Universidade do Minho

MIEI - Grupo 29

Desenvolvimento de Sistemas de Software

UPS!

Unbelievable Permutation for Shifts!

(a) Hugo Brandão A78582

(b) José Resende A77486

(c) Tânia Silva A76945

Conteúdo

1 Introdução

Neste relatório são abordadas todas as fases do desenvolvimento da Aplicação UPS! - Unbelievable Permutation for Shifts!, com o auxílio de UML, uma linguagem de modelação que permite retratar a interação, comportamento e estrutura da aplicação. Abordamos como chegamos à solução numa secção explicativa e expositiva, de maneira ao raciocínio ser mais claro.

De seguida passamos à secção de modelação onde são mostrados os diagramas mais relevantes para a execução da Aplicação, sob as normas UML.

Passamos então a uma análise da Interface em contraste com os mockups previamente existentes, de forma a também colocar um ponto de situação em termos de apresentação. Terminamos com uma análise crítica ao nosso trabalho, ponderando os pontos altos e baixos de todo o processo, além de uma reflexão sobre possível trabalho futuro.

2 Desenho da Solução

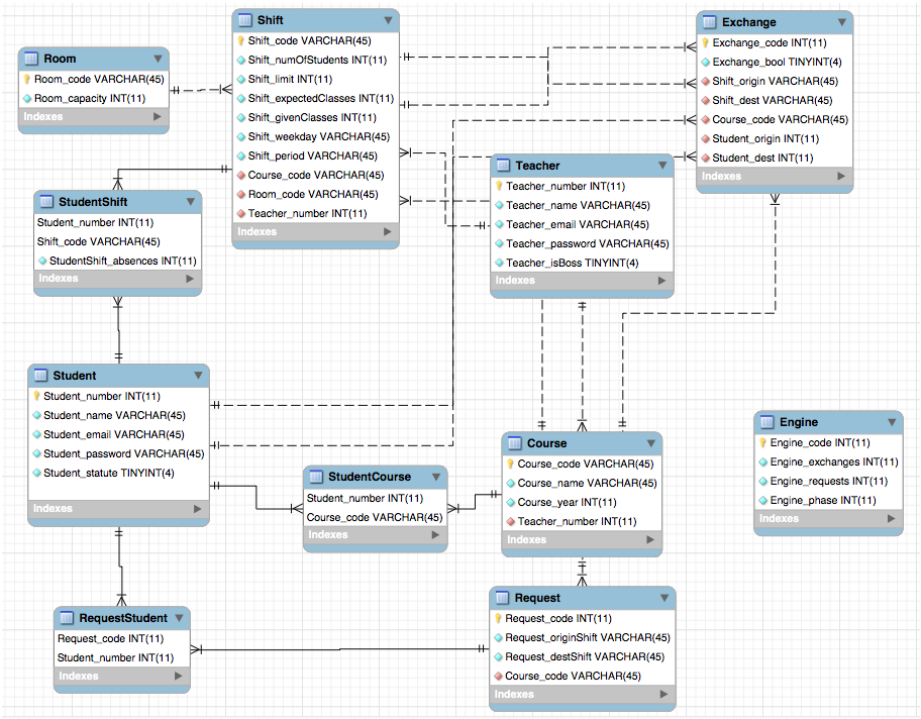
2.1 Estrutura da Aplicação

Existem 3 camadas na aplicação, que são a Presentation, Business e Data, que tratam respetivamente com a Interface Gráfica, a Lógica de Negócio e a Persistência de Dados.

Relativamente à interface utilizamos JavaFX, uma framework de criação de interfaces gráficas em Java, que, na nossa opinião, se adequava melhor às necessidades que possuíamos em termos de apresentação. Desta forma, com algum empenho, conseguimos utilizar esta ferramenta com a qual não tínhamos experiência, de forma bem sucedida.

Na camada Lógica temos um meio de comunicação (Façade) com a GUI, a classe Engine, que possui toda a estrutura e organiza a lógica da Aplicação. Foi necessário, após definir classes e rever o Modelo de Domínio, definir que tipo de estruturas a utilizar. O seu estado final possui DAOs (Data Access Objects) que comunicam com a Base de Dados e implementam as estruturas genéricas que escolhemos.

A camada de Dados foi uma das fases mais demoradas de todo o processo de desenvolvimento, já que foi necessário definir uma Base de Dados sem erros nem inconsistências, e que numa fase inicial fizesse sentido. A base de dados final ficou da seguinte forma:



Modelo lógico da Base de Dados

2.2 Abordagem ao problema

Na conceção de uma solução para a troca de turnos tínhamos de decidir exatamente como queríamos resolver este problema, de que forma iriam ser representadas as várias entidades e como se iriam relacionar de forma a ser uma solução justa para todas as partes envolvidas.

Tendo em vista a concretização de requisitos no enunciado, tentamos criar um sistema que fosse *automático*, isto é, que depois de fazer um pedido de troca qualquer pedido que satisfaça as necessidades do primeiro se efetue uma troca e que esta seja feita sem interação do utilizador, tendo sempre a certeza que não existem em nenhum momento duas trocas que se completam uma à outra.

De cada vez que um pedido é gerado efetua-se uma verificação de todos os pedidos que tenham como destino o turno de origem do pedido gerado. Se algum deles for compatível é efetuada uma troca entre as duas partes, atualizando todos os registos envolvidos.

É de notar que foram abstraídas as aulas teóricas do problema, já que essas são comuns a todos os alunos e não sujeitas a trocas. Assumimos sempre que as UCs do mesmo ano não possuem colisões com as possíveis aulas teóricas do mesmo ano.

2.3 Robustez das Alocações

Um dos principais problemas que tivemos foi alocar de maneira robusta (i.e. sem sobreposições) os diferentes turnos a um determinado aluno. O fator que nos levou à solução foram as UCs complementares dos perfis de Mestrado. Reparamos que nos ficheiros de input existiam dois campos muito particulares: *diasem* e *per*. Estes dois dão o período no qual as aulas são dadas. Desta forma, e aplicando estes campos a todas os turnos, foi fácil definir um horário (classe Schedule), que consiste em dois HashMaps para cada aluno, um para a manhã e outro para a tarde, com 5 posições e booleanos, que nos fornecem a informação se um aluno está disponível naquele determinado período ou não.

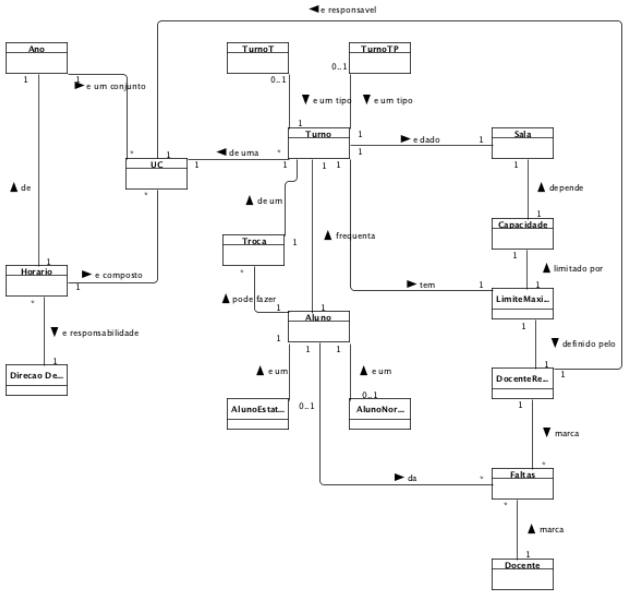
2.4 Solução Proposta

Imaginemos as UCs como grafos. Sejam TP1, TP2, TP3 nodos desse grafo, que representam os diversos turnos do Curso "DSS". Os requests (i.e. pedidos de troca) são arestas orientadas entre nodos, representando a origem e o destino daquele request. Após isso são verificados todos as arestas de saída do Nodo de destino, tentando encontrar uma aresta oposta. Caso esta seja encontrada, então efetua-se uma troca e é registada uma Exchange, que mais tarde pode ser revista pela DC e pelo professor regente da disciplina.

3 Documentação - Modelagem

3.1 Modelo de Domínio

O modelo de domínio mantém-se o mesmo da primeira fase, tendo-o como base para a criação da base de dados.



Modelo de Domínio

3.2 Diagrama de Use Cases

Tal como na primeira parte, aqui estão os Use Cases que definimos para a conceção da aplicação:

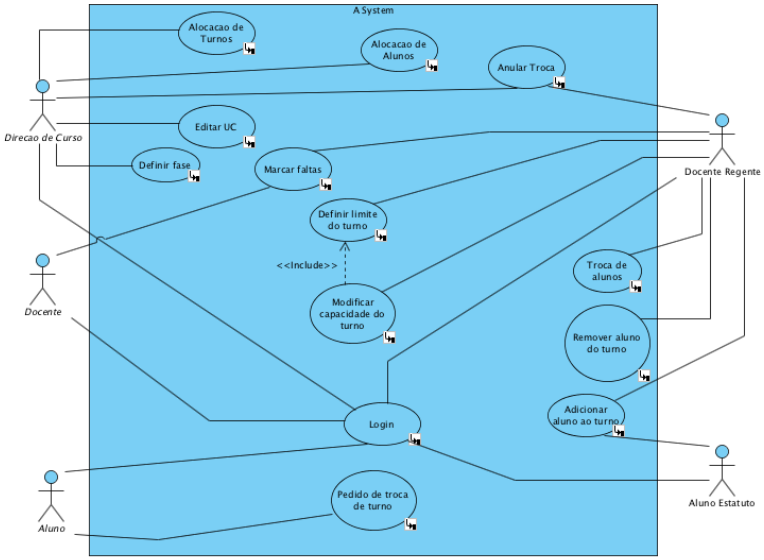


Diagrama de Use Cases

Acreditamos que a sua especificação textual seja desnecessária já que a mesma foi entregue no relatório da primeira fase e os Diagramas de Sequência abaixo explicitam de uma forma muito mais clara o que cada Use Case significa em termos de funcionamento interno do Sistema.

Houve, contudo, um Use Case que sentimos que já era tratado noutros e, portanto, decidimos não incluir em qualquer diagrama de sequência. Estamo-nos a referir ao Use Case Troca de Alunos que o Docente Regente possui, que já está tratado e verificado nos Use Cases: Remover Aluno do Turno e Adicionar Aluno ao Turno.

3.3 Diagramas de Sequência de Sistemas

Estes diagramas foram essenciais tanto na conceção de uma estrutura da aplicação bem organizada, como na visualização de uma interação clara entre objetos. Embora todos os diagramas se possam encontrar na sua íntegra na secção de Anexos, encontra-se aqui especificado um dos diagramas que melhor exemplifica a importância da modelação de DSS's.

O diagrama representa o Use Case correspondente a Modificar a Capacidade de um Turno, e nele podemos observar que a camada Business irá fazer todas as operações necessárias, servindo-se da Data para recolher informações guardadas

na base de dados. Esta é uma particularidade que se torna bastante evidente nestes diagramas, e é essencial para podermos representar todas as nossas classes no modelo de 3 camadas definido acima.

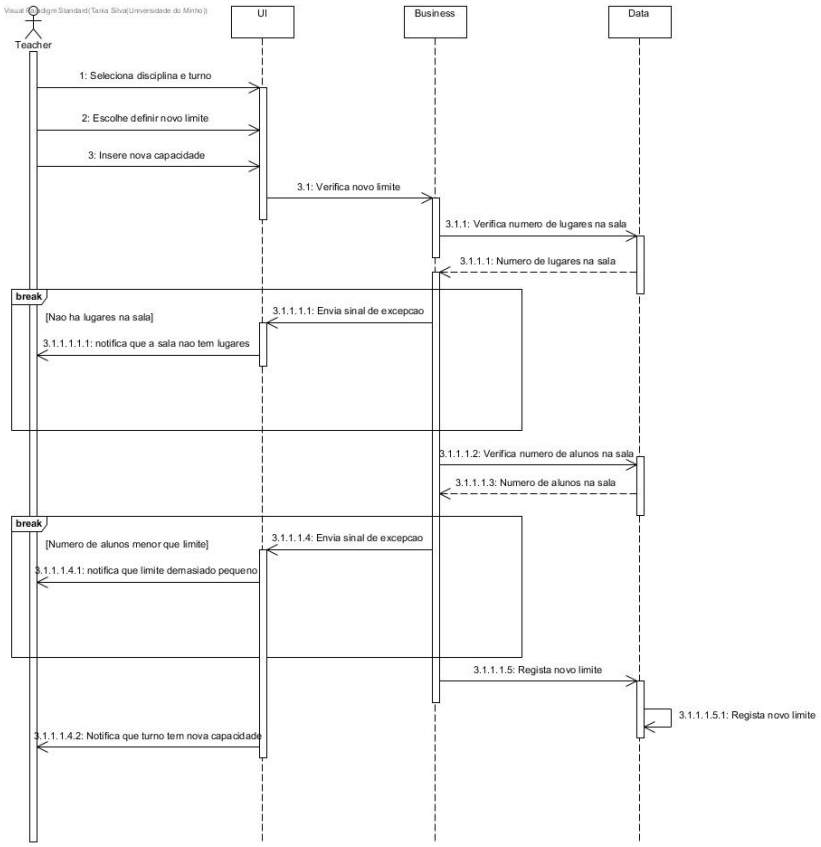
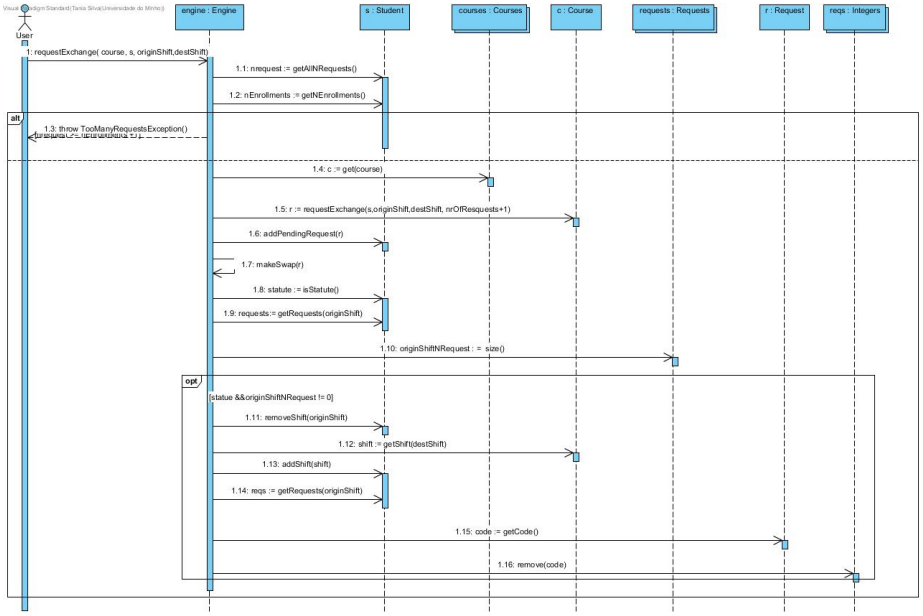


Diagrama de Sequência de Sistemas

3.3.1 Diagramas de Sequência de Subsistemas

Estes são os diagramas dos métodos mais relevantes que utilizamos para satisfazer os use cases que definimos. Na secção 6 podem-se encontrar na íntegra estes diagramas, enquanto aqui mostraremos um diagrama dos mais centrais para o bom funcionamento da Aplicação, o requestExchange. Ainda que pareça que a execução dos métodos é centralizada por parte da Engine (i.e Façade) isto não acontece, já que toda a lógica que podia não estar lá, não está, tentando sempre distribuir a carga sobre todos os objetos. Mesmo assim, quando o método envolve várias entidades que se relacionam a partir do Engine, é necessário que seja este a executar a lógica.

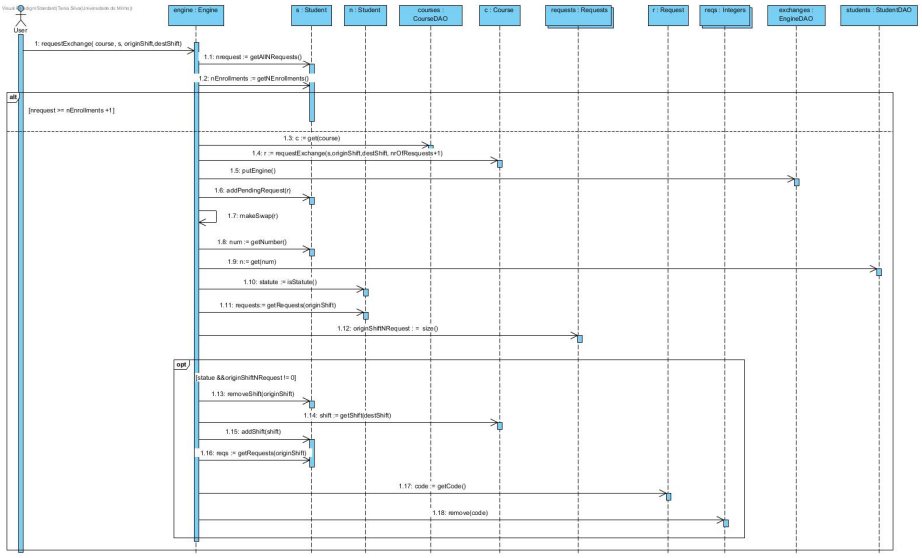
Este é o método que permite a um aluno criar um pedido de troca de turno, mas com algumas particularidades. Logo após ser criado o pedido, é feita uma verificação se existe algum pedido que o complemente. Além disso, note-se que o aluno de estatuto passa pelo mesmo processo, de maneira a favorecer o maior número de pessoas em vez de só trocar de turno. Obviamente que se nenhum pedido complementar aquele do aluno de estatuto, este trocará de turno sem ser necessária a Exchange.



requestExchange

Sendo estes os primeiros diagramas de sequência de subsistema da nossa aplicação, tivemos que mais tarde os reformular de forma a considerar a persistência de dados, ou seja, a presença de DAOs. Isto consistiu, maioritariamente, em substituir certas estruturas pelos DAOs, e deu origem aos diagramas de sequência de subsistema mais avançados que nós chamamos de 2.0.

Abaixo encontra-se representado o diagrama 2.0 correspondente ao diagrama acima mencionado, requestExchange.



requestExchange 2.0

Nestes diagramas tornam-se explícitos alguns poucos acessos à base dados, sendo que a maioria é feita em classes mais específicas. Estes acessos concentram-se em duas áreas apenas, sendo estas a recolha de dados e a persistência dos mesmos. No diagrama acima verifica-se isto na mensagem 1.3, onde se vai buscar um curso à base de dados, na mensagem 1.5, onde são guardados alguns dados do Engine, e na mensagem 1.9 onde se faz a atualização do Student guardado em memória através de um *get* à base de dados. Estas operações são bastante frequentes, especialmente em métodos dentro de classes mais particulares.

3.4 Modelação Estrutural

3.4.1 Diagramas de Classe

Esta primeira modelação das classes da aplicação dá-nos uma visão bem estruturada de como se comunicam as diferentes entidades do programa. Obter estas foi relativamente fácil, já que é um problema com o que estamos familiares, dado a sermos utilizadores de uma ferramenta similar.

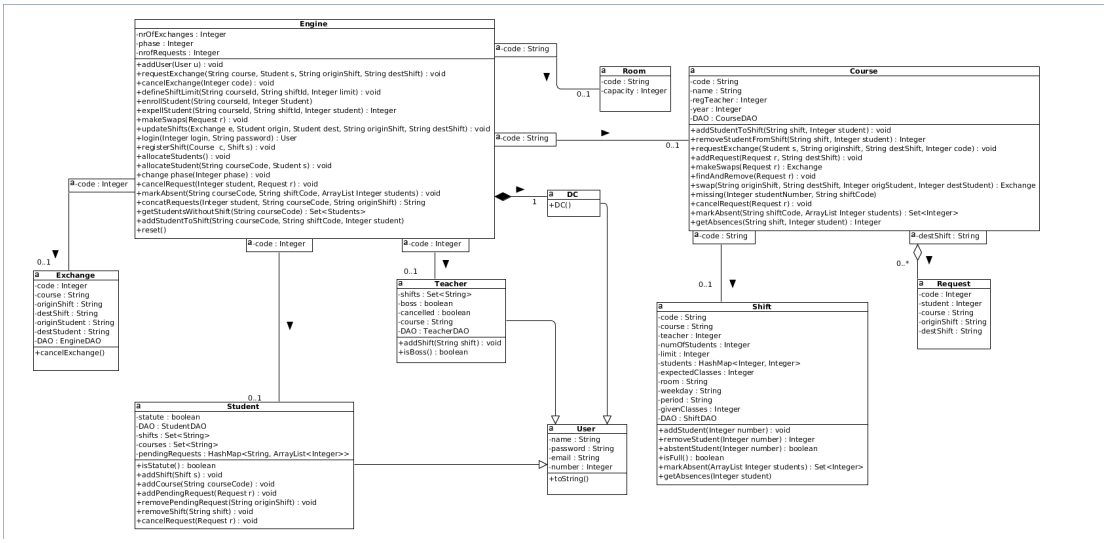


Diagrama de Classes Inicial

Contudo, a segunda versão deste diagrama, já com os DAOs nele, foi um desafio maior, já que foi necessário pensar no que cada DAO representaria. Para termos uma representação da solução mais legível e de mais fácil compreensão, modelamos vários diagramas de classes para a aplicação, os quais se encontram na íntegra na secção 6. Aqui mostraremos um dos diagramas que foi mais útil na conceção de uma aplicação suportada por DAOs, o Diagrama de Classes dos Dados.

Neste diagrama torna-se evidente que todas as classes de acesso a objetos serão sub-classes de uma classe DAO que implementará métodos comuns a todos. Também verificamos que uma classe Connect será necessária para estabelecer uma conexão com a base de dados, classe a que todas os outros DAOs estarão associados.

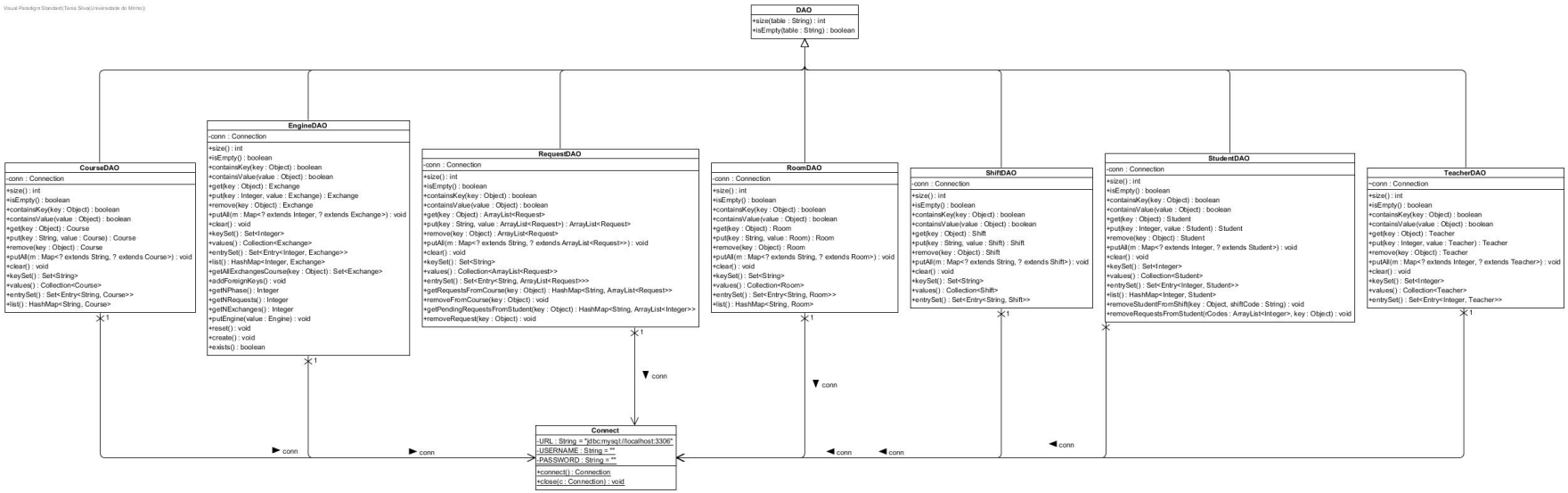


Diagrama de Classes de Data

3.4.2 Diagrama de Instalação

Como o nosso sistema corre todo na mesma máquina, o diagrama de instalação corre num só *device*. Neste apenas se comunicam 3 componentes, Presentation, Engine e Ups Database.

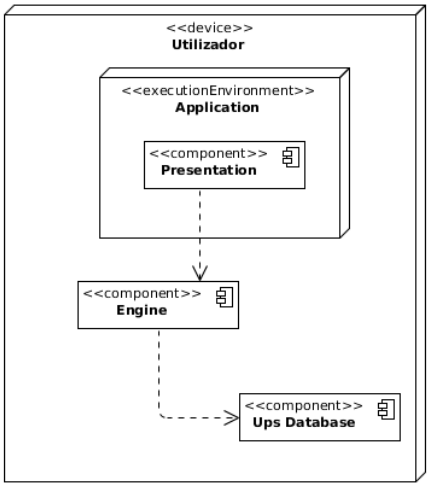


Diagrama de Instalação

3.4.3 Diagrama de Packages

Existem 3 grandes packages na aplicação, correspondentes às três camadas supra mencionadas. Dentro do package de presentation, possuímos um package de controllers, necessário para JavaFX e um package de views, que possui as templates das várias views da aplicação.

No package business, dividimos as classes em grupos aos quais estavam associados, um package users para as classes relacionadas com utilizadores, um package courses para as classes relacionadas com UCs e turnos, um package exceptions para agregar todas as exceptions da aplicação, e um package de utilities, que agrega classes de lógica genérica e o parser da aplicação.

No package data estão todas as classes DAO bem como as classes de ligação à Base de Dados.

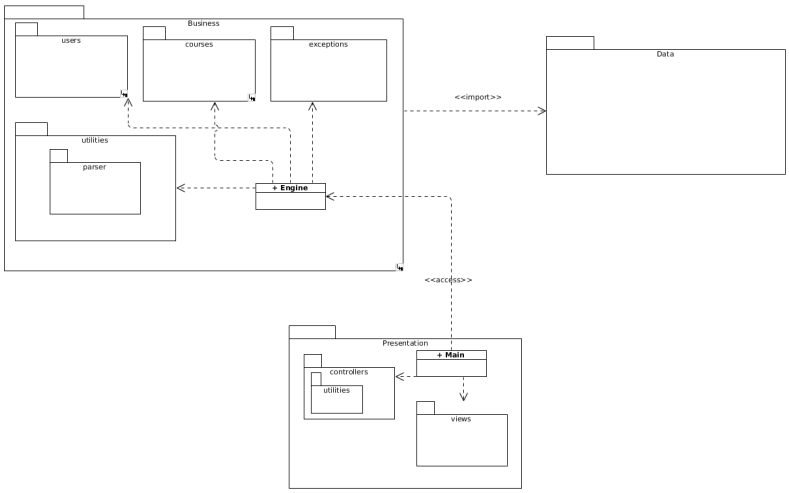


Diagrama de Packages

3.5 Modelação Comportamental

3.5.1 Diagramas de Atividade

As atividades desempenhadas pelos atores deste sistema são, na realidade, bastante simples, podendo os diagramas associados a essas atividades ser encontradas na secção 6. Aqui mostramos um diagrama de cancelamento de uma troca de turnos por parte de um professor regente:

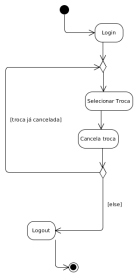
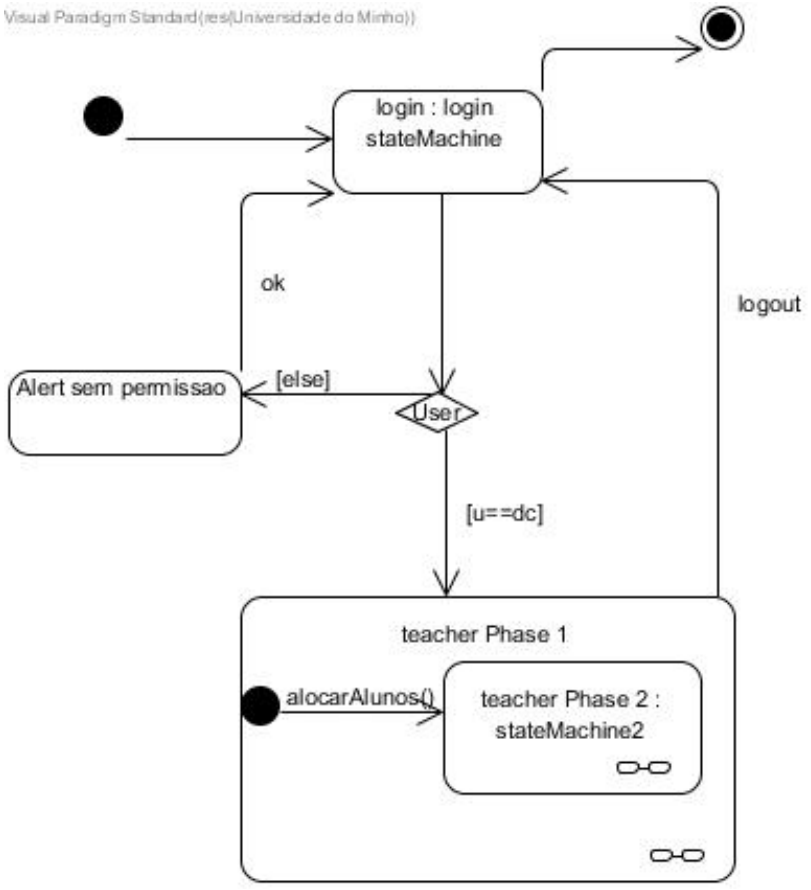


Diagrama de Atividade de um Cancelamento de Troca

4 Estado Final da Interface

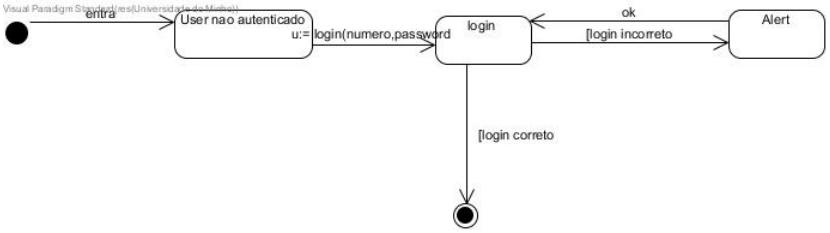
4.0.1 Máquinas de Estado

Antes de apresentar a interface finalizada, foi necessário definir que tipo de ações ou estados esta poderia ter, de maneira a facilitar a implementação lógica da aplicação e também de forma a esta reduzir o número de verificações e validações na hora de receber input. Dividimos a interface em 3 fases, tal como explicitado no enunciado do trabalho, estando apenas algumas funcionalidades permitidas em cada fase, mudando a interface de acordo com a fase que a Direção de Curso estabeleceu naquele determinado momento.

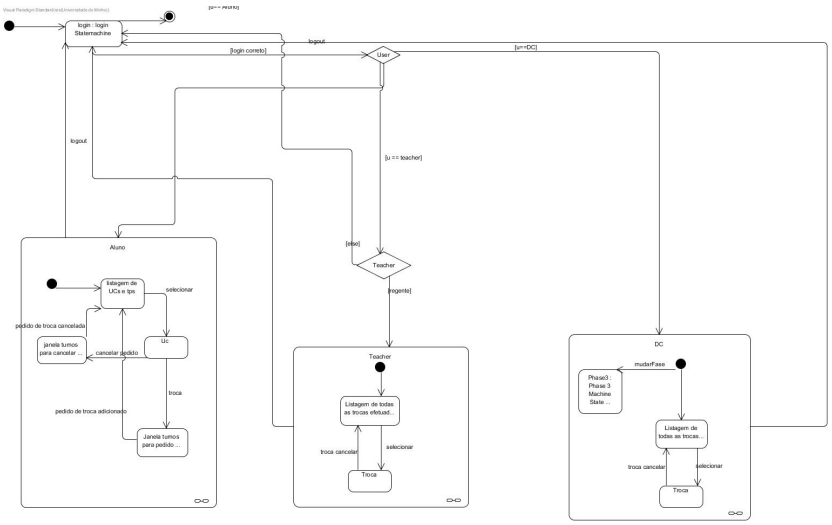


Máquina de Estado para a primeira fase

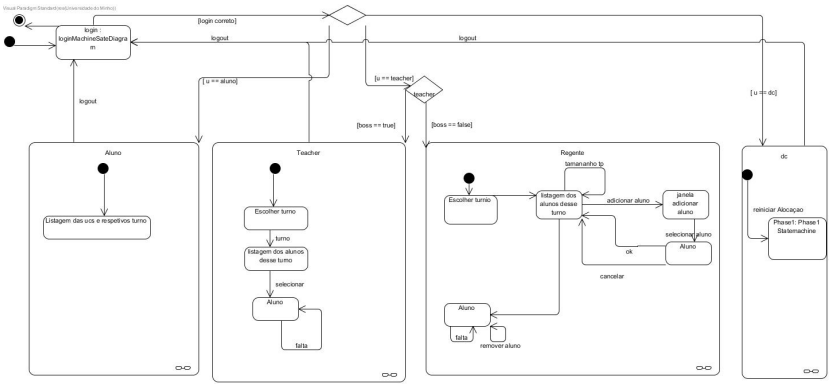
Note-se que o login é uma máquina de estado à parte, devido a ser um elemento comum às três fases sem nenhum tipo de modificação. É a porta para a aplicação e o motor de verificação das entidades que acedem à aplicação.



Máquina de Estado para o login



Máquina de Estado para a segunda fase

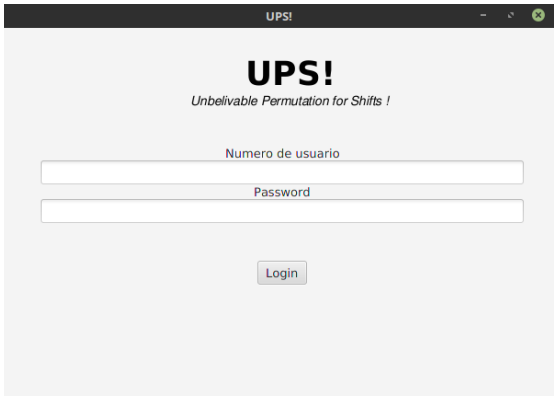


Máquina de Estado para a terceira fase

4.1 Comparação com mockup inicial

Esta secção destina-se à comparação com a nossa expectativa inicial do que seria a interface e de como realmente ficou implementada no final. Verifica-se que a nossa estimativa estava muito perto do real, já que JavaFX nos proporcionou com uma GUI muito parecida com aquela que estávamos à procura.

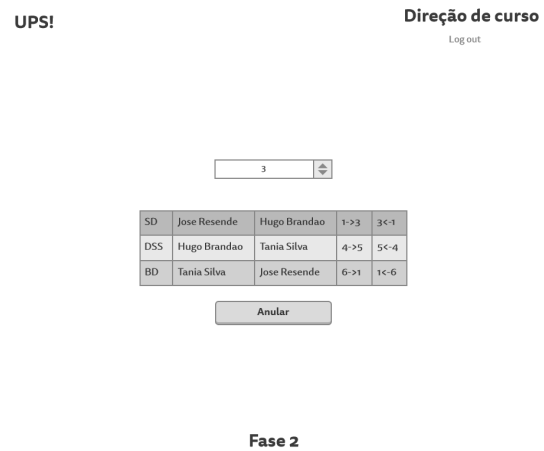
Vista de Login Mockup



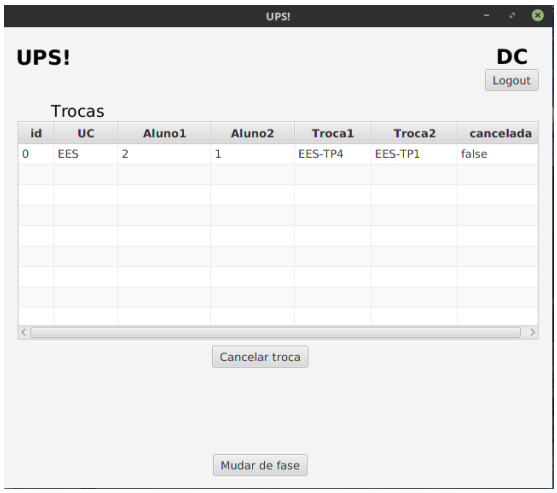
Vista de Login Implementada

Vista de Fase 1 DC Mockup

Vista de Fase 1 DC Implementada



Vista de Fase 2 DC Mockup

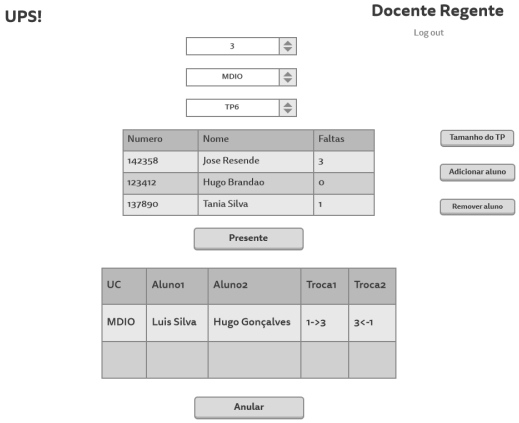


Vista de Fase 2 DC Implementada

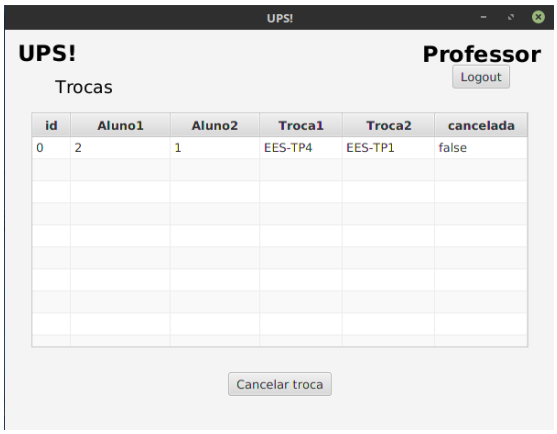
Não criamos um mockup para a terceira fase para a DC, mas isso depois provou-se necessário devido ao reset entre semestres.

Vista de Fase 3 DC Implementada

Esta mockup teve de ser fragmentada, já que juntava features de várias fases.

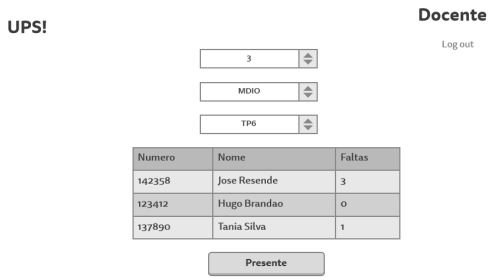


Vista do Professor Regente Mockup



Vista do Professor Regente Fase 2 Implementada

Vista do Professor Regente Fase 3 Implementada



Vista do Professor Mockup

Vista do Professor Implementada

Vista do Aluno Mockup

Vista do Aluno Implementada

5 Conclusão

Em suma, a realização deste trabalho foi demorada e um desafio, já que modelar o programa antes de o implementar é um paradigma completamente novo. Conseguimos ver que a definição dos Use Cases nos dá um *head start* na hora de saber o que precisamos para a realização do projeto e de que diferentes cenários vamos poder ter na aplicação num determinado momento.

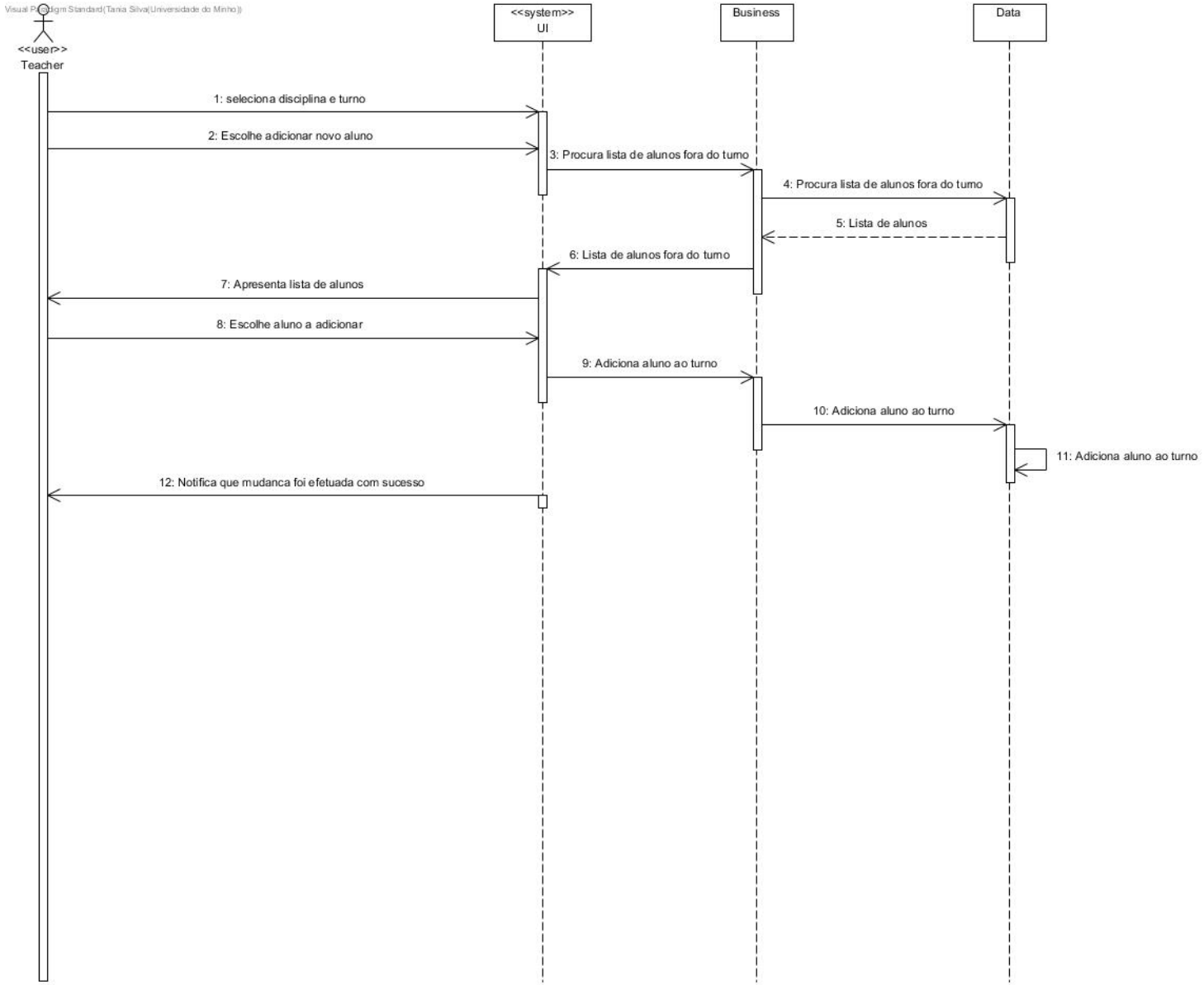
Contudo, nem todo o desenvolvimento foi bem sucedido, não conseguindo realizar um dos Use cases nem as trocas n-árias. Um dos nossos principais objetivos era conseguir modelar num Diagrama de Sequência de Sistemas uma troca n-ária, mas a informação que possuíamos mostrou não ser suficiente, e mesmo tentando implementar sem modelação, a tarefa de representar UCs como grafos, Turnos como nodos e Pedidos como Arestas, dificultava, em termos de implementação, a compreensão da solução e a conceção da mesma. Desta forma, e também por falta de tempo, apontamos este objetivo não alcançado como a farpa no nosso calcanhar de Aquiles.

Da mesma forma também não nos foi possível implementar a criação automática de turnos, apesar de termos uma alocação automática dos alunos pelos turnos pré-criados. Tendo a base técnica já efetuada mas com *deadlines* demasiado apertadas, acrescentado a outros requisitos do trabalho, fomos obrigados a definir prioridades do projeto de maneira a ter o melhor aproveitamento possível.

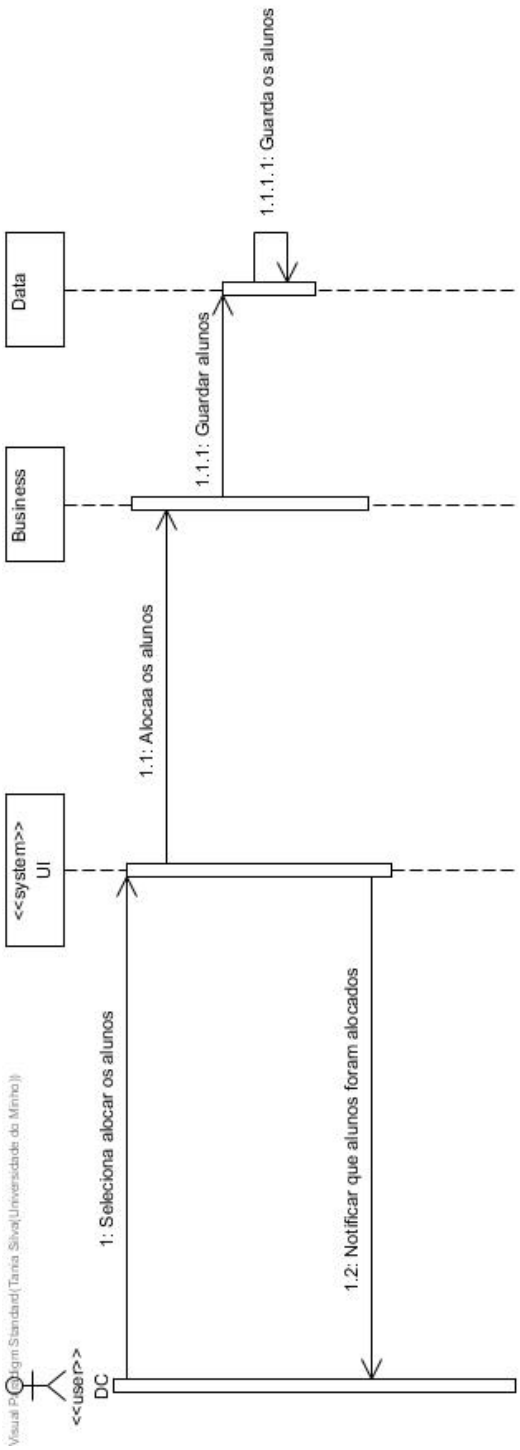
No entanto, gostaríamos de no futuro conseguir reformular o código de maneira a que trocas n-árias sejam possíveis de acontecer na nossa plataforma, de maneira a que a Direção de Curso consiga usufruir desse algoritmo e que o implemente na plataforma que atualmente utiliza.

Acreditamos, então, que o nosso desempenho foi para além das nossas expectativas, mas aquém dos nossos objetivos, absorvendo todo o conhecimento possível em todos os passos do processo.

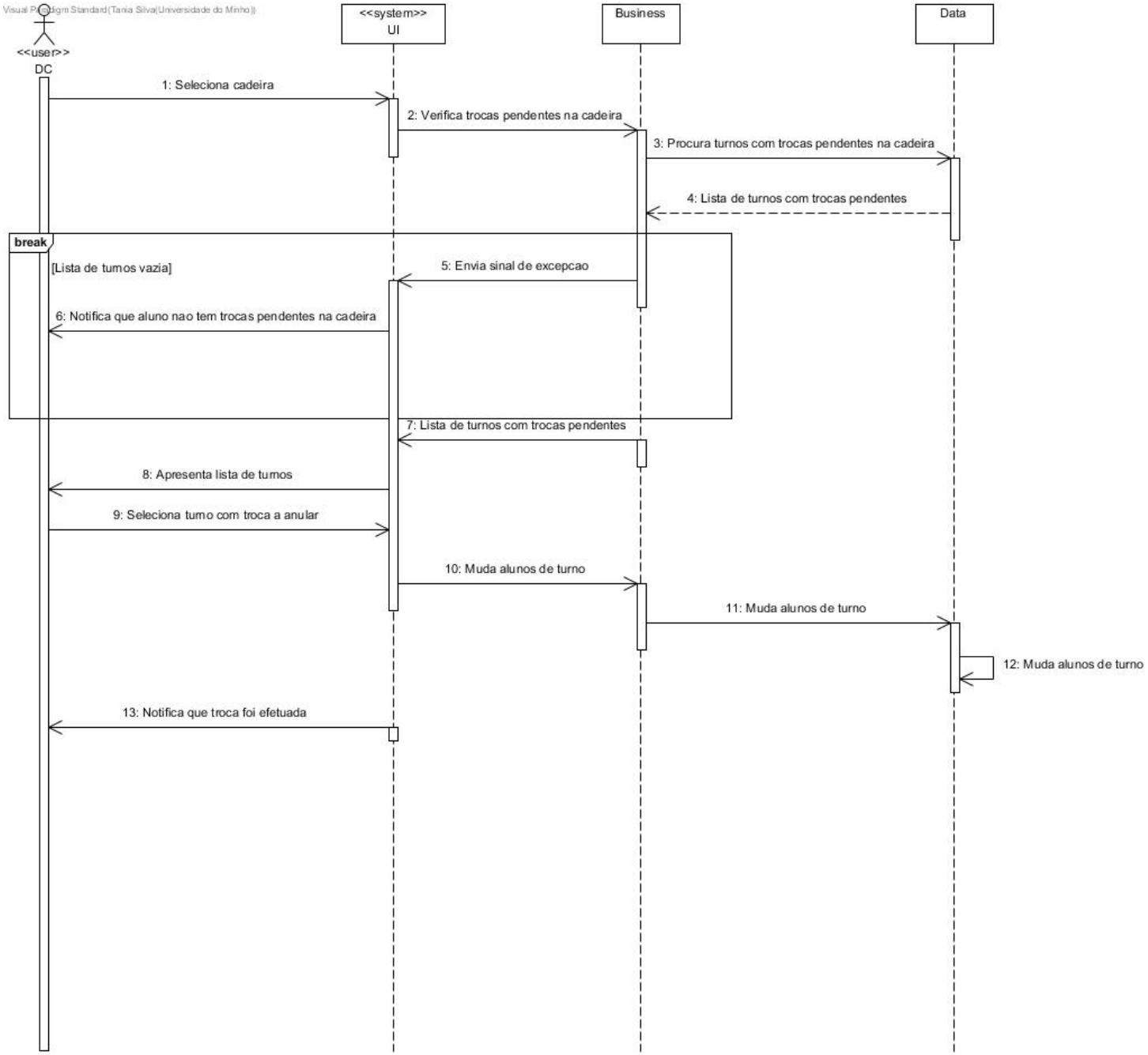
6 Anexos



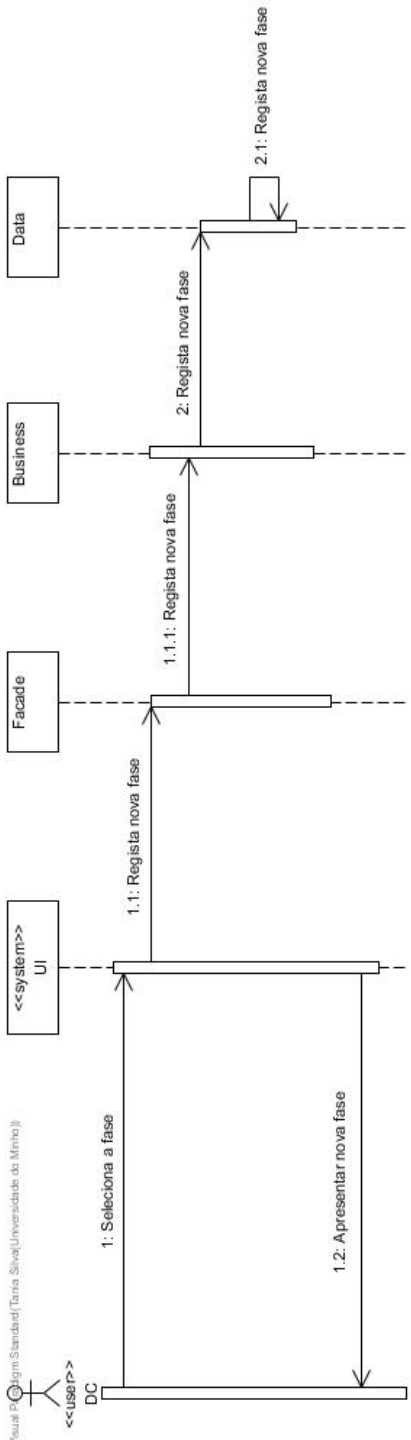
DSS - Adicionar um Aluno a um Turno



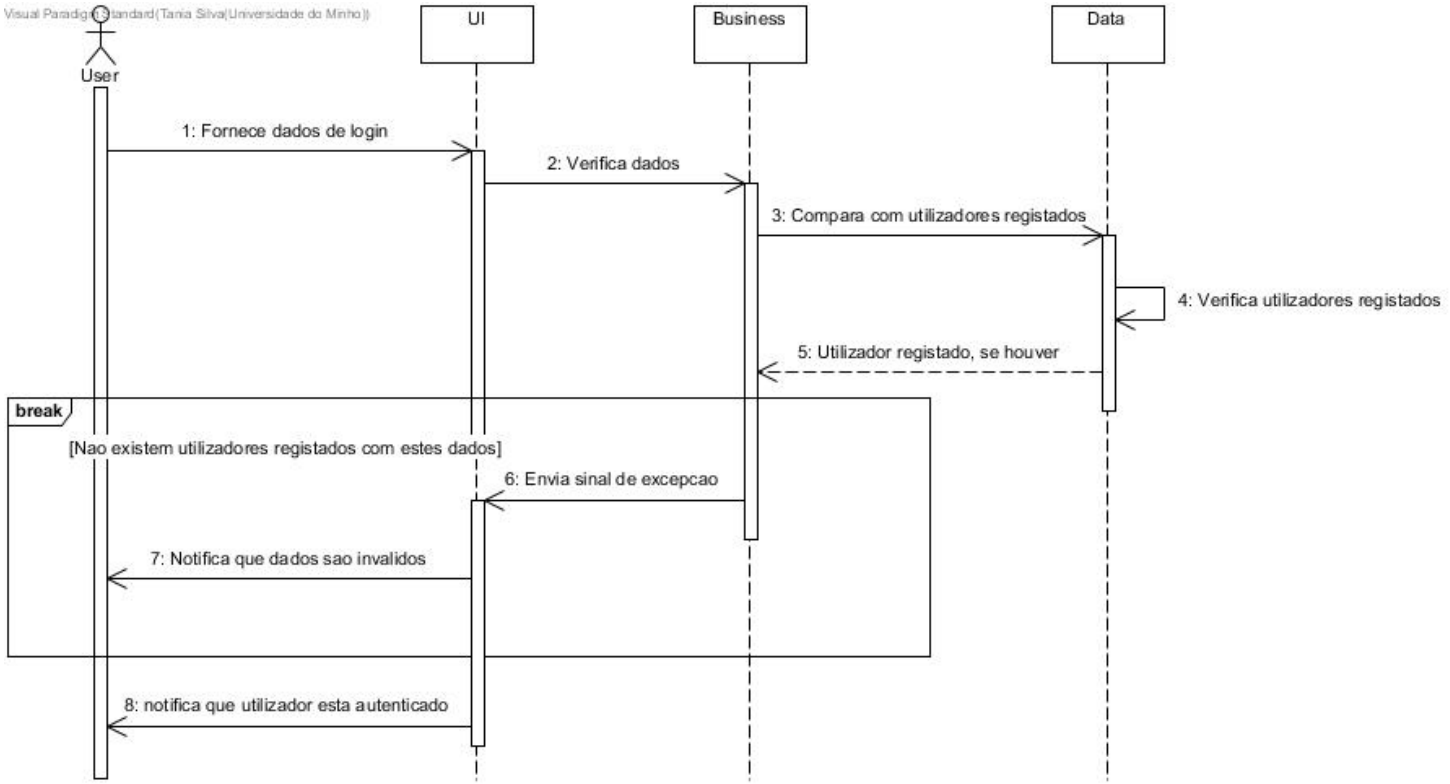
DSS - Alocação de Alunos



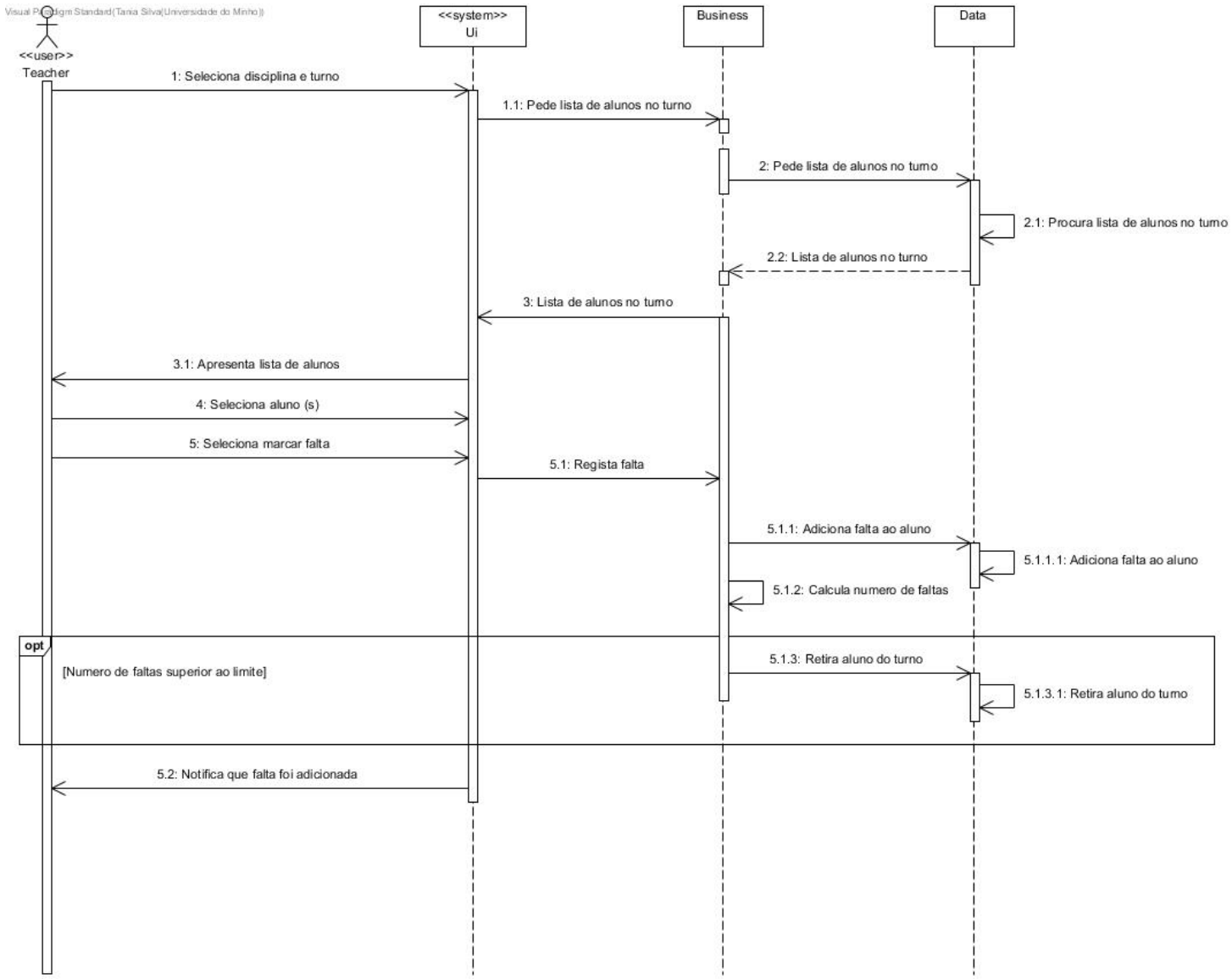
DSS - Anular uma Troca



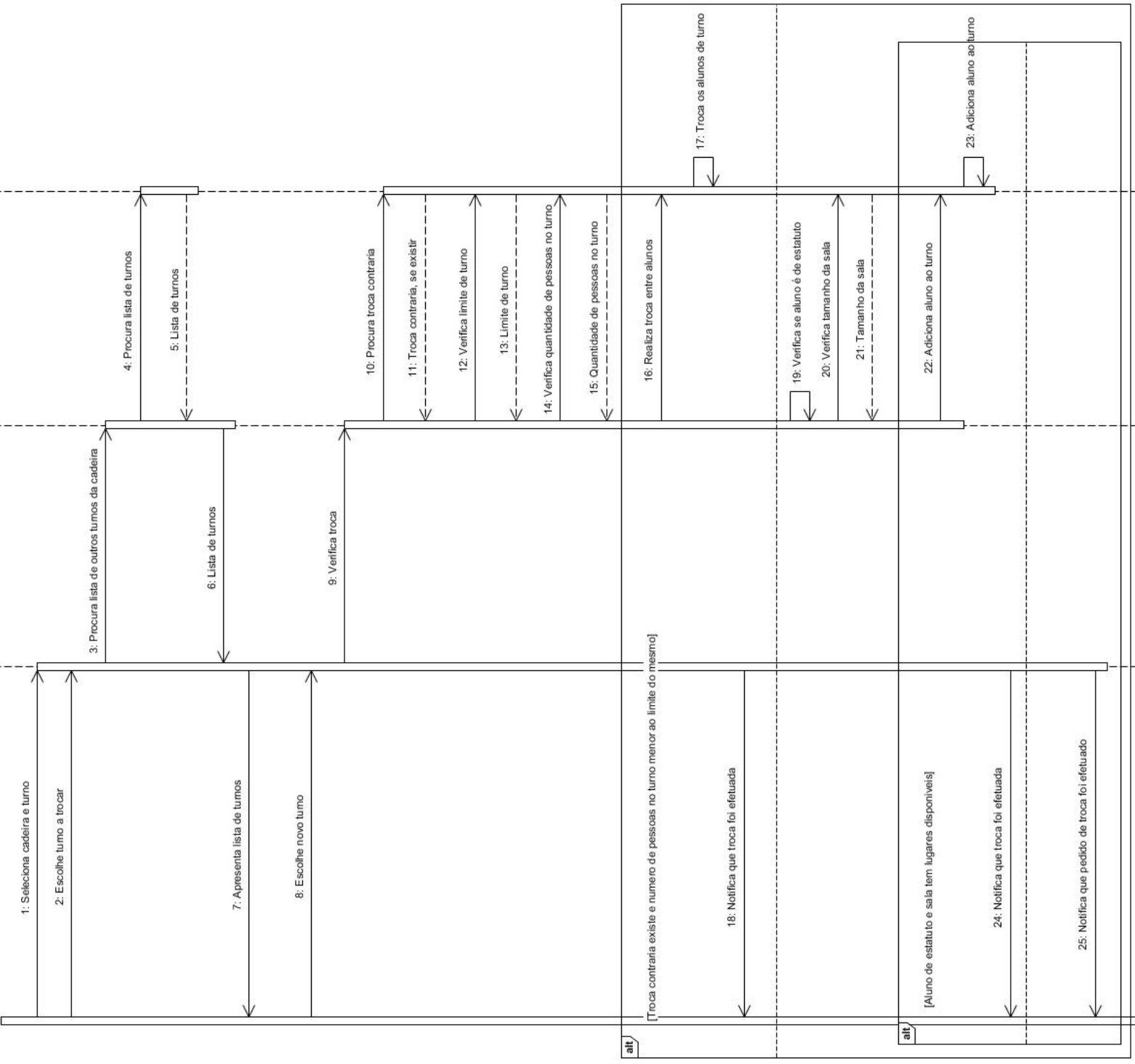
DSS - Definir a fase

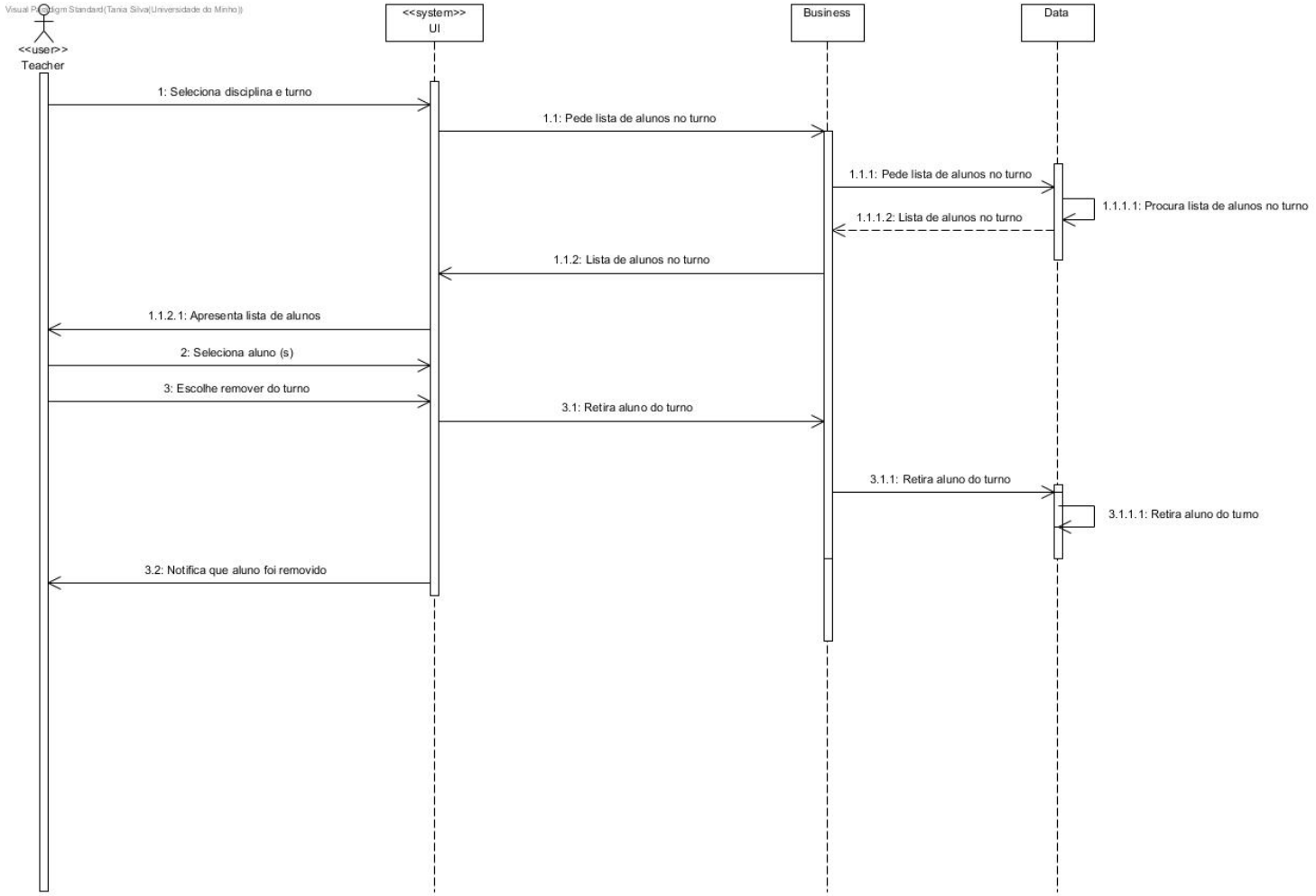

Visual Paradigm Standard(Tania Silva(Universidade do Minho))
User
UI
Business
Data
1: Fornece dados de login
2: Verifica dados
3: Compara com utilizadores registados
4: Verifica utilizadores registados
5: Utilizador registado, se houver
break
[Nao existem utilizadores registados com estes dados]
6: Envia sinal de excepcao
7: Notifica que dados sao invalidos
8: notifica que utilizador esta autenticado

DSS - Login

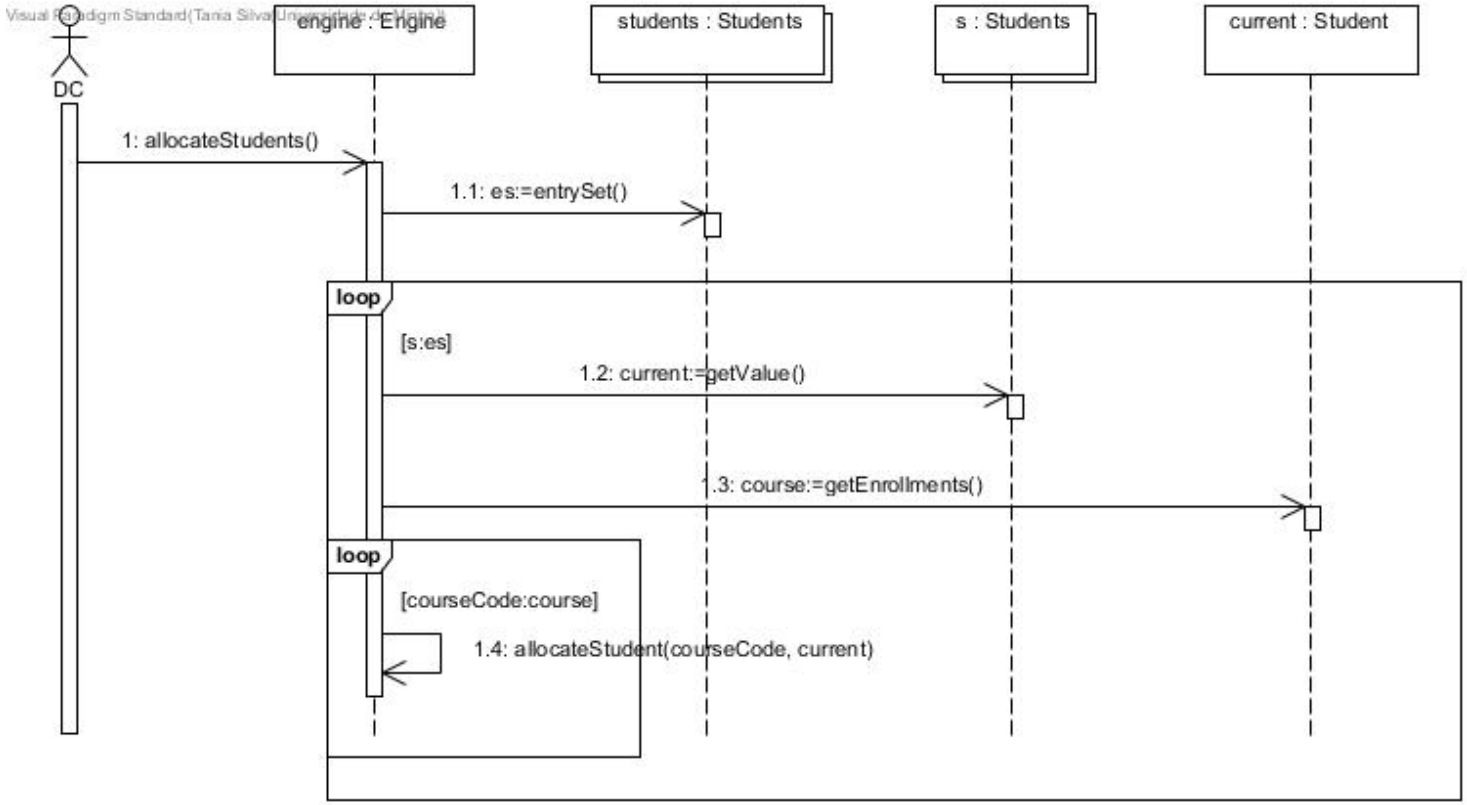


DSS - Marcar uma falta

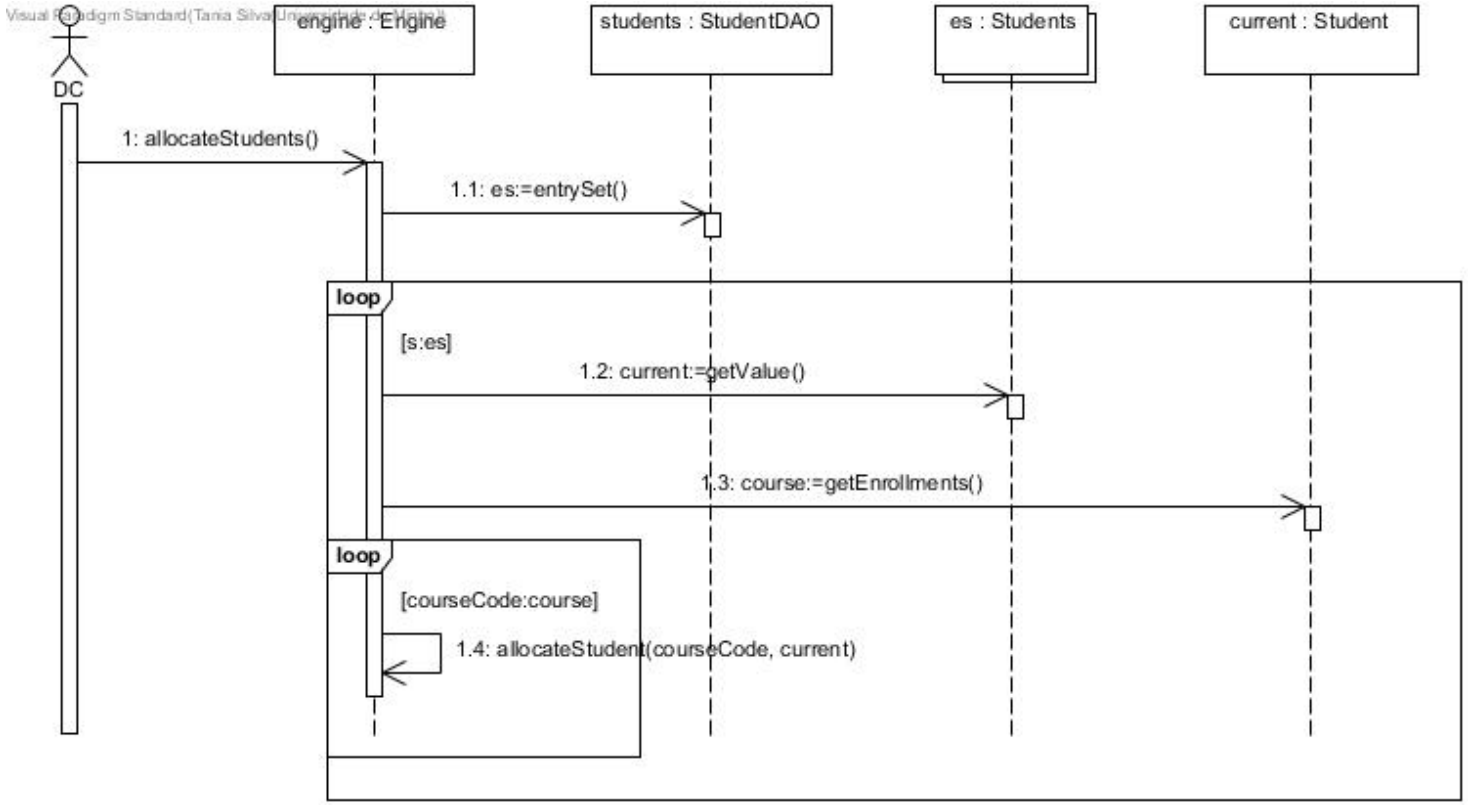

1: Seleciona cadeira e turno
2: Escolhe turno a trocar
3: Procura lista de outros turnos da cadeira
4: Procura lista de turnos
5: Lista de turnos
6: Lista de turnos
7: Apresenta lista de turnos
8: Escolhe novo turno
9: Verifica troca
10: Procura troca contraria
11: Troca contraria, se existir
12: Verifica limite de turno
13: Limite de turno
14: Verifica quantidade de pessoas no turno
15: Quantidade de pessoas no turno
16: Realiza troca entre alunos
17: Troca os alunos de turno
18: Notifica que troca foi efetuada
19: Verifica se aluno é de estatuto
20: Verifica tamanho da sala
21: Tamanho da sala
22: Adiciona aluno ao turno
23: Adiciona aluno ao turno
24: Notifica que troca foi efetuada
25: Notifica que pedido de troca foi efetuado
[Troca contraria existe e numero de pessoas no turno menor ao limite do mesmo]
[Aluno de estatuto e sala tem lugares disponiveis]
alt
alt



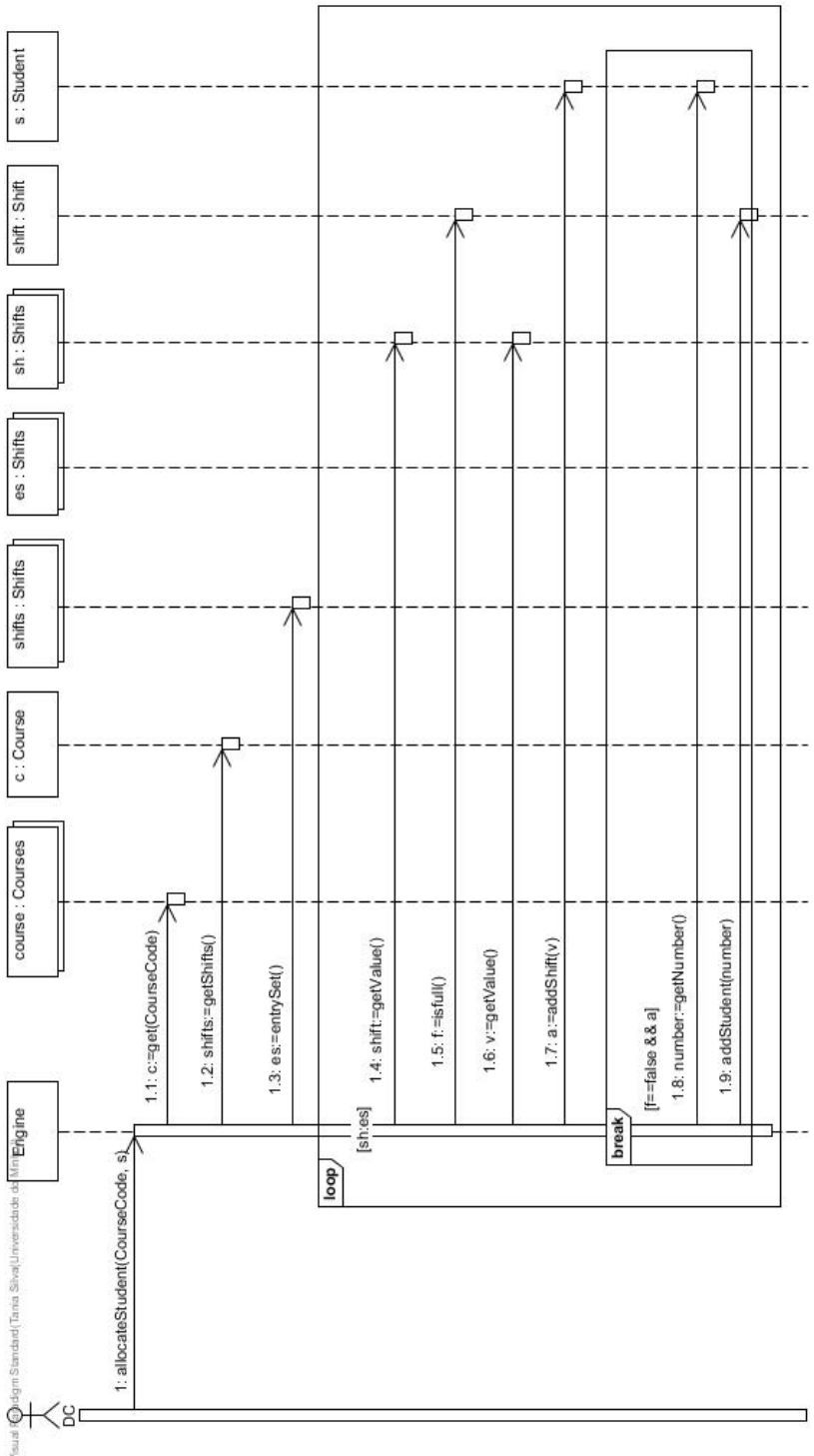
DSS - Remover um Aluno de um Turno



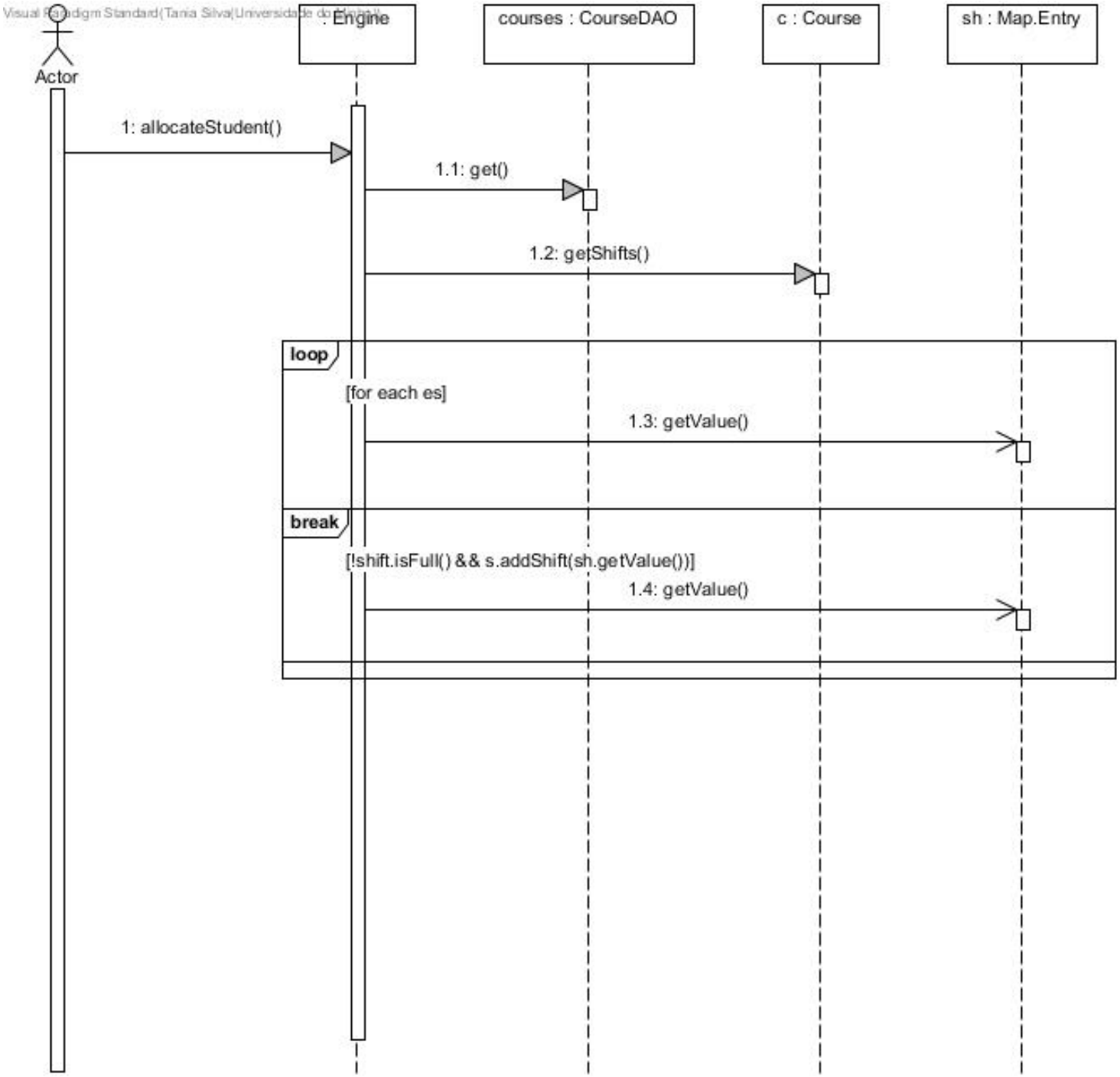
allocateStudents



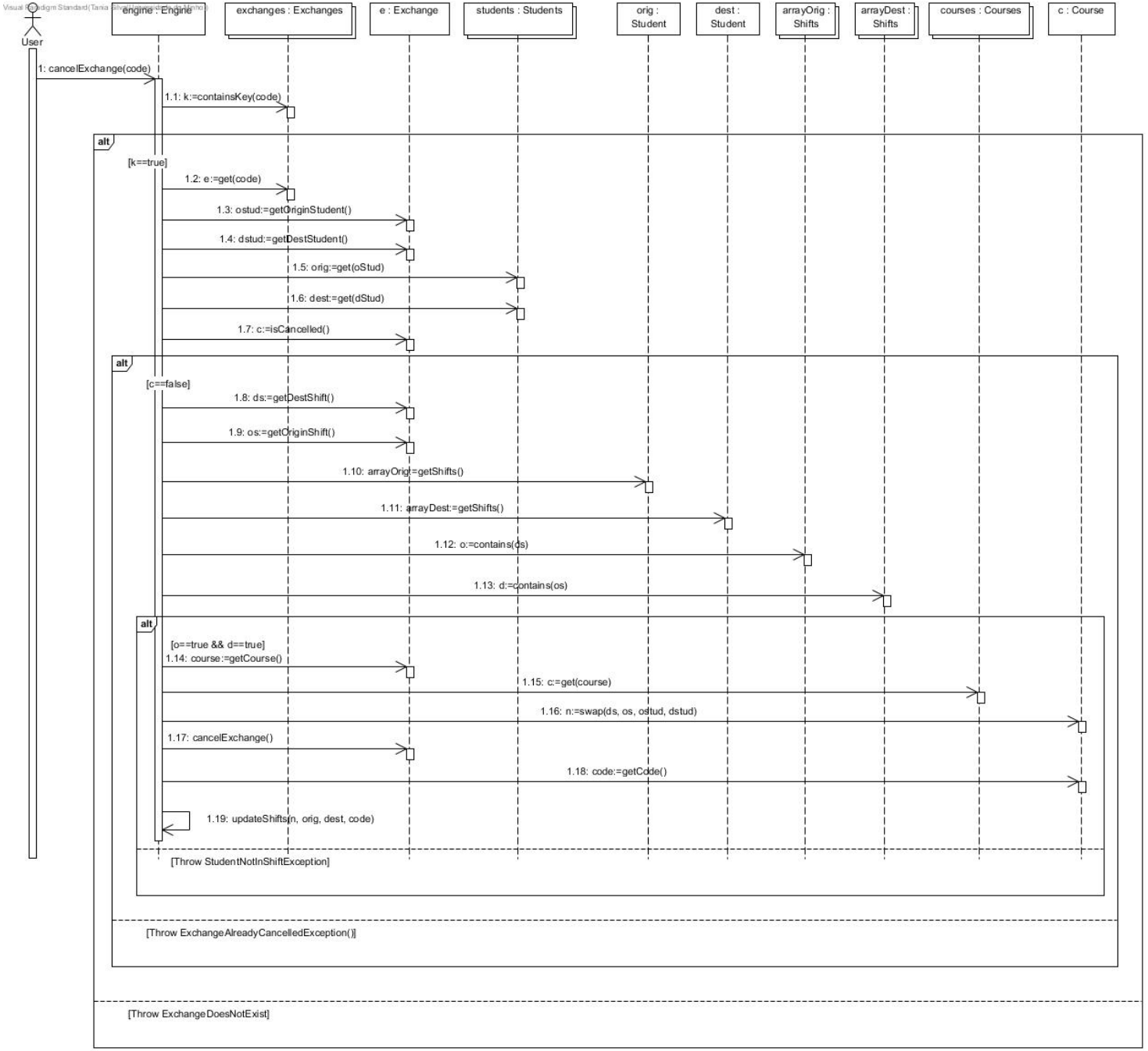
allocateStudents 2.0



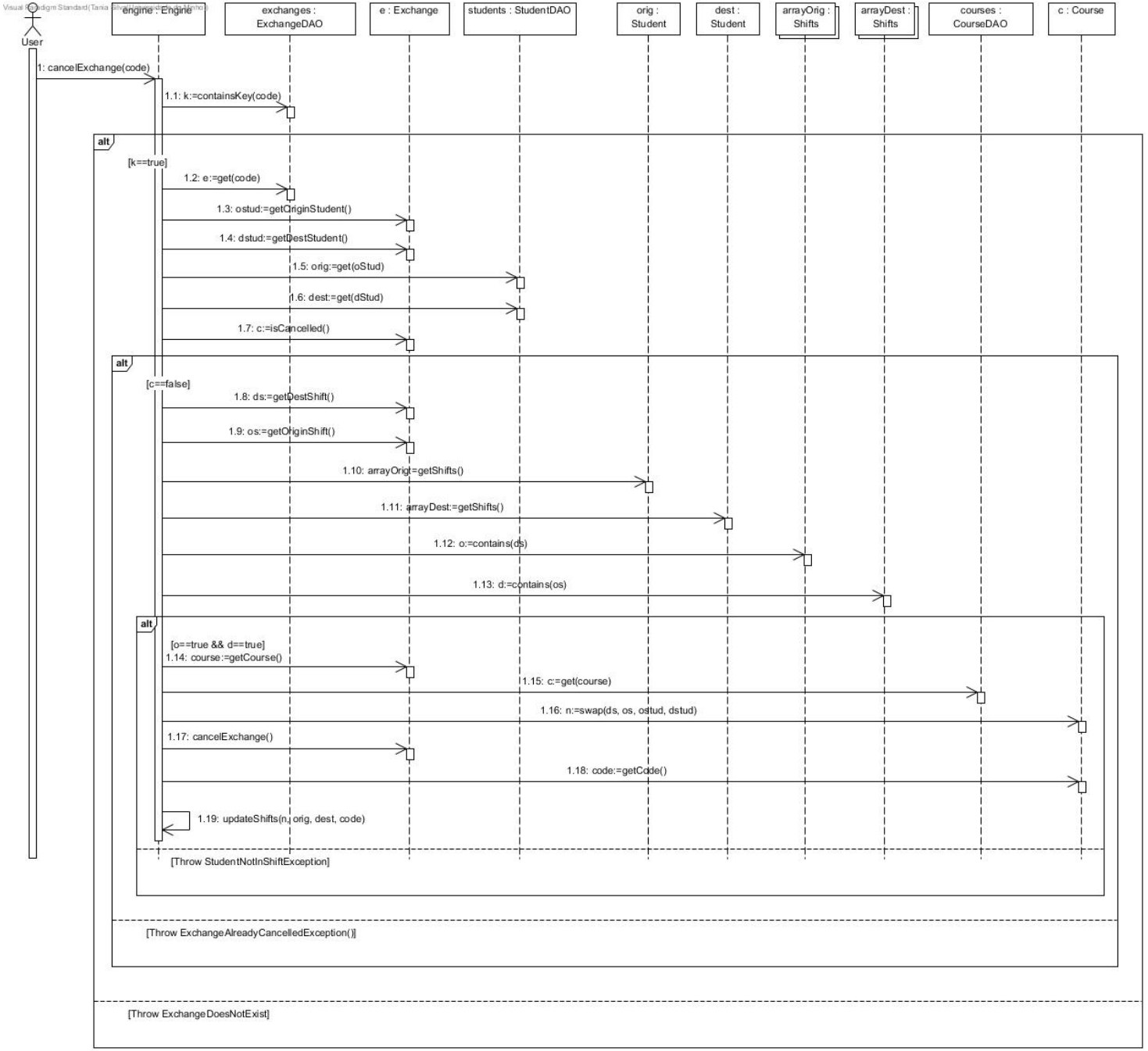
allocateStudent



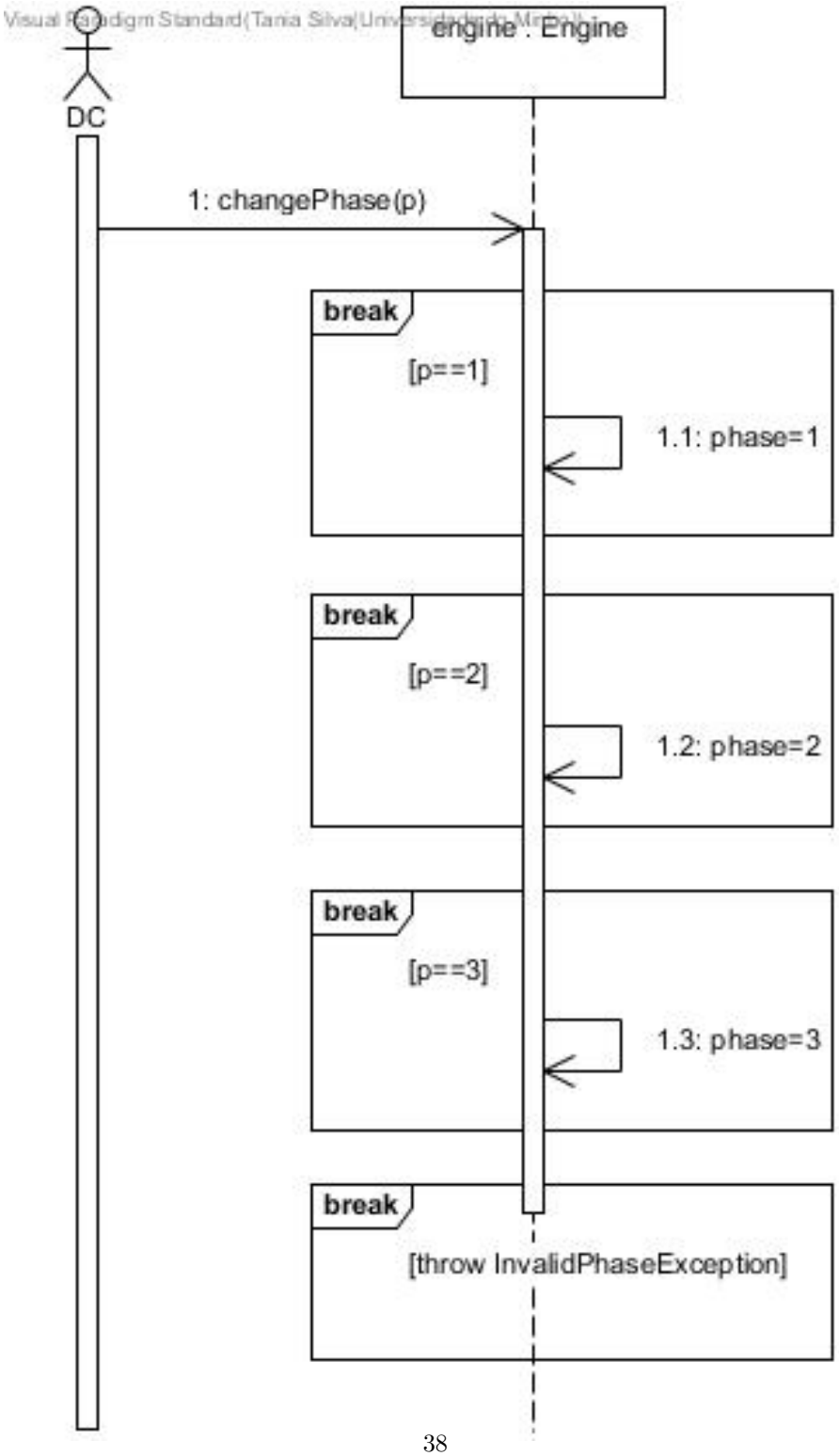
allocateStudent 2.0



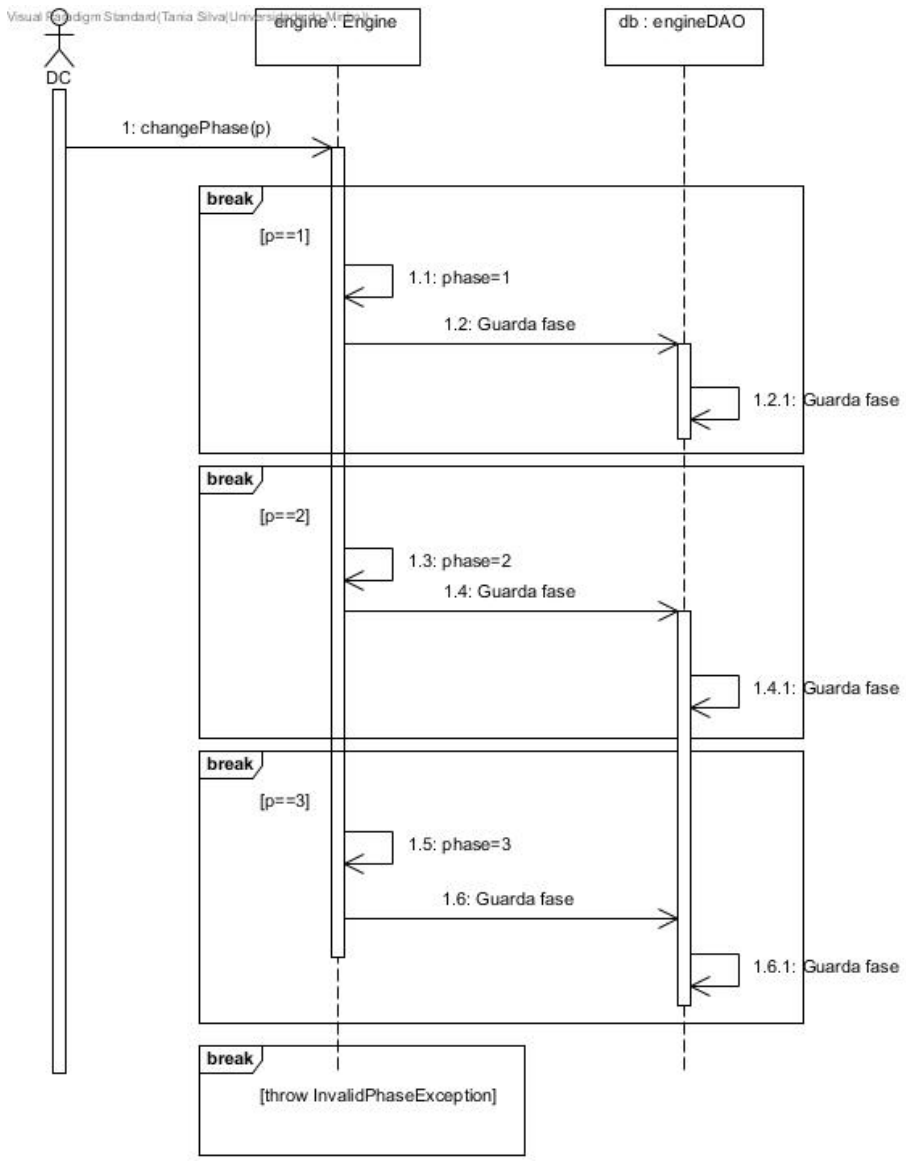
cancelExchange



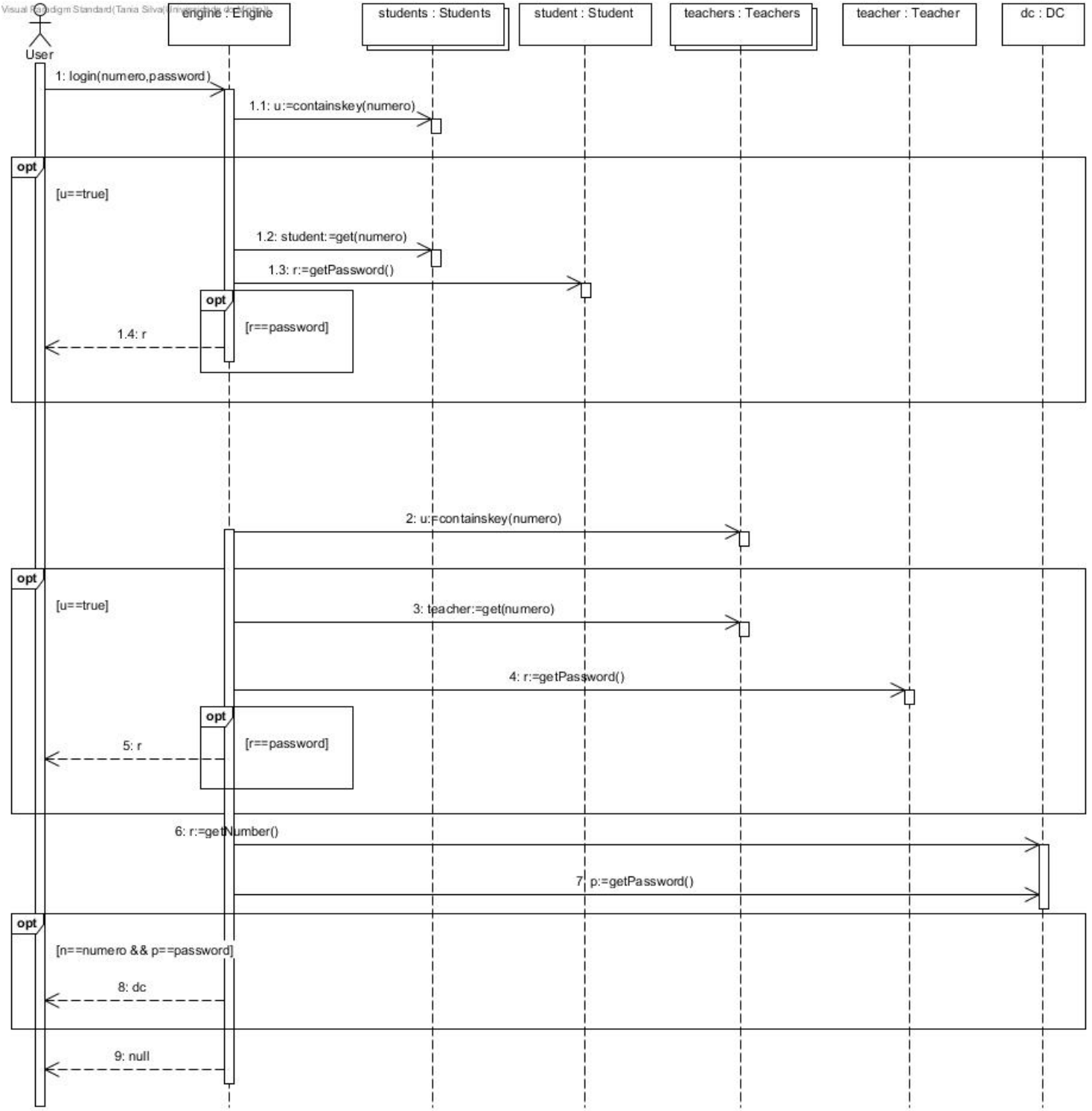
cancelExchange 2.0



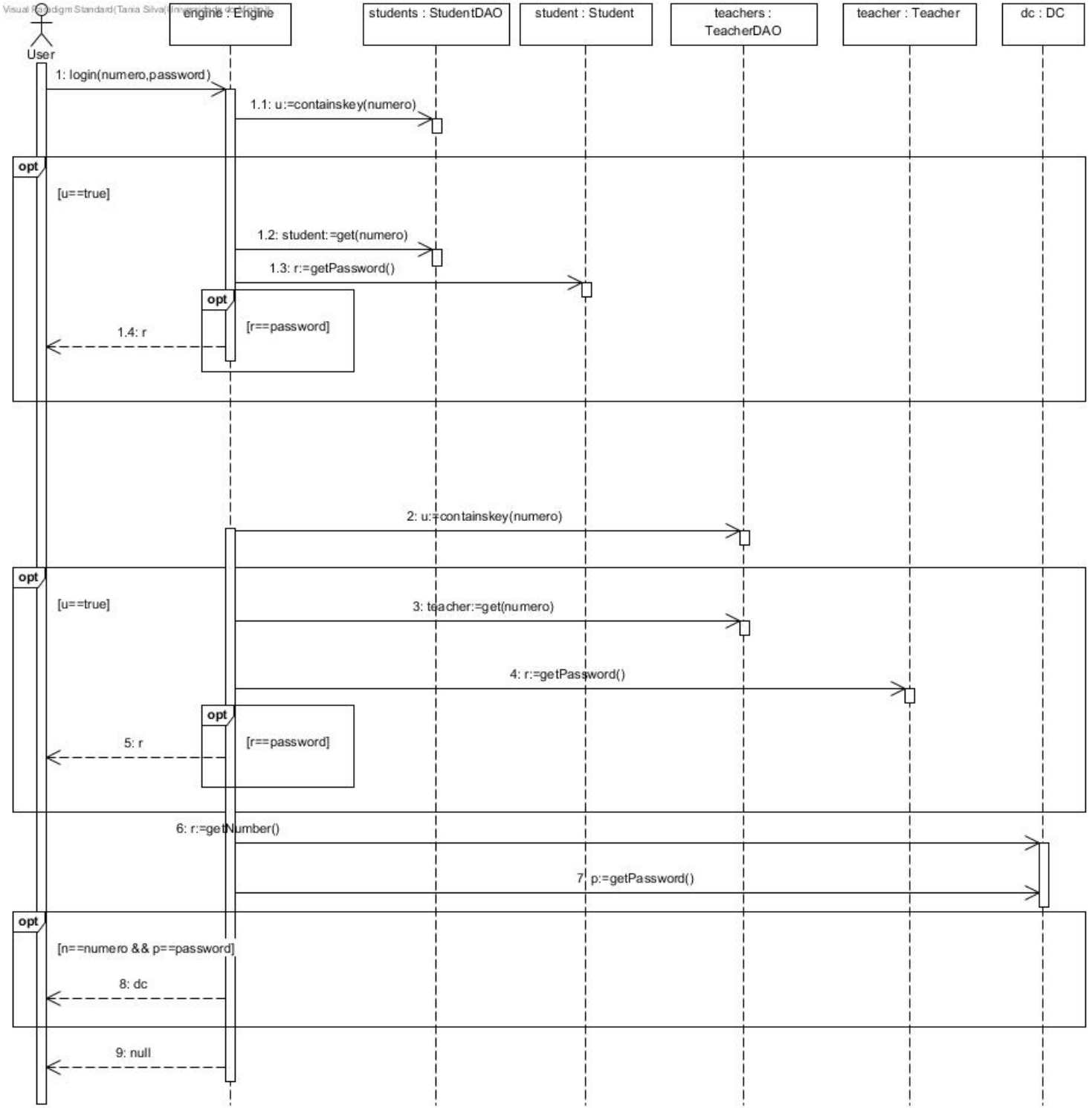
changePhase



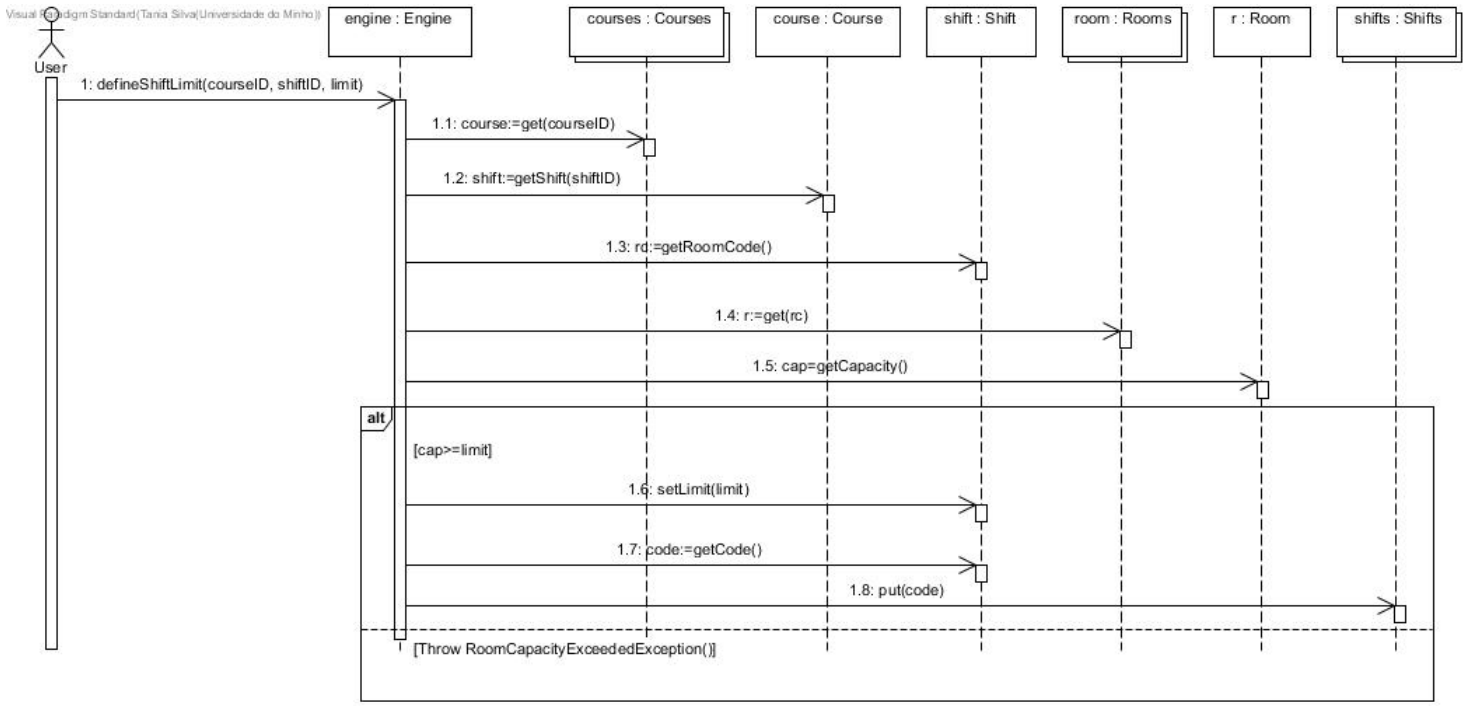
changePhase 2.0


User
engine : Engine
students : Students
student : Student
teachers : Teachers
teacher : Teacher
dc : DC
1: login(numero,password)
1.1: u:=containskey(numero)
opt
[u==true]
1.2: student:=get(numero)
1.3: r:=getPassword()
opt
[r==password]
1.4: r
2: u:=containskey(numero)
opt
[u==true]
3: teacher:=get(numero)
4: r:=getPassword()
opt
[r==password]
5: r
6: r:=getNumber()
7: p:=getPassword()
opt
[n==numero && p==password]
8: dc
9: null

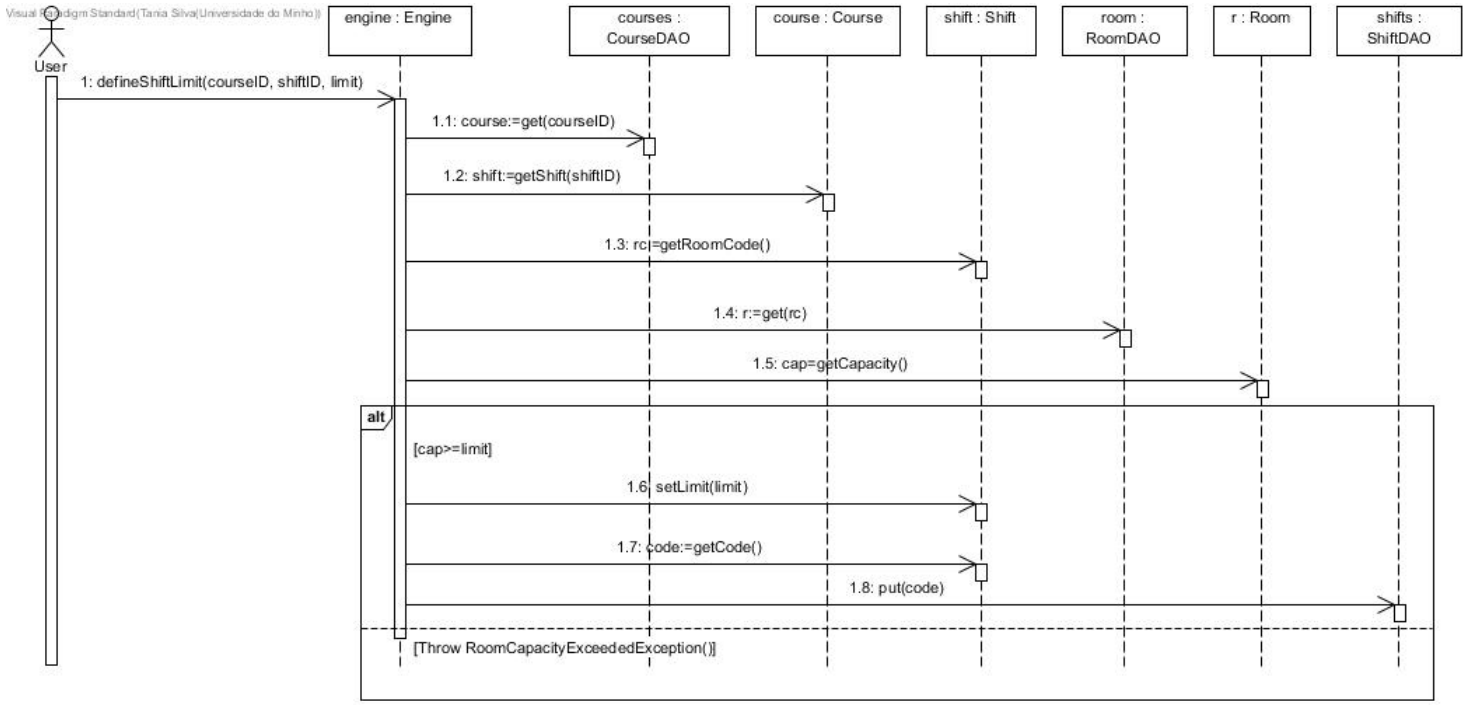
login


User
engine : Engine
students : StudentDAO
student : Student
teachers : TeacherDAO
teacher : Teacher
dc : DC
1: login(numero,password)
1.1: u:=containskey(numero)
opt
[u==true]
1.2: student:=get(numero)
1.3: r:=getPassword()
opt
[r==password]
1.4: r
2: u:=containskey(numero)
opt
[u==true]
3: teacher:=get(numero)
4: r:=getPassword()
opt
[r==password]
5: r
6: r:=getNumber()
7: p:=getPassword()
opt
[n==numero && p==password]
8: dc
9: null

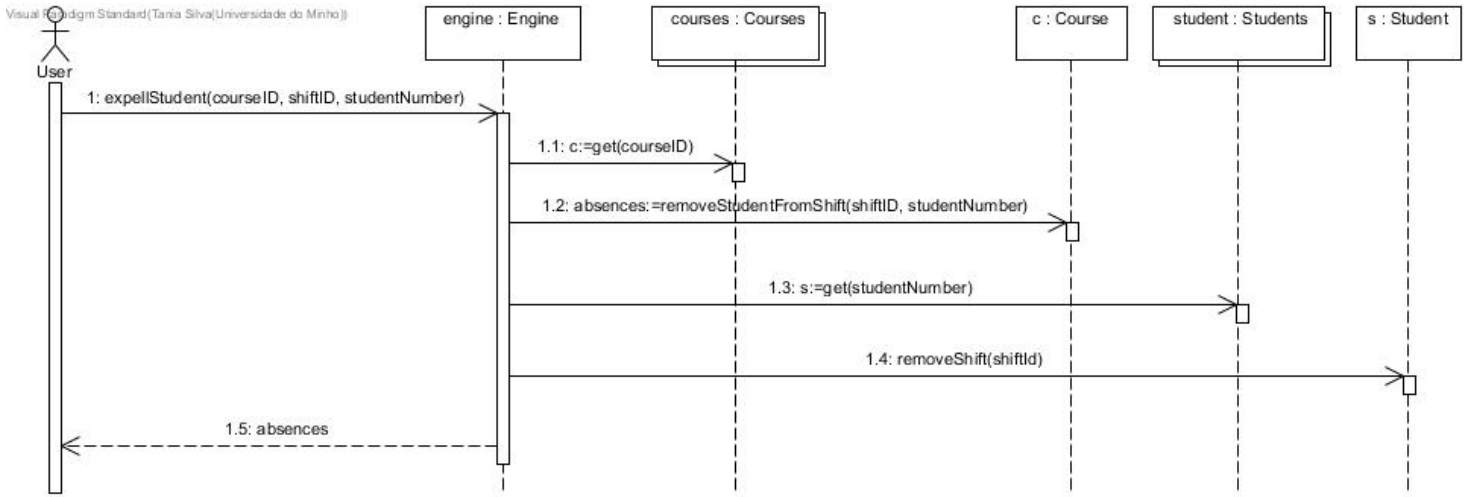
login 2.0



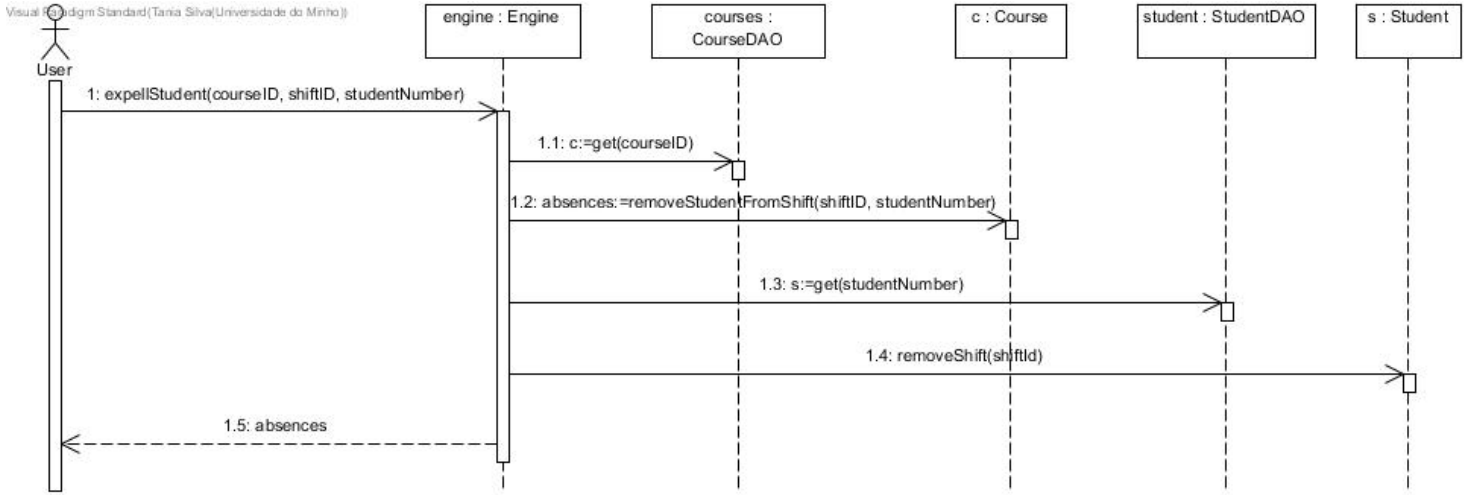
defineShiftLimit



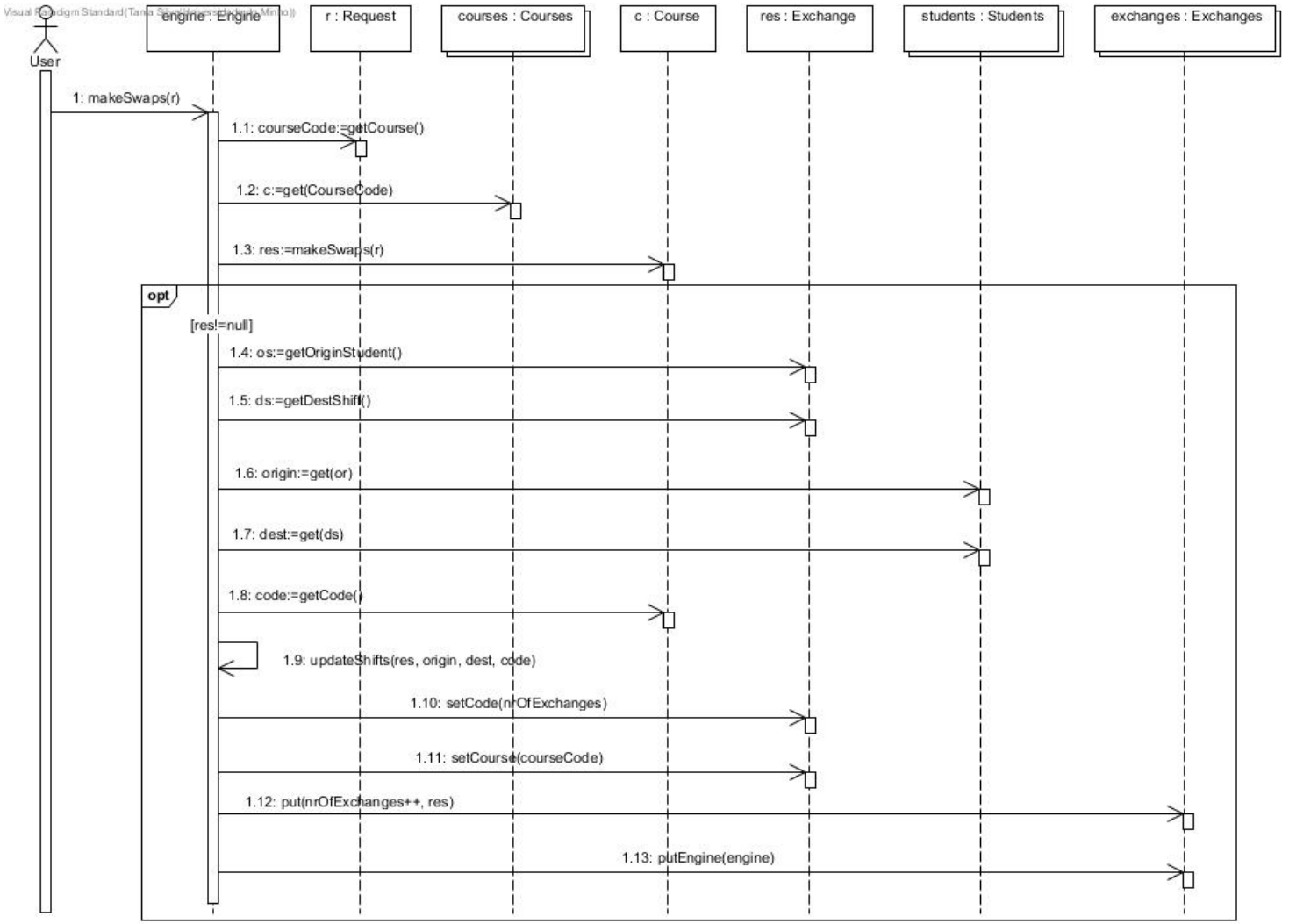
defineShiftLimit 2.0



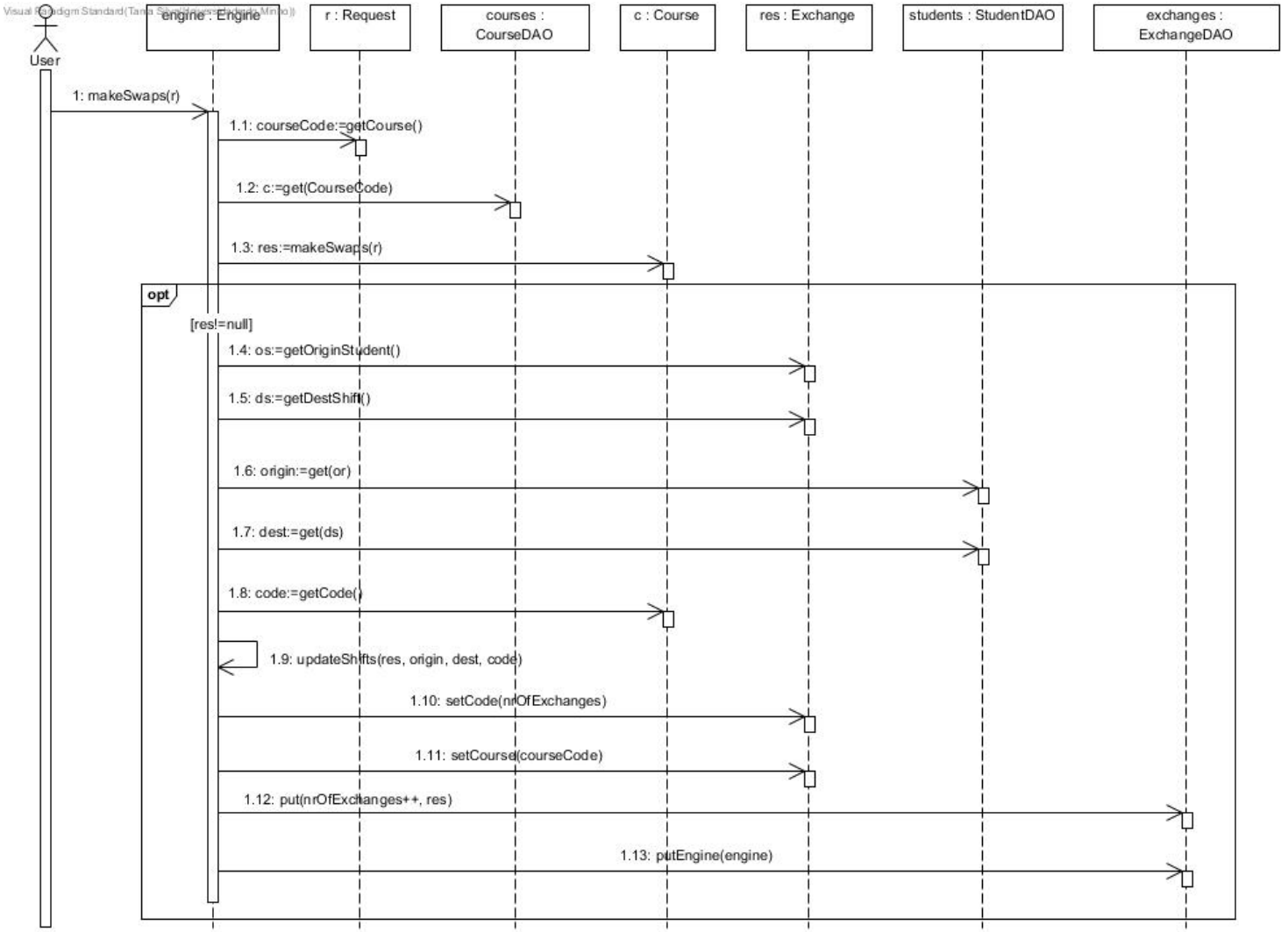
expellStudent



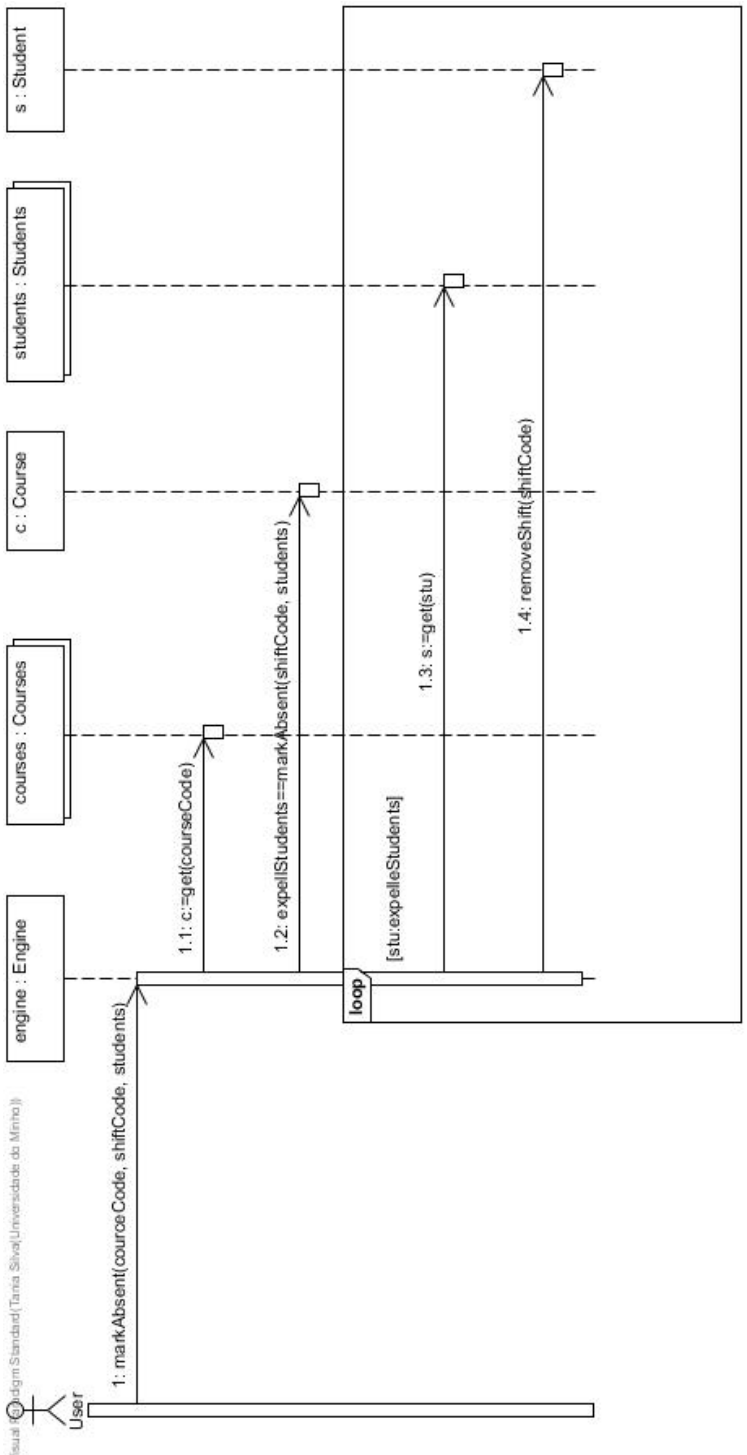
expellStudent 2.0


User
engine : Engine
r : Request
courses : Courses
c : Course
res : Exchange
students : Students
exchanges : Exchanges
1: makeSwaps(r)
1.1: courseCode:=getCourse()
1.2: c:=get(CourseCode)
1.3: res:=makeSwaps(r)
opt
[res!=null]
1.4: os:=getOriginStudent()
1.5: ds:=getDestShift()
1.6: origin:=get(or)
1.7: dest:=get(ds)
1.8: code:=getCode()
1.9: updateShifts(res, origin, dest, code)
1.10: setCode(nrOfExchanges)
1.11: setCourse(courseCode)
1.12: put(nrOfExchanges++, res)
1.13: putEngine(engine)

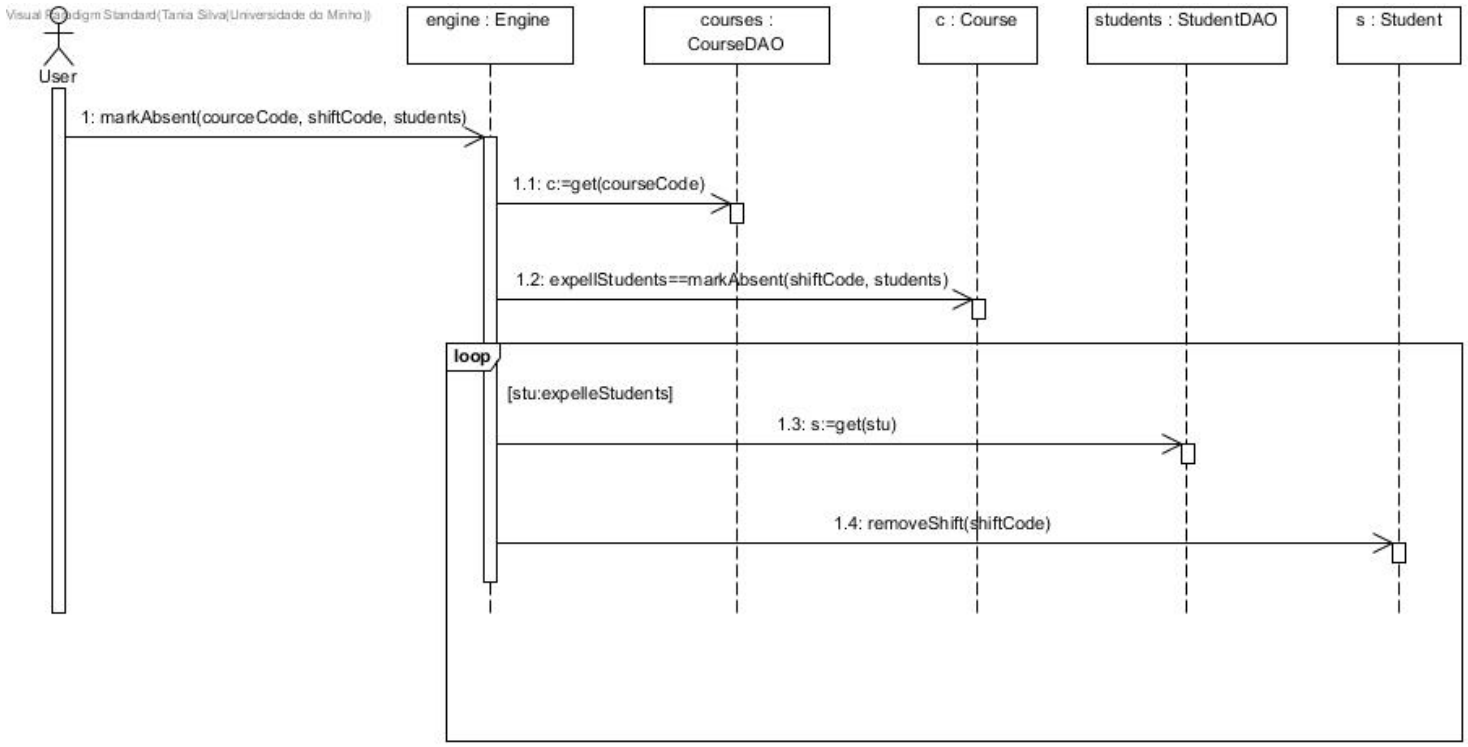
makeSwaps



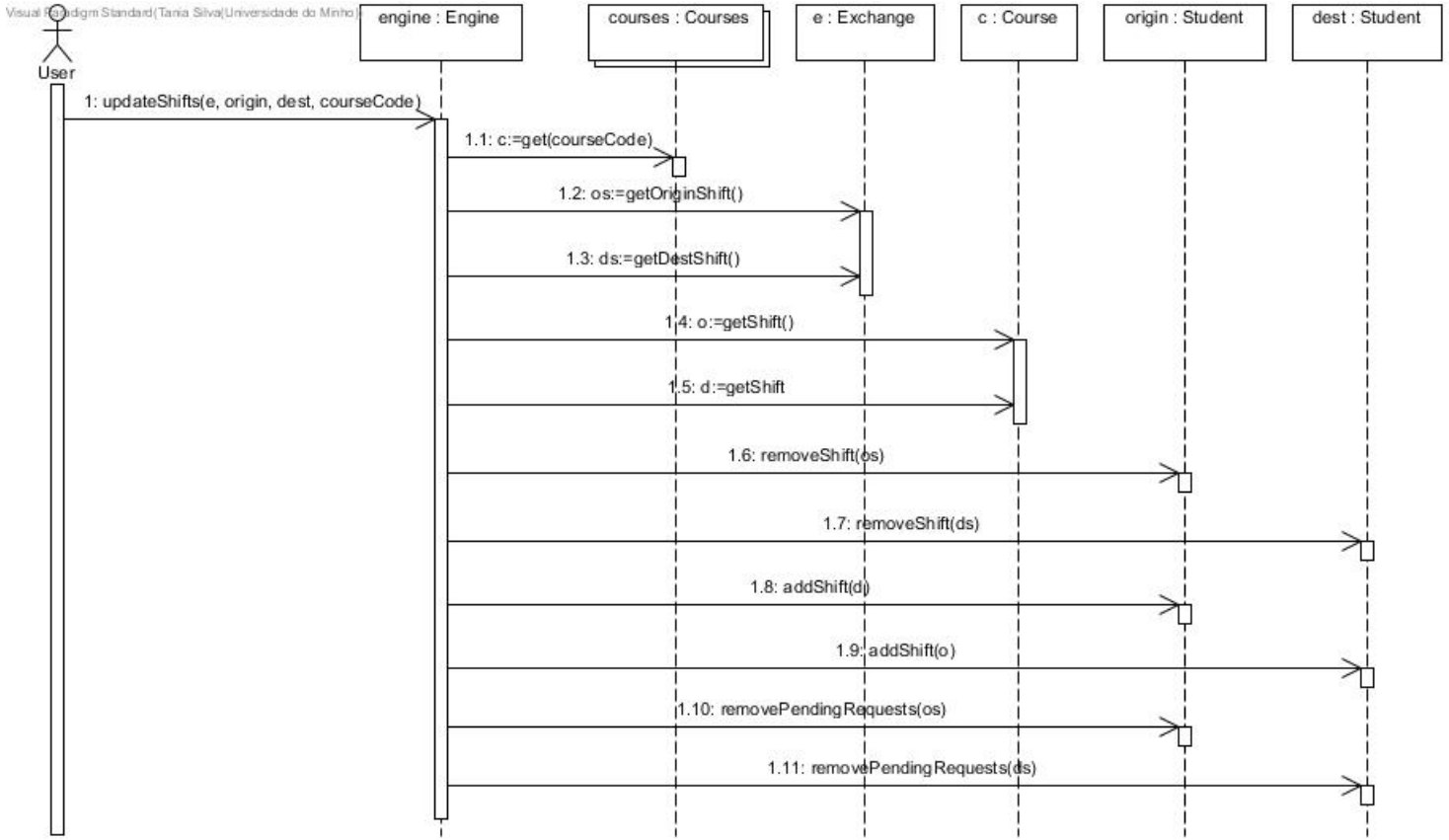
makeSwaps 2.0


User
engine : Engine
courses : Courses
c : Course
students : Students
s : Student
1: markAbsent(courseCode, shiftCode, students)
1.1: c:=get(courseCode)
1.2: expellStudents:=markAbsent(shiftCode, students)
loop
[stu:expelleStudents]
1.3: s:=get(stu)
1.4: removeShift(shiftCode)

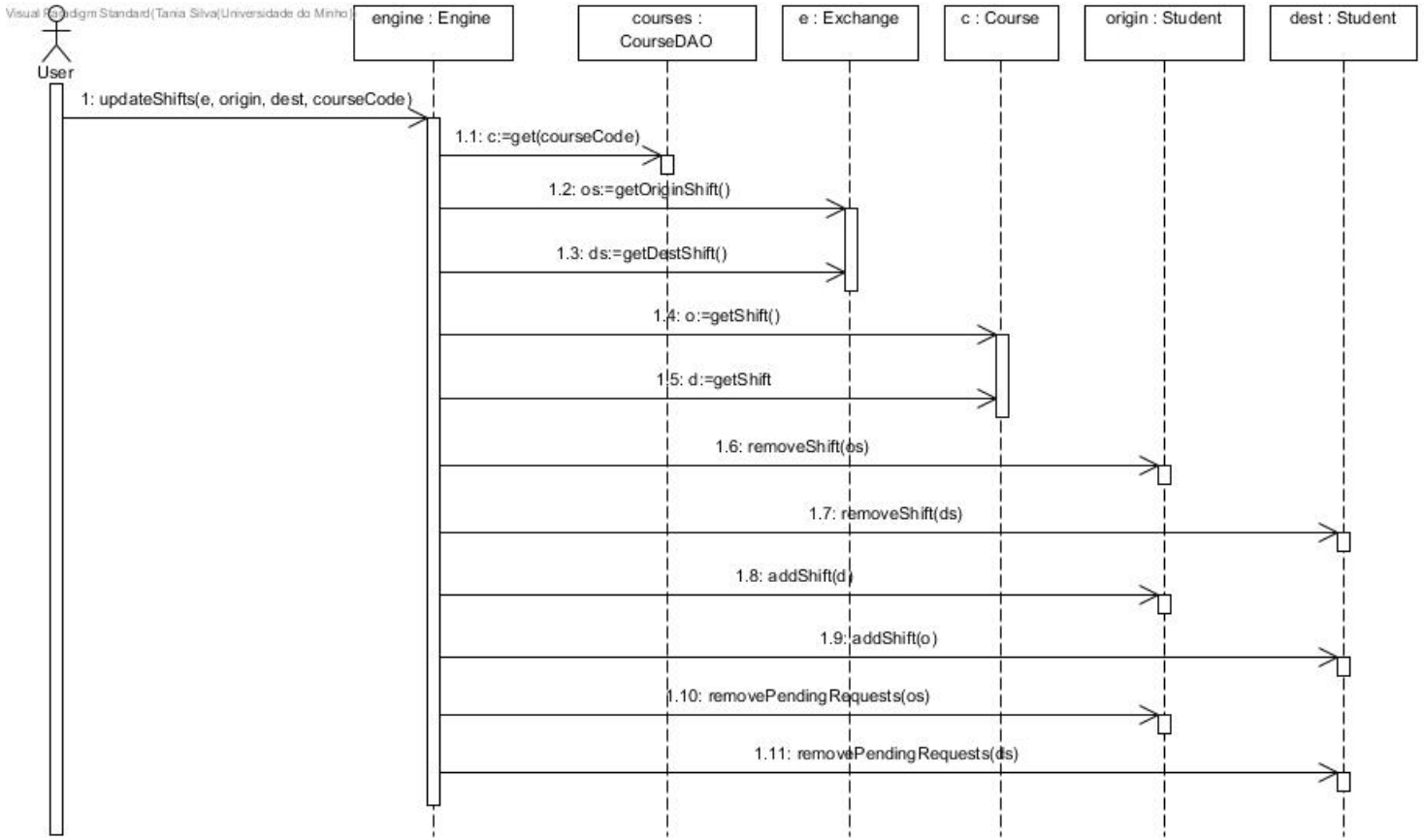
markAbsent



markAbsent 2.0



updateShifts



updateShifts 2.0

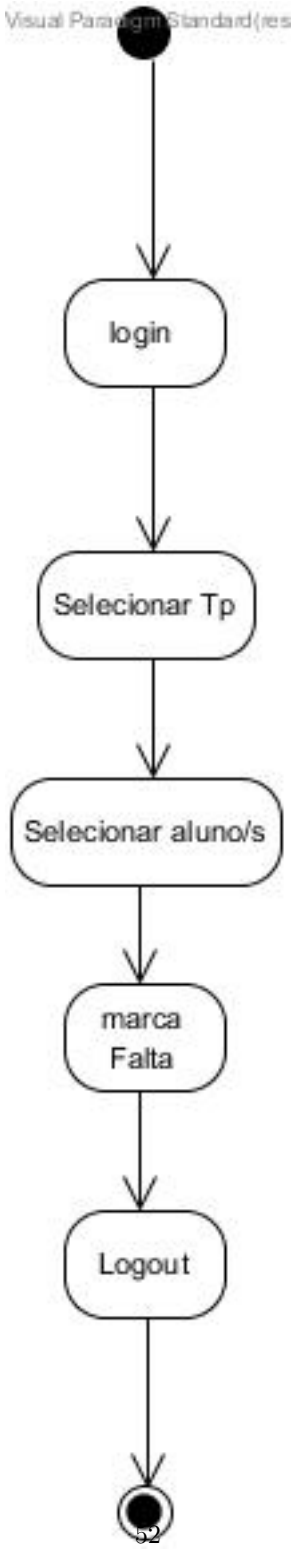


Diagrama de Atividade de Marcar uma Falta a Um ou Vários Alunos

Diagrama de Atividade de Remover um Aluno de um Turno

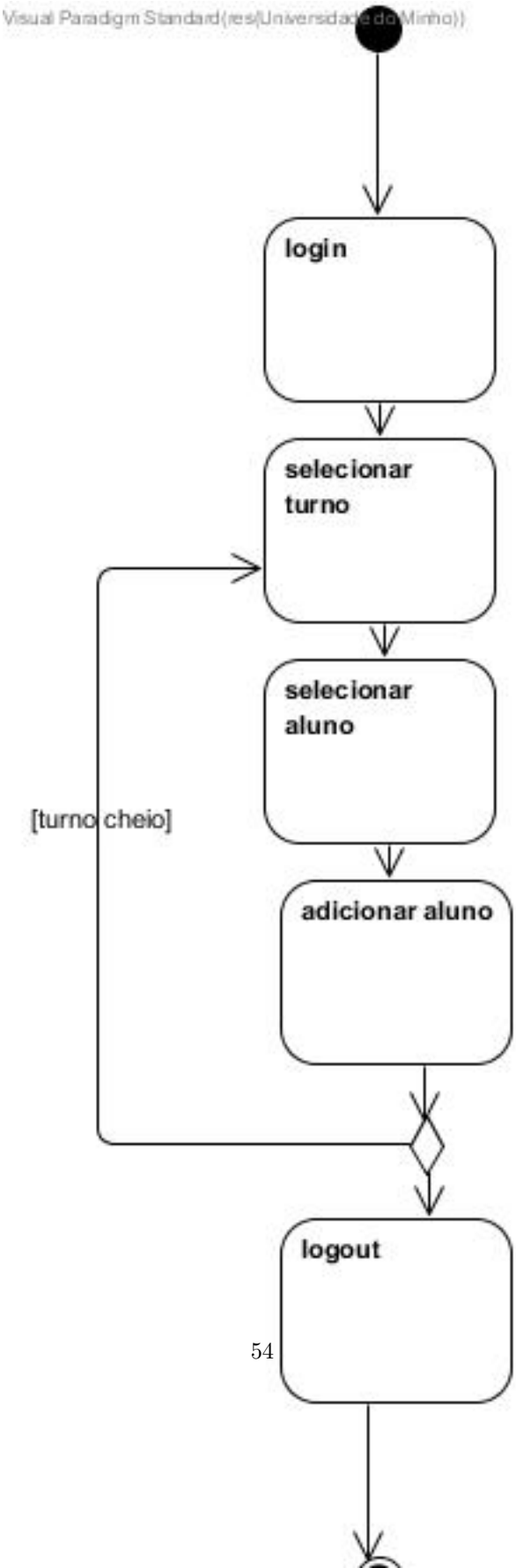
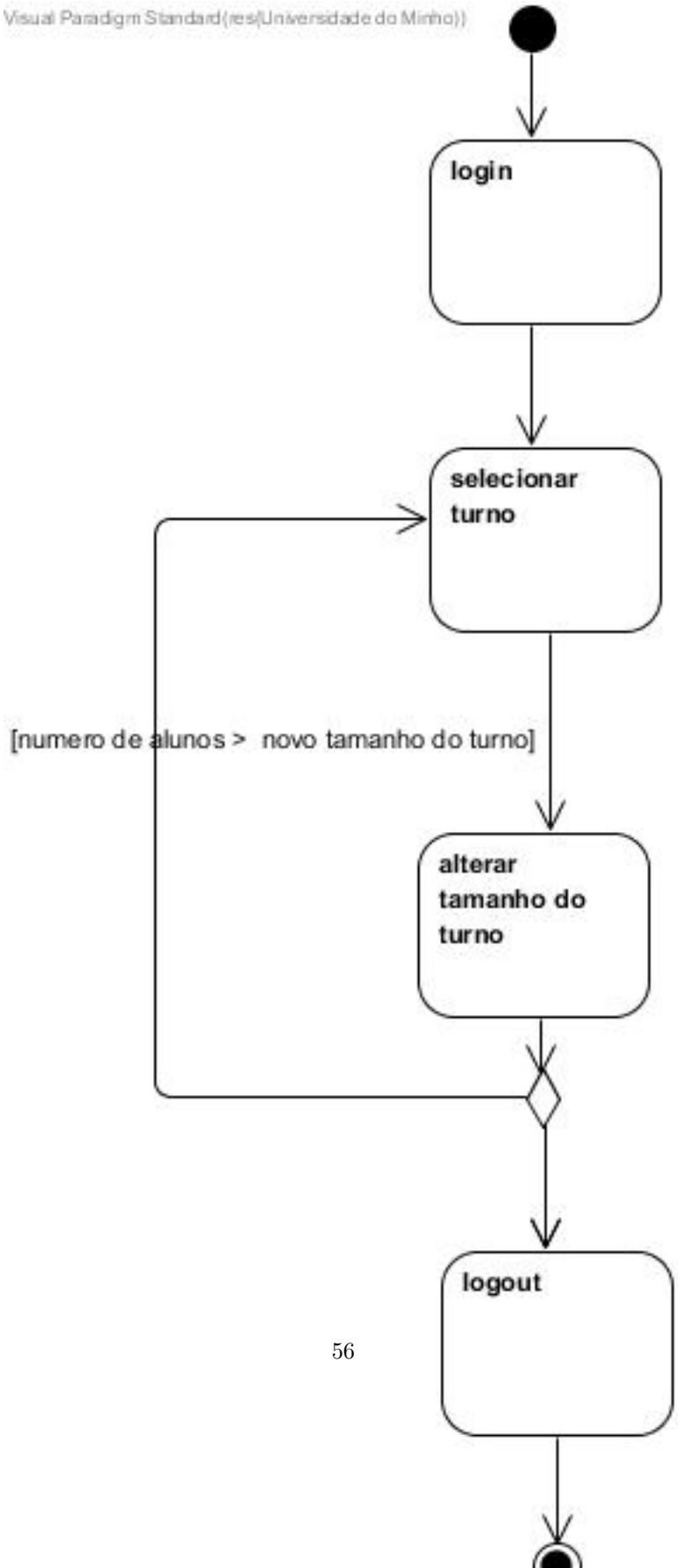

Visual Paradigm Standard(ires(Universidade do Minho))
login
selecionar turno
selecionar aluno
[turno cheio]
adicionar aluno
logout

Diagrama de Atividade de Reiniciar as Alocações


Visual Paradigm Standard(res(Universidade do Minho))
login
selecionar turno
[numero de alunos > novo tamanho do turno]
alterar tamanho do turno
logout