Trabalho Final de Técnicas de Construção de Programas

Etapa I

Bernardo Lignati, Thiago Barp e Gerônimo Acosta

Professora Erika Cota

Maio de 2017

Descrição do Sistema

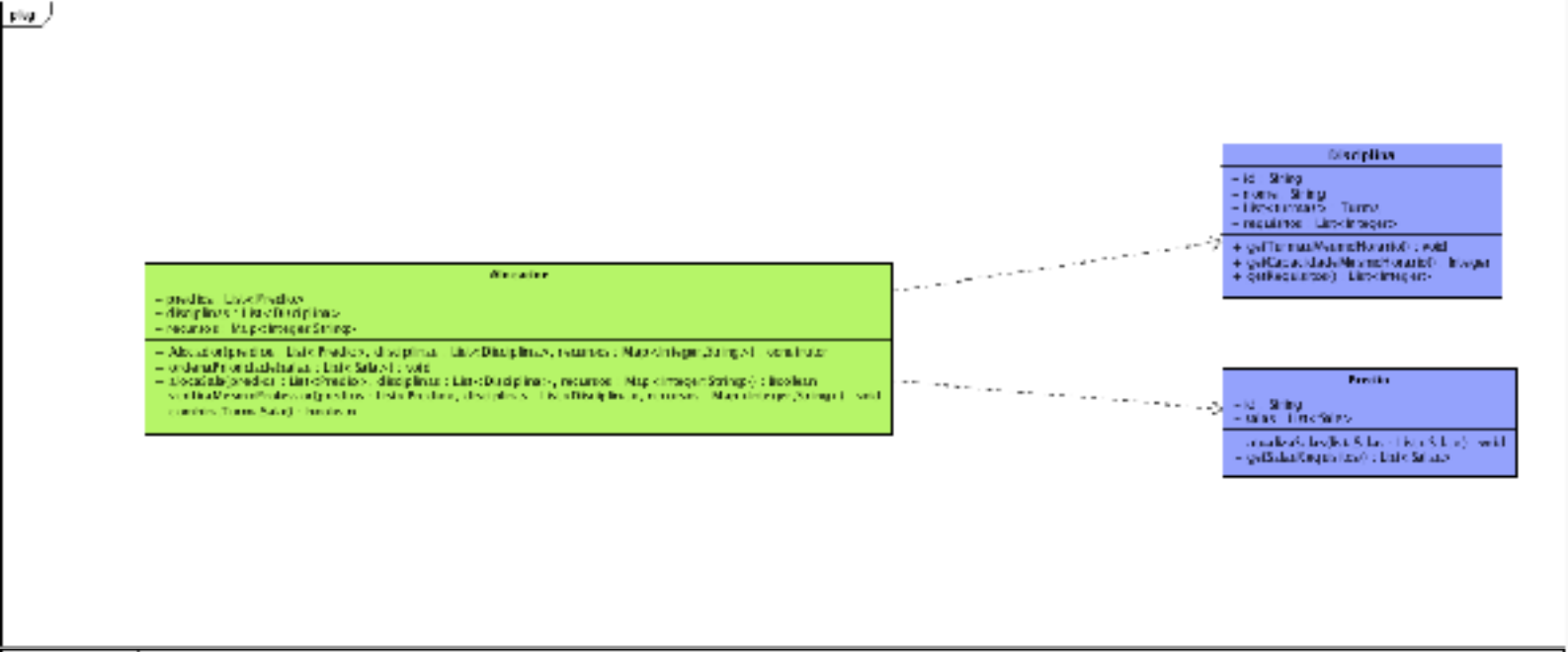
O funcionamento do trabalho é observado pelos diagramas de sequência anexados ao arquivo de entrega do trabalho. Neles, podemos observar o fluxo de funcionamento da parte da interface do programa que envolve a abertura, leitura e fechamento do arquivo .XML e da parte do projeto responsável por realizar a alocação propriamente dita das salas de aula do INF.

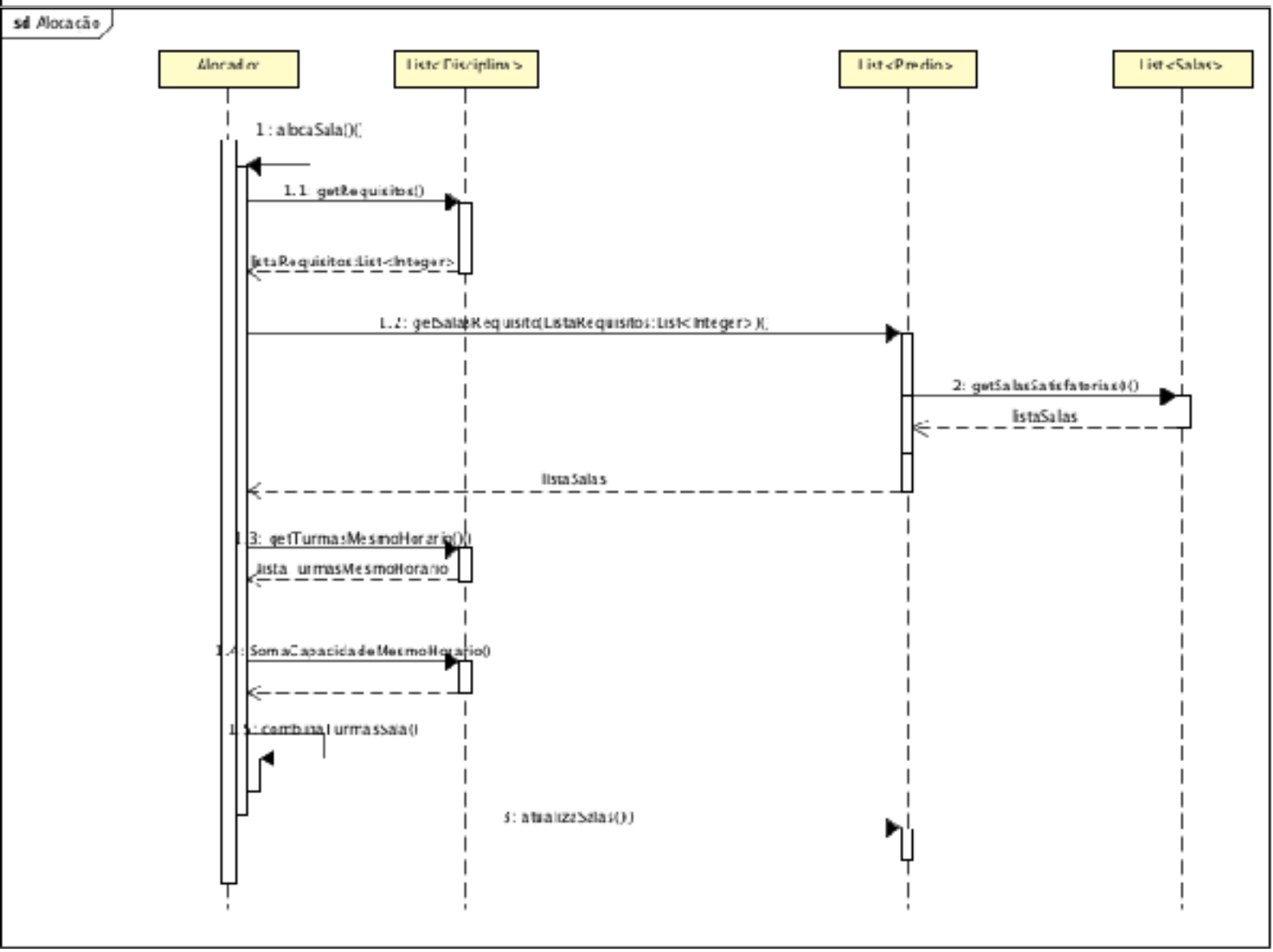
No diagrama de sequência dos Arquivos, como podemos observar, primeiramente é realizada a abertura e verificação da extensão do arquivo durante a execução da classe ArquivoEntrada encontrada no diagrama de classes Interface. Ainda nesse contexto de execução ocorre a criação de uma lista de prédios disponíveis no arquivo .XML que resulta consequentemene na criação de uma lista de de salas. Ainda nocontexto de execução de ArquivoEntrada, conseguimos criar uma lista de Disciplinas já que o arquivo éseparado em prédios e disciplinas. A partir da lista de disciplinas conseguimos criar uma lista de turmas para cada disciplina. Tendo uma lista de disciplinas e voltando para o contexto do arquivo de entrada conseguimos obter um map (semelhante a uma matriz) de recursos conforme a necessidade de cada disciplina. Tendo todas essas listas em mãos, mudamos o contexto de execução para a classe Alocador que é invocada pelo método construtor Alocador. A partir disso, criamos uma lista de prioridades conforme a necessidade que cada disciplina requisita para poder ser devidamente ministrada. Quanto mais recursos
uma determinada disciplina requer, mais alta será sua prioridade na lista de prioridades. Tendo isso pronto, ocorre a alocação de salas. Por último, ainda há a criação de um arquivo de saída .XML com a lista devidamente atualizada de alocações em

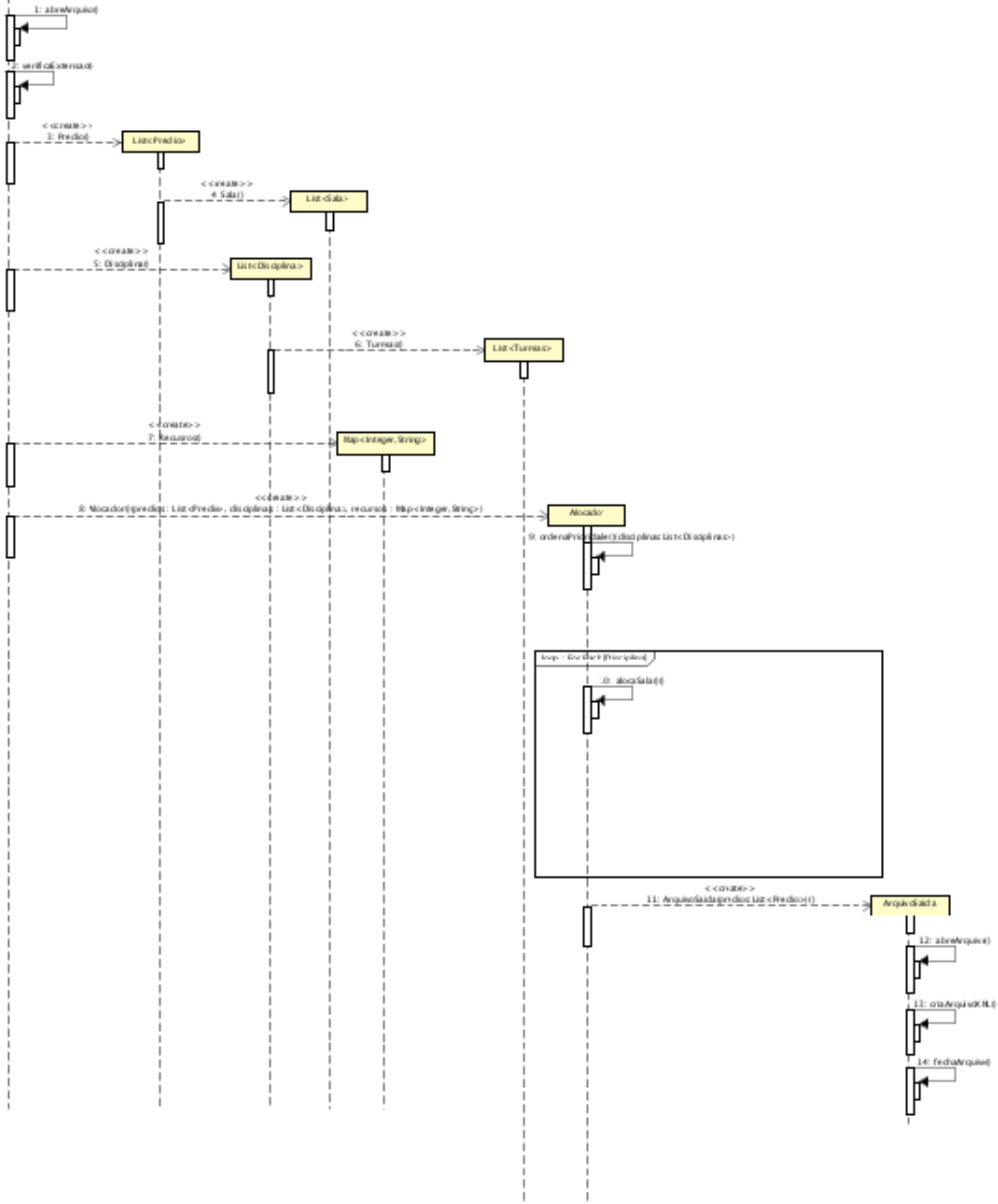
cada sala de cada prédio (informações contidas na lista de prédios) e o seu fechamento.

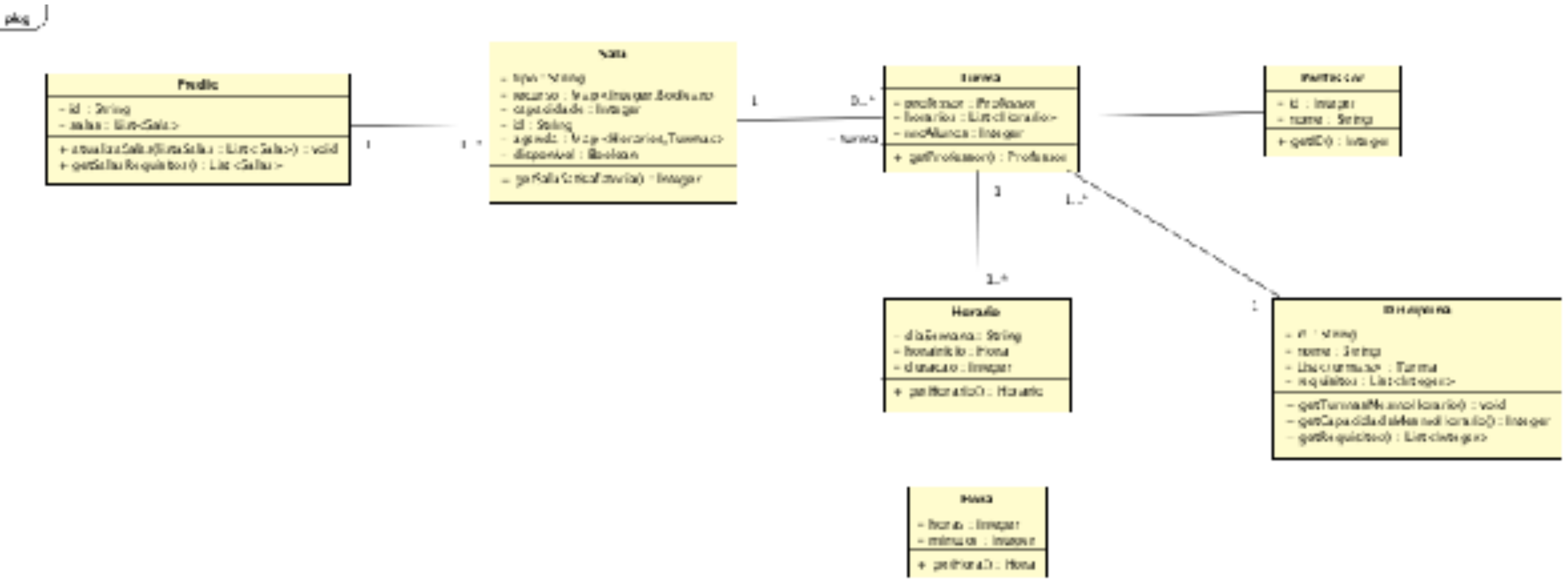
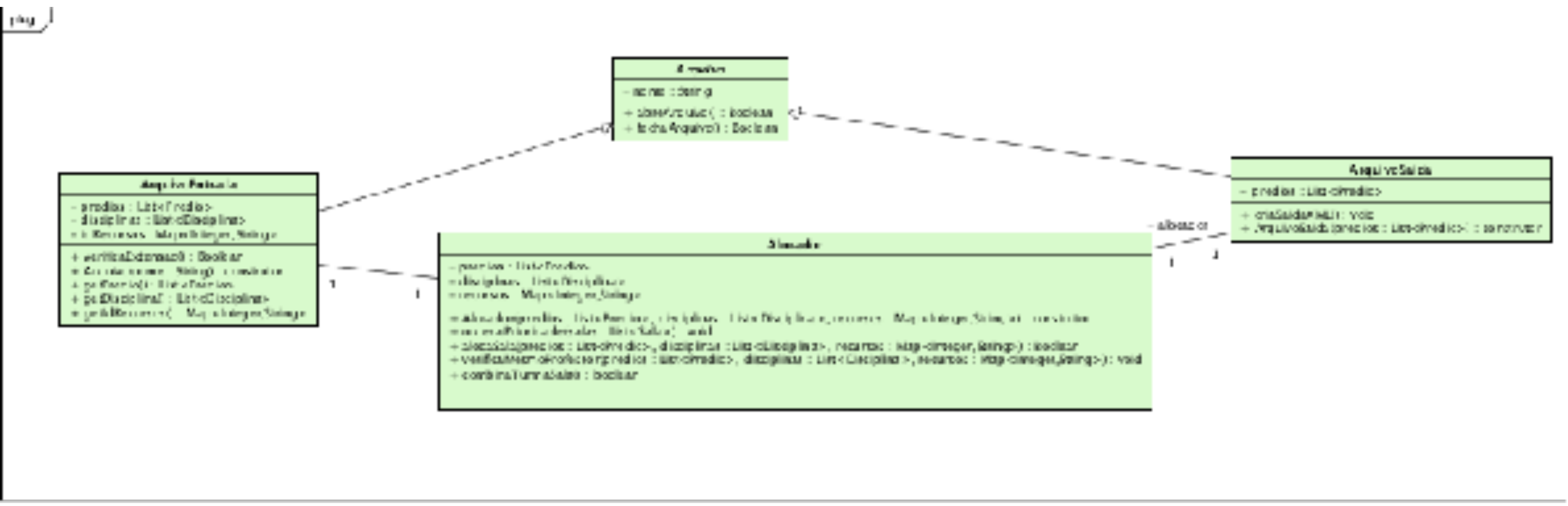
No diagrama de sequência de Alocação, chamamos o método alocaSala que retorna um valor booleano que corresponde se há ou não salas disponíveis. Depois disso devemos pegar da lista de Discilpinas todos requisitos para determinada discilpina e tendo isso em mãos, procuramos na lista de prédios todas as salas que atendam a esses requisitos requisitados invocando o método getSalasRequisito. Tendo todas salas com esses requisitos, procuramos por aquelas que ainda estejam disponíveis e criamos uma nova lista com elas. Após, verificamos se mais de uma turma terá aula na mesma sala no mesmo horário e, caso isso ocorra, somamos a quantidade de alunos por turma e verificamos na lista de salas disponíveis as que se adequam a essa capacidade. Tendo isso, conseguimos alocar uma sala que corresponda às necessidades e atualizamos a lista de salas.

Após a lista de salas ser atualizada um objeto de da classe de interface ArquivoSaida é criado e faz todas as operações necessárias para que seja criado um arquivo XML de saída, criando assim um tipo de persistência, sem a necessidade da utilização de recursos de banco de dados e deixando o programa com um fluxo de execução bem definido. Com saída e entrada com formato de genérico XML, caracterizamos o programa para fun acionar como uma API.


Disciplina
- id : String
- nome : String
- listaTurmas : Turma
- requisitos : List<Integer>
+ getTurmasMesmoHorario() : void
+ getCapacidadeMesmoHorario() : Integer
+ getRequisitos() : List<Integer>
Predio
- id : String
- salas : List<Sala>
- getSalaRequisito() : List<Sala>


sd Alocação
Alocador
Lista<Disciplinas>
List<Predios>
List<Salas>
1 : alocaSala()
1.1 : getRequisitos()
listaRequisitos:List<Integer>
1.2 : getSalaByRequisito(listaRequisitos:List<Integer>)
2 : getSalasSatisfatorias()
listaSalas
listaSalas
1.3 : getTurmasMesmoHorario()
lista_turmasMesmoHorario
1.4 : SomaCapacidadeMesmoHorario()
1.5 : cambinaTurmasSala()
5 : atualizaSalas()


1: abreArquivo()
2: verificaExtensao()
<<create>>
3: Predio()
List<Predio>
<<create>>
4: Sala()
List<Sala>
<<create>>
5: Disciplina()
List<Disciplina>
<<create>>
6: Turma()
List<Turma>
<<create>>
7: Recursos()
Map<Integer, String>
<<create>>
8: Alocador(predios : List<Predio>, disciplinas : List<Disciplina>, recursos : Map<Integer, String>)
Alocador
9: ordenaPrioridade(disciplinas: List<Disciplina>)
10: alocaSala()
<<create>>
11: ArquivoSaida(predios: List<Predio>)()
ArquivoSaida
12: abreArquivo()
13: criaArquivoXML()
14: fechaArquivo()


pkg
Predio
- id : String
+ atualizaSalas(listaSalas : List<Sala>) : void
Sala
- id : String
- disponivel : Boolean
- turma
0..*
1..*
Horario
- horaInicio : Hora
- duracao : Integer
+ getHorario() : Horario
Hora
- horas : Integer
- minutos : Integer
+ getHora() : Hora
- nome : String
+ getID() : Integer

pkg
- alocador
+ combinaTurmaSala() : boolean

Qualidade do Software

- **Qualidade Externa**
    - **Facilidade de uso**: Comandos bem explicados e simples para o Usuário feitos através da entrada de formato comum(XML);
    - **Robustez**: Evitar ‘crash’ em casos que o usuário manda entrada diferente do esperado;
    - **Eficiência**: Executar os comandos com velocidade razoável;
    - **Reusabilidade**: Alocar diferentes tipos de sala com o mesmo comando;
- **Qualidade Interna**
    - Fácil compreensão do código;
    - Fácil modificação do código;
    - Segurança do código, pequenas alterações não destroem a funcionalidade;
    - Criação do mínimo de dependência possível;

O design deste projeto foi guiado pelas boas praticas do desenvolvimento de software segundo o paradigma de Orientação a Objetos. Todas as classes foram pensadas de forma a abstrair agentes do mundo real e desenhadas sobre critérios de modularidade. Por exemplo o núcleo lógico da aplicação é independente do sistema de interfaces, sendo possível mudar o sistema de interface sem interferir na lógica interna da aplicação. Todas as classes foram desenvolvidas de forma a dividir as operações designadas a cada agente do sistema de forma a utilizar as vantagens de encapsulamento da melhor maneira possível, dessa forma os objetos das classes em um contexto global trabalham “enxergando” somente o que as outros objetos definem como sendo publico, garantindo assim a segurança dos atributos.