Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01120 - Técnicas de Construção de Programas
Prof. Érika Cota

INF UFRGS Room Allocator: Relatório da Primeira Etapa

Emily Calvet Pereira
Felipe Bertoldo Colombo de Souza
Marcello Saretta Tomazi

25 de junho de 2017

1 O projeto

O projeto consiste em um alocador de salas de aula do Instituto de Informática da UFRGS, que atende a requisitos regidos pela necessidade do professor ou disponibilidade de vagas/recursos, por exemplo.
O sistema automatiza a escolha das salas utilizadas por cada disciplina ministrada nos horários semanais visando a maior eficiência do processo, bem como a melhor utilização dos recursos físicos disponíveis nas instalações do Instituto.
Para isso, definimos classes que representam cada entidade envolvida na alocação (prédios, salas de aula, professores, disciplina, sessão) e os relacionamentos entre elas, que trazem consistência à alocação, garantindo que o resultado não comprometa a real relação disponibilidade X demanda.

2 O algoritmo

Para realizar a alocação, fazemos uso de um **algoritmo genético**.
O algoritmo gera soluções pseudo-randômicas, que estão contidas na classe **Schedules**, respeitando apenas a necessidade de dia e horário de cada Session, não considerando, inicialmente, outros fatores como número de assentos, necessidade de algum requisito físico, etc.
Cada solução é representada por um vetor que contém uma posição para cada um dos possíveis horários de uma Session (08:30, 10:30, 13:30 ou 17:30), para cada uma das salas disponíveis, para os cinco dias da semana. Logo, se uma Session ocupa a sala 214 do prédio 43412(65), na quinta-feira às 08:30, a posição equivalente do vetor apresentará o valor 1, dizendo que esta posição no espaço/tempo estará ocupada. Para saber qual Session ocupa qual posição do vetor, é então criado um HashMap para cada solução, que aponta qual posição do vetor cada Session estará ocupando.
Cada solução recebe também uma pontuação, o **fitness**, que é calculado da seguinte maneira:
Cada Session pode receber uma pontuação de 0 a 3, que é iniciada em zero e pode ser alterada de acordo com a tabela abaixo:

Tabela 1: Pontuação de uma Session

Necessidade antendida	**Pontuação**
capacidade da sala maior ou igual ao número de alunos	**+1**
sala está livre	**+1**
sala possui as features exigidas	**+1**

Cada solução então têm o seu **fitness** calculado através de (seja S o número de Sessions)

$(\sum_0^S session[i].pontuacao) \div (S \times 3)$

O que nos resultará em um número α , tal que $0 \leq \alpha \leq 1$, que pode ser interpretado da seguinte forma:

Tabela 2: Valores de α

α	**Interpretação**
0	Nenhuma Session foi satisfeita
$0 \leq \alpha \leq 1$	as Sessions foram parcialmente satisfeitas
1	todas as Sessions foram satisfeitas: ideal

Inicialmente, criamos uma população inicial de 100 indivíduos (Schedules), cujos fitness são testados a procura de um fitness = 1. Caso nenhuma solução tenha atingido um fitness de 1, são realizadas alterações na população obtida: o **crossover** e as **mutações**.
Para isso, as 10 soluções com os melhores fitness são automaticamente armazenadas em uma nova população.
Como queremos manter uma população com tamanho fixo, as outras soluções são obtidas através

das operações de crossover e mutation.
O crossover seleciona 50 pares da população e realiza uma operação de "merge" em sua organização, gerando novas 50 soluções. Este "merge" se dá realizando cortes no HasMap de cada solução e, após, unindo partes de cada HashMap em um novo HashMap. Uma novo vetor de solução é então criado a partir desse HashMap, e assim uma nova Schedule for criada. Agora, são selecionadas aleatoriamente 40 soluções da população antiga e trocadas D Sessions aleatórias de sala (mutation), gerando assim novas soluções perturbadas.

Novamente testamos se algum dos fitness dessa nova população gerada é igual a 1 e o processo é repetido até que esse valor seja atingido OU até que um mínimo aceitável seja atingido.

Caso o máximo de iterações seja atingido, retornamos então a melhor solução obtida, ainda que nem todas as Sessions tenham sido devidamente alocadas.

3 Classes

Em relação à etapa anterior, foram realizadas algumas alterações nos diagramas de classes para melhorar a coesão do projeto.
No **Domain Model** foram apenas adicionados alguns métodos em cada classe, para que cada método faça apenas o necessário. No **Architecture Model**, no entanto, foram criadas as classes **Alloc** e **Schedule** visando suportar a implementação de um algoritmo genético de baixo acoplamento.
Já a classe **InputFile** foi criada para administrar as operações feitas no arquivo de entrada, como a leitura e a escrita nesse mesmo arquivo.

3.1 Class Alloc

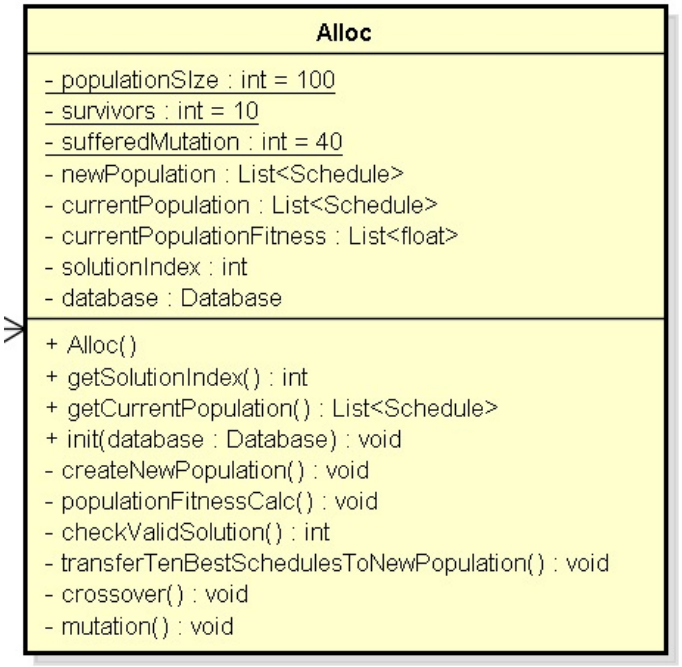

Figura 1: Class Alloc

3.2 Class Building

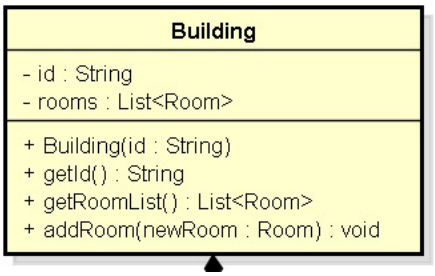

Figura 2: Class Building

3.3 Class Course

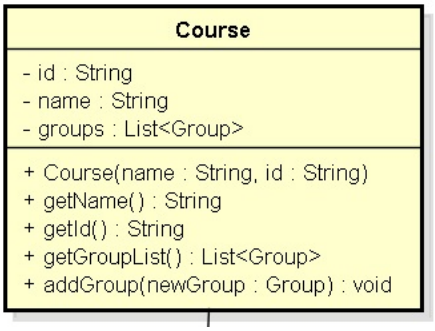

Figura 3: Class Course

3.4 Class Database

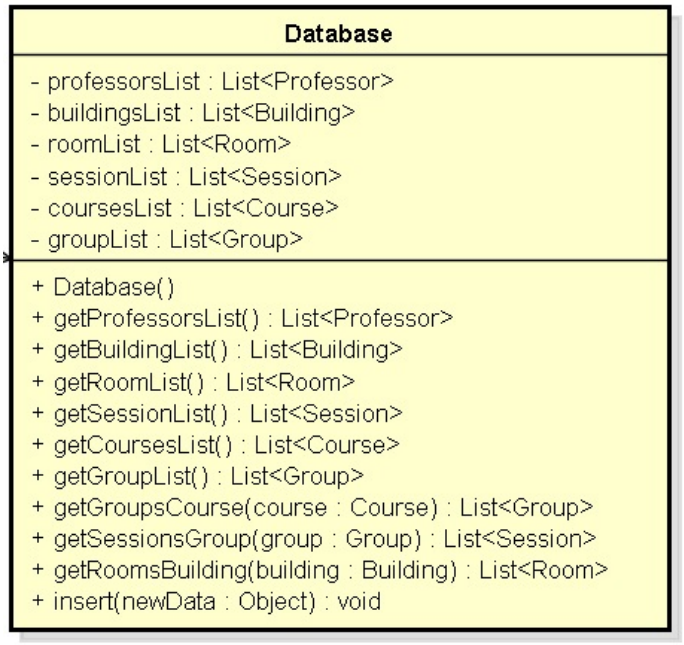

Figura 4: Class Database

3.5 Class Group

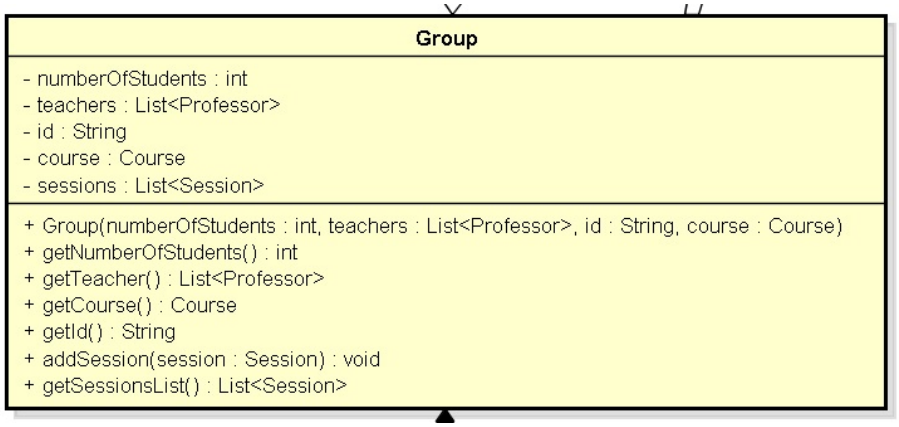

Figura 5: Class Group

3.6 Class INFOperationService - Interface

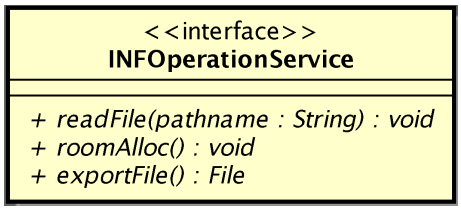

Figura 6: Class INFOperationService «interface»

3.7 Class INFOperationServiceImplementation

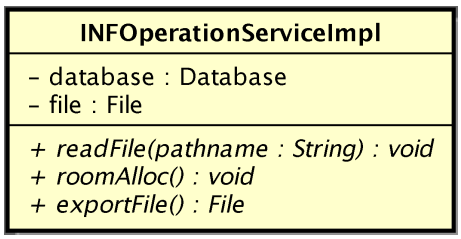

Figura 7: Class OpServImpl

3.8 Class Professor

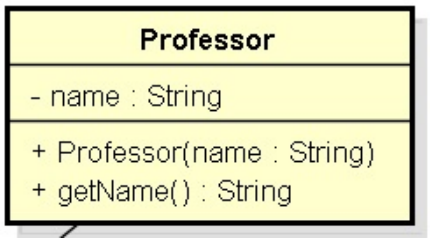

Figura 8: Class Professor

3.9 Class Room

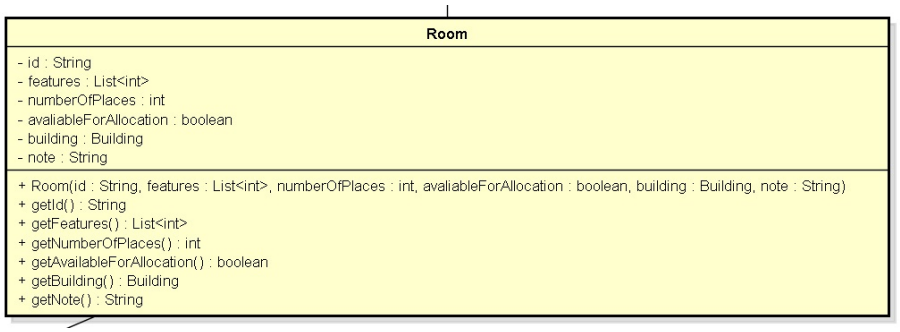

Figura 9: Class Room

3.10 Class Schedule

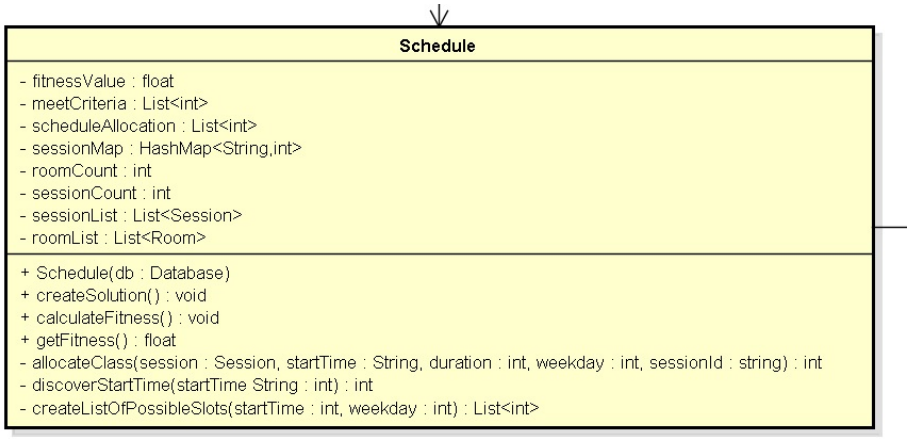

Figura 10: Class Schedule

3.11 Class Session

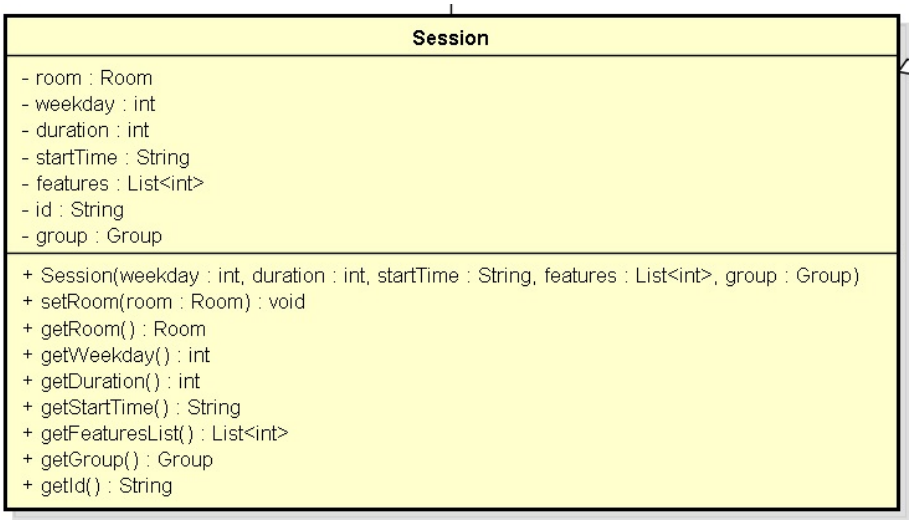

Figura 11: Class Session

3.12 Class InputFile

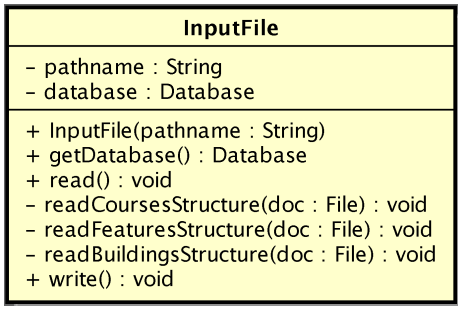

Figura 12: Class InputFile

4 Relacionamentos

4.1 Architecture model

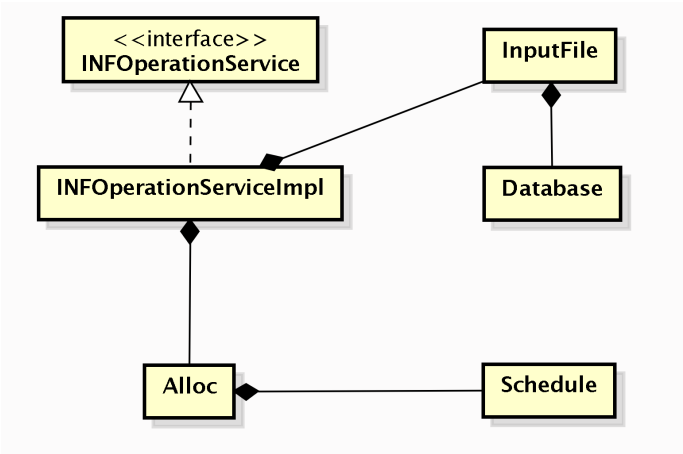

Figura 13: Relacionamentos entre as classes do Architecture Model

4.2 Domain model

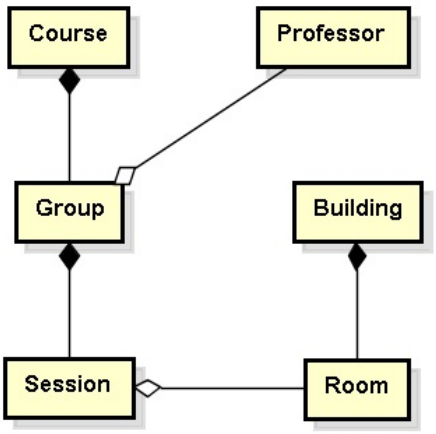

Figura 14: Relacionamentos entre as classes do Domain Model

5 Critérios de qualidade

5.1 Qualidade interna

5.1.1 Modularidade

Utilizar os critérios de modularidade visando garantir a maior independência possível entre as funções, possibilitando futuras alterações específicas que não alteram o comportamento do sistema como um todo.
As classes e os métodos a elas relacionados terão papéis específicos no sistema, com "baixa responsabilidade", evitando que uma única classe ou um único método tenham influência sobre uma grande área do sistema.

5.1.2 Legibilidade

Essencial para a manutenção e compreensão do código, a legibilidade é alcançada definindo-se alguns critérios de escrita para o código.

- Idioma: o código foi escrito em **inglês**.
- Nomenclatura:

  Classes: primeira letra de cada palavra em caixa alta.

  Ex.: MyClass
  MyClass() «construtor»
  OBS.: construtores, ainda que sejam métodos, precisam ter o mesmo nome da classe para funcionar.

  Variáveis, atributos e métodos: primeira letra minúscula e caixa alta para cada palavra nova.

  Ex.: myAttribute
  getAttribute()

  Documentação: o código foi escrito de modo a ser o mais auto-explicativo possível, trazendo mais liberdade em relação a comentários (que tendem a ficar obsoletos).
  Nesse aspecto, os critérios de nomenclatura citados acima são estendidos, uma vez que além da disposição das letras em caixa alta, o nome de cada classe/método/atributo foi explícito em relação ao papel de tal entidade.

  Manutenabilidade: Criamos funções compreensíveis e independentes de modo que sua manutenção tenha custo e tempo reduzido.

  Funcionabilidade: Visamos a entrega de um programa que executa os requisitos de forma precisa, atendendo às especificações do trabalho.

  Confiabilidade: Além da entrega conforme as especificações do trabalho, visamos também informar o usuário de falhas como entrada não suportada, solução impossível e erros em geral.

5.2 Qualidade externa

A partir dos critérios de qualidade interna acima definidos, buscamos trazer para nosso sistema alguns critérios de qualidade visíveis ao usuário, ou seja, de qualidade externa.
São eles: **robustez**, **corretude**, **compatibilidade** e **facilidade de uso**. O usuário poderá usar facilmente um sistema que responderá bem a situações adversas às que foram programadas, assim como atenderá bem às tarefas especificadas.