – ეს სისტემის სტატისტიკური სტრუქტურაა, ხოლო

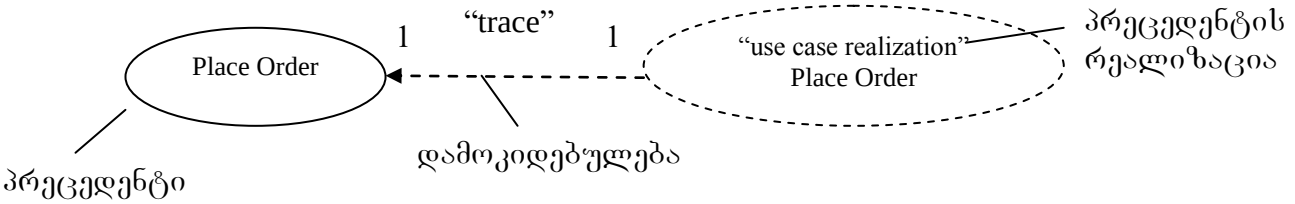
პრეცედენტების რეალიზაცია გვიჩვენებს, თუ როგორ ურთიერთქმედებენ ანალიზის კლასები სისტემის ფუნქციანალობის განხორციელებისათვის.

პრეცედენტების რეალიზებისას ანალიზის ფაზაში დამმუშავებლის მიზანია:

- გამოვავლინოთ, რომელი ანალიზის კლასების ურთიერთქმედება უზრუნველყოფს ქცევას, განსაზღვრულს პრეცედენტით; პრეცედენტების რეალიზებისას შესაძლებელია გამოვლინდეს ანალიზის ახალი კლასები.
- გამოვავლინოთ, რომელი შეტყობინებებით უნდა იცვლებოდნენ ამ კლასის ეგზემპლიარები მოცემული ქცევის რეალიზებისათვის. ეს საშუალებას მოგვცემს გამოვავლინოთ:
- ოპერაციები, ანალიზის კლასებისათვის;
- ატრიბუტები ანალიზის კლასებისათვის;
- მიმართებები ანალიზის კლასებს შორის.
- პრეცედენტების, მოთხოვნების და ანალიზის კლასების მოდელების კორექტირება, დაუმატოთ მათში ინფორმაცია რომელიც მიიღება პრეცედენტების რეალიზაციისას.

ყველა პრეცედენტების რეალიზება არ არის საჭირო. საჭიროა ამოვირჩიოთ და დავამუშაოთ მხოლოდ ყველაზე ძირითადი. მათი რეალიზაცია უნდა გაგრძელდეს მანამდე, სანამ დამმუშავებელი არ იგრძნობს რომ ფლობს საკმარის ინფორმაციას ანალიზის კლასების ერთობლივად მუშაობის გაგებისათვის. UP – იტერაციული პროცესია. ამიტომ აუცილებლობის შემთხვევაში პრეცედენტების რეალიზაცია შესაძლებელია დამუშავდეს მოგვიანებით.

მაშასადამე, პრეცედენტ, რომელიც წარმოადგენს ფუნქციონალური მოთხოვნების აღწერას, გარდაიქმნება კლასების და ურთიერთქმედების დიაგრამებში, ხოლო ეს კი სისტემის მაღალდონიანი აღწერაა. გრაფილულად მას გამოსახავენ შემდეგი სახით



პრეცედენტის რეალიზაცია გვიჩვენებს, როგორ ურთიერთქმედებენ კლასები, იმისათვის, რომ მოახდინოს სისტემის ფუნქციონალობის რეალიზება.

პრეცედენტების რეალიზების სასიცოცხლოდ მნიშვნელოვანი ნაწილია –კლასების დიაგრამა. ისინი უნდა გვაძლევდნენ წარმოდგენას სისტემის შესახებ: თუ როგორ უნდა იყვნენ დაკავშირებული კლასები, იმისათვის, რომ მათ ეგზემოლიარებს შეეძლოთ

იმოქმედონ ქცევის რეალიზებისათვის, რომელიც განსაზღვრულია ერთ ან რამოდენიმე პრეცედენტში.

ურთიერთქმედების დიაგრამები გვიჩვენებენ, თუ როგორ ურთიერთქმედებენ კლასიფიკატორების ეგზემპლიარები სისტემის ქცევის რეალიზებისათვის.

3.7.1. ურთიერთქმედება

ურთიერთქმედებას (Interaction) უწოდებენ ქცევას, რომელიც გამოიხატება მოცემულ კონტექსში მოცემული ერთობლიობის ობიექტებს შორის შეტყობინებების გაცვლაში, რის შედეგადაც მიიღწევა განსაზღვრული მიზანი. ურთიერთქმედებას ადგილი აქვს ყოველთვის, როდესაც ობიექტები დაკავშირებულნი არიან ერთმანეთთან (ეს სტრუქტურული კავშირები გამოისახებიან ობიექტების დიაგრამაზე).

შეტყობინება (Message) – ობიექტებს შორის მონაცემთა გაცვლის სპეციფიკაციაა, რომლის დროსაც გადაიცემა გარკვეული ინფორმაცია იმის გათვალისწინებით, რომ პასუხად განხორციელდება გარკვეული მოქმედება. ყველაზე ხშირად შეტყობინება დაიყვანება ოპერაციის გამოძახებამდე ან სიგნალის გაგზავნამდე, ამავე დროს მას შეუძლია შექმნას ან მოსპოს სხვა ობიექტები.

ურთიერთქმედების საშუალებით ახდენენ მართვის ნაკადების მოდელირებას ოპერაციების, კლასების, კომპონენტების, პრეცენდენტების ან მთლიანად სისტემის შიგნით. ოპერაციის კონტექსში იგი ვლინდება ოპერაციის პარამეტრების ურთიერთქმედებაში მის მიერ რეალიზებადი ალგორითმის შესრულებისას. კლასების კონტექსში ურთიერთქმედებით შესაძლებელია კლასის სემანტიკის ვიზუალირება, მაგალითად უჩვენოთ, თუ როგორ ურთიერთქმედებენ კლასის ატრიბუტები ერთმანეთში, სხვა ობიექტებთან და კლასში განსაზღვრულ ოპერაციის პარამეტრებთან. პრეცედენტების კონტექსში ურთიერთქმედებით აღიწერება სცენარი, რომელიც თავის მხრივ წარმოადგენს პრეცედენტის მოქმედების ერთ ერთ ნაკადს.

ურთიერთქმედება საშუალებას გვაძლევს მოვახდინოთ ასეთი ნაკადების ანალიზი ორი კრიტერიუმით:

- ყურადღება გავამახვილოთ შეტყობინებების დროის მიხედვით მიმდევრობაზე;
- ყურადღების აქცენტირება ხდება ურთიერთდაკავშირებული ობიექტების სტრუქტურულ მიმართებებზე და შემდეგ განიხილება თუ როგორ გადასცემენ შეტყობინებებს ამ სტრუქტურის კონტექსტში.

ურთიერთქმედებაში მონაწილე ობიექტები შეიძლება იყვნენ კონკრეტული არსები ან პროტოტიპები. კონკრეტული არსების სახით ობიექტი წარმოადგენს რაიმეს, რეალურ

სამყაროში არსებულს. მაგალითად, p, კლასი *ადამიანის* ეგზემპლიარი, შესაძლებელია აღნიშნავდეს კონკრეტულ ადამიანს. პირიქით, პროტოტიპმა p შესაძლებელია წარმოადგინოს კლასი *ადამიანის* ნებისმიერი ეგზემპლიარი.

ურთიერთქმედებას ადგილი აქვს ყოველთვის, როდესაც ობიექტები დაკავშირებულია ერთმანეთთან. კავშირი (LINK) წარმოადგენს სემანტიკურ შეერთებას ობიექტებს შორის. თუ ობიექტებს შორის არსებობს კავშირი, მაშინ ერთ ერთს შეუძლია გაუგზავნოს შეტყობინება მეორეს. ხშირად საკმარისია მიუთითოდ, რომ ასეთი გზა არსებობს. იმ შემთხვევაში თუ თქვენ გინდათ მისი უფრო დაწვრილებითი სპეციფიცირება, შესაძლებელია კავშირის შევსება შემდეგი სტანდარტული სტერეოტიპებით:

association – უჩვენებს, რომ შესაბამისი ობიექტი ხილვადია ასოციაციისათვის;

self – შესაბამისი ობიექტი ხილვადია, რადგან წარმოადგენს დისპეჩერს ოპერაციისათვის;

global – შესაბამისი ობიექტი ხილვადია, რამდენადაც იმყოფება მოქმედებათა მომცავ სფეროში;

local - შესაბამისი ობიექტი ხილვადია, რამდენადაც იმყოფება მოქმედებათა ლოკალურ სფეროში;

parameter - შესაბამისი ობიექტი ხილვადია, რამდენადაც წარმოადგენს პარამეტრს.

მოქმედება, რომელიც წარმოადგენს შეტყობინების მიღების შედეგს შესრულებადი წინადადებაა, რომელიც ქმნის გამოთვლითი პროცედურის აბსტრაქციას. მოქმედებას შეუძლია მიგვიყვანოს მდგომარეობის შეცვლამდე.

საშუალებას გვაძლევს მოვახდინოთ რამოდენიმე სახის მოქმედების მოდელირება:

call(გამოძახება) – იძახებს ოპერაციას, რომელიც გამოიყენება ობიექტზე. ობიექტს შეუძლია გაუგზავნოს შეტყობინება თავისთავს, რაც მიგვიყვანს ოპერაციის ლოკალურ გამოძახებამდე.

return(დავაბრუნოთ) – აბრუნებს მნიშვნელობას გამომძახებელ ობიექტთან.

send(გავგზავნოთ) – აგზავნის სიგნალს ობიექტთან.

create(შეიქმნას) – ქმნის ახალ ობიექტს.

destroy(განადგურდეს) – ობიექტს სპობს. ობიექტს შეუძლია გაანადგუროს თავისი თავი.

როდესაც ობიექტი იძახებს ოპერაციას ან უგზავნის სიგნალს მეორე ობიექტს, შეტყობინებასთან ერთად შეიძლება გადავცეთ მისი ფაქტიური პარამეტრები. ასევე, როდესაც ობიექტი უბრუნებს მართვას მეორე ობიექტს, შესაძლებელია მიუთითოთ დასაბრუნებელი მნიშვნელობა.

როდესაც ობიექტი უგზავნის შეტყობინებას მეორე ობიექტს, მიმღებს შეუძლია თავის მხრივ გაუგზავნოს შეტყობინება მესამე ობიექტს, ამ უკანასკნელმა მეოთხეს და ა.შ. შეტყობინებათა ასეთი ნაკადი გვაძლევს მიმდევრობას(Sequence).

ურთიერთქმედებაში მონაწილე ობიექტები არსებობენ მთელი ურთიერთქმედების მანძილზე. მაგრამ ზოგჯერ ობიექტები საჭიროა შეიქმნას (create) და განადგურდნენ (destroy). ეს ეხება კავშირებსაც: მიმართებები ობიექტებს შორის შეიძლება აღიძვრას ან გაქრეს. იმისათვის, რომ ავღნიშნოთ ობიექტების ან კავშირების ურთიერთქმედების პროცესში წარმოშობისა და გაქრობის ფაქტი, ელემენტს უერთებენ ერთერთ შემდეგ შეზღუდვას:

- **new**(ახალი) – გვიჩვენებს, რომ ეგზემპლიარი ან კავშირი წარმოიქმნება მომცავი ურთიერთქმედების შესრულების დროს;
- **destroyed**(მოსპობილი) – ეგზემპლიარი ან კავშირი ისპობა მომცავი ურთიერთქმედების შესრულების დასრულებამდე;
- **transient**(დროებითი) – ეგზემპლიარი ან კავშირი იქმნება მომცავი ურთიერთქმედების შესრულებისას და ისპობა მის დამთავრებამდე.

ურთიერთქმედებისას ობიექტების ატრიბუტების მნიშვნელობები, მათი მდგომარეობა ან როლი, როგორც წესი იცვლება. იგი შეიძლება გამოვხატოთ ობიექტის კოპიის შექმნით ატრიბუტების სხვა მნიშვნელობით, მდგომარეობით და როლით. ურთიერთქმედების დიაგრამაზე მათ აკავშირებენ შეტყობინების სტერეოტიპით become.

ყველაზე ხშირათ ურთიერთქმედებას იყენებენ მართვის ნაკადის მოდელირებისათვის, რომელიც ახასიათებს სისტემის ქცევას მთლიანობაში პრეცენდენტის, ერთი კლასის ან ცალკეული ოპერაციის ჩათვლით. ამასთან კლასები, ინტერფეისები, კომპონენტები და მათ შორის კავშირები ახდენენ სისტემის სტატიკური ასპექტების მოდელირებას.

3.7.2. ურთიერთქმედების დიაგრამა

ურთიერთქმედების დიაგრამა განსაზღვრულია ობიექტებს შორის კავშირების აღწერისათვის და გამოიყენება სისტემის დინამიკური ასპექტების მოდელირებისათვის.

როგორც წესი, ურთიერთქმედების დიაგრამები შეიცავენ:

- ობიექტებს;
- კავშირებს;
- შეტყობინებებს.

UML2-ში არსებობს ურთიერთქმედების დიაგრამების ოთხი ტიპი:

- **მიმდევრობის დიაგრამები** ყურადღების აქცენტირებას ახდენენ შეტყობინებების დროის მიხედვით მოწესრიგებაზე. ამ მიზნით გამოიყენებენ მიმდევრობის დიაგრამებს, რომლებშიც აქცენტირება ხდება შეტყობინებების დროში გადაცემაზე, რაც განსაკუთრებით სასარგებლოა პრეცენდენდების დინამიური ასპექტების მოდელირებისათის. უბრალო იტერაციები და განშტოებები მიმდევრობის დიაგრამებზე გამოისახება უფრო მოხერხებულად, ვიდრე კოოპერაციის დიაგრამაზე.
- **კომუნიკაციის დიაგრამები** გამოყოფენ ობიექტებს შორის სტრუქტურულ მიმართებებს და ძალიან სასარგებლოა ანალიზისას, განსაკუთრებით ობიექტების ერთობლივი მუშაობის ესკიზის შექმნისას. ძირითადი ყურადღება ამ დროს ეთმობა ურთიერთქმოქმედ ეგზემპლიარებს შორის სტრუქტურულ მიმართებას, რომელთა გასწვრივაც გადაიცემა შეტყობინებები. რთული იტერაციები, განშტოებები და მართვის პარალელური ნაკადების ვიზუალირებისათვის კოოპერაციები უფრო მოხერხებულია ვიდრე მიმდევრობა.
- **ურთიერთქმედების მიმოხილვის** დიაგრამები გვიჩვენებენ, თუ რთული ურთიერთქმედებები როგორ რეალიზდებიან მარტივი ურთიერთქმედებების გვერდით. ეს მოღვაწეობის დიაგრამების განსაკუთრებული შემთხვევაა, რომელშიც კვანძები მიმართავენ სხვა ურთიერთქმედებებს. ისინი სასარგებლოა სისტემის მართვის ნაკადის მოდელირებისათვის. ურთიერთქმედების მიმოხილვის დიაგრამები განხილული იყო 15,12 თავში.
- **დროითი დიაგრამები** ყურადღებას ამახვილებენ ურთიერთქმედების ფაქტიურ დროზე. მათი ძირითადი დანიშნულებაა – დროითი დანახარჯების შეფასება. დროითი დიაგრამები განხილული იყო 15,12 თავში.

მიმდევრობის და კომმუნიკაციის დიაგრამები წარმოადგენენ ყველაზე მთავარს პრეცედენტების რეალიზაციის თვალსაზრისით.

მიმდევრობის დიაგრამა. მიმდევრობის დიაგრამის ასაგებათ საჭიროა განვალაგოთ ობიექტები, რომლებიც მონაწილეობენ ურთიერთქმედებაში X ღერძის ზედა ნაწილის გასწვრივ. ჩვეულებრივ ურთიერთქმედების ინიციატორი ობიექტი თავსდება მარცხნივ, ხოლო დანარჩენები მარჯვნივ (რაც უფრო შორს არის მით უფრო დამოკიდებული ობიექტია). შემდეგ ღერძის გასწვრივ განალაგებენ შეტყობინებებს, რომლებსაც ობიექტები აგზავნიან და ღებულობენ. ამასთან რაც უფრო გვიანია მით უფრო ქვევით არის. გარდა ამისა დიაგრამაზე უჩვენებენ ობიექტის სიცოცხლის ხაზს. ეს არის ვერტიკალური წყვეტილი ხაზი, რომელიც ასახავს ობიექტის არსებობას დროში. ობიექტების უმრავლესობა, რომლებიც წარმოდგენილი არიან ურთიერთქმედების დიგრამაზე, არსებობენ მათი ურთიერთქმედების განმავლობაში, ამიტომ მათ გამოხატავენ

დიაგრამის ზედა ნაწილში, ხოლო მისი სიცოცხლის ციკლი იხაზება ზევიდნ ქვევით. ობიექტები შეიძლება იქმნებოდნენ ურთიერთქმედების პერიოდშიც. მათი სიცოცხლის ციკლი მაშინ იწყება create შეტყობინების მიღებით. ობიექტები ასევე შეიძლება დაიხურონ (მოისპონ) ურთიერთქმედების პროცესში, ასეთ შემთხვევაში მათი სიცოცხლის ციკლი მთავრდება destroy შეტყობინების მიღებით.

აღნიშნულ დიაგრამაზე უჩვენებენ აგრეთვე მართვის ფოკუსს. იგი გამოისახება სწორკუთხედით, რომელიც მიუთითებს დროის ინტერვალს, რომლის განმავლობაშიც ობიექტი ასრულებს გარკვეულ მოქმედებას უშუალოდ ან დამოკიდებული პროცედურით. სწორკუთხედის ზედა ხაზი გაუტოლდება დროის ღერძზე მოქმედების დაწყებას, ხოლო ქვედა - მის დამთავრებას.

შესაძლებელია მართვის ფოკუსის ჩართვა, გამოწვეული რეკურსიით (საკუთარ ოპერაციასთან მიმართვა) ან უკუგამოძახებით მეორე ობიექტის მხრიდან. ეს შეიძლება ვუჩვენოთ მართვის მეორე ფოკუსის აგებით, ოდნავ მარცხნივ თავისი მშობლისაგან (დასაშვებია ჩართვა ნებისმიერი სიღრმით). თუ მართვის ფოკუსის განლაგება საჭიროა მიეთითოს მაქსიმალური სიზუსტით, შეიძლება სწორკუთხედი დაიშტრიხოს, იმ დროის შესაბამისად, რომლის განმავლობაშიც მეთოდი ნამდვილად მუშაობს და არ გადასცემს მართვას მეორე ობიექტს.

ყველა შემთხვევაში თავდაპირველად უნდა მოვახდინოთ მოცემულ პრეცედენტში მონაწილე ობიექტების იდენტიფიცირება, რომლებიც გარკვეულ როლს ასრულებენ და დავადგინოთ მათი საწყისი თვისებები, მათ შორის ატრიბუტების მნიშვნელობა, მდგომარეობა და როლი. ამის შემდეგ საშუალება გვეძლევა მოვახდინოთ მათ შორის კავშირების იდენტიფიცირება. სამოქალაქო სამართალწარმოებისას მნიშვნელოვანია მართვის ნაკადების როგორც დროის მიხედვით მოწესრიგება, ასევე მართვის ნაკადების სტრუქტურული ორგანიზაცია. მიმდევრობითობის ყოველ დიაგრამაზე შეიძლება მხოლოდ ერთი მართვის ნაკადის ჩვენება, ამიტომ, როგორც წესი, ქმნიან ურთიერთქმედების რამდენიმე დიაგრამას, რომელთაგან ერთი ითვლება ძირითადად, ხოლო დანარჩენები აღწერენ ალტერნატიულ გზებს და განსაკუთრებულ პირობებს. მიმდევრობითობის დიაგრამების ასეთი ერთობლიობა, შეიძლება გაერთიანდეს პაკეტში და მიეთითოს თვითეულს შესაბამისი სახელი.

2.3.1. ნახაზზე მოყვანილია მიმდევრობის დიაგრამა მართვის ძირითადი ნაკადისათვის – სპეციალობის დატვირთვის დადგენა(იხ. § 2.1.2.). ჩვეულებრივ სცენარის ინიციალიზაციას ახდენს კლიენტი (ჩვენ შემთხვევაში დეპარტამენტის ხელმძღვანელი), რომელიც ირჩევს მთავარი ფანჯრიდან ახალი დოკუმენტის შექმნის უფლებას. დიაგრამაზე ჩანს სხვადასხვა ობიექტების ურთიერთქმედება ამ შემთხვევისათვის.

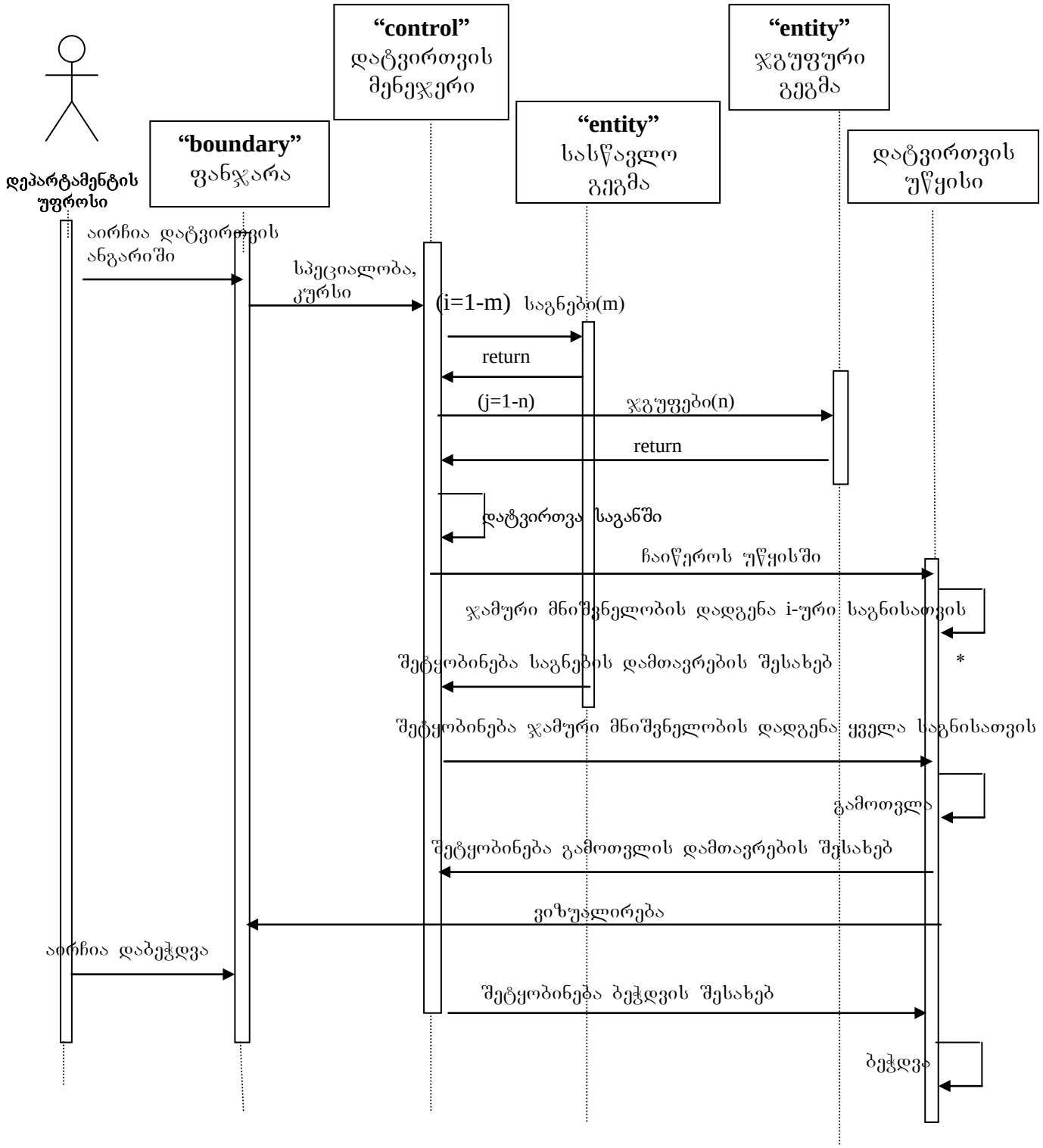
თითოეული ობიექტის მოვალეობა მითითებულია შესაბამისი სტერეოტიპით. კლიენტის არჩევანის საფუძველზე დატვირთვის მართვის ობიექტი ობიექტიდან "სასწავლო გეგმა" სპეციალობისა და კურსის მიხედვით ადგენს საგნებს(კრედიტების რაოდენობა გამოყოფილი - ლექცია, პრაქტიკული, ლაბორატორია, საკურსო სამუშაოსათვის), რომლებიც ისწავლება და ობიექტიდან "ჯგუფური გეგმა" ჯგუფებს, რომლებიც ირიცხებიან მოცემული სპეციალობის მითითებულ კურსზე (ჯგუფის ნომერი და ჯგუფში სტუდენტების რაოდენობა). მიღებული მონაცემების საფუძველზე დატვირთვის მენეჯერი ანგარიშობს მოცემულ დისციპლინაზე გეგმით გათვალისწინებულ დატვრთვას სემესტრში თითოეული ჯგუფისათვის და მის საფუძველზე ფაქტიურ დატვირთვას. მიღებული მონაცემები შეიტანება დატვირთვის უწყისში. დატვირთვის უწყისი აჯამებს მოცემული საგნის დატვირთვას თითოეული ჯგუფის მიხედვით და მთლიანად ყველა ჯგუფებისათვის.

აღნიშნულ პროცესს დატვირთვის მენეჯერი იმეორებს დისციპლინების რაოდენობის(m-ჯერ), ჯგუფების რაოდენობის (n-ჯერ) მიხედვით. ბოლოს იანგარიშება ჯამები ცალკეული ნომინალების (ლექცია, პრაქტიკული, ლაბორატორია, საკურსო სამუშაო, პრაქტიკა, მაგისტრატურა, დოქტურანტურა, რეიტინგი, მთლიანი ჯამი) მიხედვით. გამოთვლის დამთავრების შესახებ შეტყობინების მიღების საფუძველზე მომხმარებელი ქმნის ობიექტი "დატვირთვის უწყისის" ახალ ეგზემპლიარს, რომელიც შეინახება და სურვილის შემთხვევაში იბეჭდება როგორც მოცემულ შემთხვევაში.

მართვის ნაკადების დროის მიხედვით მოწესრიგება ხდება შემდეგნაირად:

1. დავადგინოთ ურთიერთქმედების კონტექსი, ეს იქნება სისტემა, ქვესისტემა, ოპერაცია ან პრეცენდენტის ერთ ერთი სცენარი.
2. განსაზღვრეთ ურთიერთმოქმედი ობიექტები და განალაგეთ მარცხნიდან მარჯვნივ, ისე რომ შედარებით მნიშვნელოვანი ობიექტები განლაგდნენ უფრო მარჯვნივ.
3. განსაზღვრეთ ყოველი ობიექტისათვის სიცოცხლის ხაზი. უფრო ხშირად ობიექტები არსებობენ მთელი ურთიერთმოქმედების განმავლ;ობაში. ხოლო იმ ობიექტებისათვის, რომლებიც ურთიერთმოქმედის პროცესში ისპობიან, სიცოცხლის ხაზზე ნათლად უჩვენეთ დაბადებისა და სიკვდილის წერტილები შესაბამისი სტერეოტიპით.
4. დაწყებული შეტყობინებიდან, რომელიც ურთიერთმოქმედების ინიციირებას ახდენს, განალაგეთ ყველა შემდეგი შეტყობინებები ზევიდან ქვევით ობიექტების სასიცოცხლო ხაზებს შორის, უჩვენეთ ყოველი შეტყობინების თვისება მაგ. მისი პარამეტრები.
5. თუ საჭიროა შეტყობინებების ჩართვა ან გამოთვლის ზუსტი ინტერვალის მითითება, შეავსეთ ობიექტების სიცოცხლის ციკლი მართვის ფოკუსით.
6. თუ საჭიროა დროითი ან სივრცობრივი შეზღუდვების სპეციფიცირება შეავსეთ შეტყობინება დროითი აღნიშვნებით და დაუკავშირეთ შესაბამისი შეზღუდვები.

7. მართვის ნაკადების უფრო ფორმალური აღწერისათვის დაუკავშირეთ ყოველ შეტყობინებას წინა და შემდგომი პირობები.

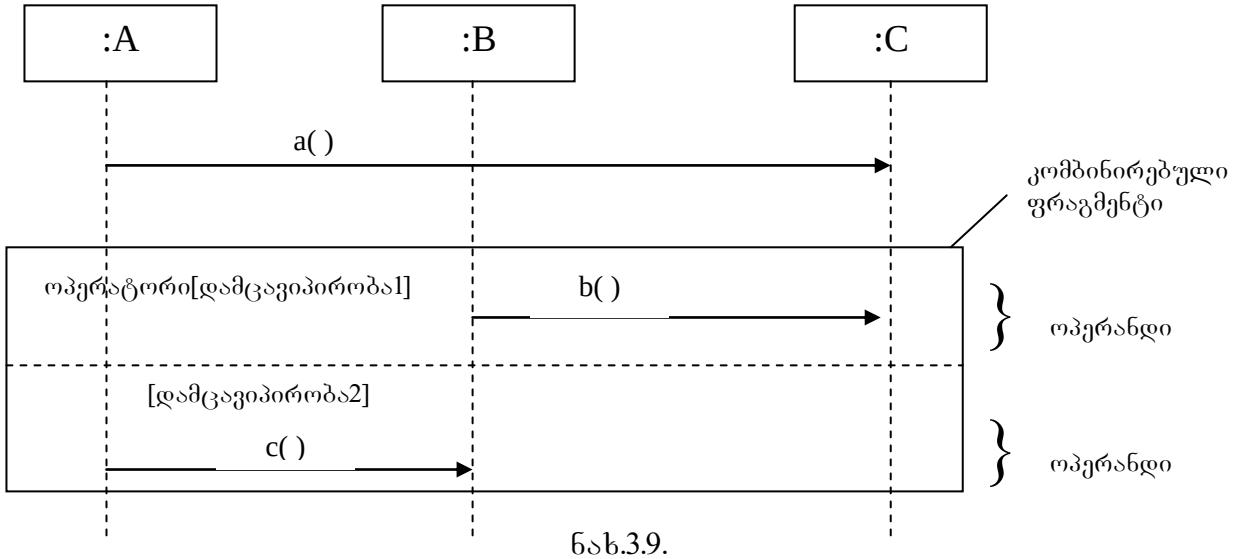


ნახ.3.8.

მიმდევრობის ყოველ დიაგრამაზე შეიძლება მხოლოდ ერთი მართვის ნაკადის ჩვენება, ამიტომ როგორც წესი ქმნიან ურთიერთქმედების რამოდენიმე დიაგრამას, რომელთაგან ერთი ითვლება ძირითადათ, ხოლო დანარჩენები აღწერენ ალტერნატიულ

გზებს და გამორიცხულ პირობებს. მიმდევრობის დიაგრამების ასეთი ერთობლიბა, შეიძლება გავაერთიანოთ პაკეტში, მივცეთ რა თვითეულს შესაბამისი სახელი.

კომბინირებული ფრაგმენტები და ოპერატორები. მიმდევრობის დიაგრამები შესაძლებელია დაიყოს უბნებად, რომლებსაც უწოდებენ კომბინირებულ ფრაგმენტებს. ყოველ კომბინირებულ ფრაგმენტს აქვს ერთი ოპერატორი, ერთი და რამოდენიმე ოპერანდები და არცერთი ან რამოდენიმე დამცავი პირობები. ოპერატორი განსაზღვრავს, თუ როგორ სრულდებიან მისი ოპერანდები.



ნახ.3.9.

დამცავი პირობები განსაზღვრავენ, შესრულდებიან თუ არა ეს ოპერანდები. დამცავი პირობა – ეს ლოგიკური გამოსახულებაა და ოპერანდი სრულდება მაშინ და მხოლოდ მაშინ, როდესაც ეს გამოსახულება ჭეშმარიტია. ერთი დამცავი პირობა შესაძლებელია გამოყენებულ იქნას ყველა ოპერანდებისათვის ან ყოველ ოპერანდს შესაძლებელია გააჩნდეს საკუთარი უნიკალური დამცავი პირობა.

ქვევით მოყვანილია ოპერატორები, რომლებიც ყველაზე ხშირად გამოიყენებიან:

- **ოპერატორი opt** გვიჩვენებს, რომ მისი ერთადერთი ოპერანდი სრულდება, თუ დამცავი პირობა ჭეშმარიტია. წინააღმდეგ შემთხვევაში შესრულება გრძელდება კომბინირებული ფრაგმენტის შემდეგ. იგი ექვივალენტურია პროგრამული კონსტრუქციის if (პირობა1) then მოქმედება1
- **ოპერატორი alt** საშუალებას იძლევა მოვახდინოთ არჩევანი რამოდენიმე ალტერნატივისას. იგი ექვივალენტურია პროგრამული კონსტრუქციის

if (პირობა1) then ოპერანდი1

else if (პირობა2) then ოპერანდი2

. . .

else if (პირობაN) then ოპერანდი N

else ოპერანდი M

- **ოპერატორი Loop** საშუალებას იძლევა მოვახდინოთ იტერაციის მოდელირება. იგი მოქმედებს შემდეგი სახით

Loop min რაოდენობით then

while (პირობა ჭეშმარიტია)

Loop (max-min) რაოდენობით

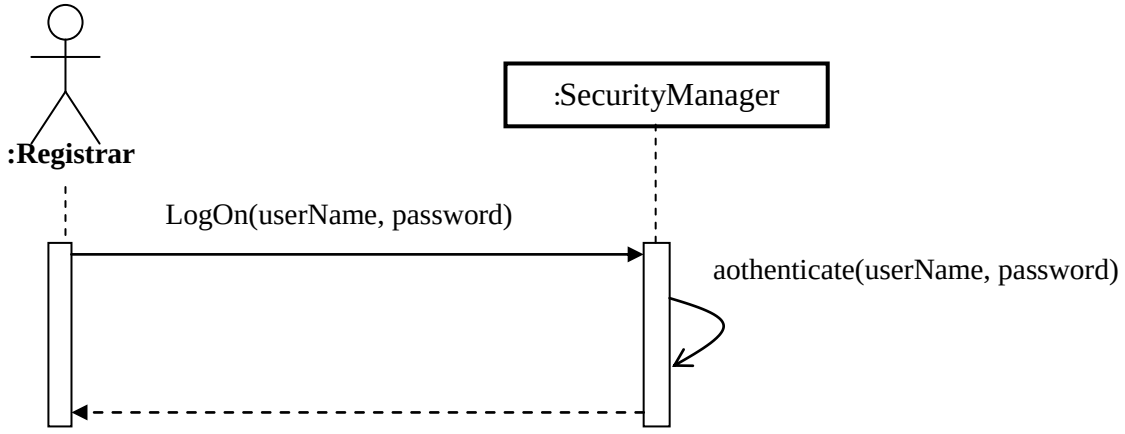
აქ უნდა აღინიშნოს, რომ თუ არ არის მოცემული max, min ოპერატორი Loop ხდება დაუსრულებელი ციკლი. თუ მოცემულია მხოლოდ min, ნიშნავს max=min.

- **ოპერატორი ref** კომბინირებული ფრაგმენტი მიმართავს სხვა ურთიერთქმედებას.
- **ოპერატორი critical** ოპერანდი სრულდება ავტომატურად უწყვეტად.

ოპერატორების სრული ნაკრები მოყვანილია [3}.

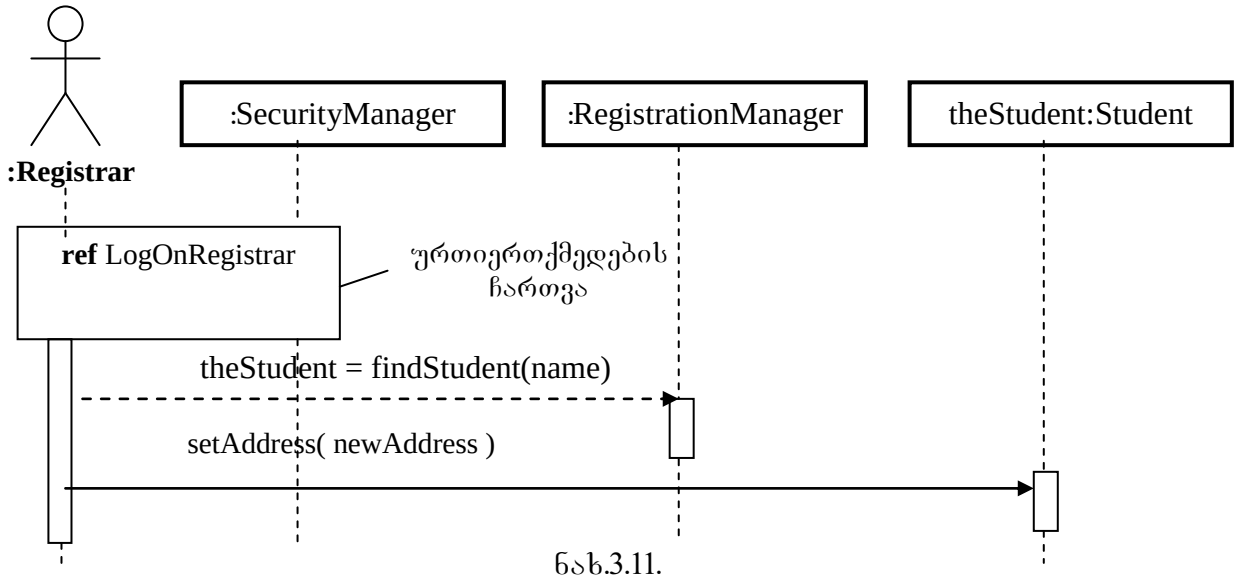
ურთიერთქმედების ჩართვა. ძალიან ხშირად შეტყობინებების ერთი და იგივე თანმიმდევრობები მრავალჯერ გამოიყენებიან სხვადასხვა მიმდევრობის დიაგრამებში. ცხადია, რომ ასეთ შემთხვევებში განმეორების თავიდან ასაცილებლად ხელსაყრელი იქნება ურთიერთქმედების ჩართვა.

მაგალითად, ხშირად ამა თუ იმ სისტემით სარგებლობა შესაძლებელია მხოლოდ ავტორიზაციის გავლის შემდეგ. სანამ აქტიორი **Registrar** (იხ.ნახ.3.10.) შეძლებს იმუშაოს სისტემასთან ის მასში უნდა შევიდეს. ამიტომ, ეს მოთხოვნა რეალიზდება კლასი SecurityManager(უშიშროების მენეჯერი)-ის მიერ. ეხლა თუ განვიხილავთ პრეცედენტს LogOnRegistrar(რეგისტრატორის შესვლა სისტემაში) იხ.ნახ.2. სისტემაში შესასვლელად იგი მოითხოვს, აქვს თუ არა გავლილი ავტორიზაცია(არის მომხმარებლის სახელი და პაროლი).



ნახ.3.10.

შესაბამისად ის ჩართული იქნება ყველა პრეცედენტში, რომელშიც **Registrar** ჯერ უნდა შევიდეს სისტემაში**.**



ნახ.3.11.

ref ოპერატორია, რომლითაც კომბინირებული ფრაგმენტი მიმართავს სხვა ურთიერთქმედებას.

ურთიერთქმედებებს შესაძლებელია გააჩნდეთ პარამეტრები. ეს საშუალებას გვაძლევს მიუთითოდ ურთიერთქმედებას სხვადასხვა მნიშვნელობები ყოველი ჩართვისას. შესაბამისად, პარამეტრები შესაძლებლობას გვაძლევენ გამოვიყენოთ ურთიერთქმედებაში კონკრეტული მნიშვნელობები.

კომუნიკაციის დიაგრამა. კოპერაციის დიაგრამა ყურადღებას ამახვილებს ურთიერთქმედებაში მონაწილე ობიექტების ორგანიზაციაზე. კომუნიკაციის დიაგრამის შექმნისათვის ურთიერთქმედებაში მონაწილე ობიექტები უნდა განვალაგოთ გრაფის მწვერვალების სახით. შემდეგ კავშირები, რომლებიც ამ ობიექტებს აკავშირებენ, გამოისახებიან ამ გრაფის წიბოების სახით. კავშირებს უმატებენ შეტყობინებებს, რომლებსაც ობიექტები ღებულობენ ან აგზავნიან. ეს საშუალებას აძლევს მომხმარებელს მიიღოს ნათელი წარმოდგენა მართვის ნაკადზე.

კომუნიკაციის დიაგრამებს გააჩნიათ ორი თვისება, რომლებიც განასხვავებენ მათ მიმდევრობის დიაგრამებისაგან:

- პირველი - ეს არის გზა. ერთი ობიექტის მეორესთან კავშირის აღწერისათვის. ამ კავშირის შუალედურ და ბოლო წერტილებს შეიძლება დაუკავშიროთ გზის სტერეოტიპი (მაგ. Local, რომელიც მიუთითებს, რომ დანიშნული ობიექტი წარმოადგენს ლოკალურს შეტყობინების გამგზავნის მიმართ).
- მეორე თვისება – შეტყობინების რიგითი ნომერია. დროის თანმიმდევრობის აღნიშვნისათვის შეტყობინებას შეიძლება დაესვას ნომერი, რომელიც თანდათან

უნდა მატულობდეს ყოველი ახალი შეტყობინებისათვის. ჩართული შეტყობინების აღნიშვნისათვის გამოიყენება ათობითი ნოტაციები 1.1., 1.2. და ა.შ.

მართვის ნაკადების სტრუქტურული ორგანიზაციის მოდელირება შედგება შემდეგი ბიჯებისაგან:

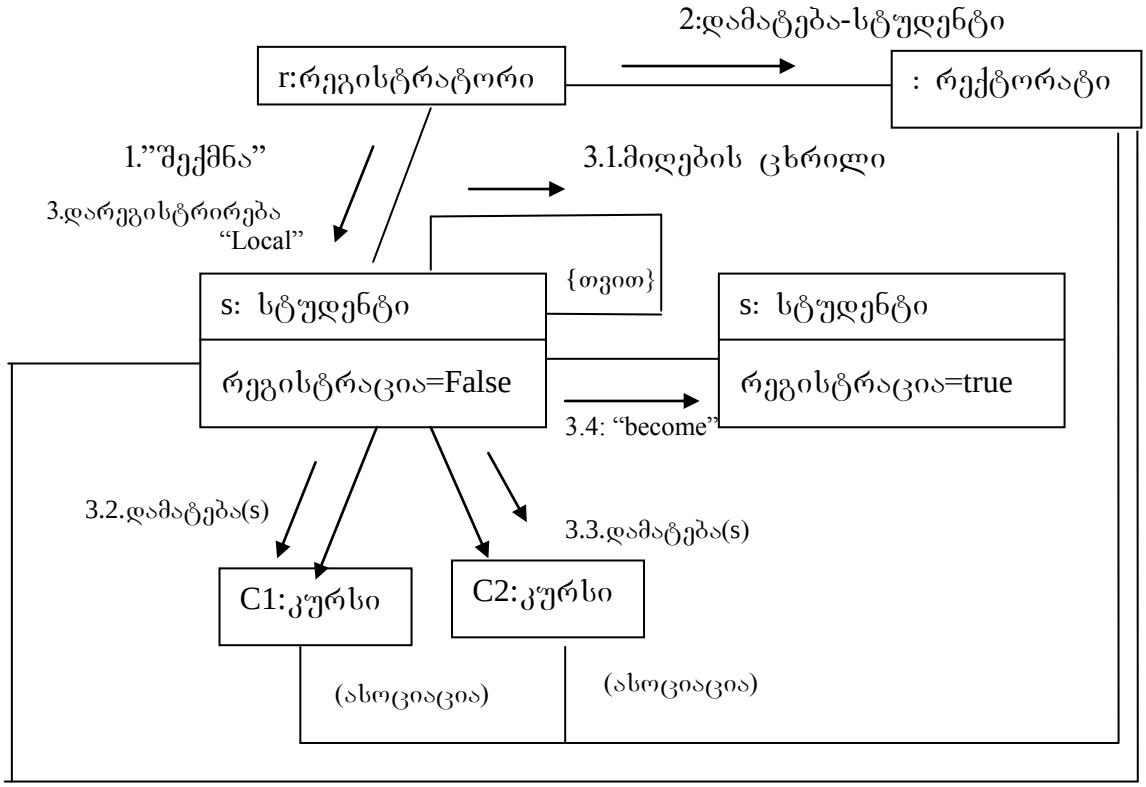
1. დავადგინოთ ურთიერთქმედების კონტექსი, ეს იქნება სისტემა, ქვესისტემა, ოპერაცია ან პრეცენდენტის ერთ ერთი სცენარი.
2. განსაზღვრეთ ურთიერთქმედების სცენა, დავადგენთ რა რომელი ობიექტები მონაწილეობენ მასში. განალაგეთ ისინი კოოპერაციის დიაგრამაზე გრაფის მწვერვალებად, ისე რომ მნიშვნელოვანი ობიექტები აღმოჩნდნენ დიაგრამის ცენტრში, ხოლო მათი მეზობლები ნაპირზე.
3. განსაზღვრეთ თითოეული ამ ობიექტის საწყისი თვისებები. თუ ატრიბუტის მნიშვნელობა, ობიექტების როლი და მდგომარეობა ურთიერთქმედებისას იცვლება, დიაგრამაზე მოათავსეთ დუბლიკატები ახალი მნიშვნელობებით და დააკავშირეთ ისინი შეტყობინების სტერეოტიპით **become** და **copy**, შესაბამისი რიგითი ნომრების დართვით.
4. დეტალურად ავღწეროთ კავშირები ობიექტებს შორის, რომელთა გასწვრივაც გადაიცემა შეტყობინებები. ამისათვის:

- თავიდან მიუთითეთ კავშირი ასოციაცია. ისინი ყველაზე მნიშვნელოვანია, რამდენადაც წარმოადგენენ სტრუქტურულ შეერთებებს.
- ამის შემდეგ მიუთითეთ დანარჩენი კავშირები, დაამატებთ რა მას გზის შესაბამის სტერეოტიპებს(როგორიც არის **global** ან **local**).

5. დაწყებული ურთიერთქმედების ინიციირების შეტყობინებიდან, დაუკავშირეთ ყველა დანარჩენ კვანძებს შეტყობინებები, მონიშნეთ რიგითი ნომრები.
6. თუ საჭიროა დროითი ან სივრცითი შეზღუდვების სპეციფიცირება დაუმატეთ შეტყობინებებს დროითი ნიშნები და დაუკავშირეთ საჭირო შეზღუდვები.

თუ საჭიროა მართვის ნაკადების უფრო ფორმალური აღწერა დაუკავშირეთ ყოველ შეტყობინებას წინა და შემდგომი პირობები.

ისევე როგორც მიმდევრობის დიაგრამაზე, აქაც ერთი კოოპერაციის დიაგრამით შეიძლება მხოლოდ ერთი მართვის ნაკადის ჩვენება, ამიტომ როგორც წესი ქმნიან ურთიერთქმედების რამოდენიმე დიაგრამას, რომელთაგან ერთი ითვლება ძირითადათ, ხოლო დანარჩენები აღწერენ ალტერნატიულ და განსაკუთრებულ პირობებს. კოოპერაციის დიაგრამების ასეთი ერთობლიბა, შეიძლება გავაერთიანოთ პაკეტში, მივცეთ რა თვითეულს შესაბამისი სახელი.



ნახ. 3.12.

მაგალითისათვის ნახ.3.12.–ზე მოყვანილია კომუნიკაციის დიაგრამა, რომელიც აღწერს მართვის ნაკადს, დაკავშირებულს ახალი სტუდენტის რეგისტრაციასთან. დიაგრამაზე წარმოდგენილია ხუთი ობიექტი: ***რეგისტრატორი, სტუდენტი, ორი ობიექტი კურსი და რექტორატი (უმაღლესი სასწავლებელი)***. მოქმედება იწყება იმით, რომ ***რეგისტრატურა*** ქმნის ობიექტს ***სტუდენტი*** და ამატებს მას ***უმაღლეს სასწავლებელს***(შეტყობინება ***დამატება სტუდენტი***), ხოლო შემდეგ აძლევს მას მითითებას დარეგისტრირდეს. ამის შემდეგ ობიექტი ***სტუდენტი*** უგზავნის შეტყობინებას თავისთავს ***მიღების ცხრილი***, მიღებული აქვს რა თავიდან ***კურსი,*** რომელზეც მას სურს ჩაწერა. შემდეგ ობიექტი ***სტუდენტი*** ამატებს თავისთავს ***კურსის*** ყოველ ობიექტს. ბოლოს ისევ არის ნაჩვენები ობიექტი ***სტუდენტი*** ატრიბუტ ***დარეგისტრირების*** განახლებული მნიშვნელობით.

3.8. მოღვაწეობის-აქტიურობის დიაგრამა

მოღვაწეობის დიაგრამებს ხშირად უწოდებენ “ობიექტ-ორიენტირებულ ბლოქ-სქემებს”. ისინი საშუალებას გვაძლევენ მოვახდინოთ პროცესის მოდელირება როგორც

მოღვაწეობის, რომელიც შედგება წახნაგებით დაკავშირებული კვანძების ერთობლიობისაგან.

UML1-ში მოღვაწეობის დიაგრამები ფაქტიურად წარმოადგენდნენ მდგომარეობის დიაგრამების განსაკუთრებულ შემთხვევას, სადაც ყოველ მდგომარეობას ქონდა შემავალი მოქმედება, რომელიც განსაზღვრავდა გარკვეულ პროცესს ან ფუნქციას, რომლებსაც ადგილი აქვთ მდგომარეობაში შესვლისას. UML2-ში მოღვაწეობის დიაგრამებს აქვთ სრულიად ახალი სემანტიკა, რომელიც ბაზირდება პეტრის ქსელების ტექნოლოგიაზე. ამ ტექნოლოგიის გამოყენებას აქვს ორი უპირატესობა:

1. პეტრის ქსელების ფორმალიზმი უზრუნველყოფს მეტ მოქნილობას სხვადასხვა ტიპის ნაკადების მოდელირებისას.
2. UML-ში ამჟამად არის მკაფიო გამიჯვნა მოღვაწეობის და მდგომარეობის დიაგრამებს შორის.

მოღვაწეობა შესაძლებელია დაუმატოთ მოდელის ნებისმიერ ელემენტს მისი ქცევის მოდელირებისათვის. მოღვაწეობას ჩვეულებრივ უმატებენ:

- პრეცედენტებს;
- კლასებს;
- ინტერფეისებს;
- კომპონენტებს;;
- კოოპერაციებს;
- ოპერაციებს.

მოღვაწეობის დიაგრამები ასევე შესაძლებელია გამოყენებულ იქნან ბიზნეს-პროცესებისა და სამუშაო პროცესების მოდელირებისათვის. აგრეთვე, მას იყენებენ UP-ს სამუშაო ნაკადებში. კერძოთ:

- ანალიზის პროცესში:
  - პრეცედენტის ნაკადის გრაფიკული მოდელირებისათვის;
  - პრეცედენტებს შორის ნაკადის მოდელირებისათვის. ამ დროს გამოიყენება მოღვაწეობის დიაგრამის განსაკუთრებული ფორმა – ურთიერთქმედების მიმოხილვის დიაგრამა.
- დაპროექტებისას:
  - ოპერაციის დეტალების მოდელირებისათვის;
  - ალგორითმის დეტალების მოდელირებისათვის.
- საქმიანი აქტიურობის მოდელირებისას:
  - ბიზნეს-პროცესების მოდელირებისათვის.

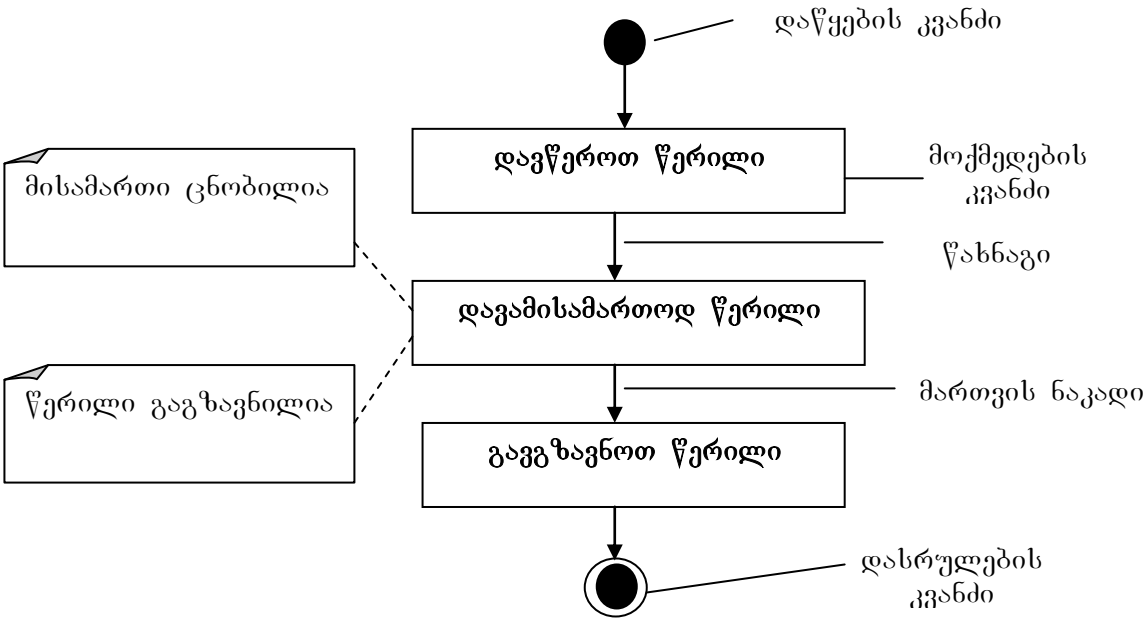
ჩვეულებრივ შემკვეთები უფრო ადვილად გებულობენ მოღვაწეობის დიაგრამებს, რამდენადაც მათი უმრავლესობა ასე თუ ისე წარმოდგენა აქვთ ბლოქ-სქემებზე.

მოღვაწეობები. მოღვაწეობები ეს კვანძების სისტემაა, დაკავშირებულნი წახნაგებით. არსებობს კვანძების სამი კატეგორია:

1. **მოქმედების კვანძები** - შედგენილია ელემენტარული გამოთვლებისაგან, რომლის შემდგომი დეკომპოზიცია შეუძლებელია. ეს ნიშნავს, რომ მათ შიგნით შეიძლება ხდებოდეს სხვადასხვა მოვლენები, მაგრამ მოქმედების მდგომარეობაში შესრულებადი სამუშაო არ შეიძლება შეწყვეტილ იქნას. ჩვეულებრივ დაშვებულია, რომ ერთი მოქმედების მდგომარეობის ხანგრძლივობა იკავებს ძალიან მცირე დროს. მოქმედება შეიძლება მდგომარეობდეს სხვა ოპერაციის გამოძახებაში, რაიმე სიგნალის გაგზავნაში, ობიექტის შექმნაში ან მოსპობაში ან უბრალო გამოსახულების გამოთვლაში.
2. **მართვის კვანძები –** მართავს მოღვაწეობის ნაკადს. შეიძლება მოღვაწეობის მდგომარეობების შემდგომი დეკომპოზიცია, ამის შედეგად შესრულებადი მოღვაწეობა შეიძლება წარმოვადგინოთ სხვა მოღვაწეობით. მოღვაწეობის მდგომარეობა არ არის ელემენტარული, ანუ შეიძლება შეწყვეტილ იქნან. ამასთან იგულისხმება, რომ მათი დამთავრებისათვის საჭიროა საკმაო დრო.
3. **ობიექტური კვანძები** - წარმოადგენენ ობიექტებს, რომლებიც გამოიყენებიან მოღვაწეობისას.

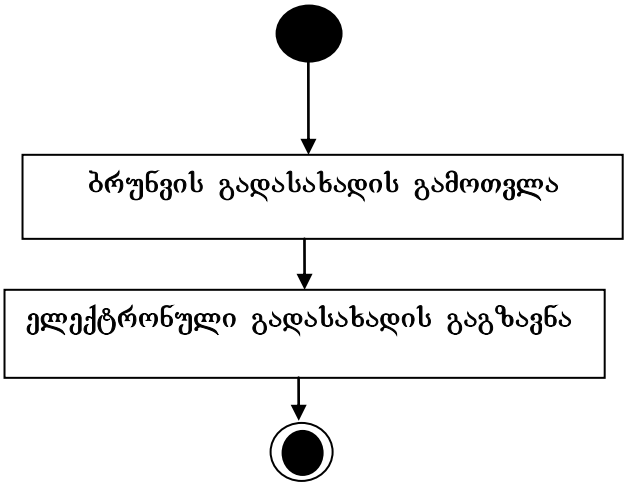
განვიხილოთ მაგალითი. ნახ.1.-ზე ნაჩვენებია მოღვაწეობის მარტივი დიაგრამა ბიზნეს-პროცესისა **გავგზავნოთ წერილი**. როგორც ჩანს, მოღვაწეობებს აქვთ წინაპირობა და შედეგი, ისევე როგოც პრეცედენტებს. წინაპირობა – ეს პირობებია, რომლებიც ჭეშმარიტი უნდა იყვნენ, იმისათვის, რომ მოღვაწეობა დაიწყოს (მოცემული მაგალითისათვის წინაპირობაა – ვიპოვოთ წერილის თემა), ხოლო შედეგი – ეს პირობაა, რომლებიც ჭეშმარიტი ხდებიან მოღვაწეობის დამთავრებისას(მოცემული მაგალითისათვის შედეგია – წერილი გაგზავნილია მისამართით). მოქმედებებს მოღვაწეობის შიგნით შეიძლება ქონდეთ საკუთარი ლოკალური წინაპირობები და შედეგები.

მოღვაწეობა ჩვეულებრივ იწყება მართვის ერთი კვანძიდან, **საწყისიდან**. ის აღნიშნავს შესრულების დაწყებას მოღვაწეობის გამოძახებისას, ერთი ან რამოდენიმე დასრულების კვანძები გვიჩვენებენ მოღვაწეობის **დასრულების** ადგილს.



ნახ.3.13.

ჩვეულებრივ მოღვაწეობის დიაგრამებს იყენებენ პრეცედენტის მოდელირებისათვის მოქმედებათა თანმიმდევრობის სახით. ნახ.2.-ზე ნაჩვენებია მოღვაწეობის დიაგრამა პრეცედენტისათვის ბრუნვის გადახახადი, რომლის წინაპირობაა - გადახდის პერიოდი, შედეგი - სამმართველო ღებულობს შესაბამის თანხას.



ნახ.3.14.

როგორც ჩანს, მოღვაწეობის დიაგრამა - ეს პრეცედენტის უფრო კომპაქტური და გრაფიკული ფორმაა. თითოეული ამ მოქმედებიდან შესაძლებელია წარმოდგენილი იქნას ცალკე მოღვაწეობის დიაგრამით.

<u>მოღვაწეობის სემანტიკა</u>. UML2-ში მოღვაწეობის დიაგრამები დაფუძნებულია პეტრის ქსელების ტექნოლოგიაზე. ქცევის მოდელირება აქ ხდება მარკერების "თამაშის" საშუალებით. ეს თამაში აღწერს მარკერების ნაკადს, რომლებიც მოძრაობენ კვანძებისა

და წახნაგების ქსელში განსაზღვრული წესით. მარკერები მოღვაწეობის დიაგრამაზე შეიძლება წარმოადგენდნენ:

- მართვის ნაკადს;
- ობიექტს;
- გარკვეულ მონაცემებს.

სისტემის მდგომარეობა დროის ნებისმიერ მომენტში განისაზღვრება მისი მარკერების განლაგებით.

მარკერები გადაადგილდებიან წახნაგების გასწვრივ საწყისი კვანძიდან მიზნობრივ კვანძამდე. მარკერების გადაადგილება ხდება ყველა აუცილებელი პირობის შესრულებისას. პირობები იცვლება კვანძის ტიპის შესაბამისად. მოყვანილი მაგალითისათვის(იხ.ნახ1) ამ პირობებს წარმოადგენენ:

- საწყისი კვანძის შედეგი;
- წახნაგის დაცვის პირობები;
- მიზნობრივი კვანძის წინაპირობა.

პირობები არსებობს არა მარტო მოქმედების კვანძებისათვის, არამედ მართვისა და ობიექტური კვანძებისათვის. მართვის კვანძებს აქვთ განსაკუთრებული სემანტიკა, რომელიც მართავს თუ როგორ გადაეცემა მარკერები შემავალი წახნაგებიდან გამოსასვლელებს. მაგალითად, საწყისი კვანძი იწყებს მოღვაწეობას, დასრულების კვანძი ასრულებს მოღვაწეობას, გაერთიანების კვანძი შეთავაზებს მარკერს მის ერთადერთ გამოსასვლელ წახნაგზე მხოლოდ იმ შემთხვევაში, თუ მარკერები შემოვიდნენ მის ყველა შემავალ წახნაგზე. ობიექტური კვანძები წარმოადგენენ კვანძებს, რომლებიც არსებობენ სისტემაში.

განვიხილოთ მარკერების “თამაში” მოღვაწეობისათვის, რომელიც მოყვანილია ნახ.3.13-ზე. მოღვაწეობის შესრულების დაწყებისას დაწყების კვანძში იწყება მართვის მარკერების მოქმედება.არც დაწყების კვანძზე, არც მის გამოსასვლელ წახნაგზე, არც მიზნობრივ კვანძზე არ არის დადებული არც ერთი პირობა, ამიტომ მარკერ ავტომატურად გადის გამოსასვლელი წახნაგით მიზნობრივ კვანძთან **დავწეროთ წერილი.** ეს ახდენს მოქმედების ინიციირებას, რომელიც განისაზღვრება მოქმედების კვანძით **დავწეროთ წერილი.** მოქმედების **დავწეროთ წერილი** დამთავრებისას მართვის მარკერების ნაკადი გადადის მოქმედების კვანძში **დავამისამართოდ წერილი** მხოლოდ მაშინ, როდესაც კმაყოფილდება მისი წინაპირობა მისამართი ცნობილია. შედეგის წერილი დამისამართდა შესრულების შემდეგ მართვა გადადის დავამისამართოდ წერილიდან გავგზავნოთ წერილი. ბოლოს, რამდენადაც პირობები, რომლებიც დააბრკოლებენ ნაკადის გამოსვლას

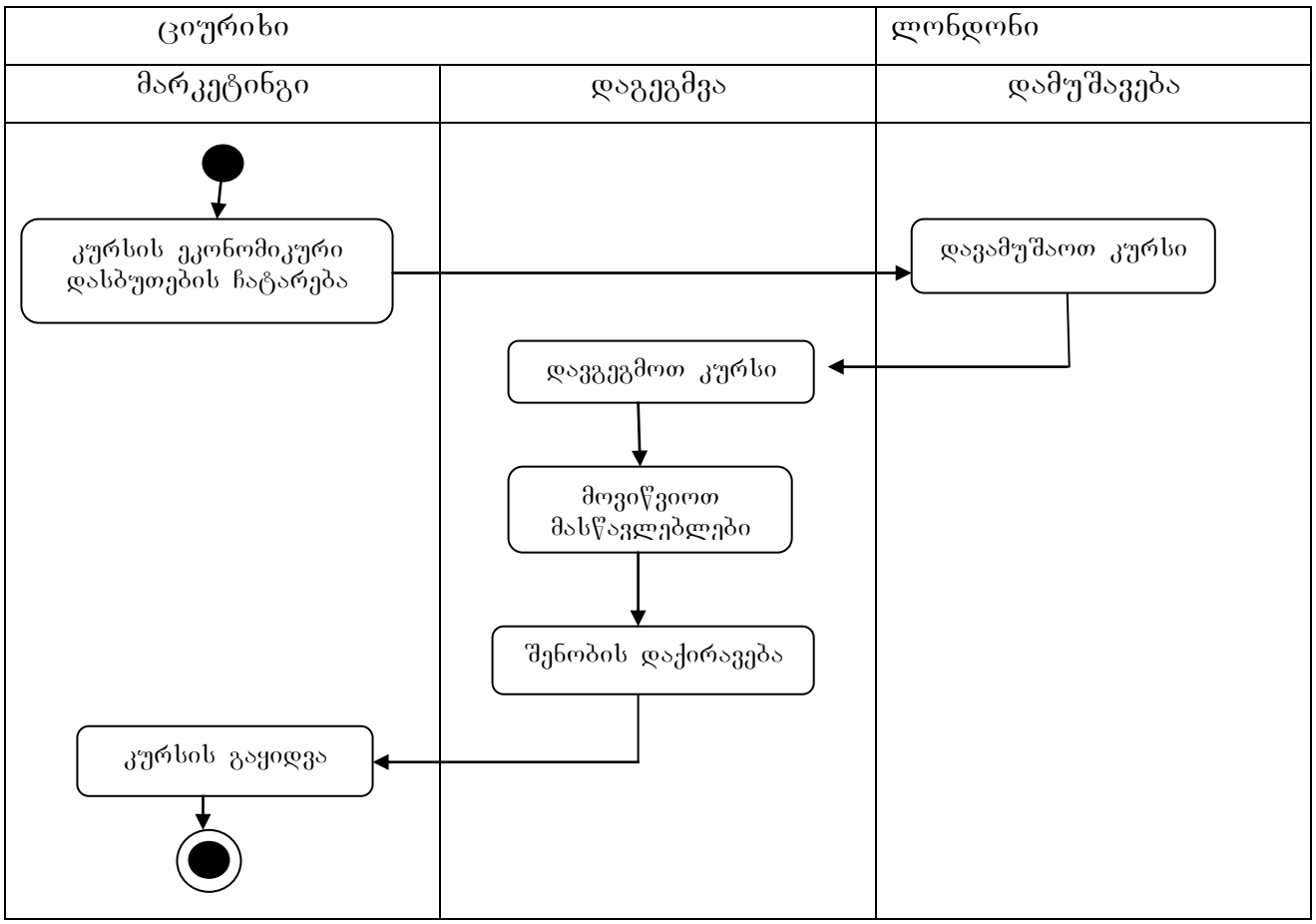
გავგზავნოთ წერილი არ არის, მართვის ნაკადი შედის დასრულების მდგომარეობაში და მოღვაწეობა მთავრდება.

თუმცა მოღვაწეობის სემანტიკა UML2-ში აღიწერება მარკერების "თამაშით", ისინი ასეთი სახით არ რეალიზდებიან. მოღვაწეობა – ეს მხოლოდ აღწერაა, რომლისთვისაც შესაძლებელია უამრავი რეალიზაცია.

<u>მოღვაწეობის განყოფილებები</u>. მოღვაწეობის დიაგრამები წაკითხვის გამარტივების მიზნით შესაძლებელია დაიყოს განყოფილებებათ ვერტიკალური ან ჰორიზონტალური ხაზებით. მოღვაწეობის ყოველი განყოფილება – ეს მოქმედებების ურთიერთდაკავშირებული მოქმედებების ერთობაა ჩართვის რამოდენიმე დონით. მოღვაწეობის განყოფილებებს კიდევ უწოდებენ ბილიკებს. მოღვაწეობის განყოფილებების სემანტიკას განსაზღვრავს მოდელის დამმუშავებელი. მას ჩვეულებრივ იყენებენ პრეცედენტების წარმოსადგენად. განყოფილებების სიმრავლეს უნდა გააჩნდეს ერთი განზომილება, რომლითაც აღიწერება მისი სემანტიკა. ამ განზომილების ჩარჩოში განყოფილებები შესაძლებელია იყოს იერარქიულად ჩართული. მაგალითად ნახ.3.-ზე ნაჩვენებია მოღვაწეობა, რომელსაც აქვს იერარქიულად ჩართული მოღვაწეობის განყოფილებების სიმრავლე.

ადგილმდებარეობა – ეს განზომილებაა და ამ განზომილების ჩარჩოში არსებობს განყოფილებების იერარქია.

ხშირად აუცილებელია უჩვენოთ მოღვაწეობის დიაგრამაზე ქცევა, რომელიც იმყოფება სისტემის მოქმედების სფეროს გარეთ, მაგალითად, სისტემის ურთიერთქმედება რომელიღაც გარე სისტემასთან. ამისათვის მოღვაწეობის დიაგრამის შესაბამისი განყოფილების დასახელების თავზე მიეთითება სტერეოტიპი "external". გარე განყოფილება არ წარმოადგენს სისტემის ნაწილს, ამიტომ იგი არ შეიძლება შედიოდეს მოდელის განყოფილების რომელიმე იერარქიაში.



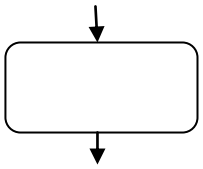
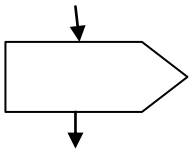
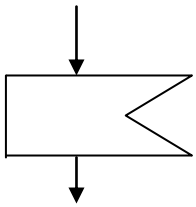
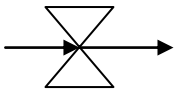
მოქმედების კვანძები. მოქმედების კვანძები სრულდებიან იმ შემთხვევაში თუ:

- მარკერები ერთდროულად შედიან ყველა შემავალ წახნაგებზე;
- შემავალი მარკერები აკმაყოფილებენ მოქმედების კვანძების ყველა ლოკალურ წინაპირობებს.

მოქმედების კვანძები ანხორციელებენ ლოგიკური **და** ოპერაციას მათ შემავალ მარკერებზე – კვანძი არ არის მზად შესრულებისათვის მანმადე, სანამ მარკერები არ იქნებიან ყველა შემავალ წახნაგებზე. მოქმედების კვანძის შესრულების დასრულებისას მოწმდება ლოკალური შედეგი. თუ ის კმაყოფილდება, კვანძი ერთდროულად სთავაზობს მარკერებს ყველა მის გამომავალ წახნაგებზე.

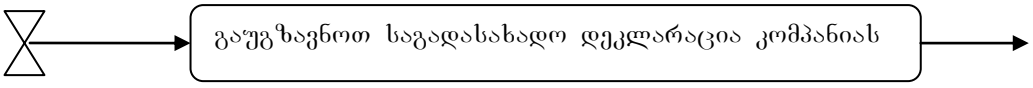
მოქმედების დეტალები ფიქსირდება მოქმედების კვანძის აღწერაში. მაგრამ პროექტირების ეტაპზე იგი გარდაიქმნება სტრუქტურირებულ ტექსტში, ფსევდოკოდში ან რეალურ კოდში.

მოქმედების კვანძის ოთხი ტიპი არსებობს, ისინი მოყვანილია ცხრ.1.

სინტაქსისი	დასახელება	სემანტიკა
	მოქმედების გამოძახების კვანძი	ახდენს მოღვაწეობის, ქცევის ან ოპერაციის ინიციირებას
	სიგნალის გაზავნა	მოქმედება სიგნალის გაგზავნა – აგზავნის სიგნალს ასინქრონულად(გამგზავნი არ ელოდება სიგნალის მიმღების დადასტურებას). სიგნალის შექმნისათვის შესაძლებელია ღებულობდეს შემავალ პარამეტრებს.
	მოქმედების კვანძი, რომელიც ღებულობს სიგნალს	ღებულობს მოვლენას – ელოდება მოვლენას, დადგენილი ობიექტი-მფლობელისაგან და გამოსცემს მოვლენას გამოსასვლელზე. აქტივირდება მარკერის მიღებისას შემავალ წახნაგზე. თუ შემავალი წახნაგი არ არის, გაიშვება მისი შემცველი მოღვაწეობის გაშვებისას და ყოველთვის არის აქტივირებული.
	მოქმედების კვანძი, რომელიც ღებულობს დროით მოვლენას	ღებულობს დროით მოვლენას – პასუხობს გარკვეულ დროით მნიშვნელობაზე. ახდენს დროითი მოვლენის გენერირებას თავისი დროითი გამოსახულების შესაბამისად.

მოქმედების გამოძახების კვანძს შეუძლია მოახდინოს მოღვაწეობის, ქცევის ან ოპერაციის ინიციირება.

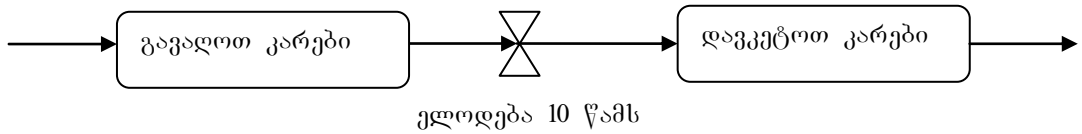
მოქმედების კვანძი, რომელიც ღებულობს დროით მოვლენას, რაეგირებს მასზე. მაგალითად, ნახ.3.-ზე ნაჩვენებია მოქმედების კვანძი, რომელიც ღებულობსდროით მოვლენას, შემავალი წახნაგის გარეშე. ეს კვანძი გახდება აქტიური და მოახდენს დროითი მოვლენის გენერირებას მოღვაწეობის მფლობელის გაშვების შემდეგ, როდესაც მისი დროითი გამოსახულება გახდება ჭეშმარიტი. მოყვანილ მაგალითში დროითი მოვლენა გენერირდება ყოველი საფინანსო წლის წლის ბოლოს და ახდენს მოღვაწეობის **გაუგზავნოთ საგადასახადო დეკლარაცია** კომპანიას.



დადგა საფინანსო წლის დასასრული

ნახ.3.15.

მაგრამ მაგალითში ნახ.4. მოქმედება, რომელიც ღებულობს დროით მოვლენას, აქვს შემავალი წახნაგი და ხდება აქტიური მხოლოდ ამ წახნაგზე მარკერის მიღების შემდეგ. იგი შესაძლებელია დაუკავშიროთ ლიფტის მართვისსისტემის ფრაგმენს. პირველი მოქმედება აღებს ლიფტის კარებს და ხსნის მოქმედებას, რომელიც ღებულობს დროით მოვლენას. ეს მოქმედება იცდის 10 წამს და შემდეგ გადასცემს მარკერს მოქმედებას დაიხუროს კარები.



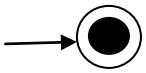
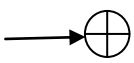
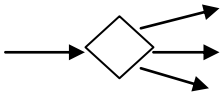
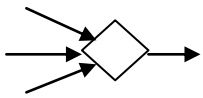
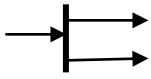
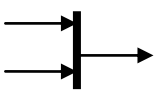
ნახ.3.16.

შესაბამისად, დროითი მოვლენა შესაძლებელია მიგვითითებდეს:

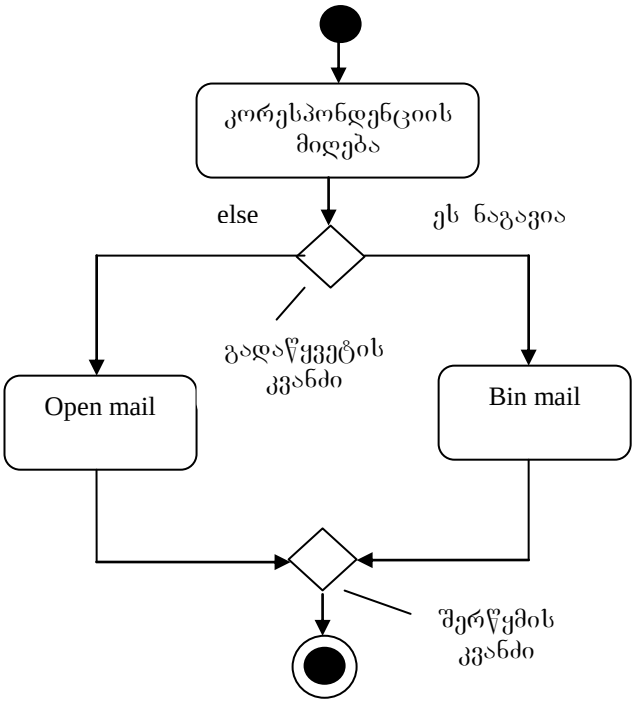
- გარკვეულ მოვლენაზე;
- დროის კონკრეტულ მომენტზე;
- დროით ინტერვალზე.

მართვის კვანძები. მართვის კვანძები აკონტროლებენ მოღვაწეობის მართვის ნაკადს. ცხრილში 4. მოყვანილია მართვის ყველა კვანძები.

ცხრ.4.

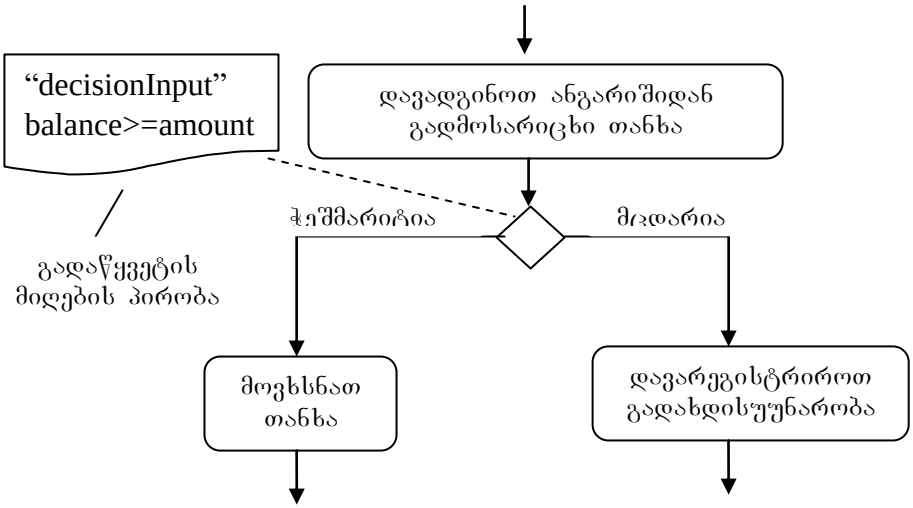
სინტაქსისი	დასახელება	სემანტიკა
	დაწყების კვანძი	მიუთითებს, თუ სად იწყება ნაკადი მოღვაწეობის გამოძახებისას
	მოღვაწეობის დასრულების კვანძი	ამთავრებს მოღვაწეობას
	ნაკადის დასრულების კვანძი	ამთავრებს მოღვაწეობის გარკვეულ ნაკადს – სხვა ნაკადების გაუთვალისწინებლად
	გადაწყვეტის კვანძი	გამომავალ წახნაგზე გადის ნაკადი, რომლის დაცვის პირობა სრულდება.
	შერწყმის კვანძი	აერთიანებს შემავალ მარკერებს ერთადერთ გამომავალ წახნაგზე.
	განშტოების კვანძი	ყოფს ნაკადს რამოდენიმე პარალელურ ნაკადებათ
	გაერთიანების კვანძი	ახდენს რამოდენიმე პარალელური ნაკადის სინქრონიზირებას

ნახ.5.-ზე მოყვანილია მარტივი მაგალითი, რომელიც წარმოგვიდგენს გადაწყვეტის და შერწყმის კვანძების გამოყენებას.



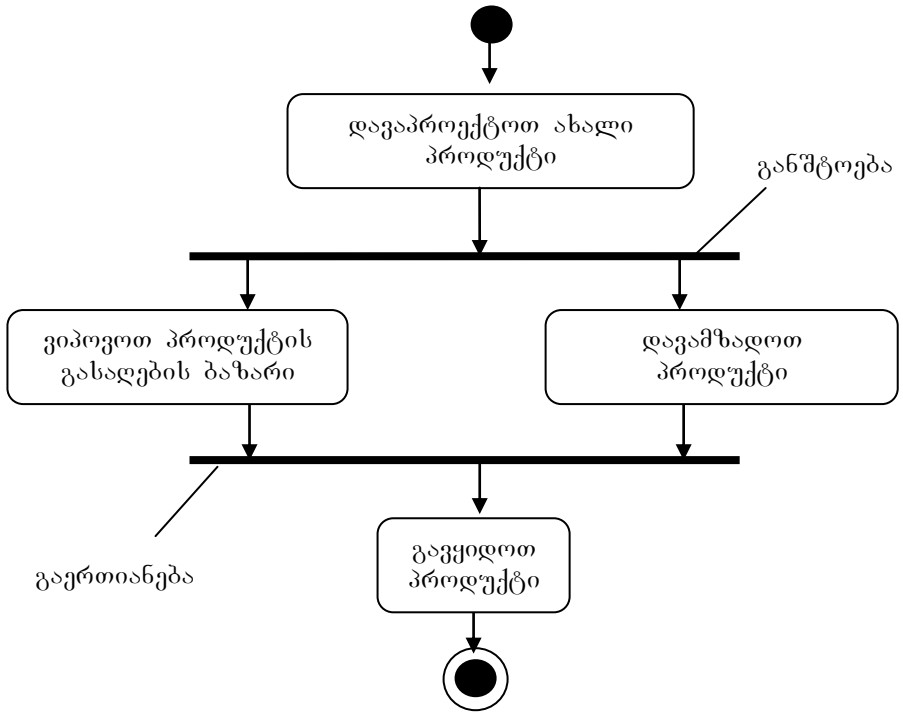
ნახ.3,17.

გადაწყვეტის კვანძისათვის შესაძლებელია გამოყენებულ იქნას სტერეოტიპი “decisionInput” (გადაწყვეტის შემავალი მონაცემები). მაგალითად:



ნახ.3.18.

განსტოების კვანძი ყოფს ნაკადს რამოდენიმე პარალელურ ნაკადათ, ხოლო გაერტიანების კვანძი ახდენს სინქრონიზირებას და აერთიანებს რამოდენიმე შემავალ ნაკადს ერთადერთ გამომავალში. მაგალითად ნახ.6.-ზე წარმოდგენილია პროდუქციის წარმოების პროცესი, რომელშიც გამოიყენება განშტოებისა და გაერთიანების კვანძები.



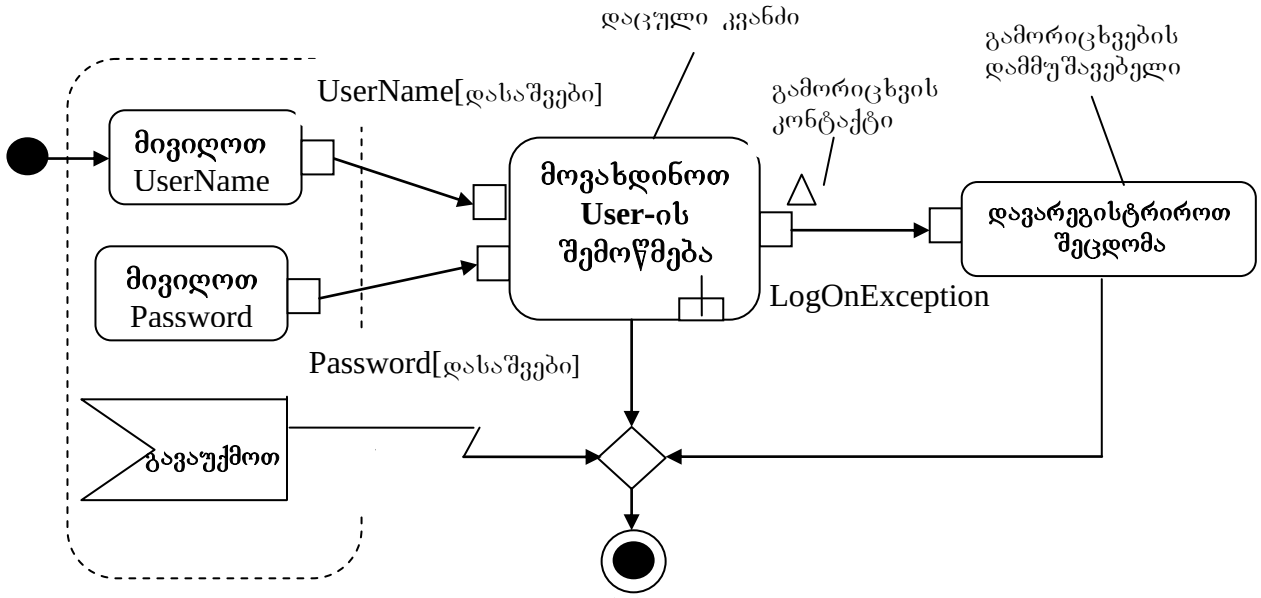
ნახ.3.19,

ობიექტების კვანძები – ეს სპეციალური კვანძებია, რომლებიც გვიჩვენებენ, რომ კონკრეტული კლასიფიკატორების ეგზემპლიარები მისაღებია მოღვაწეობის მოცემულ წერტილში. ისინი აღინიშნებიან კლასიფიკატორის სახელით და წარმოადგენენ მის ეგზემპლიარებს. ობიქტების ნაკადი წარმოადგენს ობიქტების მოძრაობას მოღვაწეობაში.

მოღვაწეობის აღწერისას საჭირო ხდება აღვწეროთ მოქმედების შეწყვეტის შესაძლებლობა. მოღვაწეობის ასეთ ნაწილებში ხდება შესაწყვეტ წახნაგზე მარკერის გადაადგილების შეწყვეტა. ისინა განსაკუთრებით სასარგებლოა ასინქრონული სიგნალებისა და წყვეტების მოდელირებისათვის.

გარდა ამისა, შეცდომების აღმოჩენისას უნდა არსებობდეს მექანიზმი, რომელიც მოახდენს მათი - გამორიცხვების დამუშავებას. შეცდომის აღმოჩენისას იქმნება გამორიცხვის ობიექტი. მართვის ნაკადი გადადის გამორიცხვების დამმუშავებელში, რომელიც გარკვეული საშუალებებით ამუშავებს გამორიცხვის ობიექტს. გამორიცხვის დამმუშავებელმა შესაძლებელია შეწყვიტოს მოღვაწეობა ან შეეცადოს აღადგინოს ნორმალური მდგომარეობა. ნახ.7.-ზე მოყვანილია მარტივი მოღვაწეობა **შევიდეთ** სისტემაში, რომელსაც აქვს მოქმედების წყვეტის ადგილი. იგი აღნიშნულია წყვეტილი სწორკუთხედით და მოიცავს მოქმედებებს **მივიღოთ** UserName, **მივიღოთ** Password და **გავაუქმოთ.** თუ მოქმედება **გავაუქმოთ** ღებულობს მოვლენას Cancel იმ მომენტში, როდესაც მართვა იმყოფება ამ ნაწილში, მას გამოყავს მარკერ წყვეტის წახნაგზე და

უცრად წყვეტს ამ ნაწილს. ყველა მოქმედებები **მივიღოთ** UserName, **მივიღოთ** Password და **გავაუქმოთ** – შეწყდება.

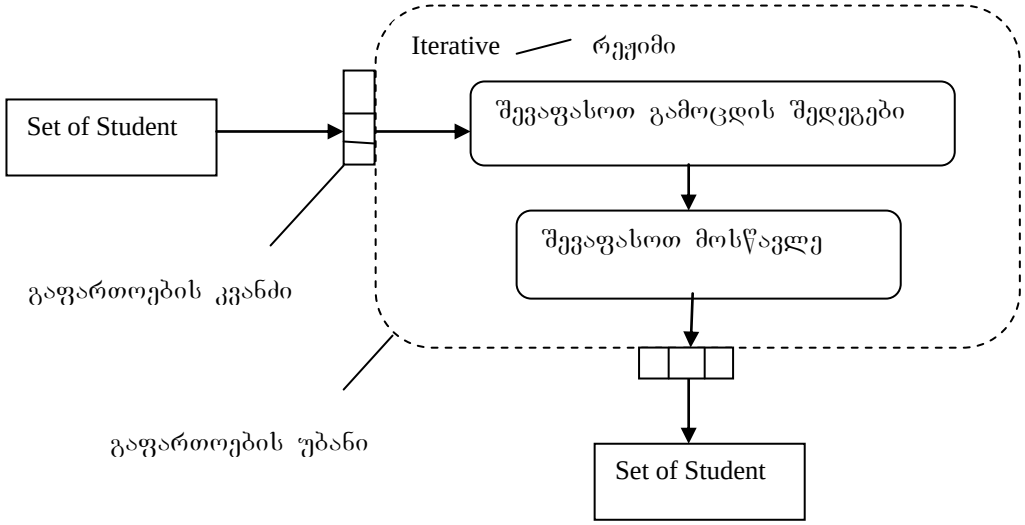


ნახ.3.20.

წყვეტადი წახნაგები გამოისახებიან ზიგზაგისებრი ისრებით(იხ.ნახ.3.20.). ნახაზზე მოქმედებები **მივიღოთ** UserName და **მივიღოთ** Password მოქმედებას მოვახდინოთ სერ-ის შემოწმება დაკავშირებულია კონტაქტებით. კონტაქტი – ეს ობიექტური კვანძია, რომელიც წარმოადგენს ერთ შესასვლელს ან გამოსასვლელს მოქმედებიდან. ჩვენს შემთხვევაში UserName[დასაშვები] და Password[დასაშვები]. მოღვაწეობა **მოვახდინოთ User-ის შემოწმება** გამოსცემს ობიექტს LogOnException (გამორიცხვა შესვლისას) მომხმარებლის შემოწმების მცდარი შედეგისას. ეს ობიექტი მიირება მოქმედებით **დავარეგისტრიროთ შეცდომა,** რომელიც წერს ინფორმაციას შეცდომის შესახებ რეგისტრაციის ჟურნალში. იმისათვის, რომ უჩვენოთ, რომ გამოსავალი კონტაქტი წარმოადგენს გარიცხვის ობიექტს, მას მიანიშნებენ მცირე სამკუთხედით. კვანძი **დავარეგისტრიროთ შეცდომა** გამოდის გამორიცხვის დამმუშავებლის როლში, რომელიც გენერირდება მოქმედებით **მოვახდინოთ User-ის შემოწმება.**

<u>გაფართოების კვანძები</u>. გაფართოების კვანძები საშუალებას გვაძლევენ უჩვენოთ ობიექტთა კოლექციის დამუშავება როგორც მოღვაწეობის დიაგრამის ნაწილი, რომელსაც უწოდებენ გაფართოების უბანს(extension regon).

გაფართოების კვანძი – ეს ობიექტური კვანძია, რომელიც წარმოადგენს ობიექტების კოლექციას, რომლებიც შედიან ან გამოდიან გაფართოების უბნიდან. ნახ.3.21-ზე მოყვანილია გაფარტოების უბნის მაგალითი. იგი წარმოდგენილია პუნქტირი სწორკუთხედით.



ნახ.3.21.

გაფართოების კვანძებზე დაიდება შემდეგი შეზღუდვები:

- გამომავალი კოლექციის ტიპი უნდა შეესაბამებოდეს შემავალი კოლექციის ტიპს..
- შემავალი და გამომავალი კოლექციების ტიპები უნდა იყვნენ ერთნაირები.

ყოველ გაფართოების უბანს აქვს რეჟიმი, რომელიც განსაზღვრავს შემავალი კოლექციების დამუსავების თანმიმდევრობას:

1. iterative (იტერაციული) - სემავალი კოლექციის ყოველი ელემენტი მუშავდება მიმდევრობით;
2. parallel (პარალელური) – შემავალი კოლექციის ელემენტები მუშავდებიან პარალელურად;
3. stream (ნაკადური) - შემავალი კოლლექცის ელემენტები მუშავდებიან კვანძში შემოსვლის მიხედვით.

გაფართოების უბანი რებულობს ობიექტების ნაკრებს Student(მოსწავლე). იგი ამუშავებს თითოეულ ობიექტს რიგის მიხედვით (რეჟიმი = iterative) და გამოსცემს დამუშსავებული ობიექტების ნაკრებს Student.

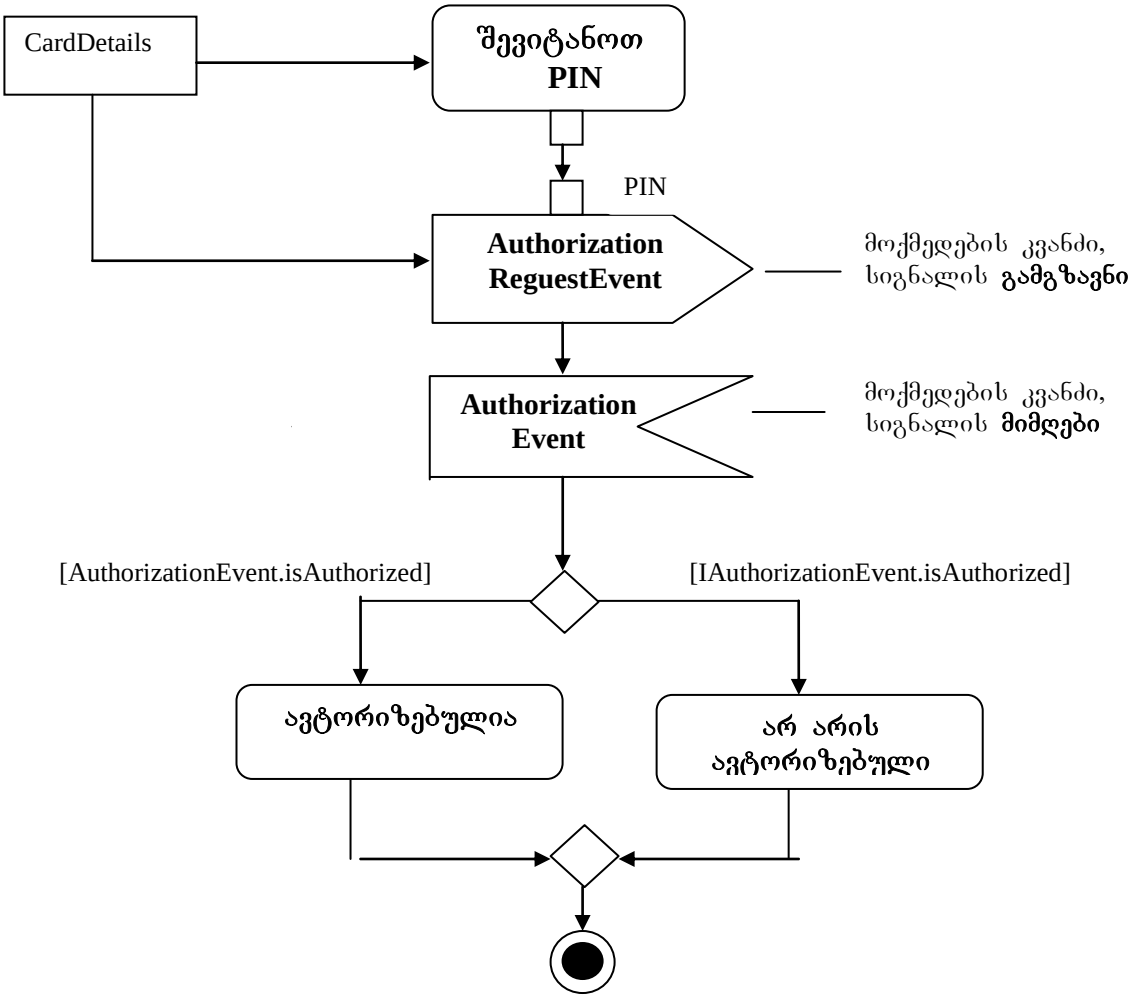
<u>სიგნლის გაგზავნა და მოვლენის მიღება.</u> სიგნალი წარმოადგენს ობიექტებს შორის ასინქრონულად გადაცემულ ინფორმაციას. სიგნალი მოდელირდება როგოეც კლასი, მონიშნული სტერეოტიპით “სიგნალ”.

ნახ.3.22.-ზე მოყვანილია მოღვაწეობა **შევამოწმოთ საკრედიტო ბარათი,** რომელიც აგზავნის მოვლენას AuthorizationReguestEvents(ავტორიზაციის დაკვეთის მოვლენა) და ღებულობს მოვლენას AuthorizationEvents (ავტორიზაციის მოვლენა). ორივე ეს სიგნალი მიეკუთვნება ტიპს SecurityEvent(უშიშროების სისტემის მოვლენა).

მოღვაწეობა **შევამოწმოთ საკრედიტო ბარათი** იწყება, მაშინ როდესაც მიიღება შემავალი პარამეტრიც CardDetails. შემდეგ იგი სთავაზობს მომხმარებელს შეიტანოთ PIN-კოდი.

მოქმედება AuthorizationReguestEvents იწყებს შესრულებას, როგორც კი მის შემავალ წახნაგებზე შემოდიან ობიექტები PIN და CardDetails(ბარათის ინფორმაცია). იყენებს რა ამ შემავალ პარამეტრებს, იგი ქმნის სიგნალს AuthorizationReguestEvents და აგზავნის მას.

სიგნალები იგზავნება ასინქრონულად და მართვის ნაკადი დაუყოვნებლივ გადადის მოვლენის მიღების მოქმედებაზე AuthorizationEvents. ეს მოქმედება ელოდება სიგნალის AuthorizationEvents მიღებას.

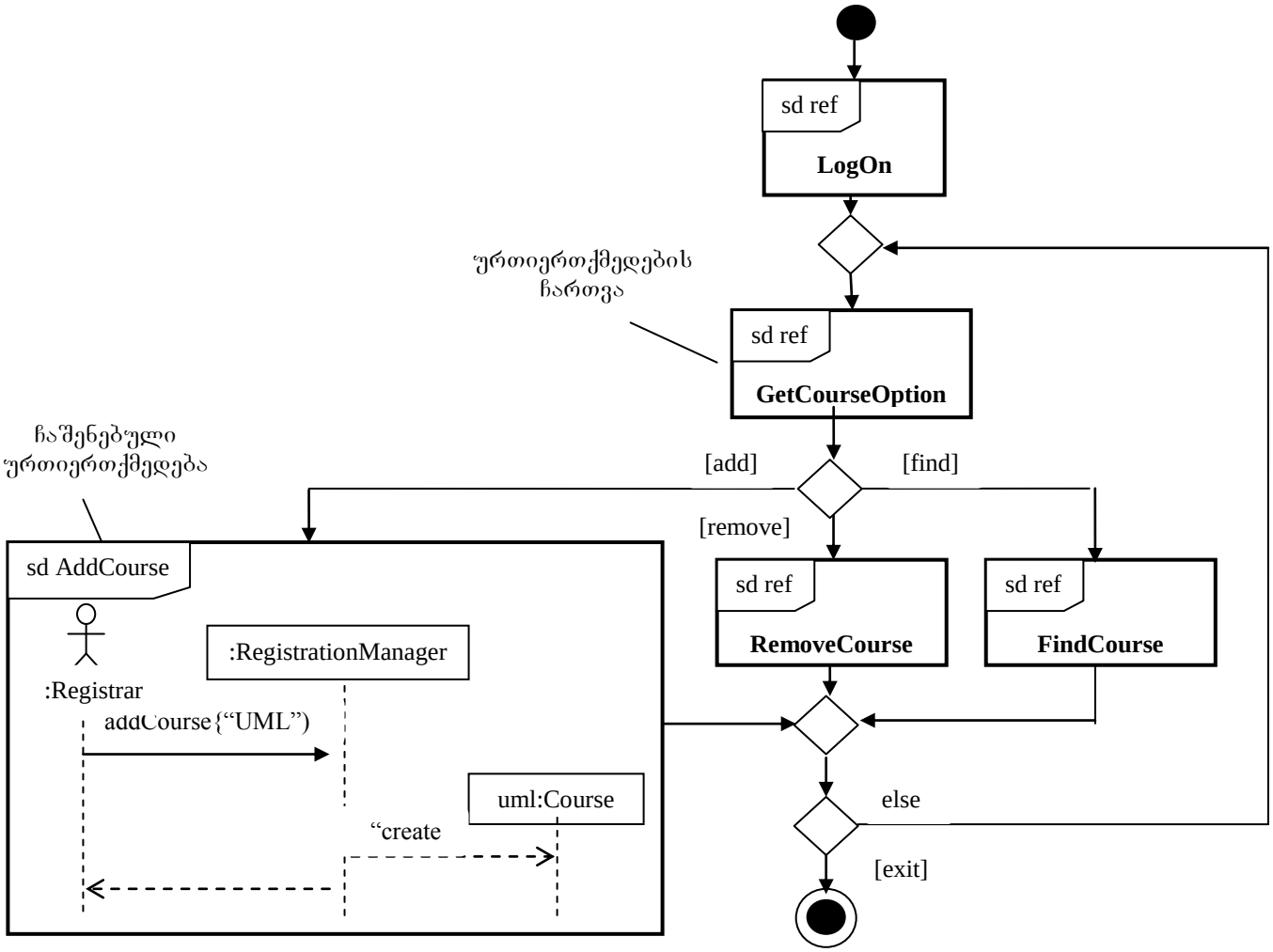


ნახ.3.22.

მიიღებს რა ამ სიგნალს, ნაკადი გადადის გადაწყვეტილებათა მიღების კვანძში თუ AuthorizationEvents. isAuthorized ჭეშმარიტია, გამოიძახება მოქმედება ავტორიზებულია, წინააღმდეგ შემთხვევაში მოქმედება არ არის ავტორიზებული.

ურთიერთქმედების მიმოხილვის დიაგრამა. ურთიერთქმედების მიმოხილვის დიაგრამა – მოღვაწეობის დიაგრამის განსაკუთრებული ფორმაა. იგი გვიჩვენებს ურთიერთქმედებებს და ურთიერთქმედების ჩართვას. იგი გამოიყენება მაღალი დონის მართვის ნაკადების მოდელირებისათვის ურთიერთქმედებებით მათ შორის. მისი გამოყენება განსაკუთრებით მნიშვნელოვანია პრეცედენტებს შორის მართვის ნაკადის აღწერისას.

ნახ.3.23.-ზე მოყვანილია ურთიერთქმედების მიმოხილვის დიაგრამა ManageCourses (კურსების ორგანიზება), რომელიც გვიჩვენებს ნაკადს დაბალდონიან ურთიერთქმედებებს შორის LogOn (შევიდეთ), GetCourseOption (მივიღოთ კურსების ვარიანტები), indCourse (ვიპოვოთ კურსი), Remove Course (მოვსპოთ კურსი) და AddCourse (დავამატოთ კურსი). ყოველი ამ ურთიერთქმედებიდან წარმოადგენს პრეცედენტს. მასასადამე, ურთიერთქმედების მიმოხილვის დიაგრამა აფიქსირებს მართვის ნაკადს პრეცედენტებს შორის.



ნახ.3.23.

ურთიერთქმედების მიმოხილვის დიაგრამის სინტაქსისი ანალოგიურია მოღვაწეობის დიაგრამის სინტაქსისის, იმის გამოკლებით, რომ აქ გამოისახება ჩაშენებული

ურთიერთქმედებები და ურთიერთქმედების ჩართვა, და არა მოღვაწეობები და ობიექტური კვანძები.

პრეცედენტების რეალიზებაში ურთიერთქმედებებით აღიწერება ქცევა, რომელიც განსაზღვრულია პრეცედენტით. შესაბამისად, ამისა ურთიერთქმედების მიმოხილვის დიაგრამა გამოიყენება ბიზნეს-პროცესების აღწერისათვის, წარმოდგენილნი პრეცედენტების სახით.

თავი 4
დაპროექტება

დაპროექტების სამუშაოთა უმრავლესობა ხორციელდება **დაზუსტების** ფაზიდან **აგების** ფაზაზე გადასვლის ზომის შესაბამისად.

დაპროექტება – მოდელირების ძირითადი მოღვაწეობაა **დაზუსტების** ფაზის ბოლო ნაწილისა და **აგების** ფაზის პირველი ნაწილისათვის. როგორც ნახ.2.-დან ჩანს, პირველი იტერაციის ძირითადი ყურადღება მიმართულია მოთხოვნების განსაზღვრაზე და ანალიზზე. ანალიზის დამთავრების შესაბამისად მოდელირების ფოკუსი გადაინაცვლებს დაპროექტებაზე. ანალიზი და დაპროექტება მნისვნელოვან წილად შესაძლებელია ჩატარდეს პარალელურადაც. მაგრამ, მნიშვნელოვანია მკაფიოდ განვასხვავოთ ამ ორი სამუშაო ნაკადის არტეფაქტები – ანალიტიკური მოდელი და საპროექტო მოდელი.

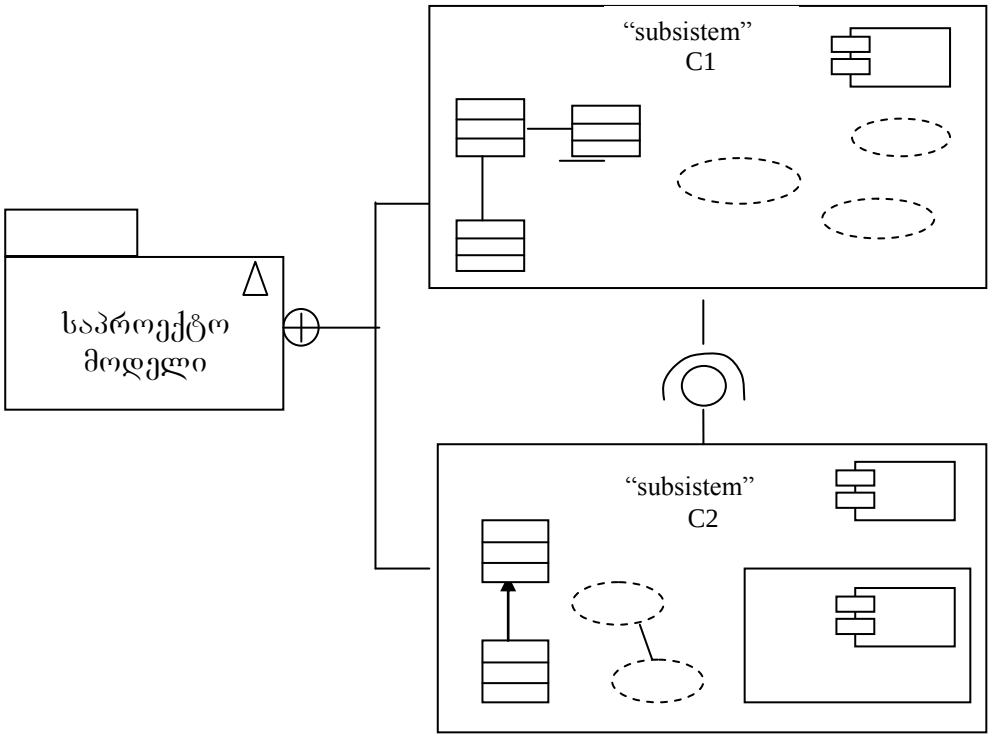
UP-ს რეკომენდაციით არ უნდა გაიყოს ანალიტიკოსებისა და დამპროექტებლების ფუნქციები. არტეფაქტების დამუშავებაზე (როგორიც არის პრეცედენტი) – მოთხოვნებიდან ანალიზითა და დაპროექტებით რეალიზაციამდე – პასუხს უნდა აგებდეს ერთი გუნდი. UP-ში მთავარია "მიზნები", და არა "ამოცანები".

ანალიზის ძირითადი მიზანია სისტემის ლოგიკური მოდელის აგება, რომელსაც უნდა წარმოადგენდეს სისტემა მომხმარებლის მოთხოვნების დაკმაყოფილების მიზნით. დაპროექტების მიზანია – დავადგინოთ სრული მოცულობით, როგორ იქნება რეალიზებული ეს ფუნქციონალობა. ამ ამოცანის გადაწყვეტის ერთ ერთი გზაა საპრობლემო სფეროსა და გადაწყვეტის სფეროს ერთდროული შესწავლა. მოთხოვნები დგინდება საპრობლემო სფეროდან, და ანალიზი შესაძლებელია განვიხილოთ როგორც გამოკვლევა სისტემის დამკვეთების თვალთახედვიდან. დაპროექტება გულისხმობს ტექნიკური გადაწყვეტების გაერთიანებას (კლასების ბიბლიოთეკები, ობიექტების შენახვის მექანიზმები და ა.შ.) სისტემის მოდელის შესაქმნელად (საპროექტო მოდელი), რომელიც შესაძლებელია რეალიზებულ იქნას სინამდვილეში.

4.1. დაპროექტების სამუშაო ნაკადი

დაპროექტებისას მიიღება სტრატეგიული გადაწყვეტილებები, როგორ შევინახოთ და გადავანაწილოთ ობიექტები, რომლის შესაბამისად იქმნება საპროექტო მოდელი.

ნახ. 4.1.-ზე წარმოდგენილია საპროექტო მოდელის მეტამოდელი. საპროექტო მოდელი შეიცავს მთელ რიგ საპროექტო ქვესისტემებს (აქ ნაჩვენებია მხოლოდ ორი ასეთი ქვესისტემა). ქვესისტემები – ეს კომპონენტებია, რომლებიც შესაძლებელია შეიცავდნენ მოდელის სხვადასხვა ტიპის ელემენტებს.



ნახ.4.1.

ანალიზისას ძირითადი ინტერფეისები შესაძლებელია დადგენილი იქნას, მიუხედავად ამისა დაპროექტებისას მათ ეთმობა უფრო მეტი ყურადღება. ეს აიხსნება იმით, რომ საპროექტო ქვესისტემების ინტერფეისები აერთიანებენ სისტემას როგორც ერთ მთლიანს და მათი მნიშვნელობა არქიტექტურის დაპროექტებისას დიდია. ამიტომ საკვანძო ინტერფეისების ძებნას და მოდელირებას ეთმობა ძალიან დიდი დრო. ნახაზზე ქვესისტემა C1 ესაჭიროება ინტერფეისი, რომელიც წარმოიდგინება ქვესისტემით C2. მოცემული და მოთხოვნილი ინტერფეისები აკავშირებენ ქვესისტემებს, როგორც

საპროექტო მოდელი შესაძლებელია განხილული იქნას როგორც ანალიტიკური მოდელის დაზუსტება, მასში ყველა არტეფაქტები დამუშავებულია უფრო საფუძვლიანად და უნდა შეიცავდნენ რეალიზაციის დეტალებს. მაგალითად, ანალიტიკური კლასი შეიძლება იყოს მხოლოდ ესკიზი მცირე ატრიბუტებით და მხოლოდ ძირითადი ოპერაციებით. მაგრამ საპროექტო კლასი უნდა იყოს მთლიანად ზუსტად განსაზღვრული – ყველა ოპერაციები და ატრიბუტები (დასაბრუნებელი ტიპებისა და პარამეტრების ცხრილიების ჩათვლით) უნდა იყვნენ სრულად აღწერილი.

საპროექტო მოდელები იქმნებიან:

- საპროექტო ქვესისტემებით;
- საპროექტო კლასებით;
- ინტერფეისებით;
- პრეცედენტების საპროექტო – რეალიზაციებით;

- განთავსების დიაგრამით.

4.2. კლასის დაპროექტება

საპროექტო კლასები – ეს კლასებია, რომელთა აღწერა იმდენად სრულია, რომ ისინი შესაძლებელია რეალიზებულ იქნან.

ანალიზისას კლასების წყაროს წარმოადგენს საგნობრივი სფერო. როგორც ვნახეთ, პრეცედენტები, მოთხოვნათა აღწერა, ტერმინოლოგია და ნებისმიერი ინფორმაცია საქმესთან დაკავსირებული შესაძლებელია გამოყენებულ იქნან როგორც წყარო ანალიზის კლასებისათვის.

საპროექტო კლასებს კი აქვთ ორი წყარო:

- საგნობრივი სფერო ანალიზის კლასების დაზუსტების მეშვეობით. ეს დაზუსტება მოიცავს რეალიზაციის დეტალების დამატებას. ამ პროცესის მიმდინარეობისას ხშირად ვლინდება, რომ მაღალ აბსტრაქტული ანალიზის კლასი საჭიროა დაიყოს ორ და მეტ დეტალიზირებულ საპროექტო კლასებათ.
- გადაწყვეტის სფერო – ეს უტილიტური კლასებისა და გამოყენებადი კომპონენტების ბიბლიოთეკებია. როგორიც არის Time, Date, String, კოლეკციები და ა.შ. აქ იმყოფება შუალედური პროგრამული უზრუნველყოფა (middleware), როგორიც არის კომმუნიკაციური პროგრამული უზრუნველყოფა(პუ), მონაცემთა ბაზები და კომპონენტური ინფრასტრუქტურები. მაგალითად .NET., CORBA ან Enterprise JavaBeans, ასევე GUI-ს აგების საშუალებები. ეს სფერო წარმოგვიდგენს სისტემის რეალიზების ტექნიკურ ინსტრუმენტალურ საშუალებებს.

ანალიზისას მოდელირდება, თუ რა უნდა აკეთოს სისტემამ. დაპროექტებისას კი მოდელირდება, თუ როგორ უნდა იყოს რეალიზებული ეს ქცევა.

ანალიზის კლასების მეშვეობით ხდება მცდელობა დააფიქსიროთ სისტემის სასურველი ქცევა მისი რეალიზების საშუალების განხილვის გარეშე. საპროექტო კლასებში კი აუცილებელია ზუსტად განვსაზღვროთ, თუ როგორ განახორციელებს კლასი თავის მოვალეობებს. ამისათვის უნდა გაკეთდეს შემდეგი:

- დავამთავროთ ატრიბუტების ნაკრები და სრულად აღვწეროთ იგი, დასახელების, ხედვისა და გამოყენებული დუმილით მნიშვნელობის ჩართვით;
- დავამთავროთ ოპერაციების ნაკრები და სრულად აღვწეროთ ისინი, დასახელების, პარამეტრების ცხრილისა და დასაბრუნებელი ტიპის ჩათვლით.

ოპერაციები ანალიზის კლასებში – ეს ფუნქციონალობის ნაწილის მაღალდონიანი ლოგიკური აღწერაა, რომელსაც თავაზობს კლასი. შესაბამის საპროექტო კლასებში ყოველი ოპერაცია ანალიზის კლასისა ზუსტდება და გარდაიქმნება ერთ ან მეტ დეტალიზირებულ და სრულად აღწერილ ოპერაციებათ, რომლებიც შესაძლებელია რეალიზებულ იქნან საწყის კოდში. შესაბამისად, ანალიზის კლასის ერთი მაღალდონიანი ოპერაცია შესაძლებელია დაიყოს ერთ ან მეტ საპროექტო ოპერაციებათ, რომლებიც შესაძლებელია რეალიზებულ იქნან. ამ დეტალიზირებულ ოპერაციებს დაპროექტების დონის უწოდებენ მეთოდებს.

საპროექტო მოდელი გადაეცემა პროგრამისტებს საწყისი კოდის ფაქტიური შექმნისათვის. კოდი ასევე შესაძლებელია გენერირებული იქნას უშუალოდ მოდელიდან, თუ გამოიყენება მოდელირების ინსტრუმენტი. ამისათვის კი საპროექტო კლასები უნდა იყვნენ საკმარისად დეტალურად აღწერილი. შესაბამისად, საპროექტო კლასი ყოველთვის უნდა ფასდებოდეს მისი მომხმარებლების თვალთახედვით. გამოყოფენ ოთხ ძირითად მახასიათებელს სწორად ფორმირებული საპროექტო კლასების დასახასიათებლად:

- სრული და საკმარისი;
- მარტივი;
- ფლობს მაღალ შინაგან შეკავშირებას;
- ფლობს დაბალ შეკავშირებას სხვა კლასებთან.

კლასის გახსნილი ოპერაციები განსაზღვრავენ კონტრაქტს კლასებსა და მის კლიენტებს შორის. სრული და საკმარისი კლასი სთავაზობს მომხმარებლებს ისეთ კონტრაქტს, როგორსაც ისინი ელოდებიან – არც მეტი და არც ნაკლები.

ოპერაციები უნდა პროექტირდებოდნენ ისე, რომ სთავაზობდნენ ერთადერთ, მარტივ, ელემენტარულ სერვისს. კლასი არ უნდა სთავაზობდეს ერთი და იგივეს მრავალი საშუალებით შესრულებას, ეს აბნევს კლიენტებს და ართულებს ტექნიკურ მომსახურებას. შესაბამისად, სერვისები უნდა იყვნენ მარტივი, ელემენტარული და უნიკალური.

ყოველი კლასი უნდა გამოსახავდეს ერთადერთ მკაფიოდ გამოსახულ აბსტრაქციას, გამოიყენებს რა შესაძლებლობების მინიმალურ ნაკრებს. შინაგანი შეკავშირება – ერთი ყველაზე სასურველი მახასიათებელია კლასის. შეკავშირებული კლასები ჩვეულებრივ უფრო ადვილი გასაგებია, ადვილია მათი გამოყენება და მომსახურება. შეკავშირებულ კლასს აქვს არც თუ ისე დიდი ურთიერთდაკავშირებული მოვალეობების სიმრავლე. ყოველი ოპერაცია, ატრიბუტი და ასსოციაციაკლასისა სპეციალურად პროექტირდება ამ მოვალეობების რეალიზებისათვის.

კონკრეტული კლასი ასოცირებული უნდა იყოს კლასების ისეთ რაოდენობასთან, რომლებიც საკმარისია იმისათვის, რომ მან შეძლოს თავისი მოვალეობების რეალიზება.

ურთიერთმიმართება უნდა დადგინდეს მხოლოდ იმ შემთხვევაში, თუ კლასებს შორის არსებობს რეალური სემანტიკური კავშირი – ეს უზრუნველყოფს დაბალ შეკავშირებას. კლასი ასოცირებული უნდა იყოს კლასების მინიმალურ რაოდენობასთან, რომლებიც საშუალებას მისცემენ მას მოახდინოს თავისი მოვალეობების რეალიზება.

პროექტირებისას მემკვიდრეობა თამაშობს ბევრათ უფრო მნიშვნელოვან როლს, ვიდრე ანალიზისას. ანალიზისას მემკვიდრეობა გამოიყენებოდა მხოლოდ იმ შემთხვევაში თუ ანალიზის კლასებს შორის არსებობდა მკაფიო და აშკარად გამოხატული მიმართება "წარმოადგენს". მაგრამ პროექტირებისას მემკვიდრეობა შესაძლებელია გამოყენებულ იქნას ტაქტიკური მიზნებისათვის კოდის ხელმეორედ გამოსაყენებლად. მემკვიდრეობა ფაქტიურად გამოიყენება შვილობილი კლასის რეალიზაციის გამარტივებისათვის, და არა ბიზნეს-მიმართების გამოსახვისათვის მშობელსა და შთამომავალს შორის.

მემკვიდრეობითობა – ძალიან ძლიერი მეთოდია. იგი წარმოადგენს ძირითად მექანიზმს პოლიმორფიზმის ფორმირებისათვის პროგრაირების მკაცრად ტიპიზირებულ ენებში, როგორიც არის Java, C# და C++.

მემკვიდრეობითობა ორ ან მეტ კლასს შორის შეკრულობის ყველაზე ძლიერი ფორმაა. კლასების იერარქიაში ინკაფსულაცია დაბალია. ბაზური კლასის ცვლილება გადაიცემა ქვევით იერარქიაში და მივყავართ ცვლილებებთან ქვეკლასებში. ყველა ობიექტ-ორიენტირებულ ენებში მემკვიდრეობითობის მიმართება შესრულებისას მუდმივია. შესრულებისას მიმართების შექმნით და მოსპობით, შესაძლებელია შევცვალოთ აგრეგაციისა და კომპოზიციის იერარქია, მაგრამ მემკვიდრეობითობის იერარქია რჩება უცვლელი.

ზოგჯერ იქმნება მემკვიდრეობის მოთხოვნა რამოდენიმე მშობლისაგან. ამავე დროს მშობლები არ უნდა იყვნენ სემანტიკურად დაკავშირებული. ასეთ მემკვიდრეობას უწოდებენ მრავლობითს. მაგრამ იგი ყველა ობიექტ-ორიენტირებული ენებისათვის არ არის მიღებული, მაგალითად Java და C# დაშვებულია მხოლოდ ერთჯერადი მემკვიდრეობა. პრაქტიკაში მრავალჯერადი მემკვიდრეობა არ წარმოადგენს პრობლემას, რამდენადაციგი ყოველთვის არის შესაძლებელი შეიცვალოს ერთჯერადით. მიუხედავად იმისა, რომ მრავალჯერადი მემკვიდრეობა გვთავაზობს დაპროექტების ამოცანის უფრო ელეგანტურ გადაწყვეტას, მისი გამოყენება შესაძლებელია გამოყენებულ იქნას მხოლოდ მაშინ როდესაც რეალიზების ენაში იგი დაშვებულია.

ბოლოს, ანალიტიკური ასოციაციები უნდა დაზუსტებული იყვნენ აგრეგაციის ერთ ერთ მიმართებაზე ყველგან სადაც ეს შესაძლებელია.

4.3. ინტერფეისები, ტიპები და როლები

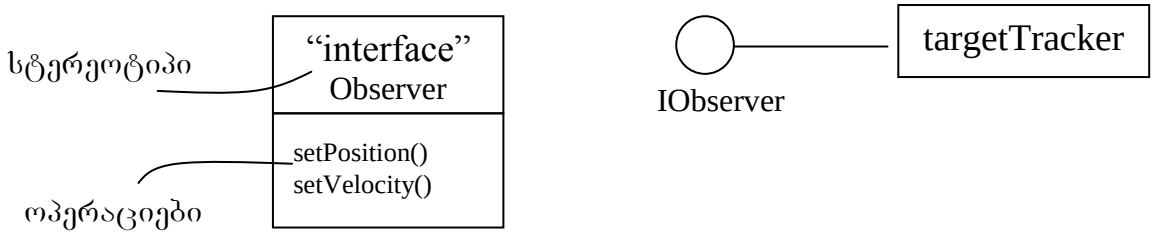
სისტემის დაპროექტებისას მნიშვნელოვანია ავაგოთ სისტემები ამოცანათა მკაფიო გამიჯვნით. ეს გულისხმობს, რომ სისტემის განვითარებისას ცვლილებას მის ერთ ნაწილში არ შეეხოს დანარჩენს. ამ მიზნის მისაღწევათ აუცილებელია სისტემის დამაკავშირებელი კვანძების სპეციფიცირება, რომლებსაც უკავშირდებიან დამოუკიდებლად ცვალებადი ნაწილები. დავადგენთ რა შემდეგში უკეთეს რეალიზაციას, თქვენ შეგეძლებათ შეცვალოთ ძველი ისე რომ არ შევაწუხოთ მომხმარებელი.

დამაკავშირებელი კვანძების მოდელირებისათვის გამოიყენება ინტერფეისები. **ინტერფეისები** – ეს ოპერაციების ერთობლიობაა, რომლებიც ახდენენ მომსახურების სპეციფიცირებას, რომელსაც წარმოადგენენ კლასები. ვაცხადებთ რა ინტერფეისს, ჩვენ გვეძლევა საშუალება დავადგინოთ აბსტრაქციის სასურველი ქცევა, მათი რეალიზაციისაგან დამოუკიდებლად. კლიენტებს შეუძლიათ დაეყრდნონ გამოცხადებულ ინტერფეისებს, ხოლო თქვენ შემდეგში შეგეძლებათ შექმნათ ან იყიდოთ მისი ნებისმიერი რეალიზაცია. მთავარია მხოლოდ მან შეასრულოს მოვალეობები, გამოცხადებული ინტერფეისით.

თითქმის ყველა თანამედროვე დაპროგრამების ენები, მათ შორის **JAVA** და **CORBA IDL**, იყენებენ ინტერფეისების კონცეფციას.

ამრიგად, ინტერფეისებს უწოდებენ ოპერაციების ნაკრებს, რომლებიც კლასების მიერ გამოიყენება მომსახურების სპეციფიცირებისათვის.

გრაფიკულად ინტერფეისი წარმოიდგინება წრის სახით. გარდა ამისა, იგი შეიძლება წარმოვადგინოთ როგორც სტერეოტიპული კლასი, რათა გაიხსნას ოპერაციები და სხვა თვისებები.

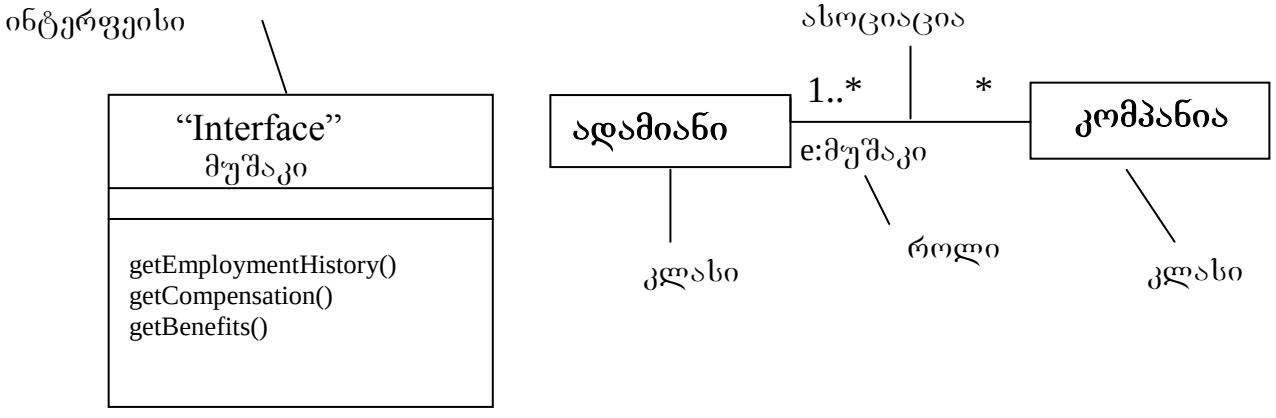


ყოველ ინტერფეისს უნდა გააჩნდეს სხვებისაგან განსხვავებული სახელი. ინტერფეისის სახელი წარმოიდგინება ტექსტური სტრიქონის სახით. იმისათვის, რომ განასხვავონ კლასისაგან ინტერფეისის სახელს დასაწყისში ემატება I.

ინტერფეისი ახდენს კლასის სპეციფიცირებას, მაგრამ არ ახდენს არავითარ შეზღუდვას მის რეალიზაციაზე. კლასი შეიძლება რამოდენიმე ინტერფეისების რეალიზებას ახდენდეს. ამასთან ისინი მოვალეობას იღებენ შეასრულონ თავიანთი ყველა

კონტრაქტები, უნდა შეიცავდნენ მეთოდებს ინტერფეისით გამოცხადებული ოპერაციების რეალიზებისათვის. ასევე კლასი შესაძლებელია დამოკიდებული იყოს რამოდენიმე ინტერფეისისაგან. ამ დროს ის ელოდება, რომ გამოცხადებული კონტრაქტები შესრულდებიან მათი რეალიზებადი გარკვეული კომპონენტების ნაკრებით. სწორედ ამიტომ ამბობენ, რომ ინტერფეისი წარმოადგენს სისტემაში დამაკავშირებელ კვანძს. იგი განსაზღვრავს კონტრაქტის პირობებს, რომლის შემდეგაც ორივე მხარე კლიენტი და მომწოდებელი შეუძლიათ იმოქმედონ ერთმანეთისაგან დამოუკიდებლად, მთლიანად დაეყრდნონ რა ურთიერთ მოვალეობებს.

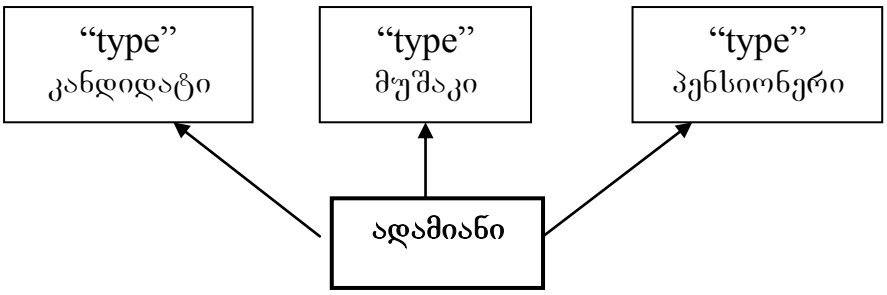
კლასი, რომელიც იყენებს ინტერფეისს, მასთან დაკავშირება ხდება დამოკიდებულების მიმართებით. შევქმნით რა ახალ ინტერფეისს, პირველ რიგში უყურებთ ოპერაციათა სიმრავლეს, რომელიც განსაზღვრავენ კლასის სერვისს. ინტერფეისი განსაზღვრავს კონტრაქტის პირობებს და კლასის ყველა ეგზემპლიარები უნდა იცავდნენ ამ პირობებს. ამასთან კონკრეტულ კონტექსში ეგზემპლიარს შეუძლია წარმოადგინოს მხოლოდ ის ინტერფეისები, რომლებიც მოცემულ სიტუაციას შეესაბამება. ეს ნიშნავს, რომ ყოველი ინტერფეისი განსაზღვრავს როლს, რომელსაც თამაშობს ობიექტი. როლი – ეს გარკვეული არსების ქცევაა კონკრეტულ კონტექსში ან სხვა სიტყვებით – პირი, რომლითაც აბსტრაქცია მიმართულია სამყაროს მიმართ. მაგალითად, განვიხილოთ კლასი *ადამიანის* ეგზემპლიარი. კონტექსისგან დამოკიდებულებით ამ კლასის ეგზემპლიარი შესაძლებელია თამაშობდეს მუშაკის, მყიდველის, მენეჯერის, მომღერლის და ა.შ. როლს. შესაბამისად, ობიექტი წარუდგენს სამყაროს ამა თუ იმ "სახეს" და მასთან ურთიერთქმედებაში მყოფი კლიენტები ელოდებიან მისგან შესაბამის ქცევას. მაგალითად, კლასი *ადამიანის* ეგზემპლიარი მენეჯერის როლში ფლობს სხვა თვისებებს, ვიდრე მას ექნება მომღერლის როლში.



როლი, რომელსაც ერთი აბსტრაქცია თამაშობს მეორეს მიმართ, შეიძლება აღვწეროთ შევავსებთ რა ასოციაციის შესაბამის ბოლო წერტილს ინტერფეისის სახელით. მაგალითისათვის ნახაზზე ნაჩვენებია ინტერფეისი *მუშაკი*, რომლის განსაზღვრა მოიცავს

სამ ოპერაციას. კლასებს შორის ადამიანი და კომპანია არსებობს ასოციაცია, რომლის კონტექსში ადამიანი თამაშობს e როლს, რომელიც ეკუთვნის ტიპს მუშაკი. სხვა ასოციაციაში ეს კლასი შესაძლებელია "სხვა სახით იყოს წარმოდგენილი". მოცემულ შემთხვევაში, კლასი ადამიანი თამაშობს კლასისათვის კომპანია მუშაკის როლს და მოცემულ კონტექსში კომპანიისათვის ხილვადია და არსებითი იქნება მხოლოდ თვისებები, რომლებიც განისაზღვრებიან მოცემული როლით.

.



ნახ.4.2.

აბსტრაქციების სემანტიკის ფორმალური მოდელირებისათვის და მათი გარკვეულ ინტერფეისთან შესაბამისობისათვის გამოიყენება სტერეოტიპი **type**. ეს არის კლასის სტერეოტიპი, რომლის მეშვეობით განისაზღვრება ობიექტების მოქმედების არე ოპერაციებთან ერთად. ტიპის კონცეფცია მჭიდროდ არის დაკავშირებული ინტერფეისის კონცეფციასთან, მხოლოდ იმ განსხვავებით, რომ ტიპის აღწერა შესაძლებელია შეიცავდეს ატრიბუტებს, ხოლო ინტერფეისის აღწერა – არა.

ნახ.4.2.-ზე მოყვანილია როლები, რომელსაც კლასი ***ადამიანი*** თამაშობს რესურსების მართვის სისტემის კონტექსში. მოცემული ნახაზიდან ჩანს, რომ კლასი ***ადამიანის*** ეგზემპლიარები შესაძლებელია მივაკუთვნოთ სამიდან ერთ ტიპს – კანდიდატი, მუშაკი და პენსიონერი, რომლებიც წარმოიდგინებიან სტერეოტიპული კლასების სახით.

4.4. პრეცედენტის რეალიზაცია დაპროექტების ეტაპზე

"პრეცედენტის საპროექტო რეალიზება" – ეს საპროექტო ობიექტებისა და საპროექტო კლასების ურთიერთქმედებაა, რომლითაც რეალიზდება პრეცედენტი. პრეცედენტის საპროექტო და ანალიტიკურ რეალიზაციას შორის დგინდება "trace" მიმართება. პრეცედენტის საპროექტო რეალიზაცია შედგება:

- ურთიერთქმედების საპროექტო დიაგრამებისაგან;
- კლასების დიაგრამისაგან, რომლებიც მოიცავენ მასში მონაწილე საპროექტო კლასებს.

ანალიზისას ძირითადი მნიშვნელობა პრეცედენტის რეალიზაციისას ეთმობა იმას, თუ რა უნდა აკეთოს სისტემამ. დაპროექტებისას ჩვენ გვაინტერესებს, თუ როგორ აპირებს

სისტემა ამის გაკეთებას. მაშასადამე, ეხლა ჩვენ გვაინტერესებს განვსაზღვროთ რეალიზაციის დეტალები, რომლებიც იგნორირებული იყო ანალიზის ეტაპზე. ამიტომ პრეცედენტის საპროექტო რეალიზაცია ბევრად უფრო დეტალიზებული და რთულია, ვიდრე საწყისი პრეცედენტის ანალიტიკური რეალიზება.

ურთიერთქმედების დიაგრამები – ძირითადი ნაწილია პრეცედენტის საპროექტო რეალიზებისა. დაპროექტებისას ისინი შესაძლებელია იყვნენ:

- ურთიერთქმედების ანალიტიკური მოდელების დაზუსტებას რეალიზების დეტალების დამატებით;
- აბსოლუტურად ახალი დიაგრამებით, რომლებიც შექმნილია ტექნიკური საკითხების ილუსტრირებისათვის, რომელიც აღიძვრება დაპროექტებისას.

იმისათვის, რომ გავიგოთ მიმდევრობის დიაგრამების როლი დაპროექტებისას, განვიხილოთ პრეცედენტი AddCourse.

მოკლე დახასიათება:

ამატებს სისტემაში ახალი კურსის დეტალებს.

მთავარი მოქმედი პირი:

Registrar

წინაპირობა:

Registrar შევიდა სისტემაში

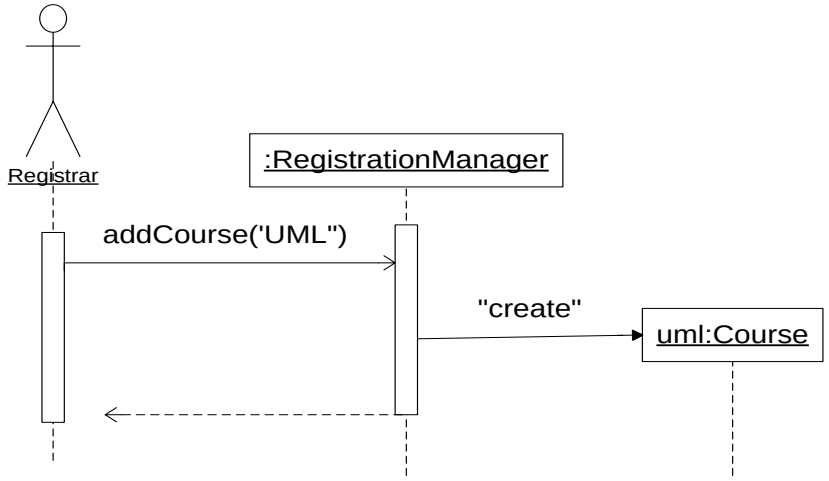
ძირითადი ნაკადი:

1. Registrar ირჩევს “add course”.

2 Registrar შეყავს ახალი კურსის დასახელება.

3. სისტემა ქმნის ახალ კურსს.

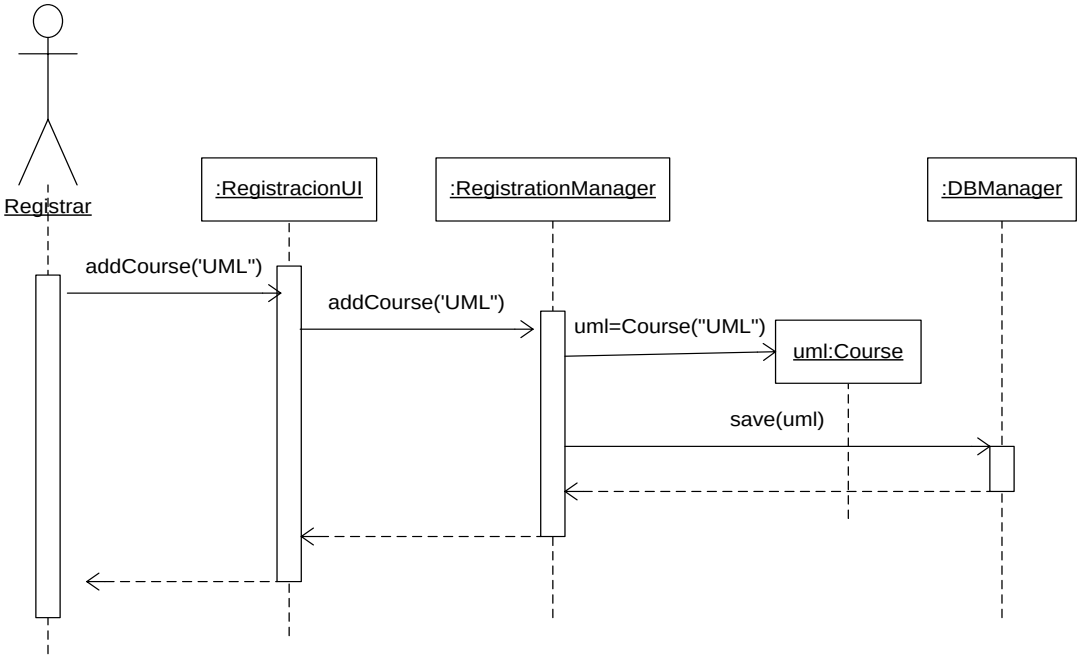
ნახ.4.3..-ზე მოყვანილია ურთიერთქმედების ანალიტიკური დიაგრამა, ხოლო ნახ.4.4.-ზე იგივე დიაგრამა დაპროექტების სხვადასხვა ეტაპებზე. მასში ანალიტიკური მოდელის აბსტრაქტული ოპერაციები გარდაქმნილია დაპროექტების დონის ოპერაციებში.



.

ნახ.4.3.

ეხლა დაწვრილებით არის ნაჩვენები ობიექტის შექმნა კონსტრუქტორის ოპერაციის გამოძახების საფუძველზე. გარდა ამისა, იგი მოიცავს ცენტრალურ მექანიზმს – ობიექტების Course შენახვის უზრუნველყოფა. მოცემულ შემთხვევაში შენახვის მექანიზმი :RegistrationManager იყენებს სერვისებს :DBManager Course ობიექტების შენახვისათვის მონაცემთა ბაზაში.



ნახ.4.4. მიმდევრობის დიაგრამა დაპროექტების სხვადასხვა ეტაპზე

4.5. პარალელიზმის მოდელირება

ერთ ერთი საკვანძო საკითხი, რომელიც განიხილება დაპროექტებისას ეს არის პარალელიზმი. პარალელიზმი ნიშნავს სისტემის ნაწილების პარალელურ შესრულებას. UML 2 უზრუნველყოფს პარალელიზმის საკმაოდ კარგ ასახვას:

- აქტიური კლასები;
- განშტოება და შერწყმა მოღვაწეობის დიაგრამაზე;
- ოპერატორი par მიმდევრობის დიაგრამაზე;
- დროითი დიაგრამების სხვადასხვა წარმოდგენები;
- რიგითი ნომრები პრეფიქსების სახით კომუნიკაციის დიაგრამაზე;
- ორთოგონალური შემადგენელი მდგომარეობები მდგომარეობის დიაგრამებზე.

4.5.1. აქტიური კლასები

პარალელიზმის მოდელირების ძირითადი პრინციპია – ყოველი მართვის ნაკადი ან პარალელური პროცესი მოდელირდება როგორც **აქტიური კლასი**, რომლის ქვეშ გაიგება ობიექტი, რომელშიც ინკაფსულირებულია საკუთარი მართვის ნაკადი. აქტიური ობიექტები წარმოადგენენ აქტიური კლასების ეგზემპლიარებს. აქტიური ობიექტები და აქტიური კლასები დიაგრამებზე გამოისახებიან როგორც ჩვეულებრივი კლასები და ობიექტები, მაგრამ ორმაგი ზოლით მარჯვნიდან და მარცხნიდან.

პრეცედენტი **DeactivateSystem:**

მოკლე აღწერა:
ახდენს სისტემის დეაქტივაციას.
მთავარი აქტიორი:
SecurityGuard
წინაპირობა:
SecurityGuard-ს აქვს აქტივაციის გასაღები.
ძირითადი ნაკადი;
3. SecurityGuard იყენებს აქტივაციის გასაღებს სისტემის გამოსართავად
4. სისტემა წყვეტს უშიშროების მიმწოდების ლოკე ენ
შედეგი:
უშიშროების სისტემა დეაქტივიზირებულია, არ ახდენს სისტემის კონტროლს.

პრეცედენტი **ActivateAll:**

მოკლე აღწერა:
ახდენს სისტემის აქტივაციას.
მთავარი აქტიორი:
SecurityGuard
წინაპირობა:
SecurityGuard-ს აქვს აქტივაციის გასაღები.
ძირითადი ნაკადი;
1. SecurityGuard იყენებს აქტივაციის გასაღებს სისტემის ჩასართავად
2. სისტემა იწყებს უშიშროების მიმწოდების კონტროლს
3. სისტემა გამოსცემს ხმოვან სიგნალს, მზადყოფნის დემონსტრირებისათვის
შედეგი:
უშიშროების სისტემა აქტივიზირებულია, ახდენს სისტემის კონტროლს.

პრეცედენტი **TriggerSensor**:

მოკლე აღწერა:
ამოქმედდა მიმწოდი..
მთავარი აქტიორი:
Fire
intruder
წინაპირობა:
უშიშროების სისტემა აქტივიზირებულია.
ძირითადი ნაკადი;
1. თუ აქტიორი Fire ახდენს სახანძრო მიმწოდის ინიციირებას
1.1. გამოიცემა სახანძრო საფრთხის სიგნალი
2. თუ აქტიორი Intruder ახდენს უშიშროების მიმწოდის ინიციირებას
2.1. გამოიცემა საფრთხის სიგნალი
შედეგი:
ისმის სირენის სიგნალი.

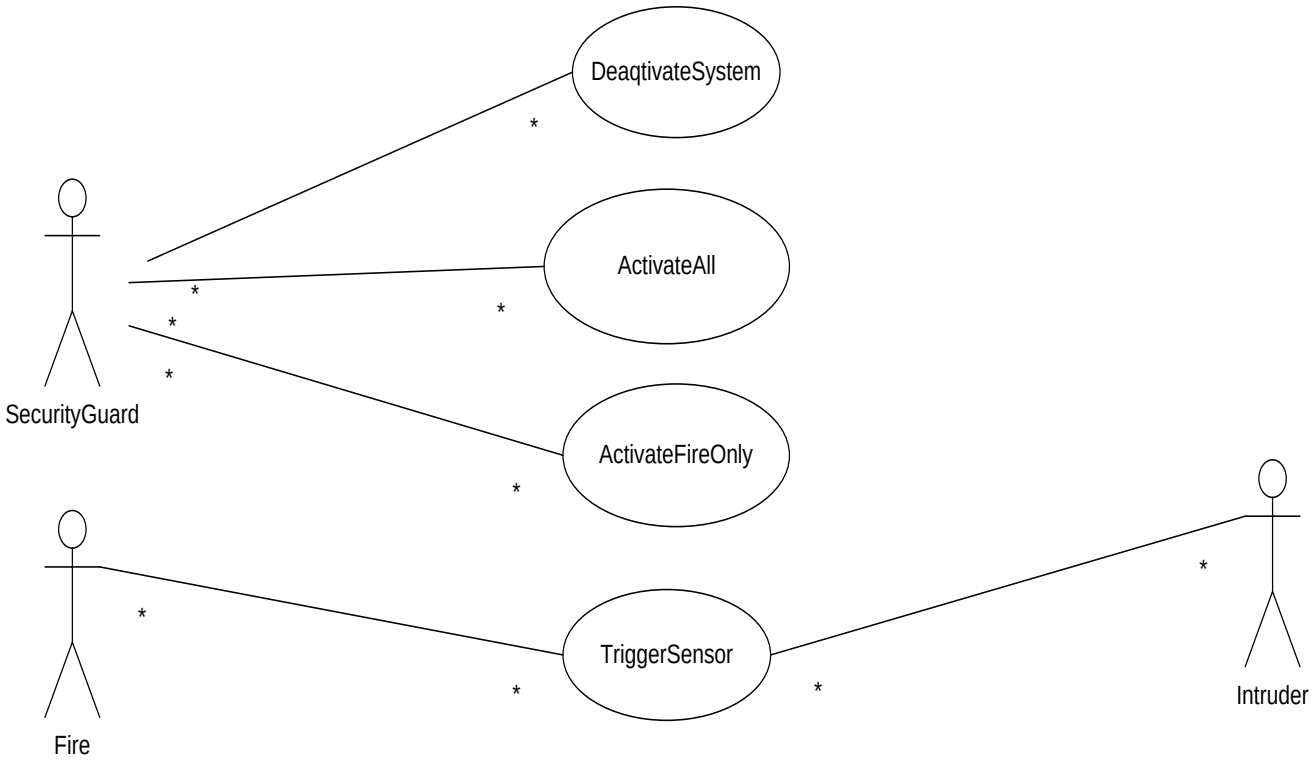
ნახ.4.5.

აქტიური კლასები ფლობენ იმავე თვისებებს, რასაც სხვა კლასები. მათ შესაძლებელია გააჩნდეთ ეგზემპლიარები, ატრიბუტები და ოპერაციები, ასევე მონაწილეობა მიიღონ დამოკიდებულების, განზოგადების და ასოციაციის მიმართებებში. ისინი შესაძლებელია რეალიზებულ იქნან კოოპერაციებით, ხოლო მათი ქცევა

სპეციფიცირებულ იქნას ავტომატების მეშვეობით. დიაგრამებზე აქტიური ობიექტები გვხვდება ყველგან სადაც გვხვდება პასიურები.

პარალელიზმს აქვს ჩვეულებრივ დიდი მნიშვნელობა ჩართული სისტემებისათვის, როგორიც არის პროგრამული უზრუნველყოფა, რომელიც მართავს ბანკომატს. პარალელიზმის შესწავლისათვის განვიხილოთ მარტივი ჩართული სისტემა – უსაფრთხოების სისტემა. იგი აკონტროლებს რიგ მიმწოდებს ხანძრის დადგენისათვის ან შენობაში უცხო პირის შესვლას. მიმწოდის ამუშავებისას სისტემა რთავს სიგნალიზაციას. პრეცედენტების მოდელი უშიშროების სისტემისათვის მოყვანილია ნახ.4.6.-ზე.

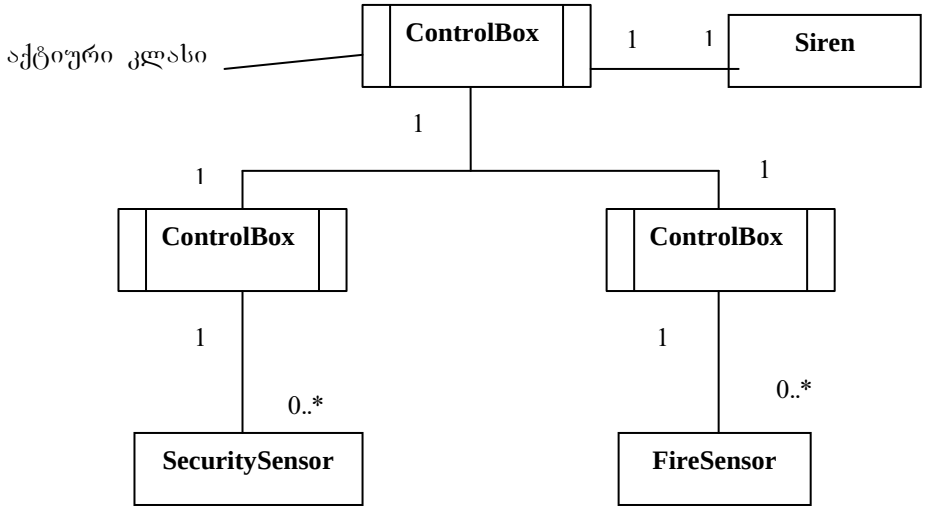
პრეცედენტი ActivateFireOnly(მოვახდინოთ მხოლოდ სახანძრო მიმწოდების აქტივირება) არ განიხილება, რადგან ძირითადი ყურადღება აქ დათმობილი აქვს სისტემის პარალელიზმს.



ნახ.4.6.

ამის შემდეგ უნდა დავადგინოთ კლასები. ჩართული სისტემებისათვის კლასების წყარო შეიძლება იყვნენ აპარატული საშუალებები, რომელზეც სისტემა სრულდება. განხილულ შემთხვევაში სიგნალიზაცია შედგება ოთხი კომპონენტისაგან: მართვის ბლოკი, სირენა, სახანძრო მიმწოდების ნაკრები და დაცვის მიმწოდების ნაკრებისაგან.

პრეცედენტების და ფიზიკური მოწყობილობების შესახებ ინფორმაციის საფუძველზე საშუალება გვეძლევა მივიღოთ კლასების დიაგრამა მოცემული სისტემისათვის, რომელიც წარმოდგენილია ნახ.4.7.-ზე.



ნახ.4.7.

უშიშროების(დაცვის) სისტემა სისტემატურად თვალყურს უნდა ადევნებდეს სახანძრო და უშიშროების(დაცვის) მიმწოდებს, ამიტომ უნდა გამოვიყენოთ მრავალნაკადიანობა (multithreading). კლასები ControlBox (მართვის ბლოკი), SecuritySensorMonitor (დაცვის მიმწოდების მონიტორი) და FireSensorMonitor (სახანძრო მიმწოდების მონიტორი) ნაჩვენებია ორმაგი ჩარჩოებით მარჯვნიდან და მარცხნიდან. ეს ნიშნავს, რომ ისინი წარმოადგენენ აქტიურ კლასებს.

აქტიური კლასი წარმოადგენს დამოუკიდებელ მართვის ნაკადს, მაშინ როდესაც ჩვეულებრივი კლასი არ არის მასთან კავშირში. აქტიურებისგან განსხვავებით, ჩვეულებრივ კლასებს უწოდებენ პასიურებს, რადგან მათ არ აქვთ საშუალება მოახდინონ დამოუკიდებელი მართვის ნაკადის ინიციირება.

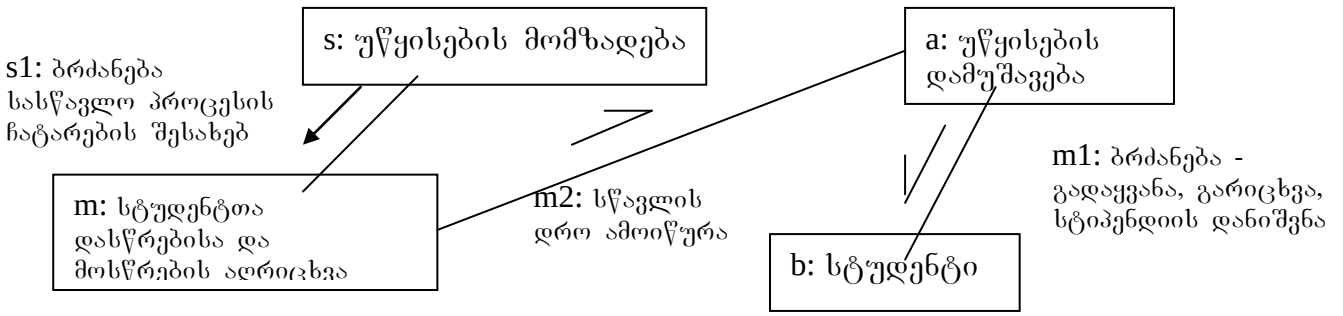
4.5.2 მართვის რამოდენიმე ნაკადის მოდელირება

სისტემის აგება რამოდენიმე მართვის ნაკადით – ადვილი ამოცანა არ არის. უნდა გადაწყდეს თუ როგორ გადავანაწილოთ სამუშაო პარალელურ აქტიურ ელემენტებს შორის, ასევე უნდა დადგინდეს კომუნიკაციისა და სინქრონიზაციის სწორი მექანიზმები სისტემის აქტიურ და პასიურ ობიექტებს შორის, რომელიც უზრუნველყოფს მათი ქცევის სისწორეს მართვის რამოდენიმე ნაკადის დროს.

კომუნიკაცია. ერთმანეთთან კოოპერირებადი ობიექტები ურთიერთქმედებენ შეტყობინებების გაცვლით. სისტემებში სადაც გვაქვს აქტიური და პასიური ობიექტები, მიზანშეწონილია განვიხილოთ ოთხი შესაძლო კომბინაცია.

პირველი, შეტყობინება შესაძლებელია გადაიცეს ერთი პასიური ობიექტიდან მეორეზე. იმის გათვალისწინებით, რომ დროის ნებისმიერ მომენტში არსებობს მხოლოდ ერთი მართვის ნაკადი, რომელიც გადის ორივე ობიექტზე, ასეთი ურთიერთქმედება შეესაბამება უბრალოდ ოპერაციის გამოძახებას.

მეორე, შეტყობინება შესაძლებელია გადაიცეს ერთი აქტიური ობიექტიდან მეორეზე. აქ ჩვენ საქმე გვაქვს პროცესებს შორის კომუნიკაციასთან, რომელიც შესაძლებელია განხორციელდეს ორი საშუალებით. პირველ ვარიანტში რომელიმე აქტიურ ობიექტს შეუძლია სინქრონულად გამოიძახოს მეორეს ოპერაცია. ასეთ საშუალებას აქვს სემანტიკა რანდევუ **(Rendezvous)**: გამომძახებელი ობიექტი მოითხოვს ოპერაციის შესრულებას და ელოდება, სანამ მიმღები მხარე მიიღებს გამოძახებას, შეასრულებს ოპერაციას და დააბრუნებს გარკვეულ ობიექტს (თუ ასეთი არის); შემდეგ ორივე ობიექტი აგრძელებენ მუშაობას ერთმანეთისაგან დამოუკიდებლად. გამოძახების შესრულების მთელი დროის განმავლობაში ორივე მართვის ნაკადი ბლოკირებულია. მეორე ვარიანტში ერთ აქტიურ ობიექტს შეუძლია ასინქრონულად გაუგზავნოს სიგნალი მეორეს ან გამოიძახოს მისი ოპერაცია. ასეთი საშუალების სემანტიკა გვაგონებს საფოსტო ყუთს **(Mailbox)**; გამომძახებელი მხარე აგზავნის სიგნალს ან იძახებს ოპერაციას, რის შემდეგ აგრძელებს შესრულებას. ამ დროს მიმღები მხარე იღებს სიგნალს ან გამოძახებას, როგორც კი იქნება ამისათვის მზად. სანამ ის ამუშავებს მოთხოვნას, ყველა ახლად შემოსული მოვლენები ან გამოძახებები დგებიან რიგში. მოახდენს რა რეაგირებას მოთხოვნაზე, მიმღები ობიექტი განაგრძობს თავის მუშაობას. საფოსტო ყუთის სემანტიკა მჟღავნდება იმაში, რომ ორივე ობიექტი არა სინქრონიზებულია, უბრალოდ ერთი უტოვებს შეტყობინებას მეორეს. **UML**-ში სინქრონული შეტყობინება გამოისახება მთლიანი ისრით, ხოლო ასინქრონული "ნახევარისრით" (იხ. ნახ.3.6.1).



ნახ.4.8.

მესამე, შეტყობინება შესაძლებელია გადაიცეს აქტიური ობიექტიდან პასიურზე. სიძნელე წარმოიშვება იმ შემთხვევაში, როდესაც ერთდროულად რამოდენიმე აქტიური ობიექტი გადასცემენ თავის მართვის ნაკადს ერთ და იმავე პასიურს. ასეთ შემთხვევაში

საჭიროა ნაკადების სინქრონიზაციის ძალიან აკურატული მოდელირება, რომელსაც განვიხილავთ ქვევით.

მეოთხე, პასიურ ობიექტს შეუძლია გადასცეს შეტყობინება აქტიურს. თუ გავითვალისწინებთ იმას, რომ ყოველი მართვის ნაკადი ეკუთვნის რომელიმე აქტიურ ობიექტს, მაშინ ხდება ნათელი, რომ პასიური ობიექტის მიერ შეტყობინების გადაცემა აქტიურზე აქვს იგივე სემანტიკა, რაც შეტყობინებების გაცვლა ორ აქტიურ ობიექტს შორის.

სინქრონიზაცია. პარალელურ სისტემაში გვაქვს რამოდენიმე მართვის ნაკადი. როდესაც ნაკადი გადის რაიმე ოპერაციაზე, ჩვენ ვამბობთ, რომ ეს ოპერაცია არის შესრულების წერტილი. თუ ოპერაცია განსაზღვრულია რომელიმე კლასში, შეიძლება ითქვას, რომ შესრულების წერტილს წარმოადგენს ამ კლასის კონკრეტული ეგზემპლიარი. ერთ ოპერაციაში (შესაბამისად ერთ ობიექტში) შესაძლებელია ერთდროულად იმყოფებოდნენ რამოდენიმე მართვის ნაკადები, ასევე, ხდება ისე, რომ სხვადასხვა ნაკადები იმყოფებოდნენ სხვადასხვა ოპერაციებში, მაგრამ ერთ ობიექტში.

პრობლემა წარმოიშვება მაშინ, როდესაც ერთ ობიექტში იმყოფება ერთდროულად რამოდენიმე მართვის ნაკადი. თუ არ გამოვიჩენთ სიფრთხილეს, ნაკადებმა შეიძლება ხელი შეუშალონ ერთმანეთს, რაც მიგვიყვანს ობიექტის მდგომარეობის არაკორექტულ შეცვლამდე. ეს არის კლასიკური პრობლემა ურთიერთგამორიცხვისა. შეცდომები ასეთი სიტუაციების დამუშავებისას შეიძლება გახდეს სხვადასხვა სახის კონკურენციების მიზეზი ნაკადებს შორის.

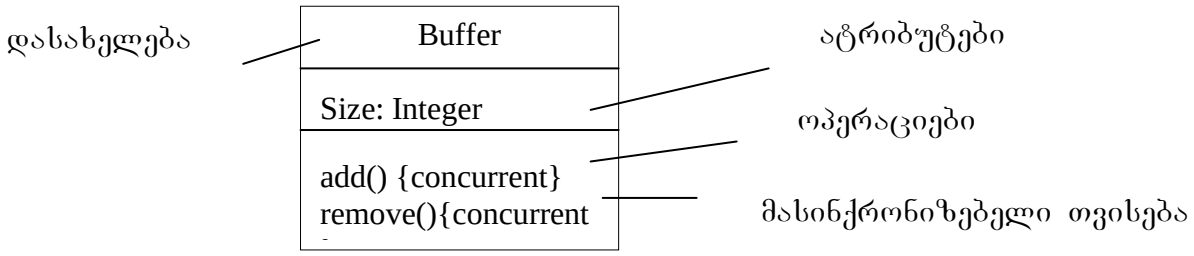
ამ პრობლემის გადასაწყვეტად ობიექტ-ორიენტირებულ სისტემებში ოპერაციებზე, რომლებიც განსაზღვრულია კლასში, ენიჭება გარკვეული მასინქრონიზებელი თვისებები.

Sequential(მიმდევრობითი) – გამომძახებელი მხარე თავისი მოქმედების შესახებ კოორდინირებას უნდა ახდენდეს გამოსაძახებელ ობიექტში შესვლამდე, ისე რომ დროის ნებისმიერ მომენტში ობიექტის შიგნით იმყოფება ერთი მართვის ნაკადი. რამოდენიმე მართვის ნაკადის არსებობისას ობიექტის სემანტიკისა და მთლიანობის გარანტია არ არის.

quarded(დაცული) - მართვის რამოდენიმე ნაკადისას ობიექტის სემანტიკა და მთლიანობა გარანტირებულია ობიექტის ყველა დაცულ ოპერაციაზე გამოძახებათა მოწესრიგების გზით. დროის ყოველ მომენტში ობიექტზე შესაძლებელია შესრულდეს მხოლოდ ერთი ოპერაცია.

Concurent(პარალელური) – მართვის რამოდენიმე ნაკადისას ობიექტის სემანტიკა და მთლიანობა გარანტირებულია იმით, რომ ოპერაცია განიხილება როგორც ატომური.

დაპროგრამების ზოგიერთ ენაში გათვალისწინებულია მოყვანილი კონსტრუქციები. Java ენაში არის თვისება synchronized, ექვივალენტური UML-ის თვისებისა concurrent. ქვევით ნახაზზე ნაჩვენებია თუ როგორ უკავშირდებიან ეს თვისებები ოპერაციებს.



აქტიური ობიექტები თამაშობენ მნიშვნელოვან როლს სისტემის წარმოდგენისას პროცესების თვალსაზრისით. ასეთი წარმოდგენა მოიცავს პროცესებსა და ძაფებს, რომლებიც ქმნიან სისტემურ პარალელიზმსა და სინქრონიზაციას. ეს კი საშუალებას გვაძლევს გამოვსახოთ ასეთი წარმოდგენის სტატიკური და დინამიკური ასპექტები იმავე დიაგრამებით, რომლებიც გამოიყენება წარმოდგენისას პროექტირების თვალსაზრისით ე.ი. კლასების, ურთიერთქმედების, მოღვაწეობის და მდგომარეობის დიაგრამებით, იმ განსხვავებით რომ ძირითადი ყურადღება მათზე ეთმობა აქტიურ კლასებს, რომლებიც წარმოადგენენ პროცესებს და ძაფებს.

მართვის რამოდენიმე ნაკადის მოდელირება ხდება შემდეგნაირად:

1. დავადგინოთ მოქმედებათა პარალელიზმის შესაძლებლობა და მოვახდინოთ მართვის ნაკადის მატერიალიზაცია აქტიური კლასის სახით. დავაჯგუფოთ აქტიური ობიექტების საერთო სიმრავლე აქტიურ კლასში.
2. განვიხილოთ მოვალეობების განაწილების ბალანსი აქტიურ კლასებს შორის, ხოლო შემდეგ განვიხილოთ, რომელ სხვა აქტიურ და პასიურ კლასებთან კოოპერირდება სტატიურად ყოველი მათგანი.
3. გამოხატეთ სტატიკური გადაწყვეტილებები კლასების დიაგრამის სახით, ნათლად გამოვყოთ აქტიური კლასები.
4. განიხილეთ, თუ როგორ კოოპერირდება დინამიურად თითოეული კლასი სხვა კლასებთან. გამოვხატოთ ეს გადაწყვეტილებები ურთიერთქმედების დიაგრამაზე. ნათლად მიუთითეთ აქტიური ობიექტები როგორც საწყისი წერტილები შესაბამისი მართვის ნაკადისა.
5. განსაკუთრებული ყურადღება მიაქციეთ აქტიურ ობიექტებს შორის კომუნიკაციებს. გამოიყენეთ საჭიროებისამებრ როგორც სინქრონული ასევე ასინქრონული შეტყობინებები.

6. ყურადღება მიაქციეთ აქტიური ობიექტების სინქრონიზაციას და იმ პასიურ ობიექტებს, რომლებთანაც ისინი კოოპერირდებიან. გამოიყენეთ ყველაზე მისადაგებული სემანტიკა – მიმდევრობითი, დაცული და პარალელური.

4.5.3. პარალელიზმი მიმდევრობის დიაგრამებზე

მიმდევრობის დიაგრამა პრეცედენტისათვის ActivateAll ნაჩვენებია ნახ.4.9.-ზე. იგი გვიჩვენებს ოპერატორების par, Loop და critical გამოყენებას.

ქვევით მოყვანილია დიაგრამის ანალიზი:

1. ობიექტი :SecurityGuard აგზავნის შეტყობინებას activate() ობიექტს :ControlBox.
2. :ControlBox აგზავნის შტყობინებას soundActivatedalarm() ობიექტს :Siren.
3. :ControlBox ბადებს ორმართვის ნაკადს, წარმოდგენილნი par ოპერატორის ოპერანდებით, ჯერ გამოიძახება 1 ოპერანდი, ხოლო შემდეგ 2 ოპერანდი.
4. par ოპერატორის 1 ოპერანდი:

    4.1.:ControlBox.უგზავნის შეტყობინებას monitor() ობიექტს :FireSensorMonitor.

    4.2. :FireSensorMonitor შედის ციკლში :FireSensor ობიექტის გამოკითხვისათვის.ციკლი პირველი შესრულებისას დგინდება fire ცვლადის საწყისი მნიშვნელობა, შემდეგ ციკლი გრძელდება მანმადე, სანამ fire აქვს მნიშვნელობა false.

    4.3. როდესაც fire ღებულობს მნიშვნელობას true:

    4.3.1. :FireSensorMonitor შედის განყოფილებაში critical, სადაც:

    4.3.1.1. იგი აგზავნის შეტყობინებას fire().:ControlBox-ში.

    4.3.1.2. .:ControlBox აგზავნის შეტყობინებას soundFireAlarm() ობიექტს :Siren.

5. ოპერანდი 1 ოპერატორისა par სრულდება.
6. par ოპერატორის 2 ოპერანდი:

    6.1. :ControlBox.უგზავნის შეტყობინებას monitor() ობიექტს :SecuritySensorMonitor.

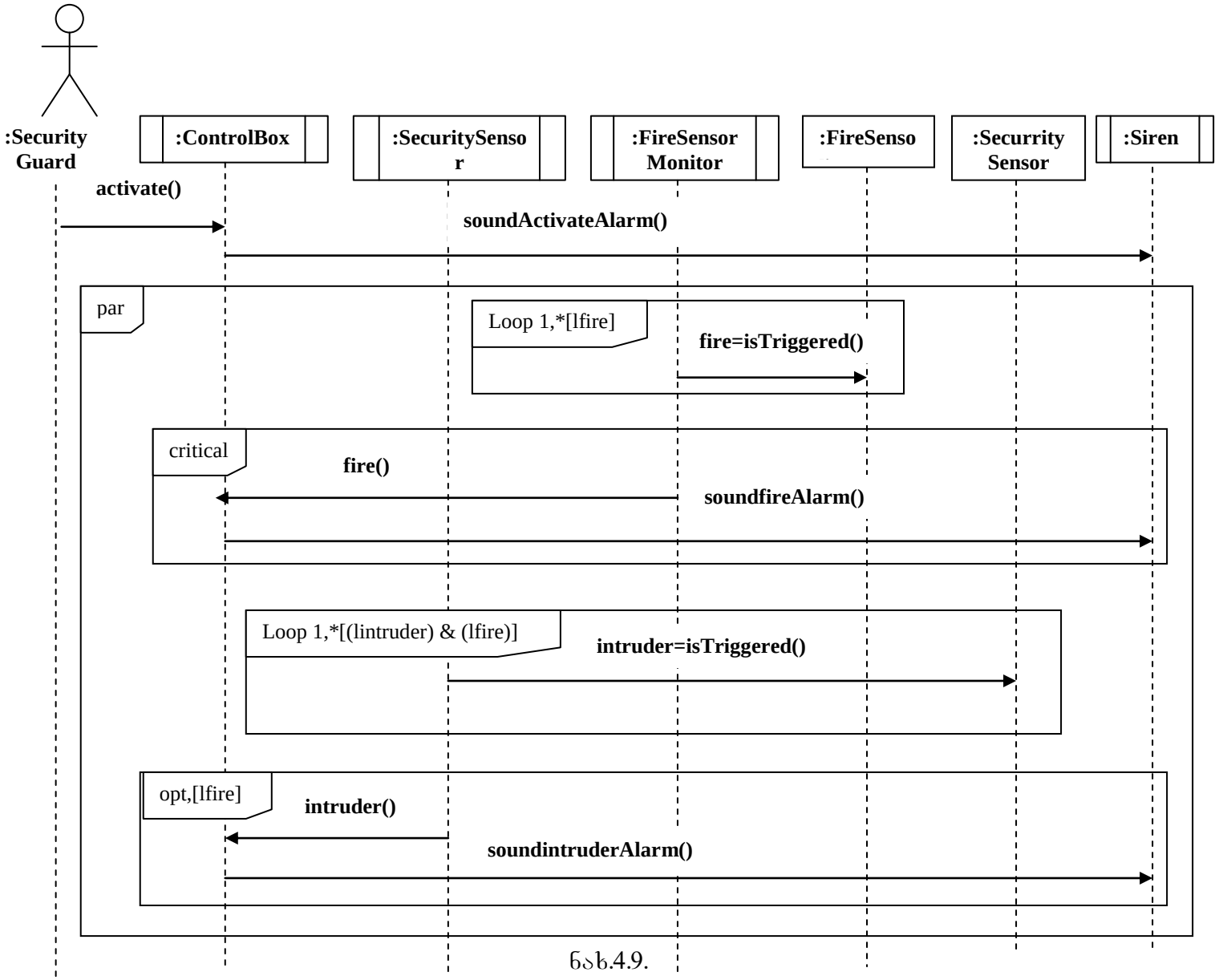
    6.2. :SecuritySensorMonitor შედის ციკლში :SecuritySensor ობიექტის გამოკითხვისათვის. ციკლი პირველი შესრულებისას დგინდება ცვლადის საწყისი მნიშვნელობა intruder, შემდეგ ციკლი გრძელდება მანამდე, სანამ intruder და (ან) fire აქვთ მნიშვნელობა false.

    6.3. როდესაც intruder ღებულობს მნიშვნელობა true:

    6.3.1. როდესაც fire აქვს მნიშვნელობა false

    6.3.1.1. :SecuritySensorMonitor უგზავნის შეტყობინებას intruder() ობიექტს .:ControlBox

    6.3.1.2. .:ControlBox აგზავნის შეტყობინებას soundIntruderAlarm() ობიექტს :Siren



ნახ.4.9.

7. ოპერანდი 2 ოპერატორისა par სრულდება

8. ურთიერთქმედება სრულდება.

ამ ურთიერთქმედებაში უნდა აღინიშნოს რამოდენიმე საინტერესო მომენტი:

- par ოპერატორის ორივე ოპერანდი სრულდება პარალელურად.
- ნაწილი critical წარმოადგენს ელემენტარულ ქცევას, რომელიც არ შეიძლება შეწყვეტილ იქნას.
- ორივე ციკლს აქვთ სემანტიკა Repeat...Until (გავიმეოროთ...სანამ) – ისინი სრულდებიან ერთხელ, იმისათვის რომ მივანიჭოთ მნიშვნელობა ცვლადს, რომელიც გამოიყენება მათ პირობაში, შემდეგ კი მეორდება მანამდე, სანამ მათი პირობები რჩებიან ჭეშმარიტი.

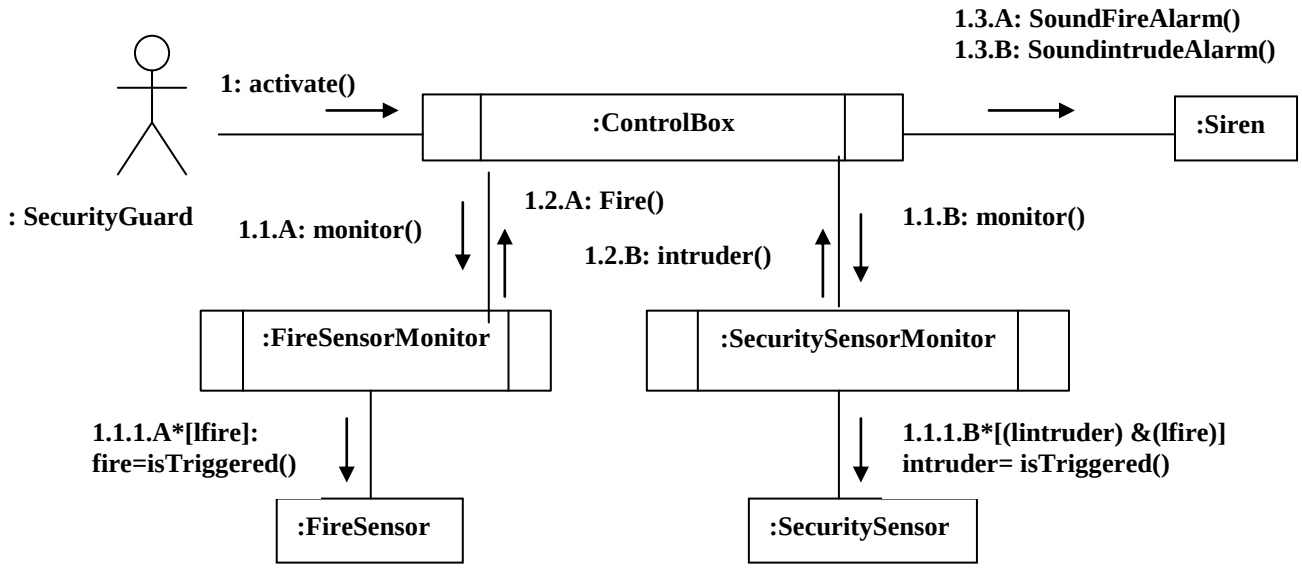
- სახანძრო განგაში ყოველთვის უფრო პრიორიტეტულია, ამიტომ par ოპერატორის ოპერანდში 2 Loop შეწყდება Fire() ან intrude() მოვლენით.

ძირითადი ყურადღება პრეცედენტის რეალიზებისას უნდა მიმართული იყოს კლასებს შორის შესაძლო ურთიერთქმედებების აღწერას პრეცედენტით მოცემული ქცევის რეალიზაციისას.

4.5.4. პარალელიზმი კომუნიკაციის დიაგრამებზე

პარალელიზმი კომუნიკაციის დიაგრამებზე გამოისახება მართვის ნაკადების ნიშნულებით, რომლებიც მიეთითება ოპერაციის რიგითი ნომრის შემდეგ, როგორც ეს ნაჩვენებია ნახ.4.10.-ზე. აქ გვაქვს ორი პარალელური ნაკადი A და B.

მოცემულ მაგალითში იგულისხმება, რომ არის ერთი ეგზემპლიარი FireSensor და ერთი SecuritySensor და იტერაცია წარმოადგენს მიმდევრობითი გამოკითხვის ოპერაციას, რომელიც მუდმივად იძახებს ოპერაციას isTriggered() მანამდე, სანამ მიმწოდი აბრუნებს მნიშვნელობას true.



ნახ.4.10.

4.6. დროითი დიაგრამები

UML 1-ს ერთ-ერთი სუსტი ადგილი იყო რეალური დროის სისტემების მოდელირება. ეს ისეთი სისტემებია, რომლებშიც დროითი თანაფარდობები კრიტიკულად არის მნიშვნელოვანი და მოვლენები უნდა თანსდევდნენ ერთმანეთს გარკვეული დროითი ფანჯრის ფარგლებში. ამბობენ "დროით ფანჯარა" და არა "დრო" რადგან აბსოლუტური დრო დამმუშავებლებისათვის მიუღებელია. როდესაც მოდელში მოცემულია დრო, ჩვეულებრივ მოცემულია დრო პლუს-მინუს გარკვეული ცდომილება, განსაზღვრული გარე ფაქტორებით, ისეთებით როგორიც არის სისტემური საათის სიზუსტე. ჩვეულებრივ ეს არ წარმოადგენს პრობლემას, დროის ძალიან ზუსტი შეზღუდვების მქონე სისტემების გამოკლებით.

UML 1-ში დროითი შეზღუდვები შესაძლებელი იყო მიგვეთითებინა სხვადასხვა დიგრამაზე(მდგომარეობის, მოღვაწეობის, განლაგების), რისთვისაც გამოიყენება დროითი(გასაღებური სიტყვით after) და ცვლილების(გასაღებური სიტყვით when) მოფლენები, მაგრამ არ იყო ცალკე დიაგრამა, რომელიც განკუთვნილი იქნებოდა დროითი შეზღუდვებისათვის. UML 2 წარუდგენს რეალური დროის სისტემების მოდელის დამმუშავებლებს დროით დიაგრამას. ეს არის ურთიერთქმედების დიაგრამის ნაირსახეობა, რომელშიც ძირითადი ყურადღება მიმართულია დროითი შეზღუდვების მოდელირებაზე და შესაბამისად იგი იდეალურად მიესადაგება რეალური დროის სისტემების ამ ასპექტს.

დროითი დიგრამები საკმაოდ მარტივია. დრო გადებულია ჰორიზონტალურად მარცხნიდან მარჯვნივ. ობიექტთა სასიცოცხლო ხაზი და მათი მდგომარეობები განლაგდება ვერტიკალურად. გადასვლები ობიექტთა სასიცოცხლო ხაზის მდგომარეობებს შორის და პირობებით წარმოიდგინება გრაფიკის სახით. ნახ.4.11.-ზე მოყვანილია დროითი დიაგრამა კლასისათვის «სტუდენტი» ტექნიკურ უნივერსიტეტში მიღებული სასწავლო გრაფიკის შესაბამისად. ეს დიაგრამა გვიჩვენებს, თუ რა ხდება, როდესაც ფორმირდება მოვლენა « სასწავლო სემესტრის დაწყება ». სემესტრის დაწყების თარიღი(T) მოძრავია, პირველი სემესტრისათვის ეს შეიძლება იყოს სექტემბრის პირველი ან მეორე კვირის დასაწყისი, ხოლო მეორე სემესტრისათვის კი მარტის პირველი კვირის დასაწყისი.
ქვემოთ მოყვანილია მოყვანილი დროითი დიაგრამის მიმდევრობითი ანალიზი:

t = 0; - სტუდენტი იმყოფება დასვენების მდგომარეობაში.

T<=t<=6 კვირა; - ხდება მოვლენა „სასწავლო სემესტრის დაწყება“ და სტუდენტი გადადის „სწავლება“ მდგომარეობაში, რაც გულისხმობს ლექცია-სემინარების დასწრებას და მათზე მიღებული დავალებების შესრულებას.

t=1 კვირა(მე-7 კვირა); - სტუდენტს ეწყება პირველი შუალედური გამოცდა.

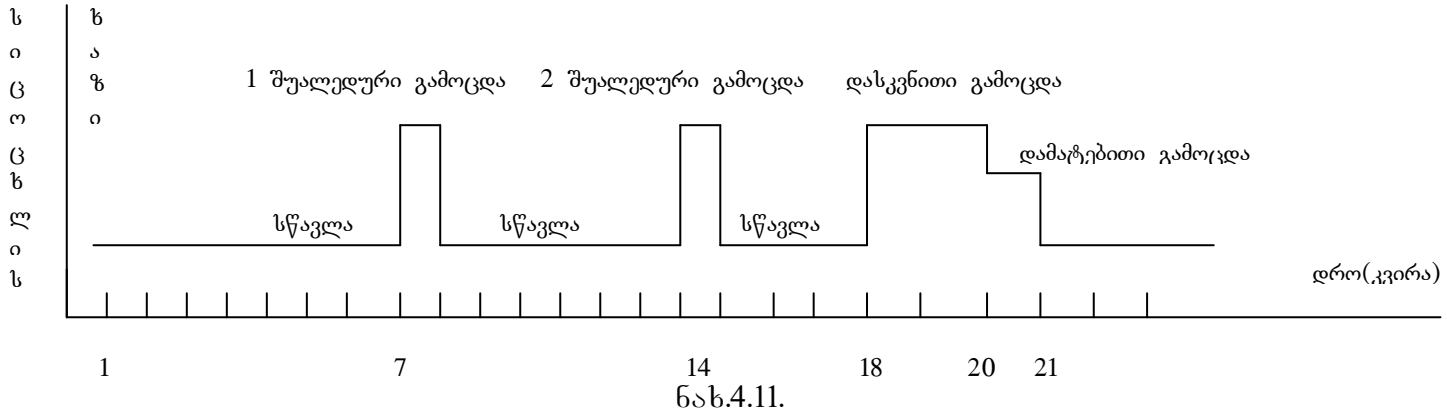
7<=t<=14: სტუდენტი უბრუნდება მდგომარეობას „სწავლა“.

t=1 კვირა(მე-14 კვირა); - სტუდენტს ეწყება მეორე შუალედური გამოცდა.

15<=t<=17: - სტუდენტი უბრუნდება მდგომარეობას „სწავლა“.

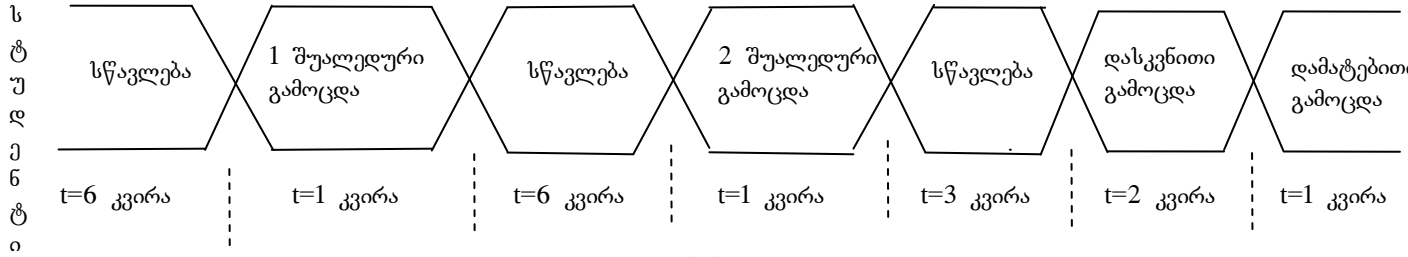
18<=t<=20: - სტუდენტს ეწყება დასკვნითი გამოცდები.

20<=t<=21: - იმ სტუდენტებს, რომლებმაც ვერ დაიმსახურეს გამსვლელი ქულები(51-ზე მეტი), მაგრამ აქვთ არა ნაკლებ 41 ქულისა ეწყებათ დამატებითი გამოცდები.



ნახ.4.11.

დროითი დიაგრამები ასევე შეგვიძლია წარმოვადგინოთ უფრო კომპაქტური ფორმით, როდესაც მდგომარეობები წარმოიდგინება ჰორიზონტალურად. ნახ.4.12.-ზე ასეთი კომპაქტური სახით წარმოდგენილია დროითი დიაგრამა ნახ.4.11.-დან. ასეთი კომპაქტური ფორმით აქცენტი ძირითადათ გადაიტანება უფრო მეტად მდგომარეობებზე და შეფარდებით დროზე, და არა აბსოლუტური დროის წარმოდგენაზე.

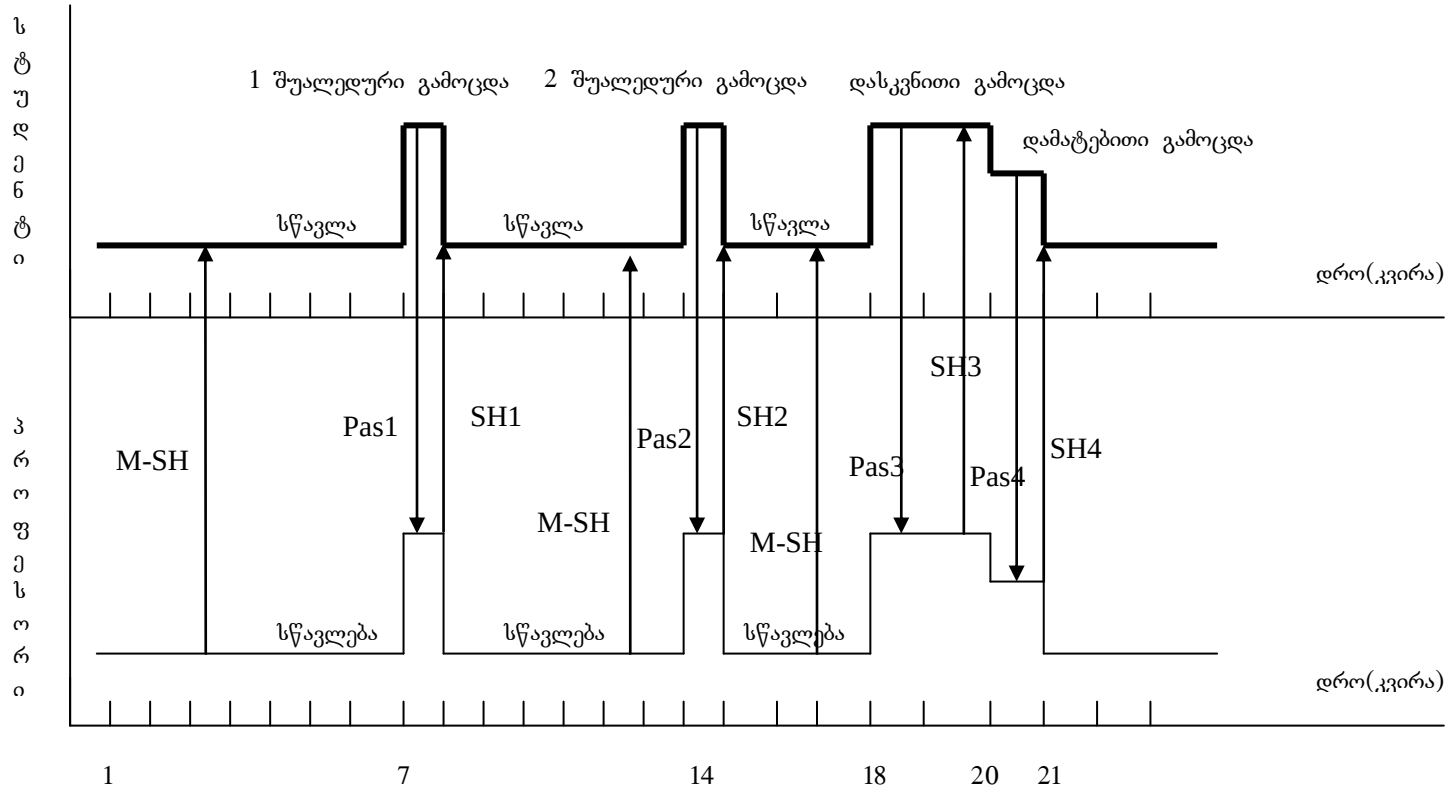


ნახ.4.12.

დროითი დიაგრამა ასევე შესაძლებელია გამოყენებულ იქნას დროითი შეზღუდვების ილუსტრირებისათვის ურთიერთმოქმედ ორ ან მეტ სასიცოცხლო ხაზს შორის. ნახ.4.13.-ზე მოცემულია ურთიერთქმედება სასიცოცხლო ხაზებს შორის: სტუდენტი და პროფესორი.

აქ შესაძლებელია აღინიშნოს, რომ პროფესორის დროითი დიაგრამა ვიზუალურად გავს სტუდენტისას, მაგრამ არა შინაარსობრივად. მდგომარეობა „სწავლება“ პროფესორის

დიაგრამაზე აღნიშნავს ლექცია-სემინარების და ლაბორატორიული მეცადინეობების ჩატარებას, ხოლო ტესტირება და გამოცდა დროითი ფაზისა და სასწავლო გეგმის შესაბამისი ტესტების მომზადებას და გამოცდის შეფასებას. დროით დიაგრამას გააჩნია ორი განყოფილება, ერთი თითოეული სასიცოცხლო ხაზისათვის. დროით დიგრამებზე შესაძლებელია შეტყობინებების ჩვენება, რომლითაც იცვლებიან სასიცოცხლო ხაზები. კერძოდ მოცემულ დიაგრამაზე მიღებულია შემდეგი აღნიშვნები: M-SH - სტუდენტთა მიმდინარე მოსწრების შეტანა(შეიტანება ყოველი კვირის ბოლოს და არ უნდა აღემატებოდეს 2 ქულას), SH1 – პირველი ტესტირების შეფასება(არ უნდა აღემატებოდეს 20 ქულას), SH2 – მეორე ტესტირების შეფასება(არ უნდა აღემატებოდეს 20 ქულას), SH3 – დასკვნითი გამოცდის შეფასება(არ უნდა აღემატებოდეს 30 ქულას და ნაკლები 8-ს), SH4 – დამატებითი გამოცდის შეფასება(არ უნდა აღემატებოდეს 20 ქულას), pas1 – პირველი ტესტირების პასუხი, pas2 – მეორე ტესტირების პასუხი, pas3 – დასკვნითი გამოცდის პასუხი, pas4 – დამატებითი გამოცდის პასუხი.



ნახ.4.13.

მოყვანილი სახეობის დროითი დიაგრამები წარმოადგენენ მოხერხებულ საშუალებას დროითი შეზღუდვების მოდელირებისათვის რეალურ დროში მომუშავე სისტემების დამუშავებისას.

4.7. ავტომატები

ურთიერთქმედებით შესაძლებელია ერთობლივად მომუშავე ობიექტის ქცევის მოდელირება. ავტომატი კი საშუალებას იძლევა მოვახდინოთ ცალკეული ობიექტის ქცევის მოდელირება. ავტომატი აღწერს ქცევას მიმდევრობითი მდგომარეობების სახით, რომლებზედაც გაივლის ობიექტი თავისი სიცოცხლის განმავლობაში.

უნიფიცირებულ პროცესში ავტომატები ძირითადათ გამოიყენებიან დაპროექტების სამუშაო ნაკადში სისტემის დინამიური ასპექტების მოდელირებისათვის. ყველაზე ხშირათ ავტომატები გამოიყენებიან **დაზუსტების** ფაზის დასრულების და **აგების** ფაზის დასაწყისში, როდესაც ხორციელდება მცდელობა უფრო დეტალურად გავერკვეთ სისტემის კლასებში მათი რეალიზების მიზნით.

ავტომატის, შესაბამისად მისი სამი ძირითადი ელემენტის ვიზუალირება UML-ში ხდება ობიექტების პოტენციალური მდგომარეობების გამოყოფით და გადასვლებით მათ შორის (მდგომარეობათა დიაგრამის სახით). უფრო დაწვრილებით მდგომარეობების, მოვლენების და გადასვლების სემანტიკა UML-ში განხილულია ნაშრომებში[2,3,4], მოცემულ შემთხვევაში განიხილება ის დამატებითი საშუალებები, რომლებიც გამოიყენება UML2-ში რეაქტიული ობიექტების სასიცოცხლო ციკლის მოდელირებისათვის. რეაქტიული ობიექტები:

- რეაგირებენ გარე მოვლენებზე(მოვლენებზე, რომლებიც ხდება ობიექტის კონტექსტის გარეთ);
- ახდენენ შიდა მოვლენების გენერირებას და რეაგირებას;
- მათი სასიცოცხლო ციკლი მოდელირდება როგორც მდგომარეობების, გადასვლების და მოვლენების თანმიმდევრობა;
- მათი მიმდინარე ქცევა შესაძლებელია დამოკიდებული იყოს წინმდებარე ქცევაზე.

ობიექტის მდგომარეობა ეს სიტუაციაა მის ცხოვრებაში, რომლის განმავლობაშიც იგი აკმაყოფილებს გარკვეულ პირობებს, ახორციელებს გარკვეულ მოღვაწეობას ან ელოდება სხვა მოვლენას.

ობიექტის მდგომარეობა იცვლება დროში, მაგრამ ნებისმიერ მომენტში იგი განისაზღვრება:

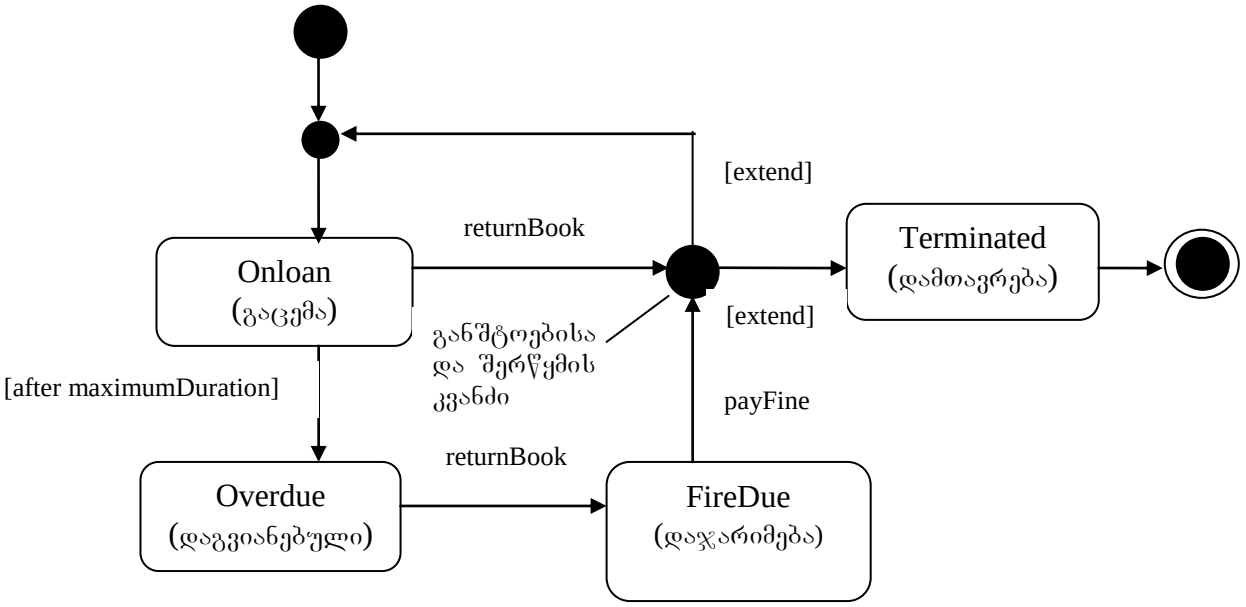
- ობიექტის მდგომარეობებით;
- სხვა ობიექტებთან მიმართებებით;
- განხორციელებული მოღვაწეობებით.

შეიძლება ითქვას, რომ მდგომარეობა ეს ობიექტის სემანტიკურად მნიშვნელოვანი მდგომარეობაა, რომელიც უნდა გამოვლენილი იქნან. ავტომატის ყოველი მდგომარეობა შესაძლებელია შეიცავდეს რამოდენიმე ან არც ერთ მოქმედებას. ყოველი მოქმედება მდგომარეობაში ასოცირდება შიდა გადასვლასთან, რომლის ინიცირებას ახდენს მოვლენა. მდგომარეობაში შესაძლებელია იყოს მოქმედებების და შიდა გადასვლების ნებისმიერი რაოდენობა.

გადასვლები გვიჩვენებენ მოძრაობას მდგომარეობებს შორის. ყოველ გადასვლას შესაძლებელია გააჩნდეს სამი ელემენტი:

1. **მოვლენა** რამოდენიმე ან არცერთი - განსაზღვრავენ გარე ან შიდა წარმოსახვას, რომლებიც ახდენენ გადასვლის ინიციირებას.
2. **დამცავი პირობა** რამოდენიმე ან არცერთი– ლოგიკური გამოსახულება, რომელიც უნდა შესრულდეს, რათა მოხდეს გადასვლა. პირობას მიუთითებენ მოვლენის შემდეგ.
3. **მოქმედებები** რამოდენიმე ან არცერთი – სამუშაოს ნაწილი, რომელიც ასოციირდება გადასვლასთან და სრულდება გადასვლის ამუშავებისას.

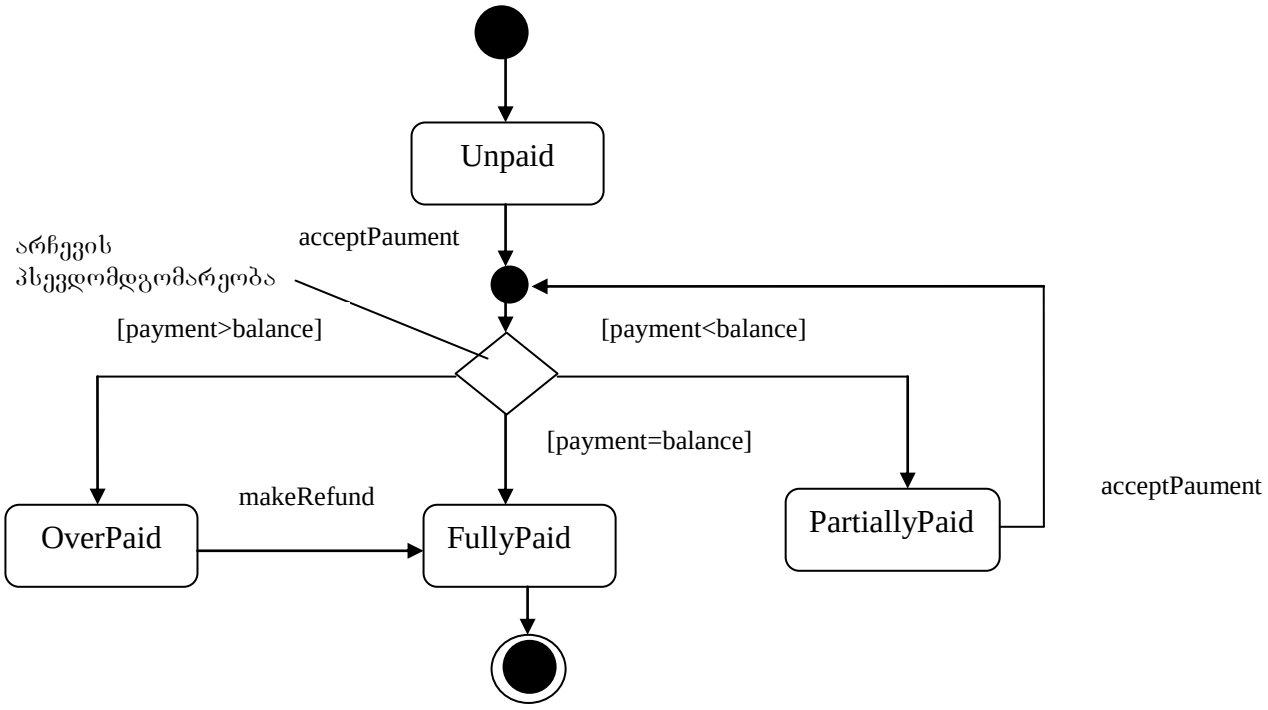
UML1 განსხვავებით UML2-ში საშუალება გვეძლევა მოვახდინოთ გადასვლების შეერთება – ეს არის გადასვლების შერწყმის ან განშტოების წერტილი. იგი გამოისახება შავი შეფერადებული წრით ერთი ან რამოდენიმე შემავალი და გამომავალი გადასვლებით. ნახ.4.14-ზე ნაჩვენებია ავტომატი კლასისათვის Loan(სესხი). Loan ახდენს წიგნის მიღების მოდელირებას ბიბლიოთეკაში. ავტომატს Loan-ს აქვს შერწყმის მარტივი კვანძი.



ნახ.4.14.

გადასასვლელ პსევდომდგომარეობას შესაძლებელია ქონდეს რამოდენიმე გამოსასვლელი გადასვლა. ამ შემთხვევაში ყოველი გამოსასვლელი გადასვლა უნდა დაცული იქნას ურთიერთგამომრიცხავი დამცავი პირობით, რომელიც უზრუნველყოფს მხოლოდ ერთი გადასვლის ამოქმედებას. ავტომატი კლასისათვის ოან ფართოვდება, რათა დამუშავდეს შემთხვევა, როდესაც წიგნის გაცემა შესაძლებელია გაგრძელდეს. ბიზნეს წესი იმაში მდგომარეობს, რომ დაბრუნების ვადის გაგრძელებისათვის გაცემული წიგნი უნდა წარდგენილი იქნას ბიბლიოთეკაში.

უბრალო განშტოების წარმოსადგენად შერწყმის გარეშე გამოიყენება არჩევის პსევდომდგომარეობა. იგი საშუალებას იძლევა ნაკადი მივმართოდ მოცემული პირობის შესაბამისად. ნახ.4,15. მოყვანილია ავტომატი კლასისათვის BankLoan(საბანკო სესხი). მოვლენის acceptPaument (გადასახადი) მიღებისას ობიექტი BankLoan გადადის მდგომარეობიდან Unpaid (არ არის გადახდილი) სამიდან ერთ ერთ მდგომარეობაში – FullyPaid(გადახდილია სრულად), OverPaid(გადახდილია ზედმეტად) ან PartiallyPaid(გადახდილია ნაწილობრივ) – გადახდის(paument) ოდენობიდან და BankLoan-ის გადაუხდელი ბალანსიდან გამომდინარე.

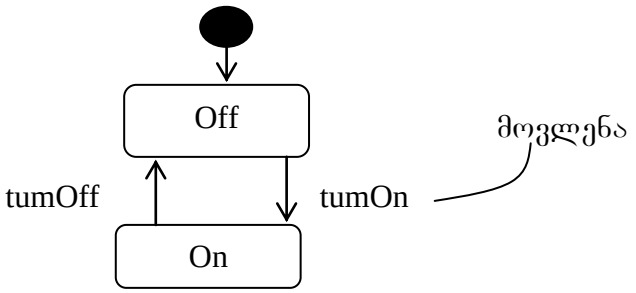


ნახ.4.15.

4.8. მოვლენები და სიგნალები

UML - ში ნებისმიერი წარმოსახვა, რომელიც შესაძლებელია მოხდეს, მოდელირდება როგორც მოვლენა.

მოვლენა – ეს არსებითი ფაქტის აღწერაა, რომელსაც აქვს გარკვეული ადგილი დროსა და სივრცეში. სიგნალის მიღება, დროითი შუალედის ამოწურვა და მდგომარეობის შეცვლა – ეს ასინქრონული მოვლენების მაგალითია, რომლებიც შეიძლება დადგეს ნებისმიერ მომენტში. გამოძახება – ეს სინქრონული მოვლენაა, რომელიც გამოიყენება რომელიმე ოპერაციის გასაშვებათ. ავტომატების კონტექსში მოვლენა ეს სტიმულია, რომელსაც შეუძლია გამოიწვიოს გადასვლა ერთი მდგომარეობიდან მეორეში (ნახ. 4.8.1).



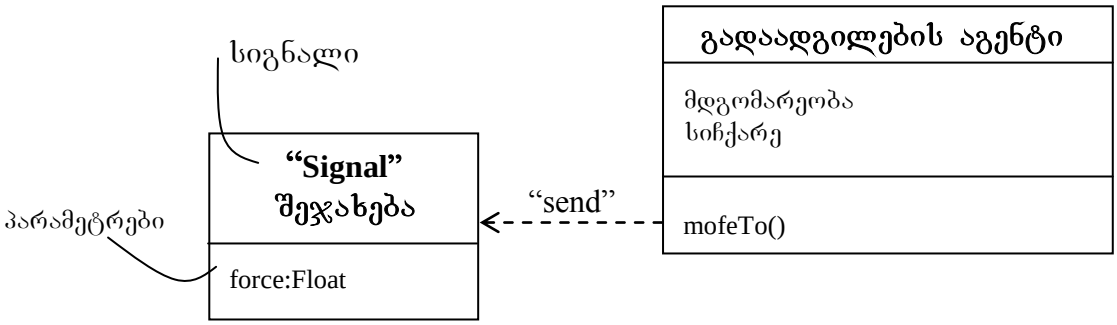
ნახ.4.8.1.

სიგნალი – ეს მოვლენის სახეა, რომელშიც სტიმული გადაეცემა ასინქრონულად ერთი ეგზემპლიარიდან მეორეს.

UML – ში შეიძლება მოვლენათა ოთხი ტიპის მოდელირება: სიგნალები, გამოძახება, დროის შუალედის ამოწურვა და მდგომარეობის შეცვლა.

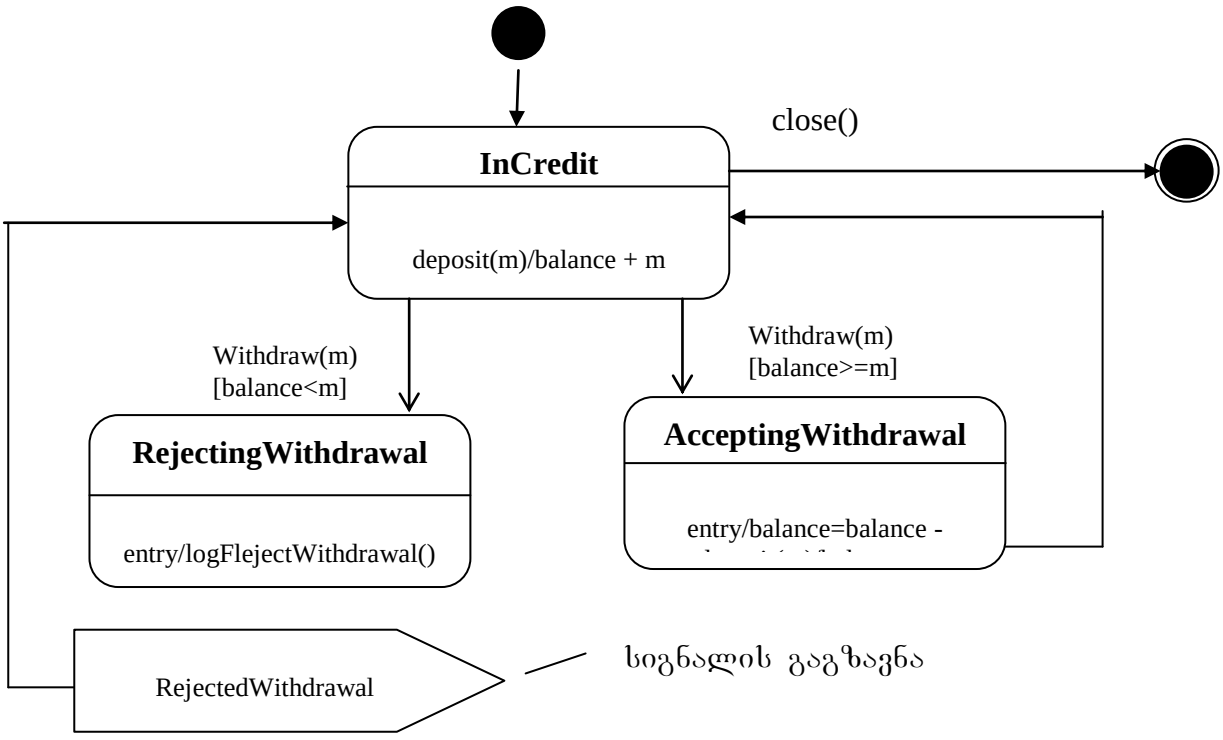
სიგნალები. სიგნალი ეს დასახელებული ობიექტია, რომელიც ასინქრონულად აღიძვრება ერთი ობიექტით და მიიღება მეორეთი. გამორიცხვა, რომელიც ფართოდ გამოიყენება თანამედროვე დაპროგრამების ენების უმეტესობაში, ყველაზე გავრცელებული სახეა შიდა სიგნალებს შორის. ობიექტების შექმნა და მოსპობაც სიგნალის ნაირსახეობას მიეკუთვნება.

სიგნალი შეიძლება გაიგზავნოს როგორც მდგომარეობის გადასვლის მოქმედება ავტომატში ან როგორც შეტყობინების გაგზავნა ურთიერთქმედებისას. ოპერაციების შესრლებისას ასევე შეიძლება გაიგზავნოს სიგნალი. იმისათვის, რომ მიუთითოდ ოპერაციის მიერ სიგნალის გაგზავნა, შეიძლება ვისარგებლოთ მიმართებით დამოკიდებულება სტერეოტიპით **send**.



ნახ.4.8.2.

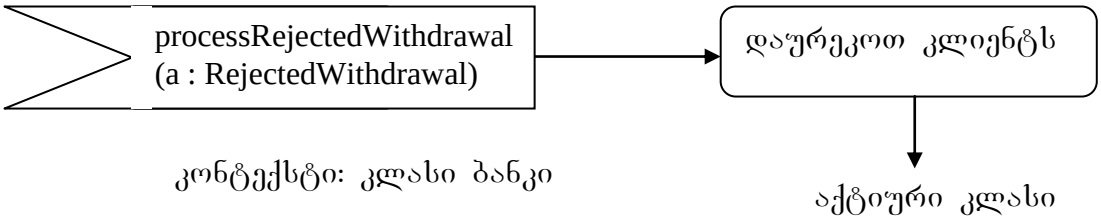
როგორც ნახ.4.8.2.-დან ჩანს სიგნალები მოდელირდებიან სტერეოტიპული კლასებით, რომლებიც თავის ატრიბუტებში შეიცავენ ინფორმაციას, რომელიც უნდა გადაიცეს. სიგნალს ჩვეულებრივ არ გააჩნია ოპერაციები, რადგან ის განკუთვნილია მხოლოდ ინფორმაციის გადასაცემად.



ნახ.4.8.3.

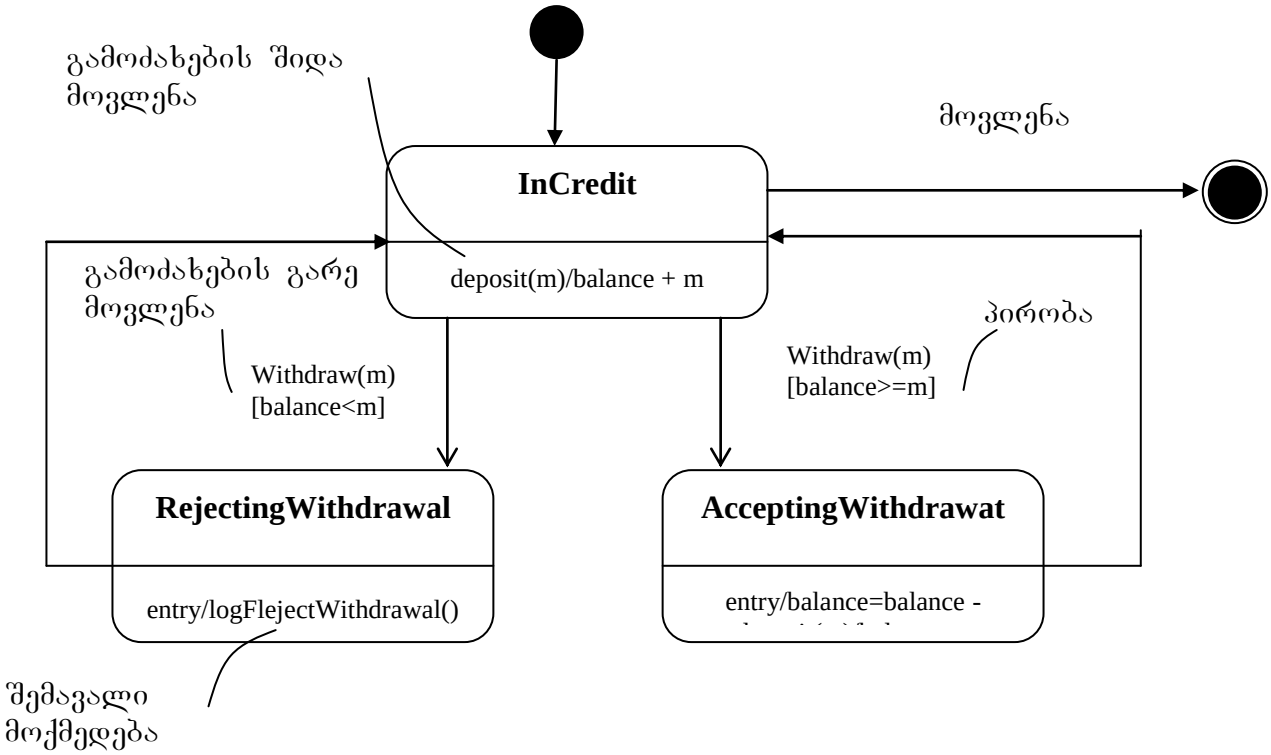
ნახ.4.8.3.-ზე წარმოდგენილია ავტომატი SimpleBankAccount(უბრალო საბანკოანგარიში). ანგარიშიდან თანხის მოხსნის მოთხოვნის გადახრისას იგზავნება სიგნალი. სიგნალი აღინიშნება ხუთკუთხედით, რომლის შიგნით მიეთითება სიგნალის დასახელება.

სიგნალის მიღება აღინიშნება შეზნექილი ხუთკუთხედით, როგორც ეს ნაჩვენებია ნახ.4.8..4.-ზე.



ნახ.4.8.4. სიგნალის მიღება

გამოძახების მოვლენები. თუ სიგნალის მოვლენა წარმოადგენს სიგნალის აღძვრას, გამოძახების მოვლენა განკუთვნილია ოპერაციის შესრულების აღწერისათვის. ორივე შემთხვევაში მოვლენას შეუძლია გამოიწვიოს მდგომარეობის შეცვლა ავტომატში. იმ დროს როდესაც სიგნალი წარმოადგენს ასინქრონულ მოვლენას, გამოძახების მოვლენა ჩვეულებრივ სინქრონულია. ეს ნიშნავს იმას, რომ როდესაც ერთი ობიექტი ინიციირებას ახდენს ოპერაციის შესრულებაზე მეორე ობიექტზე, რომელსაც გააჩნია თავისი ავტომატი, მართვა გადაიცემა გამომგზავნიდან მიმღებზე, ამუშავდება შესაბამისი გადასვლა, შემდეგ ოპერაცია სრულდება, მიმღები გადადის ახალ მდგომარეობაში და უბრუნებს მართვას გამომგზავნს.



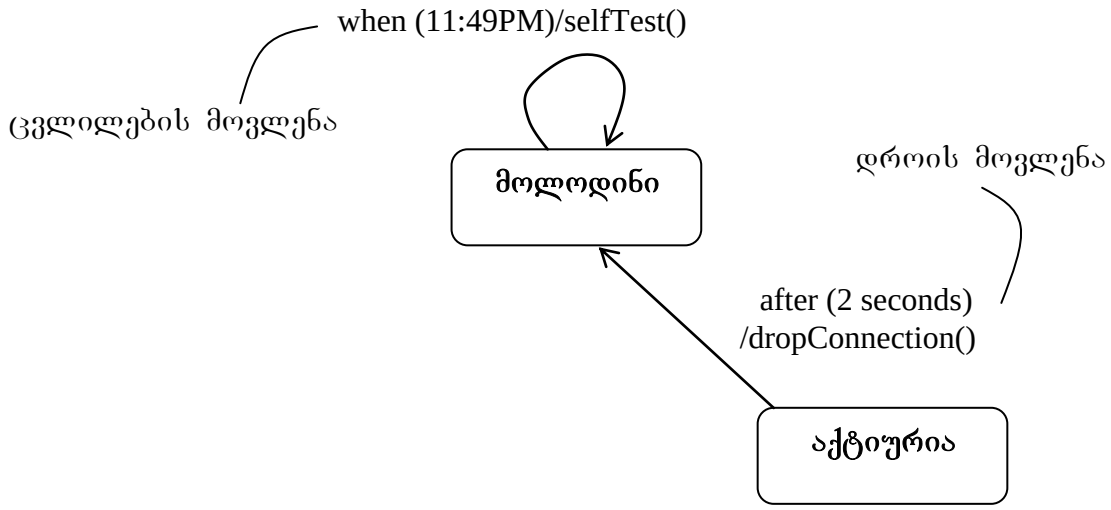
ნახ.4.8.5.

როგორც ნახ.4.8.5. – დან ჩანს ვიზუალურად სიგნალისა და გამოძახების მოვლენები არ განსხვავდებიან ერთმანეთისაგან. მაგრამ მოვლენის მიმღებმა რასაკვირველია იცის ამ განსხვავების შესახებ (ოპერაციის გამოცხადების საფუძველზე თავის ოპერაციათა ცხრილში). სიგნალი როგორც წესი მუშავდება ავტომატის დონეზე, ხოლო გამოძახების

მოვლენა – მეთოდით. მოვლენიდან სიგნალზე ან ოპერაციაზე გადასასვლელად შესაძლებელია ვისარგებლოთ ინსტრუმენტალური პროგრამით.

დროითი და ცვლილების მოვლენები. დროითი მოვლენა წარმოადგენს დროითი შუალედის ამოწურვას. იგი წარმოიდგინება გასაღებური სიტყვით after(შემდეგ), მას მოსდევს გამოსახულება, რომელიც ითვლის დროის გარკვეულ შუალედს. გამოსახულება შეიძლება იყოს მარტივი ან რთული. თუ არ არის მითითებული, დროის ათვლა იწყება მიმდინარე მდგომარეობაში შესვლის მომენტიდან.

ცვლილების მოვლენით აღიწერება მდგომარეობის შეცვლა ან გარკვეული პირობის შესრულება. იგი წარმოიდგინება გასაღებური სიტყვით when, რომელსაც მოსდევს ბულის გამოსახულება. ასეთი გამოსახულება შესაძლებელია აბსოლუტური დროის მომენტის (მაგალითად, when=11;59) ან პირობის შესამოწმებლად (მაგალითად, when altitude<1000).



ნახ.4.8.6.

მოვლენის გაგზავნა და მიღება. სიგნალისა და გამოძახების მოვლენებში უკიდურეს შემთხვევაში მონაწილეობენ ორი ობიექტი: ობიექტი, რომელიც აგზავნის სიგნალს ანუ ოპერაციის ინიციირებას ახდენს, და ობიექტი, რომელსაც ეგზავნება მოვლენა. რამდენადაც სიგნალები თავისი ბუნებით ასინქრონულია, ხოლო ასინქრონული გამოძახებები თავისთავად წარმოადგენენ სიგნალებს, მოვლენათა სემანტიკა უკავშირდება აქტიური და პასიური ობიექტების სემანტიკას.

ნებისმიერი კლასის ეგზემლიარს შეუძლია გაგზავნოს სიგნალი მიმღებ ობიექტთან ან მოახდინოს მასში ოპერაციის ინიციირება. გაგზავნის რა სიგნალს მიმღებთან, იგი აგრძელებს თავის მართვის ნაკადს, არ ელოდება რა მისგან პასუხს.. პირიქით, თუ ობიქტი ახდენს ოპერაციის ინიციირებას, იგი უნდა დაელოდოს მიმღებისაგან პასუხს.

ნებისმიერი კლასის ნებისმიერი ეგზემპლიარი შეიძლება იყოს მოვლენა სიგნალის ან მოვლენა გამოძახების მიმღები. თუ ეს სინქრონული მოვლენაა, გამგზავნი და მიმღები იმყოფებიან მდგომარეობაში რანდევუ ოპერაციის შესრულების მთელი ხანგრძლივობისას. ეს ნიშნავს, რომ გამგზავნის მართვის ნაკადი ბლოკირდება მიმღების მართვის ნაკადით, სანამ ოპერაცია არ დასრულდება. თუ ეს სიგნალია, გამგზავნი და მიმღები არ შედიან მდგომარეობაში რანდევუ: გამგზავნი აგზავნის სიგნალს, მაგრამ არ ელოდება პასუხს მიმღებისაგან. გამოძახების მოვლენები, რომლებსაც ღებულობს ობიექტი, მოდელირდება როგორც ოპერაციები ამ ობიექტის კლასებზე. დასახელებული სიგნალები, რომლებსაც ღებულობს ობიექტი, მოდელირდება ჩამონათვალის სახით კლასის დამატებით განყოფილებაში.

თუ გარე მოვლენები გადაიცემა სისტემასა და მის მომხმარებელს(აქტიორს) შორის, შიდა მოვლენები გადაიცემა სისტემის შიგნით არსებულ ობიექტებს შორის, რომლის მაგალითი შესაძლებელია იყოს გამორიცხვა.

გამორიცხვები. კლასების ან ინტერფეისების ქცევის ვიზუალირების მნიშვნელოვან ნაწილს წარმოადგენს გამორიცხვების სპეციფიცირება, რომლებსაც შეუძლიათ აღაგზნონ მისი ოპერაციები. თუ გაქვთ კლასი ან ინტერფეისი, მაშინ ოპერაციები, რომლებიც შესაძლებელია მათზე გამოვიძახოთ ნათლად ჩანს აღწერიდან, მაგრამ იმის გაგება თუ რომელ გამორიცხვებს აღაგზნებენ ისინი არ არის ადვილი, თუ ეს ნათლად არ არის მითითებული მოდელში.

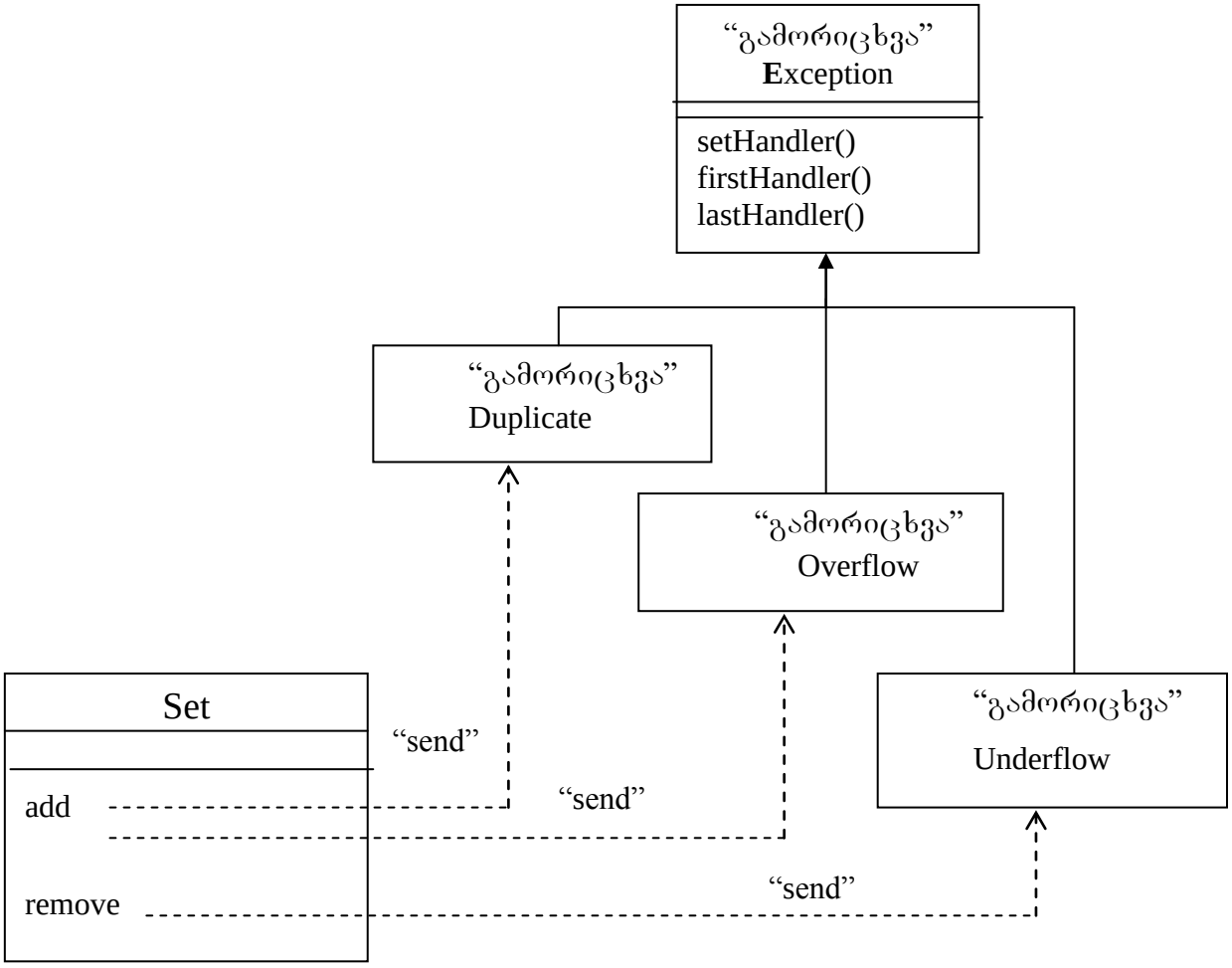
გამორიცხვები წარმოადგენენ სიგნალების კერძო შემთხვევებს და მოდელირდებიან სტერეოტოპული კლასებით. გამორიცხვები შესაძლებელია მიუერთოდ ოპერაციების სპეციფიკაციებს. გამორიცხვების მოდელირება წარმოადგენს ოპერაციას, გარკვეული აზრით საწინააღმდეგოს სიგნალთა სიმრავლის მოდელირებისა. სიგნალთა ოჯახის მოდელირების ძირითადი მიზანია– მოვახდინოთ იმის სპეციფიცირება, თუ რომელი სიგნალები შეუძლია მიიღოს აქტიურმა ობიექტმა; გამორიცხვების მოდელირების მიზანია უჩვენოთ, რომელი გამორიცხვები შეიძლება აღიძვრეს ობიექტით თავისი ოპერაციებიდან.

გამორიცხვები მოდელირება ხდება შემდეგნაირად:

1. ყოველი კლასისა და ინტერფეისისათვის და ყოველი მათში განსაზღვრული ოპერაციისათვის განიხილეთ, რომელი გამორიცხვების აღძვრა არის შესაძლებელი.
2. მოახდინეთ გამორიცხვების ორგანიზება იერარქიულად. მაღალ დონეებზე განვალაგოთ ზოგადი გამორიცხვები, ქვედაზე – სპეციალიზირებულები, საჭიროების შემთხვევაში შემოიტანეთ შუალედური გამორიცხვები.

3. მიუთითედ ყოველი ოპერაციისათვის, რომელი გამორიცხვები შეუძლია მან აღაგზნოს. ეს შეიძლება მიუთითოდ დიაგრამაზე (დამოკიდებულებით send ოპერაციიდან მის გამორიცხვამდე) ან მოვათავსოთ გამორიცხვების ჩამონათვალი ოპერაციათა სპეციფიკაციაში.

ნახ.4.8.8.–ზე წარმოდგენილია გამორიცხვების იერარქიული მოდელი, რომლებიც შესაძლებელია აღიძვრას კლასით Set. ამ იერარქიის თავში იმყოფება აბსტრაქტული კლასი Exception, ხოლო ქვევით სპეციალიზირებული გამორიცხვები Duplicate, Overflow და Underflow. როგორც ჩანს, ოპერაცია add(დამატება) აღაგზნებს გამორიცხვებს Duplicate(ასლი) და Overflow(გადავსება), ხოლო ოპერაცია remove(ამოგდება) – მხოლოდ ოპერაციას Underflow(ცდა ცარიელი ბუფერის წაკითხვისა). შეგვეძლო მოგვეხსნა დამოკიდებულებები წინა ხედიდან და შეგვეტანა გამორიცხვები ყოველი ოპერაციის აღწერაში. როგორც არ უნდა იყოს, ვიცით რა, თუ რომელი გამორიცხვა აღიგზნება ყოველი ოპერაციის შემდეგ, ჩვენ შეგვეძლება შევქმნათ პროგრამები Set კლასის გამოყენებით.



ნახ.4.8.8.

4.9. მდგომარეობათა დიაგრამები

მდგომარეობათა დიაგრამა გამოიყენება სისტემის დინამიური ასპექტების მოდელირებისათვის. ფაქტიურად იგი გვიჩვენებს ავტომატს. ობიექტის სასიცოცხლო ციკლის მოდელირებისას სასარგებლოა როგორც მოღვაწეობის დიაგრამები, ისე მდგომარეობათა დიაგრამები. მაგრამ თუ მოღვაწეობის დიაგრამები გვიჩვენებენ მართვის ნაკადს მოღვაწეობიდან მოღვაწეობამდე, მდგომარეობათა დიაგრამაზე წარმოდგენილია მართვის ნაკადი მდგომარეობიდან მდგომარეობამდე.

მდგომარეობათა დიაგრამა გამოიხატება გრაფის სახით, მასზე ძირითადათ გამოისახება:

- მარტივი და შედგენილი მდგომარეობები;
- გადასვლები ასოციირებულ მოვლენებთან ან მოქმედებებთან ერთად.

მდგომარეობათა დიაგრამა შედგენილია ელემენტებისაგან, რომელიც გვხვდება ნებისმიერ ავტომატში და მათზე გამოიყენება ავტომატის ყველა მახასიათებლები. როგორც ყველა დიაგრამა, მდგომარეობათა დიაგრამაც შეიძლება შეიცავდეს შენიშვნებსა და შეზღუდვებს.

ყველაზე ხშირათ მდგომარეობათა დიაგრამებს იყენებენ რეაქტიული ობიექტების მოდელირებისათვის, განსაკუთრებით კლასების, პრეცენდენტების ან მთლიანად სისტემის. რეაქტიულს უწოდებენ ისეთ ობიექტებს, რომელთა ქცევა ყველაზე კარგად გამოიხატება მისი რეაქციით საკუთარი კონტექსის გარეთ მომხდარ მოვლენებზე. რეაქტიულ ობიექტს აქვს მკაფიოდ გამოხატული სასიცოცხლო ციკლი, როდესაც მიმდინარე ქცევა გამოწვეულია წარსულით. როგორც წესი რეაქტიული ელემენტები იმყოფებიან ლოდინის მდგომარეობაში, სანამ არ მიიღებს მოვლენას, ხოლო როდესაც ეს მოხდება მისი რეაქცია დამოკიდებულია წინმდებარე მოვლენებზე. მას შემდეგ რაც ობიექტი მოახდენს მასზე რეაგირებას, იგი ხელახლა გადადის შემდეგი მოვლენის ლოდინის მდგომარეობაში. ასეთი ობიექტებისათვის ინტერესს წარმოადგენს პირველ რიგში მდგრადი მდგომარეობები, მოვლენები რომლებიც ახდენენ გადასვლის ინიციირებას ერთი მდგომარეობიდან მეორეში და მოქმედებები, რომლებიც სრულდება მდგომარეობის შეცვლისას.

რეაქტიული ობიექტების მოდელირებისას უნდა მოვახდინოთ ძირითადათ სამი საგნის სპეციფიცირება: მდგრადი მდგომარობები, რომელშიც შესაძლებელია იმყოფებოდეს ობიექტი, მოვლენები, რომლებიც ახდენენ გადასვლების ინიციირებას და მოქმედებები, რომლებიც სრულდება მდგომარეობის ყოველი შეცვლისას. რეაქტიული ობიექტების მოდელირება გულისხმობს მთელი მისი სასიცოცხლო ციკლის მოდელირებას, დაწყებული შექმნის მომენტიდან და დამთავრებული მისი მოსპობით, განსაკუთრებული აქცენტით მდგრად მდგომარეობებზე, რომლებშიც შეიძლება იმყოფებოდეს ობიექტი.

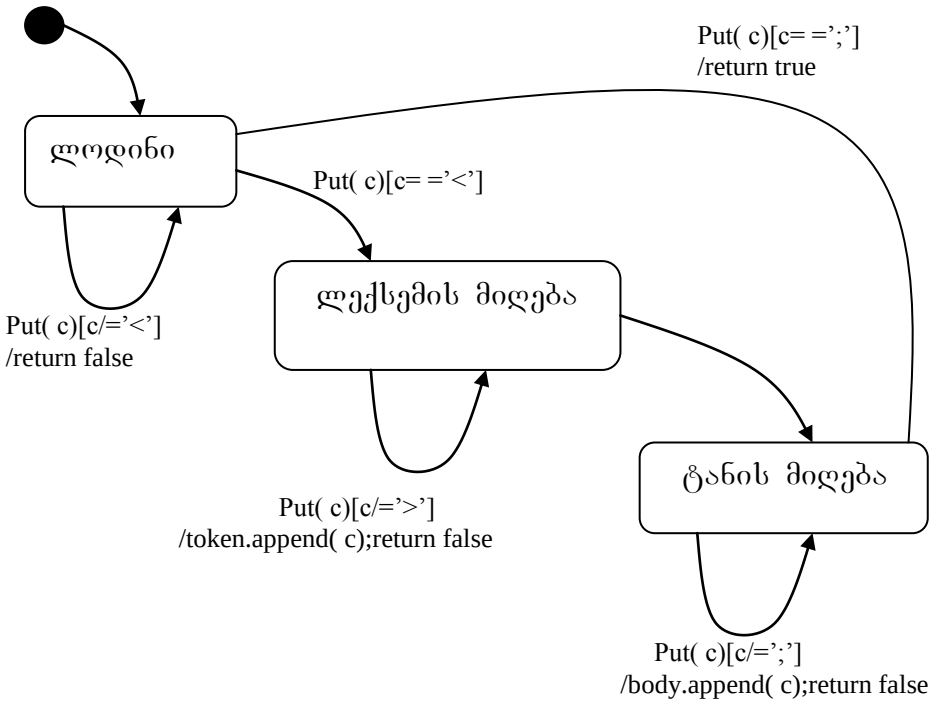
მდგრადი მდგომაეობა ეს ისეთი მდგომარეობაა, რომელშიც ობიექტი იმყოფება განუსაზღვრელად დიდი დროის განმავლობაში. მოვლენებს შეუძლიათ აგრეთვე გადასვლების ინიციირება თავის თავში და მოახდინონ შიდა გადასვლები, როდესაც საწყისი და მიზნობრივი მდგომარეობები ერთმანეთს ემთხვევა. მოვლენებზე ან მდგომარეობის ცვლილებაზე რეაქციისას ობიექტმა შეიძლება შეასრულოს გარკვეული მოქმედება.

რეაქტიული ობიექტის ქცევის მოდელირებისას ჩვენ გვეძლევა საშუალება მოვახდინოთ მოქმედების სპეციფიცირება, მივაბამთ რა მას გადასვლასთან ან მდგომარეობის შეცვლასთან. ლიტერატურაში ავტომატი, რომლის ყველა მოქმედება მიბმულია გადასვლებთან, უწოდებენ მილის მანქანას, ხოლო ავტომატი რომლის მოქმედებები მიბმულია მდგომარეობებთან, მურის მანქანას. მდგომარეობათა დიაგრამის დამუშავებისას ჩვეულებრივ გამოიყენება მილის და მურის მანქანების კომბინაცია.

რეაქტიული ობიექტის მოდელირება შედგება შემდეგი პროცესებისაგან:

1. ავირჩიოთ ავტომატისათვის კონტექსი – კლასი, პრეცენდენტი ან სისტემა მთლიანობაში.
2. ავირჩიოთ ობიექტისათვის საწყისი და საბოლაო მდგომარეობები.

.



ნახ.4.9.1.

3. დაადგინეთ ობიექტის მდგრადი მდგომარეობები, ისეთი რომლებშიც ის შეიძლება იმყოფებოდეს განუსაზღვრელად დიდი დროის განმავლობაში. დაიწყეთ ზედა დონის მდგომარეობებით, ხოლო შემდეგ გადადით ქვემდგომარეობებზე.

4. მოახდინეთ მდგრადი მდგომარეობების მოწესრიგება ობიექტის მთელი სასიცოცხლო ციკლის განმავლობაში.
5. დაადგინეთ რომელ მოვლენებს შეუძლიათ მდგომარეობებს შორის გადასვლების ინიციირება. წარმოვადგინოთ ეს მოვლენები როგორც გადასვლების ტრიგერები.
6. დაუკავშირეთ მოქმედებები გადასვლებს(მილის მანქანა) და/ან მდგომარეობებს(მურის მანქანა).
7. განიხილეთ როგორ შეიძლება ავტომატის გამარტივება ქვემდგომარეობების, განშტოებების, შერწყმის და ისტორიული მდგომარეობებით
8. გასინჯეთ, რომ ნებისმიერი მდგომარეობის მიღწევა შესაძლებელია მოვლენათა გარკვეული კომბინაციისას.
9. დარწმუნდით ჩიხური სიტუაციების არ არსებობაში, ანუ ისეთების რომლებიდანაც არ არის გადასვლა მოვლენის არც ერთი კომბინაციისას.
10. შეამოწმეთ ავტომატი ხელით ან ინსტრუმენტალური საშუალებით და დაადგინეთ როგორ იქცევა ის მოვლენათა მოსალოდნელი თანმიმდევრობისას და მათზე რეაქციისას.

ნახ.4.9.1.-ზე მოყვანილია მდგომარეობათა დიაგრამა მარტივი კონტექსურად თავისუფალი ენის გარჩევისათვის. ასეთი ენის მაგალითები შეიძლება ვნახოთ სისტემებში, რომლებშიც შემავალი ან გამომავალი ნაკადი შეადგენენ XML- შეტყობინებები. ასეთ შემთხვევაში პროექტირდება ავტომატი სიმბოლოთა ნაკადის გარჩევისათვის, რომელიც დააკმაყოფილებს ენის სინტაკს:

შეტყობინება: ’ < ’ სტრიქონი ‘ > ’ შეტყობინება ‘ ; ‘

პირველი სტრიქონი წარმოადგენს ტეგს, მეორე – შეტყობინების ტანია. მოცემული სიმბოლოების ნაკადიდან ამოირჩევა მხოლოდ შეტყობინებები, რომლებიც აკმაყოფილებენ სინტაკსის წესებს.

ნახაზიდან ჩანს, რომ ავტომატისათვის გათვალისწინებულია სამი მდგრადი მდგომარეობა: ლოდინი, ლექსემის მიღება და ტანის მიღება. მდგომარეობათა დიაგრამა დაპროექტებულია მილის მანქანის სახით – მოქმედებები მიბმულია გადასვლებთან. სინამდვილეში ავტომატში არის მხოლოდ ერთი მოვლენა, რომელიც იმსახურებს ინტერესს – გამოძახება put ფაქტიური პარამეტრით c (სიმბოლოთი). მდგომარეობაში ლოდინი ავტომატი უკუაგდებს ყველა სიმბოლოს, რომლებიც არ შეესაბამებიან ლექსემის დასაწყისს (ეს სპეციფიცირებულია დამცავი პირობით). ლექსემის დასაწყისის აღმოჩენისას ობიექტის მდგომარეობა იცვლება ლექსემის მიღებაზე. იმყოფება რა ამ მდგომარეობაში, ავტომატი ინარჩუნებს ყველა სიმბოლოს, რომლებიც არ წარმოიდგინებიან როგორც ლექსემის დასასრული (ესეც სპეციფიცირებულია დამცავი

პირობით). აღმოაჩენს რა ლექსემის დასასრულს, ობიექტი გადადის მდგომარეობაში ტანის მიღება. ამ მდგომარეობაში ავტომატი ინარჩუნებს ყველა სიმბოლოს, რომლებიც არ წარმოიდგინებიან როგორც შეტყობინების დასასრული (იხ. დამცავი პირობა). როგორც კი მიიღება შეტყობინების დასასრული, ობიექტის მდგომარეობა იცვლება მდგომარეობით ლოდინი. ბრუნდება მნიშვნელობა, რომელიც გვიჩვენებს, რომ შეტყობინება გარჩეულია და ავტომატი მზად არის შემდეგის მისაღებათ.

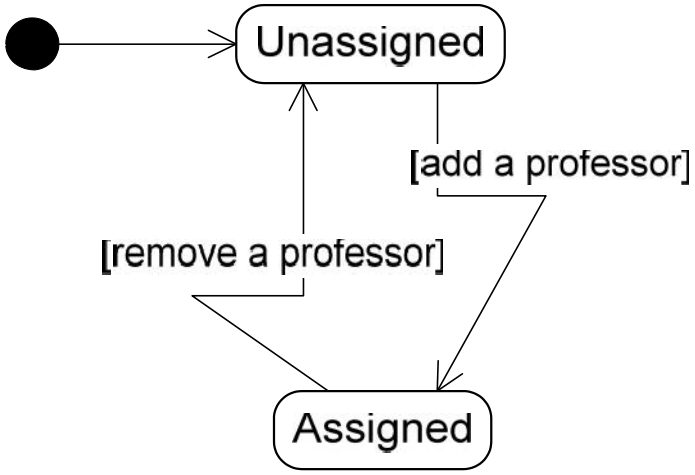
ყურადღება უნდა მივაქციოთ იმ გარემოებას, რომ ეს დიაგრამა აღწერს ავტომატს, რომელიც მუშაობს უწყვეტად – მასში არ არის საბოლაო მდგომარეობა.

მდგომარეობების მოდელირება კლასებისათვის. თუ სისტემაში გვაქვს მდგომარეობისაგან დამოკიდებული ობიექტები ქცევის რთული დინამიკით, მაშინ მათთვის შესაძლებელია ავაგოთ მოდელი, რომელიც აღწერს ობიექტის მდგომარეობებს და გადასვლებს მათ შორის. ეს მოდელი წარმოსდგება მდგომარეობის დიაგრამის სახით.

მაგალითისათვის განვიხილოთ კლასი CourseOffering – ის ობიექტის ქცევა. მდგომარეობის დიაგრამა იგება რამოდენიმე ეტაპად.

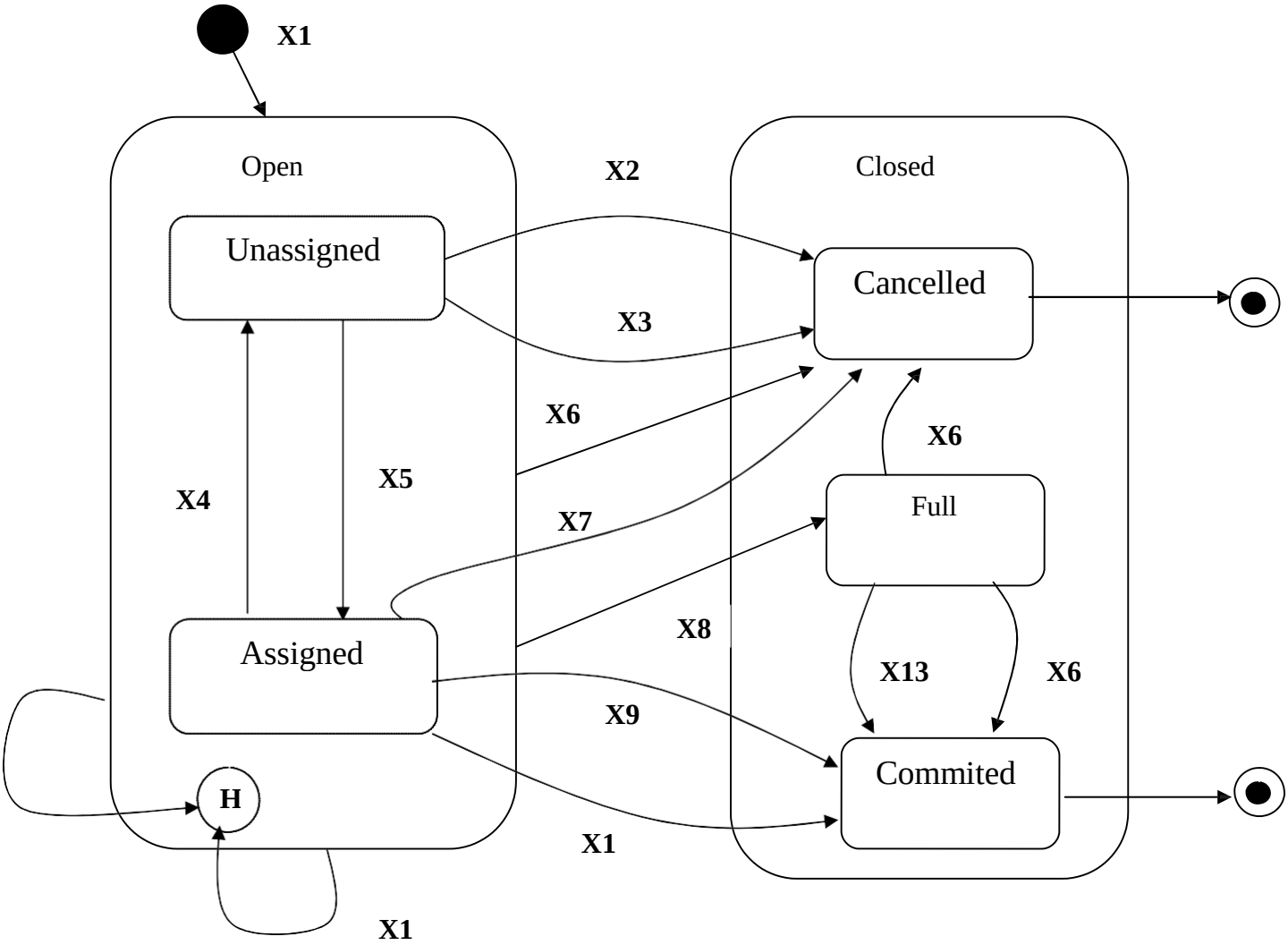
ეტაპი 1. მდგომარეობების იდენტიფიცირება. მდგომარეობათა გამოვლენის ნიშნებია ობიექტის ატრიბუტების მნიშვნელობების შეცვლა და კავშირების გაწყვეტა სხვა ობიექტებთან. ასე მაგალითად, ობიექტი CourseOffering შეიძლება იმყოფებოდეს მდგომარეობაში Open (კურსზე მიღება გახსნილია) მანამდე, სანამ მასზე დარეგისტრირებულ სტუდენტთა რაოდენობა არ გადააჭარბებს 10-ს, ხოლო როგორც კი გადააჭარბებს 10-ს, ობიექტი გადადის მდგომარეობაში Closed (კურსზე მიღება დახურულია). ამას გარდა, ობიექტი CourseOffering შესაძლებელია იმყოფებოდეს მდგომარეობაში Unassigned (იგი არავის არ მიყავს ანუ არ არსებობს კავშირი Professor – ის რომელიმე ობიექტთან) ან Assigned (ასეთი კავშირი არსებობს).

ეტაპი 2. მოვლენათა იდენტიფიცირება. მოვლენები, როგორც წესი დაკავშირებული არიან გარკვეული ოპერაციების შესრულებასთან. პრეცედენტ „ავირჩიოთ კურსები სწავლებისათვის“ ანალიზისას კლასში CourseOffering მოვალეობების განაწილების შედაგად განისაზღვრა ორი ოპერაცია – addProfessor და removeProfessor, დაკავშირებულნი გარკვეული პროფესორის მიერ კურსების არჩევასთან (ახალი კავშირის შექმნა) და არჩეული კურსზე უარის თქმა (კავშირის გაწყვეტა). ამ ოპერაციებს შეუსაბამებენ ორ მოვლენას – addProfessor და removeprofessor.



ნახ.4.9.2. გადასვლები მდგომარეობებს შორის

ეტაპი 3. მდგომარეობებს შორის გადასვლების იდენტიფიცირება. გადასვლები გამოიძახებიან მოვლენებით. მაშასადამე, Unassigned და Assigned ერთდებიან ორი გადასვლით (ნახ.4.9.2.).



ნახ.4.9.3. მდგომარეობის დიაგრამა კომპოზიციური მდგომარეობებით

CourseOfering ობიექტის ქცევის შემდგომი დეტალიზებას მივყავართ მდგომარეობის დიაგრამის აგებამდე (ნახ.4.9.3.). მოცემულ დიაგრამაზე გამოყენებულია მოდელირების ისეთი შესაძლებლობები, როგორიც არის კომპოზიციური (composite state) და ისტორიული მდგომარეობები (history state). მოცემულ შემთხვევაში კომპოზიციური მდგომარეობებია Open და Closed, ხოლო ჩართული მდგომარეობებია – Unassigned, Assigned, Cancelled (კურსი გაუქმდა), Full (კურსი შევსებულია) და Committed (კურსი ჩართულია განრიგში). კომპოზიციურ მდგომარეობები საშუალებას იძლევიან გავამარტივოთ დიაგრამა, შევამციროთ რა გადასვლების რაოდენობა, რამდენადაც ჩართული მდგომარეობები მემკვიდრეობით იძენენ კომპოზიციური მდგომარეობების ყველა თვისებებსა და გადასვლებს.

ისტორიული მდგომარეობა – ეს პსევდომდგომარეობაა, რომელიც აღადგენს წინამდებარე აქტიურ მდგომარეობას კომპოზიტურ მდგომარეობაში. იგი საშუალებას აძლევს კომპოზიტურ მდგომარეობას Open დაიმახსოვროს რომელი ჩართული მდგომარეობა (Unassigned ან Assigned) იყო მიმდინარე Open – დან გამოსვლის მომენტში, იმისათვის რომ ნებისმიერი გადასვლა პენ – ში (ადდ სტუდენტ ან რემოვე სტუდენტ) ბრუნდებოდეს სწორედ ამ ჩართულ მდგომარეობაში, და არა საწყის მდგომარეობაში.

დიაგრამის გადატვირთვის თავიდან ასაცილებლად, ქვემოთ მოყვანილია იმ მოვლენების აღწერა, რომლებიც განაპირობებენ ერთი მდგომარეობიდან გადასვლას მეორეში:

X1 – სტუდენტების რაოდენობა=0;

X2 - რეგისტრაციის დახურვა;

X3 – დახურვა;

X4 - პროფესორის დამატება;

X5 - პროფესორის ამოგდება;

X6 – დამთავრება;

X7 – დახურვა[სტუდენტების რაოდენობა<3]; X8 – სტუდენტების რაოდენობა =10;

X9 – რეგისტრაციის დახურვა[სტუდენტების რაოდენობა>=3]; X10 – დაიხურა {სტუდენტების რაოდენობა>=3};

X11 – სტუდენტის დამატება /სტუდენტის–რაოდენობა=სტუდენტის–რაოდენობა+1;

X12 – სტუდენტის დამატება /სტუდენტის–რაოდენობა=სტუდენტის–რაოდენობა+1;

X13 - რეგისტრაციის დახურვა.

თავი 5
სისტემის რეალიზება

ავტომატიზებული სისტემის პროექტირებისას საჭირო ხდება განვიხილოთ როგორც ლოგიკური, ისე მისი ფიზიკური ასპექტები. ლოგიკურ ელემენტებს განეკუთვნებიან ისეთი არსები, როგორიც არის კლასები, ინტერფეისები, კოოპერაციები, ურთიერთქმედებები და ავტომატები, ხოლო ფიზიკურს – კომპონენტები (ლოგიკური არსების ფიზიკური დაჯგუფება) და კვანძები (აპარატურა, რომელზედაც განლაგდებიან და სრულდებიან კომპონენტები).

ზემოთ განხილული ყველა დიაგრამები გამოსახავდნენ სისტემის მოდელის აგების კონცეპტუალურ ასპექტებს და მიეკუთვნებოდნენ წარმოდგენის ლოგიკურ დონეს. ლოგიკური წარმოდგენის დამახასიათებელი ნიშანია ის, რომ იგი ოპერირებს ცნებებით, რომლებსაც არა აქვთ დამოუკიდებელი მატერიალური განხორციელება. სხვა სიტყვებით, ლოგიკური წარმოდგენის სხვადასხვა ელემენტები, როგორიც არის კლასები, ასოციაცია, მდგომარეობა, შეტყობინება, არ არსებობენ მატერიალურად ან ფიზიკურად. ისინი მხოლოდ ასახავენ ფიზიკური სისტემის სტრუქტურის გაგებას ან მისი ქცევის ასპექტებს მომხმარებლის თვალთახედვით.

ლოგიკური წარმოდგენის ძირითადი დანიშნულება მდგომარეობს იმაში, რომ მოახდინონ სისტემის მოდელის ელემენტებს შორის სტრუქტურული და ფუნქციონალური მიმართებების ანალიზი. მაგრამ კონკრეტული ფიზიკური სისტემის შესაქმნელად აუცილებელია რაიმე საშუალებებით მოხდეს ლოგიკური წარმოდგენის ყველა ელემენტების რეალიზება კონკრეტულ მატერიალურ არსებში.

5.1. რეალიზების სამუშაო ნაკადი

რეალიზების სამუშაო ნაკადში სამუშაო ობიექტ-ორიენტირებული ანალიტიკოსისათვის ან დამპროექტებელისათვის საკმაოდ მცირეა. მაგრამ არის საკითხები, რომლებიც მოითხოვენ ყურადღებით განხილვას.

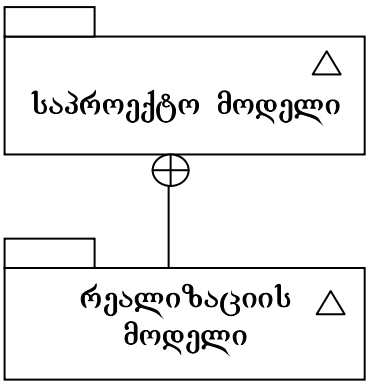
რეალიზაციის სამუშაო ნაკადი იწყება დაზუსტების ფაზაში და წარმოადგენს აგების ფაზის ძირითად ნაკადს. რეალიზაცია შედგება საპროექტო მოდელის გარდაქმნაში შესრულებად კოდში. ანალიტიკოსის ან დამპროექტებლის თვალსაზრისით რეალიზაციის მიზანია – რეალიზაციის მოდელის წარმოება. ეს მოდელი მოიცავს საპროექტო კლასების განაწილებას კომპონენტების მიხედვით. ეს როგორ ხორციელდება უმეტესად დამოკიდებულია დაპროგრამების კონკრეტულ ენაზე.

რეალიზაციის პროცესში ძირითადი ყურადღება მიმართულია შესრულებადი კოდის წარმოებაზე. რეალიზაციის მოდელის შექმნა შეიძლება იყოს გვერდითი პროდუქტი ამ პროცესის, მაგრამ მოდელირების არა აშკარა მოღვაწეობა. ჩვეულებრივ მოდელირების მრავალი ინსტრუმენტალური საშუალება იძლევა საშუალებას შევქმნათ რეალიზაციის მოდელი საწყისი კოდიდან უკუ დაპროექტების საშუალებით. ეს საშუალებას აძლევს პროგრამისტებს ეფექტურად მოახდინონ რეალიზაციის მოდელირება.

მაგრამ არის ორი შემთხვევა, როდესაც ძალიან მნიშვნელოვანია, რომ გამოცდილმა ანალიტიკოსებმა ან დამპროექტებლებმა ჩაატარონ რეალიზაციის მოდელირება.

- როდესაც გათვალისწინებულია კოდის პირდაპირი გენერირება მოდელიდან, საჭირო იქნება განვსაზღვროთ ისეთი დეტალები, როგორიც არის საწყისი ფაილები და კომპონენტები.
- თუ ხორციელდება კომპონენტურ-ორიენტირებული დამუშავება კომპონენტების ხელმეორედ გამოყენების მიზნით, საპროექტო კლასების და ინტერფეისების განაწილება კომპონენტების მიხედვით ხდება სტრატეგიულად მნიშვნელოვანი საკითხი. ალბათ, თქვენ მოგინდებათ პირველად პროექტის ამ ნაწილის მოდელირება, და არა ყველაფერი დავაკისროთ პროგრამისტებს.

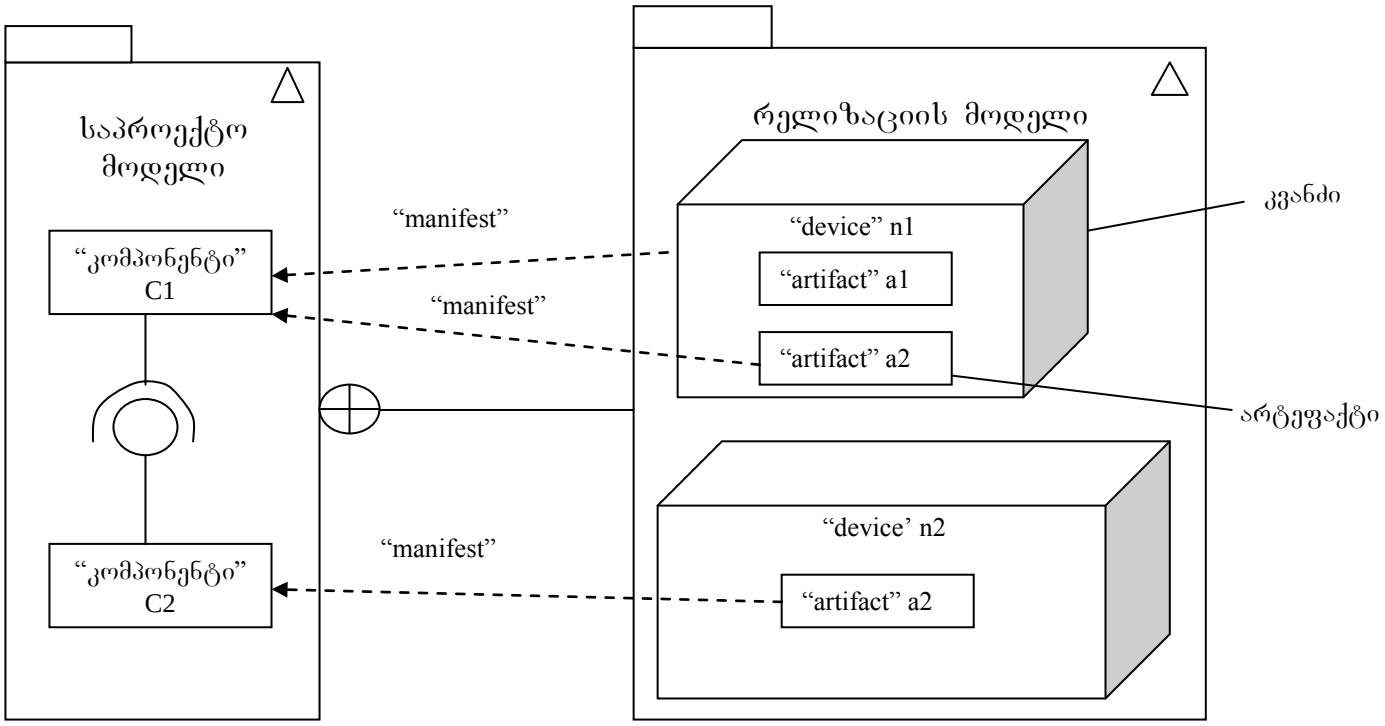
დამოკიდებულება რეალიზაციის მოდელსა და საპროექტო მოდელს შორის ძალინ მარტივია. ფაქტიურად რეალიზაციის მოდელი – ეს საპროექტო მოდელის წარმოდგენაა რეალიზაციის თვალთახედვით. სხვა სიტყვებით, რეალიზაციის მოდელი – ეს საპროექტო მოდელის ნაწილია, რომელიც დაკავებულია რეალიზაციის საკითხებით.



ნახ.5.1.

იგი განსაზღვრავს, თუ საპროექტო ელემენტები როგორ წარმოიდგინებიან არტეფაქტებით და ეს არტეფაქტები როგორ განლაგდებიან კვანძებზე. არტეფაქტები წარმოადგენენ რეალური არსების აღწერებს, როგორიც არის საწყისი ფაილები, ხოლო კვანძები წარმოადგენენ მოწყობილობების აღწერას ან შესრულების გარემოს, რომელშიც

ეს არსები განლაგდებიან. მიმართებები საპროექტო მედელსა და რეალიზაციის მოდელს შორის მოყვანილია ნახ.5.2.



ნახ.5.2.

მიმართება “manifest” არტეფაცტებსა და კომპონენტებს შორის მიუთითებს იმაზე, რომ არტეფაცტები წარმოადგენენ კომპონენტების ფიზიკურ წარმოდგენებს. მაგალითად, კომპონენტი შესაძლებელია შედგებოდეს კლასისა და ინტერფეისისგან, რომლებიც რეალიზებული არიან ერთი არტეფაქტით: ფაილით, რომელიც შეიცავს საწყის კოდს.

საპროექტო კომპონენტები – ეს ლოგიკური არსებია, რომლებიც აჯგუფებენ საპროექტო ელემენტებს. ხოლო რეალიზაციის არტეფაქტები პროეცირდებიან რეალურებზე., რეალიზაციის მიზნობრივი ენის დაჯგუფების ფიზიკური მექანიზმები.

ძირითადი არტეფაქტი რეალიზაციის სამუშაო ნაკადის ობიექტ- ორიენტირებული ანალიტიკოსისა ან დამპროექტებლის თვალსაზრისით – რეალიზაციის მოდელია. ეს მოდელი შედგება კომპონენტების დიაგრამისაგან, რომლებიც გვიჩვენებენ, არტეფაქტები როგორ წარმოადგენენ კომპონენტებს, და ახალი ტიპის დიაგრამისაგან – განლაგების დიაგრამისაგან. განლაგების დიაგრამა ახდენს ფიზიკური გამოთვლითი კვანძების მოდელირებას, რომლებზეც განლაგდებიან პროგრამული არტეფაქტები, და მიმართებები ამ კვანძებს შორის.

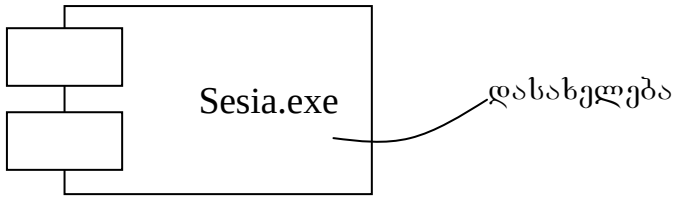
5.2. კომპონენტები. კომპონენტების დიაგრამა

მაშასადამე, სისტემის სრული პროგრამული პროექტი შესდგება ლოგიკური და ფიზიკური წარმოდგენების ერთობლიობისაგან, რომლებიც ერთმანეთთან შეთანხმებული უნდა იყოს.

ფიზიკური არსების მოდელირებისათვის გამოიყენებიან კომპონენტები. მათ მიეკუთვნებიან შესრულებადი მოდულები, ბიბლიოთეკები, ცხრილები, ფაილები და დოკუმენტები.

ხიდს ლოგიკურ და ფიზიკურ მოდელებს შორის ქმნიან ინტერფეისები. კომპონენტი ახდენს გარკვეული ინტერფეისების რეალიზებას. შესაბამისად, ლოგიკურ მოდელში სპეციფიცირებული რომელიმე კლასის ინტერფეისი, შემდეგ რეალიზებული უნდა იქნას გარკვეული კომპონენტით.

გრაფიკულად კომპონენტი გამოისახება შემდეგი სახით

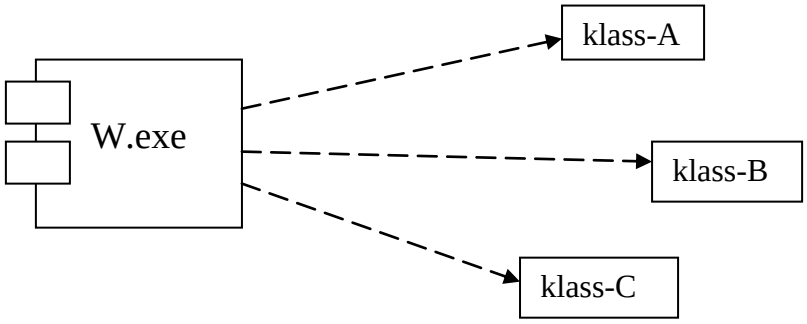


ნახ.5.3.

ბევრი მიმართებით კომპონენტები მსგავსია კლასების. ერთსაც და მეორესაც გააჩნია სახელი, შეუძლიათ ინტერფეისების რეალიზება, შევიდნენ დამოკიდებულების, განზოგადების და ასოციაციის მიმართებებში, მიიღონ მონაწილეობა ურთიერთქმედებაში. მაგრამ კომპონენტებსა და კლასებს შორის არის არსებითი განსხვავებაც. კერძოდ, კლასები წარმოადგენენ ლოგიკურ აბსტრაქციებს, ხოლო კომპონენტები – ფიზიკურ არსებს. ეს კი მიუთითებს, რომ კომპონენტებსა და კლასებს შორის არსებობს გარკვეული მიმართება, კერძოდ კომპონენტები ახდენენ კლასების რეალიზებას. კავშირს კომპონენტებსა და მათში რეალიზებულ კლასებს შორის წარმოადგენენ დამოკიდებულების მიმართებით. ასეთ ინფორმაციას აქვს დიდი მნიშვნელობა ლოგიკური და ფიზიკური წარმოდგენების შეთანხმებისათვის. ნახ.5.4.-ზე მოყვანილია ასეთი დამოკიდებულების ფრაგმენტი, როდესაც გარკვეული კომპონენტი დამოკიდებულია შესაბამისი კლასებისაგან. ცხადია, ცვლილებებმა კლასების აღწერის სტრუქტურაში შეიძლება გამოიწვიოს კომპონენტის შეცვლა.

მოყვანილი განმარტებიდან გამომდინარე კომპონენტები შესაძლებელია განლაგდნენ კვანძებში, ხოლო კლასები – არა. კლასებს შეიძლება გააჩნდეთ ატრიბუტები და

ოპერაციები. კომპონენტები ფლობენ მხოლოდ ოპერაციებს, რომლებიც მისაღწევია მათი ინტერფეისებიდან.

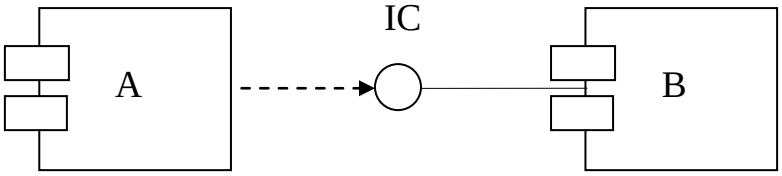


ნახ.5.4.

ინტერფეისი, რომელიც რეალიზდება კომპონენტით, უწოდებენ ექსპორტირებულ ინტერფეისს. ეს ნიშნავს, რომ კომპონენტი მოცემული ინტერფეისიდან წარუდგენს რიგ მომსახურებებს სხვა კომპონენტებს. კომპონენტს შეუძლია ბევრი ინტერფეისების ექსპორტირება. ინტერფეისი, რომლითაც კომპონენტი სარგებლობს უწოდებენ იმპორტირებულს. ეს ნიშნავს, რომ კომპონენტი შეთავსებულია ინტერფეისთან და დამოკიდებულია მისგან თავისი ფუნქციების შესრულებისას. კომპონენტს შეუძლია სხვადასხვა ინტერფეისების იმპორტირება, ამასთან მისთვის დასაშვებია ერთდროულად ექსპორტირება და იმპორტირება.

კონკრეტული ინტერფეისი შეიძლება ექსპორტირებული იყოს ერთი კომპონენტით და იმპორტირებული მეორეს მიერ. თუ ორ კომპონენტს შორის განლაგებულია ინტერფეისი, მათი უშუალო ურთიერთდამოკიდებულება იყოფა. კომპონენტი, რომელიც იყენებს მოცემულ ინტერფეისს, იფუნქციონირებს კორექტულად (განურჩევლად იმისა, რომელი კომპონენტით იქნება რეალიზებული ინტერფეისი). ცხადია, კომპონენტი შეიძლება გამოყენებულ იქნას გარკვეულ კონტექსტში მხოლოდ მაშინ, როდესაც ყველა მისი იმპორტირებული ინტერფეისები ექსპორტირებულნი არიან რომელიღაც სხვა კომპონენტებით.

ნახ.4.5.-ზე მოყვანილია კომპონენტები, რომელთაგან კომპონენტი ახდენს IC



ნახ.5.5.

ინტერფეისის რეალიზებას (ექსპორტირებას), ხოლო კომპონენტი A ახდენს მის იმპორტირებას (უკავშირდება მას დამოკიდებულების მიმართებით).

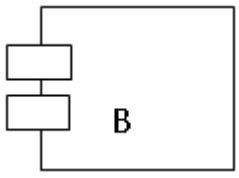
სისტემა შესაძლებელია შეიქმნას კომპონენტებისაგან, ხოლო შემდეგ განვავითაროთ ის, დავამატოთ რა ახალი კომპონენტები ან შევცვალოთ ძველი – დამატებითი კომპილაციის გარეშე. ამის მიღწევა შესაძლებელია ინტერფეისების მეშვეობით. დადგინდება რა ინტერფეისი შესაძლებელია ჩაისვას მუშა სისტემაში მუშა კომპონენტი, რომელიც შეთავსებულია ამ ინტერფეისთან ან წარმოადგენს მას. სისტემა შესაძლებელია გაფართოვდეს ახალი კომპონენტების ჩართვით, რომლებიც უზრუნველყოფენ ახალ მომსახურებას დამატებითი ინტერფეისების მეშვეობით, ასევე კომპონენტებით, რომლებსაც უნარი აქვთ გამოიცნონ და გამოიყენონ ეს ახალი ინტერფეისები. ასეთი სემანტიკიდან გამომდინარე, კომპონენტი ეს სისტემის ფიზიკური შესაცვლელად დასაშვები ნაწილია, რომელიც შეთავსებულია ერთ ინტერფეისთან და რეალიზებას უწევს სხვებს.

გამოყოფენ სამი სახის კომპონენტებს:

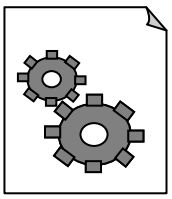
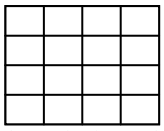
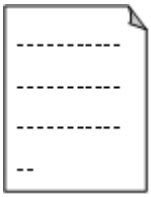
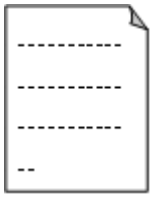
1. **განლაგების კომპონენტები**, რომლებიც აუცილებელია და საკმარისი შესრულებადი სისტემის აგებისათვის. მათ რიცხვს მიეკუთვნება დინამიურად დამაკავშირებელი ბიბლიოთეკები (**DLL**) და შესრულებადი პროგრამები (**EXE**).
2. **კომპონენტები – მუშა პროდუქტები**. ეს დამუშავების პროცესის გვერდითი შედეგია. მას შეიძლება მივაკუთვნოთ ფაილები პროგრამის საწყისი ტექსტებით და მონაცემებით, რომლებისგანაც იქმნება განლაგების კომპონენტები. ასეთი კომპონენტები არ ღებულობენ უშუალო მონაწილეობას შესრულებადი სისტემის მუშაობაში, მაგრამ წარმოადგენენ სამუშაო პროდუქტს, რომლებისგანაც იქმნება შესრულებადი სისტემა.
3. **შესრულების კომპონენტები**. ისინი იქმნებიან როგორც სისტემის მუშაობის შედეგი.

კომპონენტები შესაძლებელია დაჯგუფებულ იქნას პაკეტში და მათ შორის შესაძლებელია დამოკიდებულების, განზოგადების, ასოციაციის და რეალიზაციის მიმართებები.

მოდელირების უნიფიცირებულ ენაში**(UML)** განსაზღვრულია ხუთი სტანდარტული სტერეოტიპი კომპონენტებთან მიმართებაში და თვითეულს შეესაბამება თავისი გრაფიკული გამოსახვა:

- **executable**(შესრულებადი) 

– განსაზღვრავს კომპონენტს, რომელიც შესაძლებელია შესრულდეს კვანძში.

- **Library**(ბიბლიოთეკა)  – განსაზღვრავს სტატიკურ ან დინამიურ ობიექტურ ბიბლიოთეკას;
- **table**(ცხრილი)  - განსაზღვრავს კომპონენტს, რომელიც წარმოადგენს მონაცემთა ბაზის ცხრილს.
- **file**(ფაილი)  - განსაზღვრავს კომპონენტს, რომელიც წარმოადგენს დოკუმენტს, საწყისი ტექსტით და მონაცემებით.
- **document**(დოკუმენტი)  - განსაზღვრავს კომპონენტს, რომელიც წარმოადგენს დოკუმენტს.

დამუშავებული კომპონენტების ორგანიზაციისა და მათ შორის დამოკიდებულებების ასახვისათვის გამოიყენება კომპონენტების დიაგრამა.

აქ შედის კვანძზე განლაგებული ფიზიკური არსების, მაგალითად პროგრამების, ბიბლიოთეკების, ცხრილების, ფაილების და დოკუმენტების მოდელირება. არსებითად, კომპონენტების დიაგრამა – ეს კლასების დიაგრამაა, ფოკუსირებული სისტემურ კომპონენტებზე.

გრაფიკულად კომპონენტების დიაგრამა ჩვეულებრივ შეიცავს კომპონენტებს, ინტერფეისებს და მიმართებებს მათ შორის (დამოკიდებულების, განზოგადების, ასოციაციის და რეალიზაციის).

<u>კომპონენტების ფორმირება და მათი გამოყენება სისტემის რეალიზებისათვის.</u>

თუ სისტემა შედგება ერთი შესრულებადი ფაილისაგან, კომპონენტების მოდელირება საჭირო არ არის. ხოლო თუ სისტემა შედგება რამოდენიმე კომპონენტისაგან და მასთან ასოცირებული ობიექტური ბიბლიოთეკებისაგან, მაშინ კომპონენტების მოდელირება დაგვეხმარება ფიზიკური სისტემის აწყობისათვის.

ზემოთ მოყვანილი დაყოფიდან გამომდინარე, სისტემის რეალიზება გულისხმობს განლაგების, მუშა პროდუქტების და შესრულების კომპონენტების ფორმირებას.

ყველაზე ხშირად კომპონენტები გამოიყენებიან განლაგების კომპონენტების მოდელირებისათვის, რომლებიც შეადგენენ სისტემის რეალიზებას. განლაგების

კომპონენტების ფორმირებისათვის თავდაპირველად განისაზღვრება ოპერაციული სისტემის გარემო(ჭინდოწს, ინუხ) და შეირჩევა დაპროგრამების ენა, რომლის გარემოშიც უნდა მოხდეს კოდების აგება.

პროგრამული კოდების შემდეგ მუშავდება მონაცემთა ბაზის, მისი ცხრილებისა და ინდექსური ფაილების კომპონენტები. მონაცემთა ბაზების მართვის სისტემები (მბმს) ფართოდ გავრცელებული დაპროგრამების სისტემებია, რომელთა დანიშნულებას ინფორმაციის მიღება, შენახვა და კომპიუტერის საშუალებით დამუშავება წარმოადგენს. მონაცემთა ბაზების მართვის ზოგიერთი თანამედროვე სისტემა მონაცემთა ბაზების შექმნის ძლიერი მექანიზმებითაა აღჭურვილი. მათ განეკუთვნება მონაცემთა ბაზების დაპროექტების ავტომატიზებული ინსტრუმენტული საშუალებები მრავალფანჯრიანი რეჟიმითა და შეტანილი ინფორმაციის სინტაქსური კონტროლის გათვალისწინებით.

დამუშავებული კომპონენტები ერთიანდება კონტექსტურ დიაგრამებში და მზადდება სერვერ-მონაცემთა ბაზებში განსათავსებლად. ცენტრალურ სერვერ-ბაზასთან ერთად გათვალისწინებული უნდა იყოს სარეზერვო, არქივირებული ბაზის არსებობა. სერვერ-ბაზების ადმინისტრირების რეგლამენტი უნდა შემუშავებული იქნას სისტემის დანერგვის შემდეგ, გამომდინარე ფუნქციური ავტომატიზებული სამუშაო ადგილების რაოდენობისა და მენეჯმენტის მიზნებიდან.

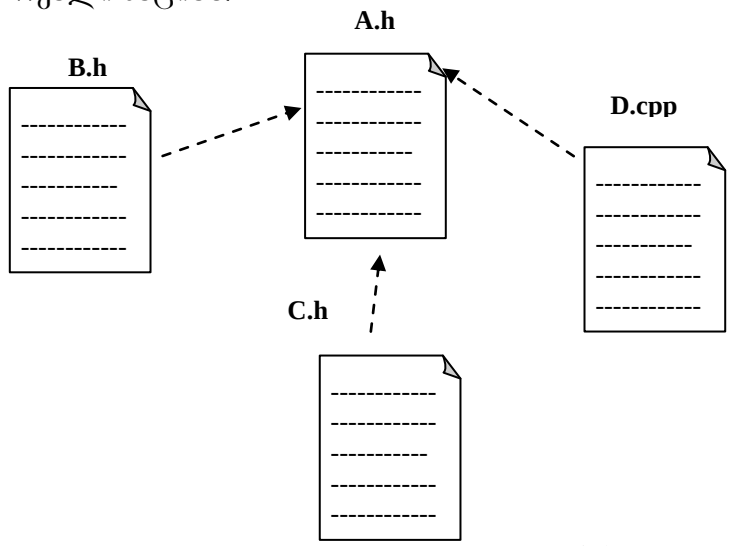
გარდა სპეციალური პროგრამული და საბაზო ფაილების დამუშავებისა, ფართოდ გამოიყენება სტანდარტული და სერვისული ბიბლიოთეკების პროგრამები, რაც ხორციელდება ობიექტ-ორიენტირებული და სტრუქტურული დაპროგრამების პრინციპებით.

კომპონენტების ფორმირება აუცილებლად ითვალისწინებს მათ ტესტირებას და საბოლოო მუშა სახემდე მიყვანას, აგრეთვე დოკუმენტირებისა და ინსტრუქციების მომზადებას მომხმარებლებისათვის, სისტემისა და მონაცემთა ბაზების ადმინისტრატორებისათვის.

კომპონენტების დიაგრამა შესაძლებელია გამოვიყენოთ სისტემის რეალიზების სხვადასხვა დონეზე, პირველ რიგში საწყისი კოდის მოდელირებისათვის. უმრავლესობა ობიექტ- ორიენტირებულ პროგრამირების ენებში კოდი იწერება დამუშავების ინტეგრირებულ გარემოში, რომლებიც ინახავენ საწყის ტექსტებს ფაილებში. კომპონენტების დიაგრამები შესაძლებელია გამოვიყენოთ ამ ფაილების კონფიგურაციის მართვის მოდელირებისათვის, რომლებიც წარმოადგენენ კომპონენტებს – სამუშაო პროდუქტებს.

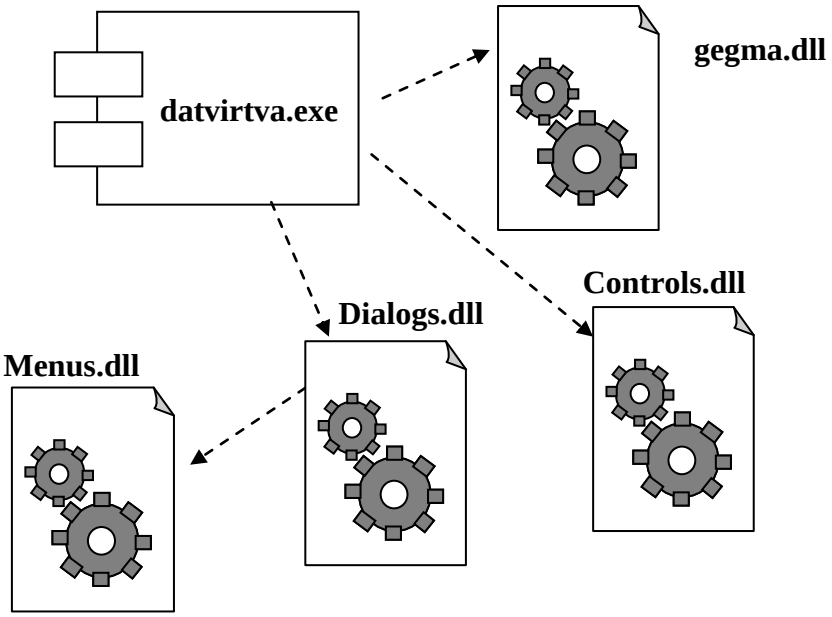
ნახ.5.6.-ზე ნაჩვენებია საწყისი ფაილები, რომლებიდანაც იქმნება ბიბლიოთეკა A.dll. ნახაზზე ჩანს სამი სათაო ფაილები(A.h, B.h, C.h), რომლებიც წარმოადგენენ გარკვეული

კლასების სპეციფიკაციებს, ასევე ფაილი D.cpp, რომელიც წარმოადგენს ერთ-ერთი კლასის რეალიზაციას.



ნახ.5.6.

ნახ. 5.7.-ზე მოყვანილია კომპონენტების ნაკრები სასწავლო პროცესის ორგანიზებისათვის. დიაგრამა შედგება ერთი შესრულებადი პროგრამისაგან (დატვირტვა.ეხე) და ბიბლიოთეკებისაგან (გეგმა, ჩონტროლს, ენუს, იალოგს). კომპონენტების გამოსახვისათვის გამოყენებულია სტანდარტული ელემენტები შესრულებადი პროგრამებისა და ბიბლიოთეკებისათვის.

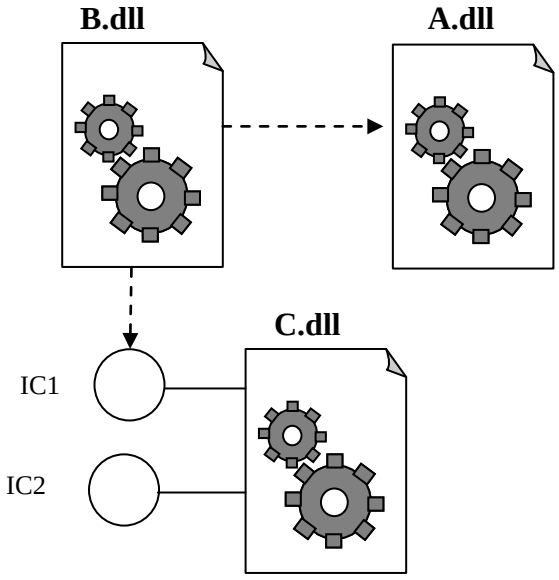


ნახ.5.7.

კომპონენტების მოდელირების მნიშვნელობა კიდევ უფრო იზრდება, თუ სისტემის განვითარებასთან ერთად საჭირო ხდება მისი სხვადასხვა ნაწილების კონფიგურაციის მართვა და ვერსიების კონტროლი. ვერსია – ეს შედარებით სრული და შეთანხმებული არტეფაქტების ნაკრებია, რომელიც წარედგინება შიდა ან გარე მომხმარებელს.

სისტემისათვის, რომელიც შედგენილია კომპონენტებისაგან, ვერსია უპირველეს ყოვლისა გულისხმობს იმ ნაწილებს, რომლებიც აუცილებელია დავაყენოთ მუშა სისტემის მისაღებათ. ვერსიების მოდელირებისას კომპონენტების დიაგრამის მეშვეობით ხდება გადაწყვეტილებების ვიზუალირება, რომლებიც მიიღება სისტემის ფიზიკური შემადგენლების, ესე იგი კომპონენტების განლაგების მიმართ.

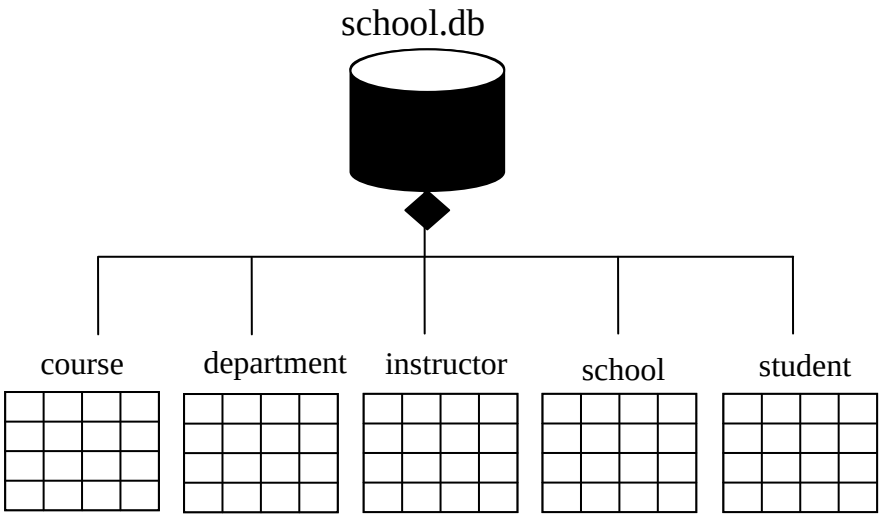
ნახ.5.8.-ზე მოყვანილია შესრულებადი ვერსიის ნაწილის მოდელი. ნახაზიდან ჩანს ერთი კომპონენტი A.dll, რომელიც ახდენს ინტერფეისის IC1 ექსპორტირებას, რომლის იმპორტსაც ახდენს სხვა კომპონენტი B.dll. C.dll ახდენს კიდევ ერთი ინტერფეისის ექსპორტირებას IC2, რომელიც შესაძლებელია გამოყენებულ იქნას სხვა კომპონენტების მიერ სისტემაში. დიაგრამაზე ნაჩვენებია კიდევ ერთი კომპონენტი .დლლ, რომელიც ასევე ახდენს ინტერფეისებს, თუმცა მათი დეტალები დამალულია, ნაჩვენებია მხოლოდ დამოკიდებულება B.dll –სა A.dll-გან.



ნახ.5.8.

მნიშვნელოვანია კომპონენტების დიაგრამის როლი მონაცემთა ბაზის ფიზიკური მოდელირებისას. მონაცემთა ბაზის ლოგიკური სქემა აღწერს შენახული მონაცემების ლექსიკონს, ასევე მათ შორის კავშირის სემანტიკას. ფიზიკურად ყველაფერი ეს ინახება მონაცემთა ბაზაში. ლოგიკური სქემის ასახვა ობიექტ-ორიენტირებულ მონაცემთა ბაზაზე სიძნელეს არ წარმოადგენს, შედარებით რთულია ასახვა რელაციურზე. სიძნელე განპირობებულია იმით თუ როგორ გამოვსახოთ კლასები ცხრილების მეშვეობით და ოპერაციები, განსაზღვრულნი ლოგიკურ სქემაზე. ამ დროს ხელმძღვანელობენ შემდეგი მოსაზრებით - უბრალო ოპერაციები შექმნა, განახლება და ამოგდება რეალიზდება SQL და DBC სტანდარტული საშუალებებით, ხოლო უფრო რთული ქცევა (მაგალითად, ბიზნეს წესები) გამოისახებიან ტრიგერებზე და შესანახ პროცედურებზე.

5.9. ნახაზზე ნაჩვენებია მონაცემთა ბაზის რამოდენიმე ცხრილი, აღებული უმაღლესი სასწავლებლის საინფორმაციო სისტემიდან. მასზე ასახულია ერთი ბაზა (გამოსახული კომპონენტით სტერეოტიპით database), რომელიც შედგება ხუთი ცხრილისაგან (გამოსახულნი კომპონენტების სახით სტერეოტიპით interface).



ნახ.5.9.

ვიზუალირებისას კომპონენტების დიაგრამაზე კომპონენტები გამოისახებიან ცხრილის სტერეოტიპით და შესაძლებელია უჩვენოთ თითოეული ცხრილის შედგენილობა, თუმცა ეს მოცემულ მაგალითში ნაჩვენები არ არის. კომპონენტებს შესაძლებელია გააჩნდეთ ატრიბუტები, ამიტომ ფიზიკური მონაცემთა ბაზის მოდელირებისას ფართოდ გამოიყენება ატრიბუტები ცხრილის თითოეული სვეტის აღწერისათვის. ასევე კომპონენტებს შესაძლებელია ქონდეთ ოპერაციები, რომლითაც შეიძლება ვისარგებლოთ შესანახი პროცედურების აღნიშვნისათვის.

ბოლოს, კომპონენტების დიაგრამით შესაძლებელია ადაპტური სისტემების მოდელირება. ზოგიერთი სისტემები აბსოლუტურათ სტატიკურნი არიან - მათი კომპონენტები ჩნდებიან, ღებულობენ მონაწილეობას შესრულებაში, ხოლო შემდეგ ქრებიან სცენიდან. სხვები უფრო დინამიურნი არიან. ისინი შეიცავენ კომპონენტებს, რომლებიც მიგრაციას განიცდიან. ასეთი სისტემების წარმოდგენისათვის გამოიყენება კომპონენტების დიაგრამა.

დიდი სისტემებისათვის, რომლებიც რამოდენიმე კომპიუტერზე იმართება, საჭირო ხდება კომპონენტების განაწილების საშუალებების მოდელირება, დაუნიშნოთ რა თითოეულს კვანძი, რომელზედაც ის განლაგდება.

5.3. განლაგება. განლაგების დიაგრამა

კომპონენტები, რომელსაც ვიყენებთ, უნდა განლაგდნენ რომელიმე აპარატურაზე(კვანძზე), წინააღმდეგ შემთხვევაში ისინი ვერ შესრულდებიან. ავტომატიზებული სისტემა ამიტომაც შედგება პროგრამული და აპარატული უზრუნველყოფისაგან.

არტეფაქტები. არტეფაქტი წარმოადგენს რეალური არსის აღწერას, მაგალითად, როგორიც არის ფაილი. არტეფაქტების მაგალითებია:

- საწყისი ფაილები;
- შესრულებადი ფაილები;
- სცენარები;
- მონაცემთა ბაზის ცხრილები;
- დოკუმენტები;
- დამუშავების პროცესის შედეგები, მაგალითად UML-მოდელი.

არტეფაქტის ეგზემპლიარი წარმოადგენს კონკრეტული არტეფაქტის კონკრეტულ ეგზემპლიარს. არტეფაქტის ეგზემპლიარები განლაგდებიან კვანძების ეგზემპლიარებზე. არტეფაქტებს შეუძლიათ წარმოადგინონ ერთი ან მეტი კომპონენტები. არტეფაქტებს შეუძლიათ უზრუნველყონ *ნებისმიერი* ტიპის UML-ელემენტის ფიზიკური წარმოდგენა. ჩვეულებრივ ისინი წარმოადგენენ ერთ ან რამოდენიმე კომპონენტებს

UML-ში განლაგება - ეს არტეფაქტების კვანძების მიხედვით ან არტეფაქტების ეგზემპლიარების კვანძების ეგზემპლიარების მიხედვით განაწილებაა. განლაგების დიაგრამა განსაზღვრავს ფიზიკურ მოწყობილობას, რომელზედაც შესრულდება პროგრამული სისტემა, ასევე აღწერს, თუ როგორ განლაგდება პროგრამული უზრუნველყოფა ამ მოწყობილობაზე.

განლაგების დიაგრამის ორი ფორმა არსებობს:

1. დესკრიფტორული ფორმა – შეიცავს კვანძებს, მიმართებებს კვანძებს შორის და არტეფაქტებს. კვანძი წარმოადგენს მოწყობილობის ტიპს (მაგ. პკ). ანალოგიურად არტეფაქტი წარმოადგენს ფიზიკური პროგრამული არტეფაქტის ტიპს., მაგ. Java JAR- ფაილი.
2. ეგზემპლიარული ფორმა – შეიცავს კვანძების ეგზემპლიარებს, მიმართებებს კვანძების ეგზემპლიარებსა და არტეფაქტების ეგზემპლიარებს. კვანძების ეგზემპლიარები წარმოადგენენ კონკრეტულ, მოწყობილობის იდენტიფიცირებულ ნაწილს.(მაგ. ჯიმის პკ). არტეფაქტის ეგზემპლიარი წარმოადგენს პროგრამული

უზრუნველყოფის კონკრეტული ეგზემპლიარის ტიპს. მაგ. კონკრეტული JAR-ფაილი.

მართალია განლაგების დიაგრამა განიხილება როგორც რეალიზაციის სამუშაო ნაკადი, დაპროექტებისას, მისი პირველი მიახლოება ხშირად იქმნება როგორც აპარატული მოწყობილობის არჩევის პროცესი. შესაძლებელია დავიწყოთ განლაგების დიაგრამის დესკრიფტორული ფორმიდან, შემოვისაზღვროთ კვანძებითა და მათ შორის კავშირებით, ხოლო შემდეგ დავაზუსტოდ იგი და გადავაქციოთ ერთ ან რამოდენიმე ეგზემპლიარულ ფორმაში. როდესაც გახდება ცნობილი უფრო მეტი მოწყობილობის საიტის შესახებ, რომელზედაც განლაგდება პროექტი, აუცილებლობის შემთხვევაში შესაძლებელია შეიქმნას განლაგების დიაგრამის ეგზემპლიარული ფორმა, რომელიც გვიჩვენებს ამ საიტზე ფაქტიურად გამოყენებულ კომპიუტერებსა და მოწყობილობებს.

მაშასადამე, განლაგების დიაგრამის შექმნა – ეს არის პროცესი ორი ეტაპისაგან:

1. დაპროექტების სამუშაო ნაკადში ძირითადი ყურადღება ეთმობა კვანძებს ან კვანძების ეგზემპლიარებს და კავშირებს მათ შორის.
2. რეალიზაციის სამუშაო ნაკადში – არტეფაქტების ეგზემპლიარების კვანძების ეგზემპლიარების მიხედვით (ეგზემპლიარული ფორმა) ან არტეფაქტების კვანძების მიხედვით განაწილებას (დესკრიფტორული ფორმა).

კვანძები. კვანძი წარმოადგენს გამოთვლითი რესურსის ტიპს. UML2-ის სპეციფიკაციით – კვანძი წარმოადგენს გამოთვლითი რესურსის ტიპს, რომელზედაც განლაგდებიან არტეფაქტები შესრულებისათვის. არსებობს ორი სტერეოტიპი კვანძებისათვის:

- “device” (მოწყობილობა) – კვანძი წარმოადგენს გამოთვლითი მოწყობილობის ტიპს, მაგალითად პკ ან სერვერი Fire Sun კორპორაციის.
- “execution environment” (შესრულების გარემო) – კვანძი წარმოადგენს პროგრამული უზრუნველყოფის შესრულების გარემოს ტიპს. მაგალითად ვებ-სერვერი Apache .

ასოციაცია კვანძებს შორის წარმოადგენს კავშირის არხს, რომლითაც შესაძლებელია ინფორმაციის გადაცემა ორივე მიმართულებით

კვანძის ეგზემპლიარი წარმოადგენს კონკრეტულ გამოთვლითი რესურსს. თუ საჭიროა უჩვენოთ კვანძების კონკრეტული ეგზემპლიარები, შესაძლებელია გამოვიყენოთ განლაგების დიაგრამის ეგზემპლიარული ფორმა. დიაგრამის ეგზემპლიარულ ფორმაში ეგზემპლიარები წარმოადგენენ რეალურ ფიზიკურ მოწყობილობებს ან გამოთვლის გარემოს რეალურ ეგზემპლიარებს, რომლებიც სრულდებიან ამ მოწყობილობებზე.

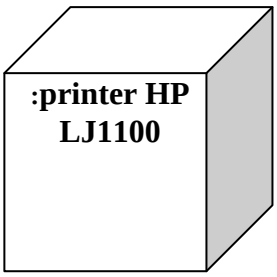
განლაგების დიაგრამის დესკრიფტორული ფორმა კარგია ფიზიკური არქიტექტურის ტიპის მოდელირებისათვის, ხოლო ეგზემპლიარული ფორმა – ამ არქიტექტურის ფაქტიური განლაგებისათვის კონკრეტულ საიტზე.

არტეფაქტი წარმოადგენს რეალური არსის აღწერას, მაგალითად ისეთის როგორიც არის ფაილი. არტეფაქტის ეგზემპლიარი წარმოადგენს კონკრეტული არტეფაქტის კონკრეტულ ეგზემპლიარს.

არტეფაქტებმა შესაძლებელია წარმოადგინონ ერთი ან რამოდენიმე კომპონენტი.

მაშასადამე, კვანძი – ფიზიკური ელემენტია, რომელიც არსებობს შესრულების დროს და წარმოადგენს გამოთვლით რესურს, რომელიც ჩვეულებრივ ფლობს როგორც მინიმუმ მეხსიერების გარკვეულ მოცულობას, ხოლო ხშირათ ასევე პროცესორსაც.

გრაფიკულად კვანძი გამოისახება კუბის სახით. ყოველ კვანძს გააჩნია სახელი, რომელიც წარმოადგენს ტექსტურ სტრიქონს(იხ.ნახ.5.10.).



ნახ.5.10.

ობიექტების ან კომპონენტების სიმრავლე, რომლებიც მიწერილია კვანძზე როგორც ჯგუფზე, უწოდებენ განაწილების ელემენტს. კვანძებს შორის ყველაზე გავრცელებული მიმართებაა ასოციაცია. მოცემულ კონტექსტში ასოციაცია წარმოადგენს კვანძების ფიზიკურ შეერთებას.

კვანძები ძირითადათ გამოიყენებიან:

- პროცესორებისა და მოწყობილობების მოდელირებისათვის, რომლებიც ქმნიან ავტონომიური, კლიენტ-სერვერული და განაწილებული სისტემების ტოპოლოგიას.
- სისტემაში შემავალი კომპონენტების ფიზიკური განაწილების ვიზუალირებისა და სპეციფიცირებისათვის პროცესორებისა და კვანძების მიხედვით.

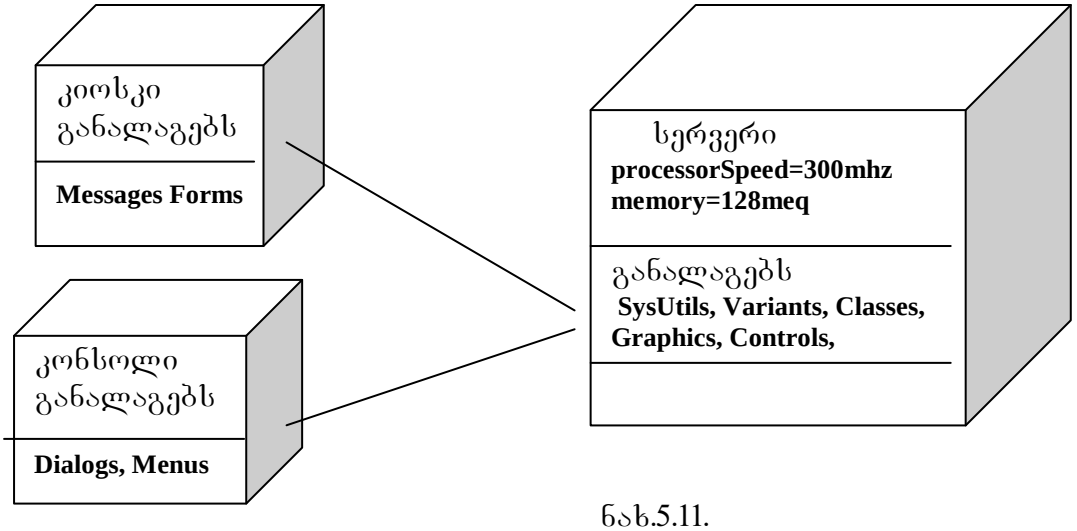
პროცესორი ეს კვანძია, რომელსაც შეუძლია მონაცემების დამუშავება, ანუ შეასრულოს კომპონენტი. მოწყობილობა ეს კვანძია, რომელსაც არ შეუძლია მონაცემების დამუშავება და საერთო ჯამში გამოიყენება რეალურ სამყაროსთან დაკავშირებული რაიმეს წარმოდგენისათვის.

პროცესორებისა და მოწყობილობების მოდელირებისას თავიდან ახდენენ გამოთვლითი ელემენტების იდენტიფიცირებას. თუ ეს ელემენტები წარმოადგენენ საერთო სახის პროცესორებსა და მოწყობილობებს მიუწერენ მათ შესაბამის სტანდარტულ სტერეოტიპებს. მაგრამ, თუ ეს პროცესორები და მოწყობილობები შედიან საპრობლემო

სფეროს ლექსიკონში, შეუთავსებენ მათ შესაბამის სტერეოტიპებს. ისევე როგორც კლასების მოდელირებისას, დადგინდება ატრიბუტები და ოპერაციები, გამოყენებული თითოეული კვანძისათვის.

სისტემების ტოპოლოგიის მოდელირებისას ხშირათ სასარგებლოა სისტემის შემადგენლობაში შემავალ პროცესორებსა და მოწყობილობებზე კომპონენტების ფიზიკური განაწილების ვიზუალირება. ეს განსაკუთრებით შეეხება განაწილებულ სისტემებს, რათა გამოირიცხოს კომპონენტების განლაგების დუბლირების შესაძლებლობა. საკმაოდ გავრცელებულია შემთხვევა, როდესაც ერთი და იგივე კომპონენტები განლაგებულნი არიან ერთდროულად რამოდენიმე კვანძზე. ამიტომ სასურველია მიეწეროს ყოველი მნიშვნელოვანი კომპონენტი შესაბამის კვანძს ან ჩამოითვალოს კომპონენტები, განლაგებულნი კვანძზე, დამატებით განყოფილებაში.

ნახ.5.11.-ზე წარმოდგენილია დიაგრამა, რომლებზეც მოყვანილია შესრულებადი პროგრამები განლაგებულნი კვანძების მიხედვით.



ნახ.5.11.

სერვერი – ეს კვანძია სტერეოტიპით პროცესორი საერთო დანიშნულებით, კიოსკი და კონსოლი – კვანძებია სტერეოტიპით სპეციალიზირებული პროცესორები. ყოველი პროცესორისათვის გამოყოფილია დამატებითი განყოფილება, მათზე განლაგებული კომპონენტების აღნიშვნისათვის. ამასთან სერვერი წარმოდგენილია სიჩქარის და მეხსიერების ატრიბუტებით.

აუცილებელი არ არის კომპონენტები კვანძების მიხედვით განაწილებული იყვნენ სტატიურად. შესაძლებელია კომპონენტების ერთი კვანძიდან მეორეში დინამიური მიგრაციის მოდელირება.

ბევრ განაწილებულ სისტემებში კომპონენტები და ობიექტები არ იცვლიან თავის ადგილმდებარეობას თავდაპირველი განლაგების შემდეგ. მაგრამ გვხვდება განაწილებული სისტემების კატეგორია, რომლებშიც სხვადასხვა არსები გადაადგილდებიან.

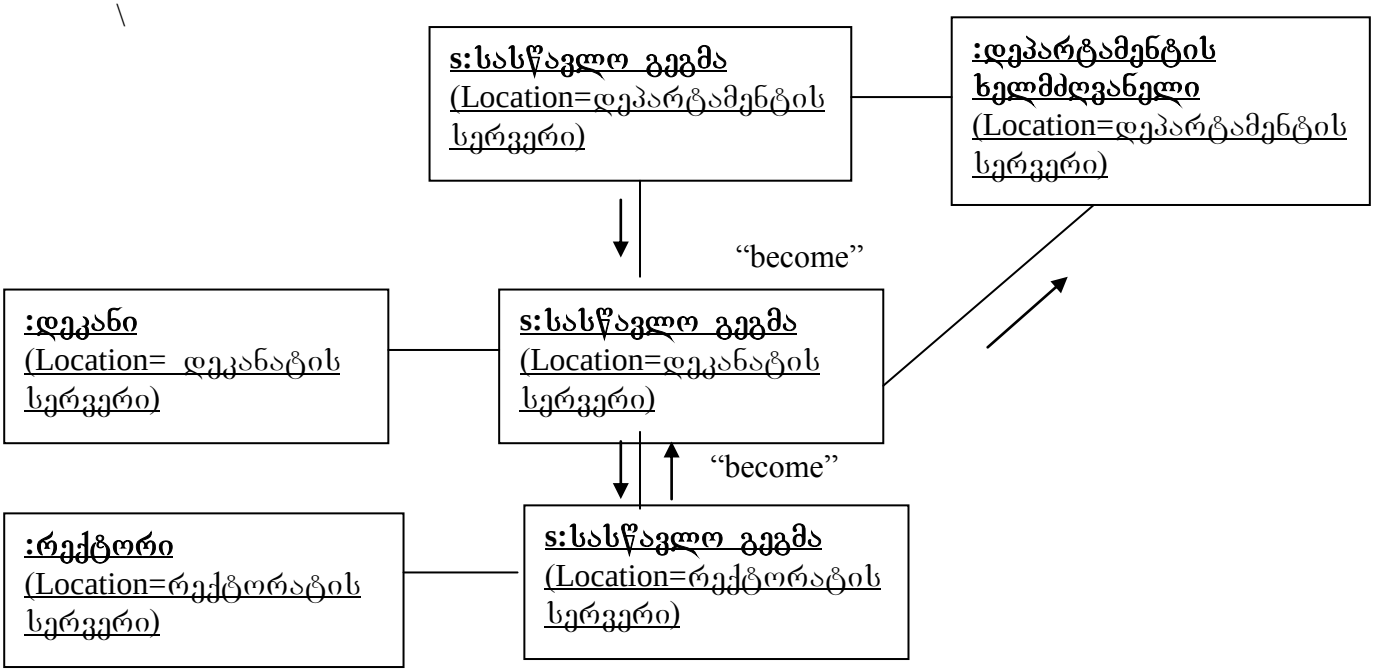
პირველ რიგში, ობიექტები მიგრირებენ, რათა მიუახლოვდნენ აქტიორებს და სხვა ობიექტებს, რომლებთანაც ისინი კოოპერირდებიან სამუშაოს უკეთესი შესრულებისათვის. მაგალითად, სასწავლო პროცესის ორგანიზების სისტემაში ***სასწავლო გეგმა*** გადაადგილდება დეპარტამენტიდან დეკანატში და იქიდან შესაძლებელია რექტორატში.

მეორეს მხრივ, ობიექტები მიგრაციას განიცდიან ამა თუ იმ კვანძის მწყობრიდან გამოსვლის გამო ან დატვირთვის ბალანსირებისათვის სხვადასხვა კვანძებს შორის. ამიტომ სისტემის მდგრადობის უზრუნველსაყოფად ხდება ყველა ელემენტების გადაადგილება სხვა კვანძებზე. ამ დროს შესაძლებელია წარმადობის და გამტარუნარიანობის შემცირება, მაგრამ უზრუნველყოფილი იქნება მისი მდგრადი - ნორმალური მუშაობა.

5.12. ნახაზზე მოყვანილია კომუნიკაციის დიაგრამა, რომელიც წარმოადგენს სასწავლო გეგმის მიგრაციას კვანძებს შორის. კლასის ***სასწავლო გეგმის*** ეგზემპლარი (სახელით **s)** მიგრაციას განიცდის მისი დამტკიცების შესაბამისად. გზაზე ეს ობიექტი ურთიერთქმედებს კლასების ***მიმართულების ხელმძღვანელის, დეკანის, რექტორის*** ეგზემპლიარებთან ყოველ კვანძზე და ბოლოს, დამტკიცების შედეგს აბრუნებს ობიექტზე ***მიმართულება***, რომელიც განთავსებულია კვანძზე დეპარტამენტის სერვერი.

კვანძების კონფიგურაციის და აგრეთვე მათზე განლაგებული კომპონენტების გამოსახვისათვის გამოიყენება განლაგების დიაგრამა. იგი საშუალებას იძლევა მოახდინოთ აპარატული საშუალებების ტოპოლოგიის მოდელირება, რომელზედაც სრულდება სისტემა. ამრიგად, განლაგების დიაგრამა შეიცავს კვანძებს და მიმართებებს (დამოკიდებულების და ასოციაციის) მათ შორის.

\

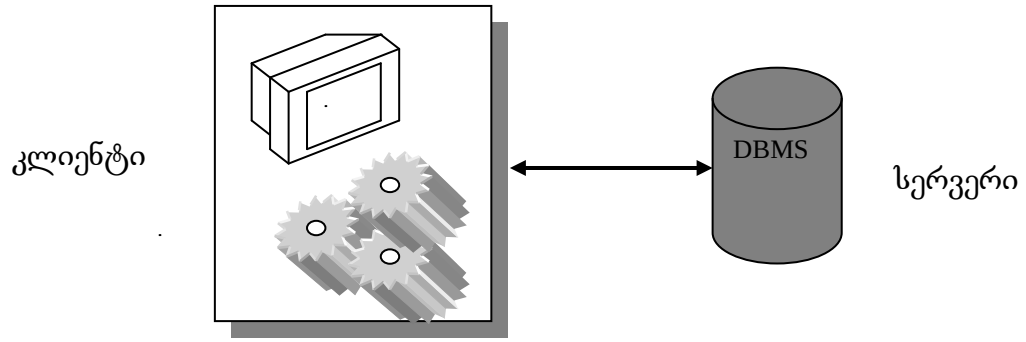


ნახ.5.12.

განლაგების დიაგრამას განსაკუთრებული მნიშვნელობა ენიჭება კლიენტ-სერვერული და მთლიანად განაწილებული სისტემების მოდელირებისას. კლიენტ-სერვერული სისტემა ტიპიური მაგალითია არქიტექტურის, სადაც ძირითადი ყურადღება ეთმობა მოვალეობების მკაფიო გამიჯვნას მომხმარებლის ინტერფეისსა, რომელიც არსებობს კლიენტზე, და სისტემის შენახულ მონაცემებს შორის, რომელიც არსებობენ სერვერზე. ამ დროს მნიშვნელოვანია გადაწყდეს თუ როგორ დაკავშირდნენ კლიენტები და სერვერები ქსელით, ასევე დადგინდეს თუ როგორ არიან განაწილებულნი პროგრამული კომპონენტები კვანძებს შორის. განლაგების დიაგრამები საშუალებას იძლევიან მოახდინოთ ასეთი სისტემების ტოპოლოგიის მოდელირება.

კლიენტ - სერვერული სისტემების მოდელირება თანამედროვე მიდგომით რთული ორგანიზაციული სისტემების ავტომატიზაცია უნდა განხორციელდეს კომპიუტერული ქსელის გამოყენებით, განაწილებული სამუშაო ადგილებით. განაწილებულ სისტემებში პროგრამული კომპონენტების ოპტიმალური განაწილებისათვის გამოიყენება კლიენტ – სერვერული არქიტექტურა.

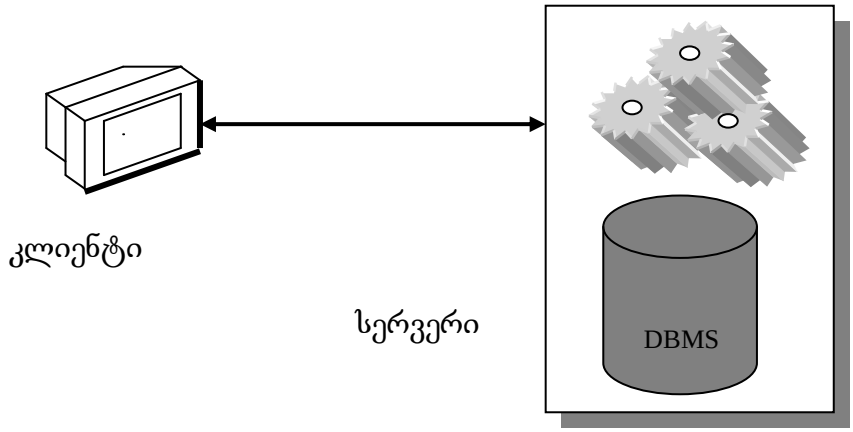
თავიდან, როდესაც კლიენტ-სერვერულმა არქიტექტურამ დაიწყო გავრცელება, ძირითადათ იყო ორ რგოლიანი სისტემები, რომლებიც შედგებოდა რამდენიმე კლიენტისა და ერთი სერვერისაგან. სერვერი ასრულებდა მხოლოდ მონაცემთა სერვერის როლს. ყველაფერი, რაც სისტემის მუშაობის ლოგიკას, ინფორმაციის სისწორეს და დაცვას ეხებოდა, თავმოყრილი იყო კლიენტურ ნაწილში. ასეთი არქიტექტურა წინგადადგმული ნაბიჯი იყო მონოლიტურ პროგრამებთან შედარებით, რადგან ასეთ სისტემებში ინფორმაცია თავმოყრილი იყო ერთ ადგილზე და ყველა კლიენტი მუშაობდა მონაცემთა ერთდაიმავე ასლთან.



ნახ.5.13.

მაგრამ იმის გამო, რომ ბიზნეს-ლოგიკა თავმოყრილი იყო კლიენტურ ნაწილში, საჭირო ხდებოდა შედარებით ძლიერი კლიენტური კომპიუტერის შეძენა, რადგან ყველა კლიენტური პროგრამა თვითონ ახდენდა როგორც ამოცანის ლოგიკასთან დაკავშირებულ გამოთვლებს, ასევე ინფორმაციის ასახვას მომხმარებლის ეკრანზე (ნახ.5.13.).

კლიენტ-სერვერული სისტემების შემდეგ თაობაში მოხდა ბიზნეს-ლოგიკის გადატანა სერვერის მხარეს. ამან გამოიწვია სერვერული რგოლის საკმაოდ გართულება და საჭირო გახდა კიდევ უფრო მძლავრი კომპიუტერი, რომელზეც იგი იმუშავებდა. სამაგიეროდ, ყველა ის კომპიუტერი, რომელზეც სისტემის კლიენტურ რგოლს უნდა ემუშავა, შეიძლება ყოფილიყო დაბალი სიმძლავრის, რადგან სისტემის კლიენტურ ნაწილს მხოლოდ ეკრანზე ინფორმაციის გამოტანა და ბრძანებების მიღება ევალებოდა (ნახ.5.14.). ამან საგრძნობლად გააიაფა ასეთი სისტემის ექსპლუატაცია, იმიტომ რომ ქსელში მხოლოდ ერთი კომპიუტერი იყო მძლავრი და შესაბამისად, შედარებით ძვირი, ხოლო დანარჩენი შეიძლებოდა ყოფილიყო გაცილებით იაფი.



ნახ5.14.

შემდეგი თაობის კლიენტ-სერვერულ სისტემებში ბიზნეს-ლოგიკა გამოიტანეს ცალკე, შუალედურ რგოლში. ასეთ არქიტექტურას ის უპირატესობა აქვს, რომ მისი ყველა ნაწილი მაქსიმალურად დამოუკიდებელია ერთმანეთისაგან. ამიტომ მის რომელიმე რგოლში ცვლილება არ იწვევს სხვა რომელიმე რგოლის ცვლილებას.

მაგალითად, თუ მოხდა ცვლილება პროგრამის მუშაობის ლოგიკაში, მაშინ ეს გამოიწვევს მხოლოდ შუალედური რგოლის ცვლილებას და არ შეეხება სხვა რგოლებს. აგრეთვე ასეთი სტრუქტურის დროს შესაძლებელია არსებობდეს რამოდენიმე შუალედური რგოლი და რამოდენიმე მონაცემთა რგოლი, რაც იძლევა დატვირთვის ბალანსისა და სისტემის მდგრადობის კონტროლის საშუალებას. აგრეთვე გაცილებით ადვილია ასეთი სისტემის მოდერნიზაცია, რადგან შეიძლება ახალი ფუნქციონალურობის დამატება უბრალოდ ახალი კომპონენტის დამატებით.

განასხვავებენ კლიენტ-სერვერული სისტემის სხვადასხვა ვარიაციებს. მაგალითად, “თხელი” კლიენტი, რომლის გამოთვლითი რესურსები შეზღუდულია და ძირითადათ დაკავებულია მომხმარებელთან ურთიერთქმედებით და ინფორმაციის ასახვით. ”თხელ” კლიენტებს შეიძლება არ გააჩნდეთ საკუთარი კომპონენტები. ამის მაგიერ ისინი

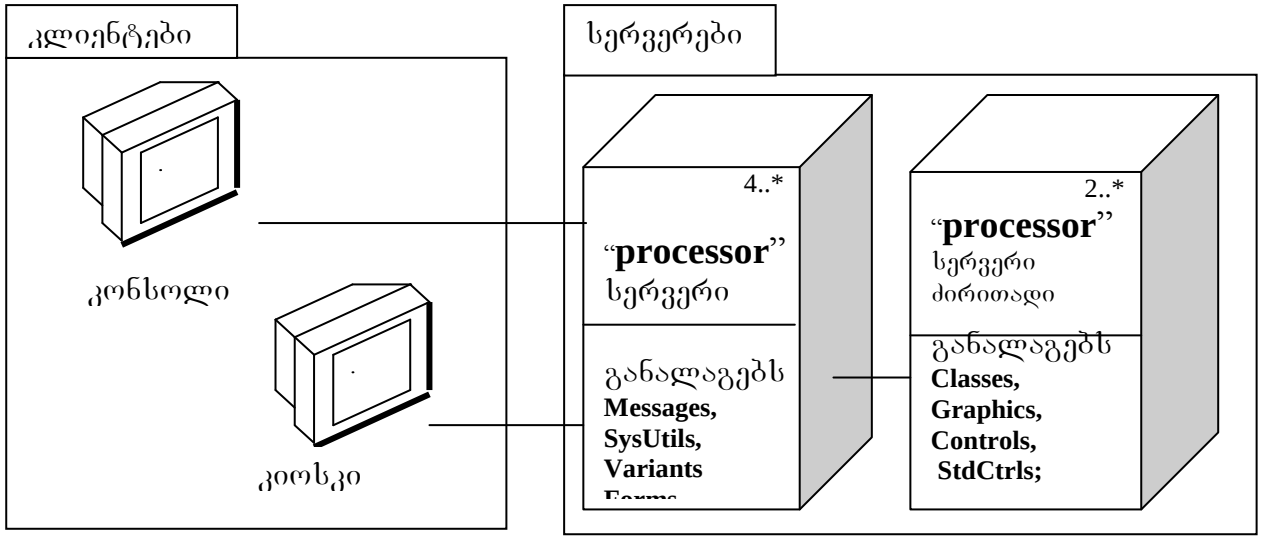
ტვირთავენ კომპონენტებს სერვერიდან მოთხოვნილებისამებრ. მეორეს მხრივ, შეიძლება ავირჩიოთ "მსხვილი" კლიენტი, რომელსაც გამოთვლითი რესურსები მეტი აქვს და რომელსაც ამის გამო შეუძლია დაკავდეს არა მარტო ვიზუალირებით. ამორჩევა "თხელ" და "მსხვილ" კლიენტს შორის ეს არქიტექტორული გადაწყვეტაა, რომელზეც გავლენას ახდენს სხვადასხვა ტექნიკური, ეკონომიკური და პოლიტიკური ფაქტორები.

ნებისმიერ შემთხვევაში სისტემის გაყოფისას კლიენტურ და სერვერულ ნაწილად გვიხდება გადაწყვეტილების მიღება, თუ სად განვათავსოთ ფიზიკურად კომპონენტები და როგორ უზრუნველვყოთ პასუხისმგებლობის ბალანსი მათ შორის. მაგალითად, მართვის სისტემების უმრავლესობის არქიტექტურა ინფორმაციით სამდონიანია, ანუ მომხმარებლის ინტერფეისი, ბიზნეს-ლოგიკა და მონაცემთა ბაზა ფიზიკურად განცალკევებულია ერთმანეთისაგან. გადაწყვეტილება იმის შესახებ, თუ სად უნდა განლაგდეს ინტერფეისი და მონაცემთა ბაზა გასაგებია, ხოლო სად უნდა იმყოფებოდნენ კომპონენტები, რომლებიც ბიზნეს - ლოგიკის რეალიზებას ახდენენ, შედარებით რთულია.

განლაგების დიაგრამა შესაძლებელია გამოვიყენოთ კლიენტ-სერვერული სისტემის ტოპოლოგიის აღწერისა და დადგენისათვის, ასევე პროგრამული კომპონენტების განაწილებისათვის კლიენტსა და სერვერს შორის.

კლიენტ-სერვერული სისტემის მოდელირებისას თავიდან ახდენენ კვანძების იდენტიფიცირებას, რომლებიც წარმოადგენენ კლიენტისა და სერვერის პროცესორებს. გამოყოფენ იმ მოწყობილობებს, რომლებიც ასე თუ ისე გავლენას ახდენენ სისტემის ქცევაზე. სტერეოტიპების მეშვეობით დაამუშავებენ ვიზუალურ აღნიშვნებს პროცესორებისა და მოწყობილობებისათვის. ბოლოს, დაამოდელირებენ კვანძების ტოპოლოგიას განლაგების დიაგრამაზე.

ნახ.5.15.-ზე ნაჩვენებია სისტემის ტოპოლოგია, რომელიც შეესაბამება კლასიკურ კლიენტ-სერვერულ არქიტექტურას. როგორც ნახაზიდან ჩანს, კლიენტი და სერვერი მკაფიოდ არის გამიჯნული პაკეტების(*კლიენტი,სერვერი*) გამოყენებით. პაკეტი *კლიენტი* შეიცავს ორ კვანძს(*კონსოლი, კიოსკი*). პაკეტი სერვერი ასევე შეიცავს ორ კვანძს(*სერვერი, ძირითადი სერვერი*). თითოეულზე აღწერილია კომპონენტები, რომლებიც განლაგებულია კვანზძე. ასევე, ორივე კვანძისათვის მითითებულია ჯერადობა, რომელითაც მიეთითება თუ რამდენი ეგზემპლიარია მოსალოდნელი კონკრეტულ კონფიგურაციაში.



ნახ.5.15.

მთლიანად განაწილებული სისტემის მოდელირება. განაწილებული სისტემები შეიძლება იყვნენ სხვადასხვა სახის - დაწყებული უბრალო ორპროცესორიანიდან დამთავრებული განშტოებული, განლაგებულნი მრავალ გეოგრაფიულად დაშორებულ კვანძებში.

უკანასკნელნი როგორც წესი არ არიან სტატიკურნი. კვანძები ჩნდებიან და ქრებიან პროცესორების მწყობრიდან გამოსვლის გამო. იქმნება ახალი, უფრო ჩქარი კავშირგაბმულების არხები, რომლებიც ფუნქციონირებენ ნელის პარალელურად, რომელთა ბოლოს და ბოლოს დემონტირებას ახდენენ. იცვლება არა მარტო სისტემის ტოპოლოგია, არამედ პროგრამული კომპონენტების განაწილება. მაგალითად, მონაცემთა ბაზის ცხრილები შეიძლება გადაადგილდეს სერვერებს შორის რათა დაუახლოვოთ ისინი ინფორმაციის მომხმარებლებს.

განაწილებული სისტემები თავისი ბუნებით შესდგებიან კომპონენტებისაგან, ფიზიკურად გაბნეულნი სხვადასხვა კვანძებზე. ბევრი სისტემებისათვის ადგილმდებარეობა (**Location**) კომპონენტების ფიქსირდება სისტემის დაყენების მომენტში. მაგრამ გვხვდება ისეთი სისტემებიც, რომლებშიც კომპონენტები მიგრირებენ ერთი კვანძიდან მეორეში.

ელემენტის ადილმდებარეობის მითითება შესაძლებელია ორი საშუალებით - მიუთითოდ კვანძის დამატებით განყოფილებაში ან ვისარგებლოთ მონიშნული მნიშვნელობით **Location** იმ კვანძის აღნიშვნისათვის, რომელზეც განლაგდება კლასი.

განაწილებული სისტემის ტოპოლოგიის მოდელირებისას მიზანშეწონილია განვიხილოთ ეგზემპლიარების როგორც კომპონენტების, ასევე კლასების ფიზიკური განლაგება. თუ ყურადღების ცენტრში არის გაშლილი სისტემის კონფიგურაციის მართვა, კომპონენტების განაწილების მოდელირება განსაკუთრებით მნიშვნელოვანია ისეთი ფიზიკური არსების სპეციფიცირებისათვის, როგორიც არის შესრულებადი მოდულები, ბიბლიოთეკები და ცხრილები. თუ უფრო აინტერესებთ ფუნქციონალურობა,

მასშტაბურობა და სისტემის გამტარუნარიანობა, მაშინ უფრო მნიშვნელოვანია ობიექტთა განაწილების მოდელირება.

გადაწყვეტილება თუ როგორ განაწილდეს ობიექტები სისტემაში მნიშვნელოვანი პრობლემაა და მჭიდროდ არის დაკავშირებული პარალელიზმის საკითხებთან.

ობიექტების განაწილების მოდელირებისას ყოველი ინტერესმქონე კლასისათვის სისტემაში განიხილავენ მიმართვების ლოკალურობას – სხვა სიტყვებით, გამოავლენენ ყველა მეზობლებს და მათ ადგილმდებარეობას. ძლიერკავშირიანი ლოკალურობა ნიშნავს, რომ ლოგიკურად მეზობლური ობიექტები იმყოფებიან გვერდიგვერდ, ხოლო სუსტკავშირიანი – რომ ისინი ფიზიკურად დაშორებულნი არიან ერთმანეთისაგან. ასევე უნდა ეცადოთ განალაგოთ ობიექტები აქტიორების გვერდით, რომლებიც მანიპულირებენ მათზე.

შემდეგ განიხილავენ ტიპიურ ურთიერთქმედებებს ურთიერთდაკავშირებულ ობიექტთა სიმრავლეებს შორის და განალაგებენ ობიექტების სიმრავლეს ურთიერთქმედების მაღალი ხარისხით გვერდიგვერდ, იმისათვის რომ შემცირდეს კომუნიკაციის ღირებულება. ობიექტები, რომლებიც სუსტად ურთიერთქმედებენ ერთმანეთს შორის, განაცალკევებენ სხვადასხვა კვანძებზე. ბოლოს, განიხილავენ პასუხისმგებლობის განაწილებას სისტემაში. გადაანაწილებენ ობიექტებს ისე, რომ დაბალანსდეს ყოველი კვანძის დატვირთვა. უნდა გათვალისწინებული იქნას მომსახურების უსაფრთხოება, მობილურობა და ხარისხი. ეს შეხედულებები უნდა აისახოს ობიექტების განლაგების დროს. ობიექტები დიაგრამაზე შესაძლებელია გამოიხატოს ერთი ორი შესაძლო საშუალებიდან:

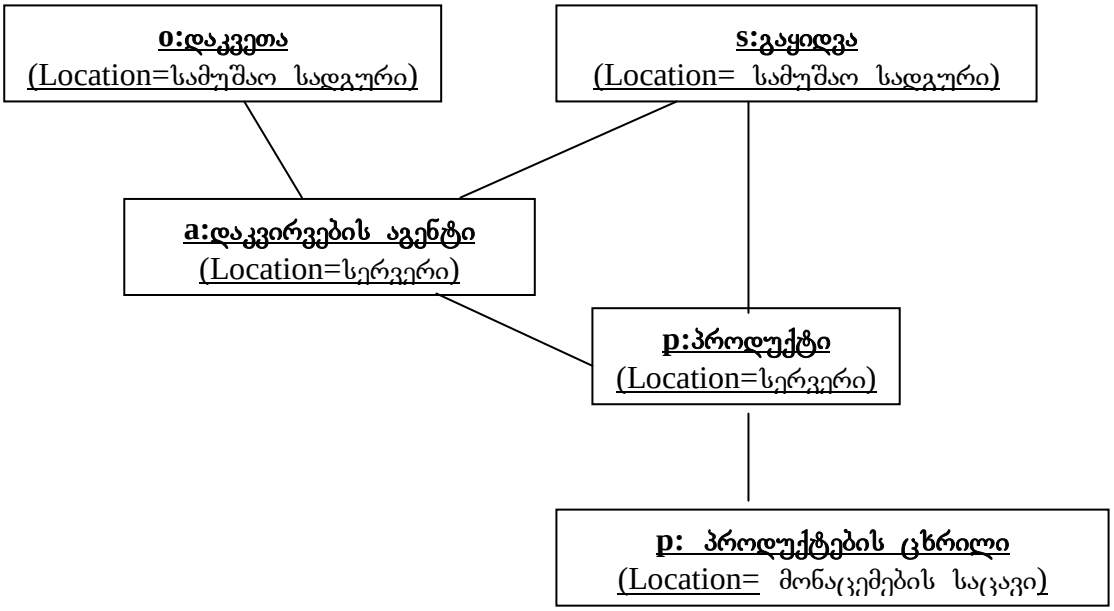
- ჩავრთოთ ობიექტები უშუალოდ კვანძებში განლაგების დიაგრამაზე.
- ნათლად მიუთითოდ ობიექტის მდგომარეობა მონიშნული მნიშვნელობის მეშვეობით.

ნახ.5.16.-ზე მოყვანილია ობიექტების დიაგრამა, რომელიც ახდენს საცალო ვაჭრობის სისტემაში ობიექტების განაწილების მოდელირებას. ამ დიაგრამის ფასი იმაშია, რომ იგი საშუალებას იძლევა მოვახდინოთ ძირითადი ობიექტების ფიზიკური განლაგების ვიზუალირება.

როგორც ჩანს, ორი ობიექტი *დაკვეთა* და *გაყიდვა* იმყოფებიან კვანძში *სამუშაო სადგური*, ორი სხვა *დაკვირვების აგენტი* და *პროდუქტი* – კვანძში *სერვერი* და ერთი *პროდუქტების ცხრილი* – კვანძში *მონაცემების საცავი*.

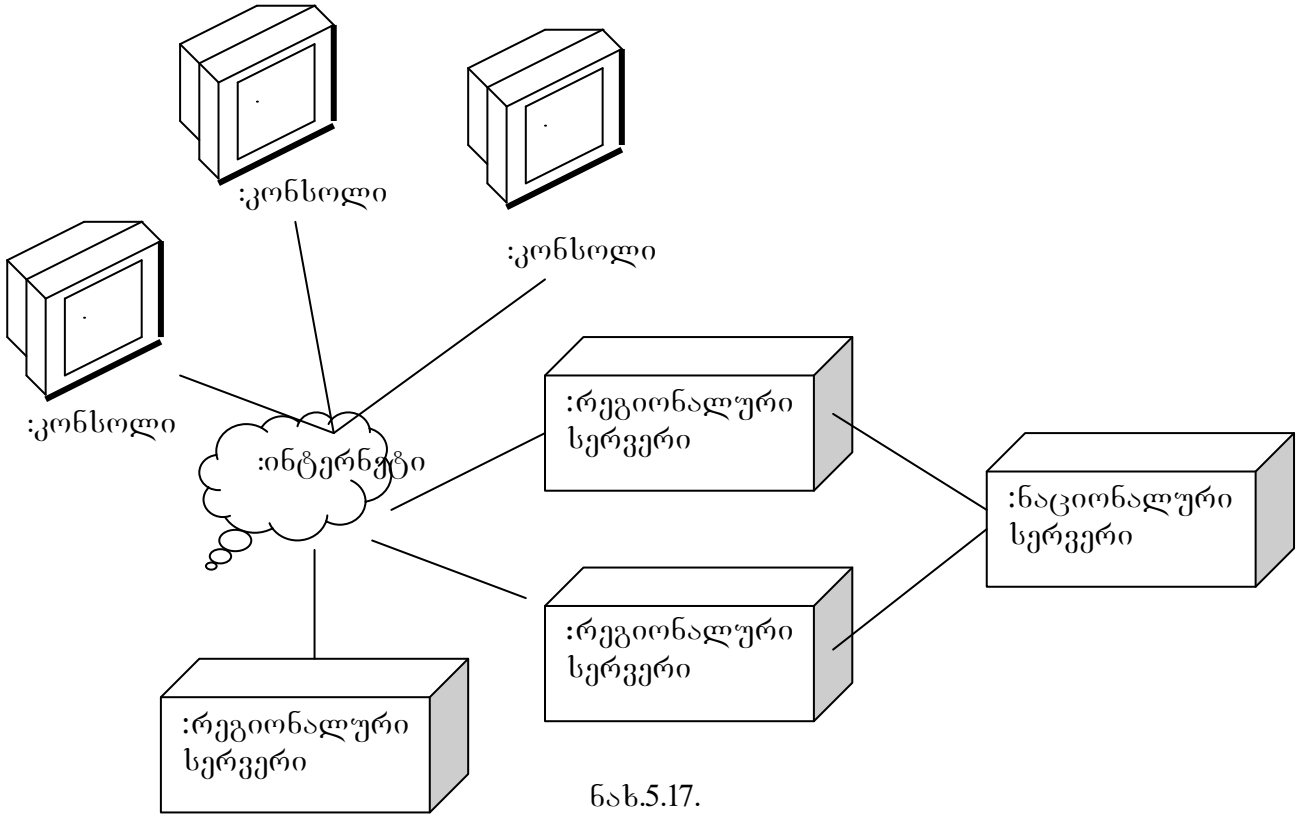
მთლიანად განაწილებული სისტემის მოდელირებისას მოახდენენ მოწყობილობების და პროცესორების იდენტიფიცირებას, ისევე როგორც კლიენტ-სერვერული სისტემის შემთხვევაში. განსაკუთრებული ყურადღება უნდა მიექცეს კვანძების დაჯგუფებას, რისთვისაც შესაძლებელია პაკეტების გამოყენება. წარმოადგენენ რა მოწყობილობებს და პროცესორებს განლაგების დიაგრამის სახით, ყველგან სადაც შესაძლებელია უნდა

გამოყენებულ იქნას ინსტრუმენტალური საშუალებები სისტემის ქსელური ტოპოლოგიის გახსნისათვის.



ნახ.4.16. ობიექტების განაწილების მოდელირება

ნახ.5.17.-ზე გამოსახულია მთლიანად განაწილებული სისტემის ტოპოლოგია. მასზე გამოსახულია სამი კონსოლი, რომლებიც დაკავშირებულნი არიან ინტერნეტთან. მეორეს მხრივ, გვაქვს სამი რეგიონალური სერვერი, რომლებიც ანხორციელებენ ინტერფეისს ნაციონალურ სერვერთან.



ნახ.5.17.

ლიტერატურა

1. Джим Арлоу и Айла Нейштадт, **UML 2** н Унифицированный процесс. 2-е издание, Практичеслий объектно- ориентированный анализ и проектирование, Санкт-Петербург - Москва, 2008.
2. Буч Г., Рамбо Д., Джекобсон А. Язык **UML**. Руководство пользователя.// Серия "Объектно-ориентированные технологии в программировании". Москва, 2004.
3. Леоненков. Самоучитель UML. UML Teach Yourself. ... Особенности реализации языка UML в CASE-инструментарии Rational Rose 98/2000 · Заключение · Каталог · Индекс раздела.
   khpi-iip.mipk.kharkiv.edu/library/case/leon/index.html - 3k –
4. სისტემების ობიექტზე ორიენტირებული ანალიზი. დამხმარე სახელმძღვანელო. ლაბორატორიული პრაქტიკუმი. დამტკიცებულია სტუ-ს სარედაქციო-საგამომცემლო საბჭოს მიერ. გამომცემლობა "ტექნიკური უნივერსიტეტი", 20013, 90 გვ.
5. სისტემების ობიექტზე ორიენტირებული დაპროექტება. დამხმარე სახელმძღვანელო. ლაბორატორიული პრაქტიკუმი. დამტკიცებულია სტუ-ს სარედაქციო-საგამომცემლო საბჭოს მიერ. გამომცემლობა "ტექნიკური უნივერსიტეტი", 20013, 95 გვ.
6. პროგრამული სისტემის დამუშავების CASE საშუალებები. დამხმარე სახელმძღვანელო. ლაბორატორიული პრაქტიკუმი. 20013, 130 გვ.http://gtu.ge/book/Uml1.pdf
7. გოგიჩაიშვილი გ,, სუხიაშვილი თ.. სისტემების ობიექტ-ორიენტირებული ანალიზი და დაპროექტება. სახელმძღვანელო. დამტკიცებულია სტუ-ს სამეცნიერო-ტექნიკური საბჭოს მიერ. გამომცემლობა "ტექნიკური უნივერსიტეტი", 2009. 165 გვ. ბიბლ. 681.3.06. 214 975 067-071
8. სუხიაშვილი თ. მოდელირების უნიფიცირებული ენა (UML). პრაქტიკული ობიექტ-ორიენტირებული ანალიზი და დაპროექტება. თბილისი, "ტექნიკური უნივერსიტეტი". 2018, 210 გვ. ელ-სახელმძღვანელო. http://gtu.ge/book/Uml1.pdf
9. სუხიაშვილი თ. სისტემების ობიექტ-ორიენტირებული დაპროექტება. საკურსო პროექტის მეთოდ. სახელმძღვ. 2016, 97გვ. http://gtu.ge/book/sak_pro.pdf
10. სუხიაშვილი თ. პროგრამული სისტემის დამუშავება ინსტრუმენტალური საშუალება IBM Rational Rose გამოყენებით. ლაბორატორიული პრაქტიკუმი. თბილისი, "ტექნიკური უნივერსიტეტი". 2018, 125გვ. ელექტრონული სახელმძღვანელო.
11. სუხიაშვილი თ. ანალიზის სამუშაო ნაკადი პროგრამული სისტემის დამუშავების უნიფიცირებულ პროცესში. პროფ. კ.კამკამიძის 90 წლისადმი მიძღვნილი საერთაშორისო კონფერენცია. თბილისი, "ტექნიკური უნივერსიტეტი". 2018,
12. სუხიაშვილი თ. ანალიზის კლასები და მათი გამოვლენა ბიზნეს-პროცესების მართვისას. სტუ-ს შრომები, 2019, №,3(27) გვ.83-86
13. სუხიაშვილი თ. გამორიცხვების დამუშავების მოდელირება ბიზნესპროცესების მართვისას. სტუ-ს შრომები, 2019, №,3(27) გვ.83-86

14. რეალური დროის მოდელირება ბიზნეს პროცესების მართვისას. სტუ-ს შრომები, 2015, № 1(19), გვ. 56-60.
15. . მონაცემთა ბაზის ლოგიკური სქემის დამუშავება ბიზნეს-პროცესების კომპიუტერული მართვისათვის. სტუ-ს შრომები, 2016, # 1(21), გვ. 116-122.
16. . ადაპტაციის საშუალებების დამუშავება ობიექტ-ორიენტირებული დაპროექტებისას. სტუ-ს შრომები, 2016, № 2(22), გვ. 135-140.