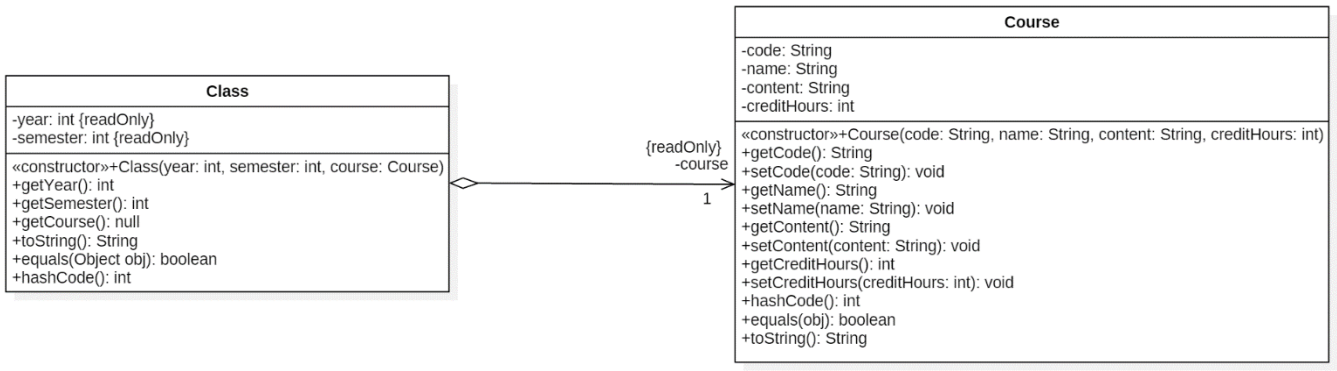
Roteiro 08 – Atividade 01/07

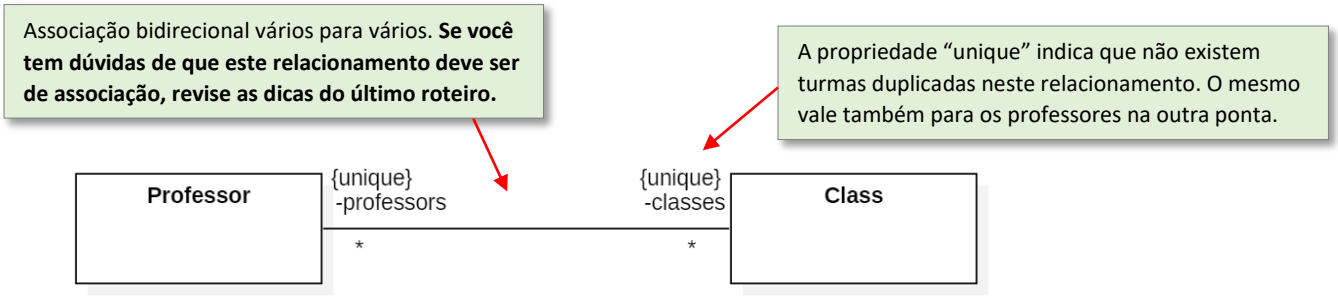
1. Como nosso sistema acadêmico está sendo modelado e programado em Inglês, precisamos padronizar alguns conceitos para facilitar o entendimento comum. Isso é necessário, não apenas por causa da tradução para o Português, mas porque existem diferenças de terminologia entre países de língua inglesa.

Termo em Inglês	Termo correspondente em Português
Person	Pessoa
Employee	Empregado; Funcionário; Colaborador
Student	Estudante; Aluno
Professor	Professor (usualmente universitário)
Dean	Reitor
Chair	Chefe de departamento
Advisor	Orientador (papel de um professor)
Advisee	Orientado (papel de um estudante)
Department	Departamento (oferece um conjunto de disciplinas)
Major	Curso (tipicamente bacharelado)
Course	Disciplina (definida na grade curricular e oferecida por um departamento). Ex: POO1
Class	Turma (específica em um determinado semestre/ano/turno). Ex: POO1.2017/1.01
Content	Ementa (assuntos tratados na disciplina)
Degree plan	Grade curricular (disciplinas organizadas por semestre)
Credit hours	Horas de aula (carga horária semanal)
Enrollment	Matrícula
Marks	Nota

2. Vamos considerar o conceito de uma disciplina (`Course`) e turma (`Class`), onde uma turma é a ocorrência de uma disciplina em um determinado ano e semestre. Modele e implemente estas classes, conforme o diagrama abaixo.



3. Agora, vamos considerar o relacionamento entre professores e turmas. Um professor pode ministrar várias turmas (inclusive nenhuma). Por outro lado, uma turma pode ser ministrada por vários professores (por exemplo, um professor para a parte teórica e outro para a parte prática). Também considere que uma turma pode ser criada sem que um professor ainda esteja definido para ela.



Roteiro 08 – Atividade 01/07

Um relacionamento bidirecional significa navegável nos dois sentidos (na StarUML, as propriedades da associação ***end1*** e ***end2.navigable*** precisam estar marcadas). Note também que as setas não são mostradas. Esta é uma limitação da ferramenta, pois seria adequado que as setas fossem mostradas em ambos os lados. Visualmente, a omissão das setas pode ser interpretada como navegabilidade ainda indefinida. Nesta disciplina, quando o relacionamento não tiver setas nas duas pontas, consideraremos que ele é bidirecional.

Para implementarmos o relacionamento descrito, precisaremos de uma coleção de turmas na classe `Professor` e uma coleção de professores na classe `Class`. Mais do que isso, precisamos garantir a consistência do relacionamento. Se um professor A ministra as turmas X e Y, estas duas turmas precisam também apontar para A. Se A deixar de ministrar estas turmas, X e Y precisam deixar de apontar para A também. Esta lógica vale nos dois sentidos. Imagine que a turma X foi excluída. A coleção do professor A precisa ser também atualizada.

Logo, precisaremos fornecer operações para adicionar ou excluir uma turma (na ponta `Professor`) e adicionar ou excluir um Professor (na ponta `Class`). Também serão necessárias operações que retornem os professores de uma turma e as turmas de um professor. Faça os ajustes na sua modelagem e implemente as alterações.

4. Abaixo, é apresentado um trecho da implementação da classe `Professor`.

```
public List<Class> getClasses() {
    return classes;
}

public List<Class> getClasses(int Year, int Semester) {
    List<Class> filteredClasses = new ArrayList<>();
    // código omitido
    return filteredClasses;
}

public List<Class> getClasses(Course course) {
    List<Class> filteredClasses = new ArrayList<>();
    // código omitido
    return filteredClasses;
}
```

Diferentes formas para recuperar as classes de um professor.

```
public void addClass(Class c) {
    new Validator().notNull(c, "class");
    if (!classes.contains(c)) {
        classes.add(c);
        c.addProfessor(this);
    }
}
```

O professor adiciona a turma...

... e pede para a turma fazer o mesmo.

```
public void delClass(Class c) {
    new Validator().notNull(c, "class");
    if (classes.contains(c)) {
        classes.remove(c);
        c.delProfessor(this);
    }
}
```

Esta condição funciona como ponto de parada. Caso contrário teremos um loop infinito, onde o objeto `Professor` invoca o objeto `Class` e vice-versa. Experimente sem esta linha.

Veja que os métodos das operações `addClass` e `delClass` garantem a consistência discutida anteriormente (as duas pontas são sempre atualizadas na mesma operação). Conceitualmente, estas operações deveriam ser atômicas. Uma **operação atômica** (ou **transação**) deve ser indivisível. Isso indica que se houver alguma exceção durante a sua execução, o estado do sistema deveria retornar exatamente ao ponto antes da invocação da operação. No nosso caso, como estamos trabalhando com objetos em memória, desconsideraremos esta restrição. Entretanto, quando você estiver manipulando informações armazenadas em banco de dados ou trabalhando com processos em sistemas operacionais, este conceito passará a ser altamente relevante.

A implementação em `Class` é similar. Entretanto, para a recuperação dos professores, como a turma já é de um determinado ano/semestre, é necessário apenas a operação `getProfessors`.

Roteiro 08 – Atividade 01/07

5. Para testar o programa, você pode fazer algo como:

```
State s1 = new State("Santa Catarina", "SC");
City c1 = new City("Florianópolis", s1);
Address a1 = new Address("Rua Floriano, 2012", null, c1, "88015200");
Professor bruce = new Professor("Bruce Wayne", "02/05/1996", "wayne@com", a1, "01/03/2017", 40, "20.34", AcademicDegree.BACHELOR);
Professor emma = new Professor("Emma Grace Frost", "23/09/1994", "frost@com", a1, "31/07/2016", 40, "20.34", AcademicDegree.MASTER);
Professor strange = new Professor("Stephen Vincent Strange", "23/09/1994", "frost@com", a1, "31/07/2016", 40, "20.34", AcademicDegree.MASTER);

Course pool = new Course("POO1", "Programação OO", "Ementa de OO", 4);
Course es1 = new Course("ES1", "Engenharia de Software", "Ementa de ES1", 4);
Course es2 = new Course("ES2", "Engenharia de Software", "Ementa de ES2", 4);

Class pool_171 = new Class(2017, 1, pool);
Class es1_171 = new Class(2017, 1, es1);
Class es2_171 = new Class(2017, 1, es2);

bruce.addClass(pool_171);
emma.addClass(pool_171);
pool_171.delProfessor(bruce);

System.out.println(bruce);
System.out.println(pool_171);
System.out.println(emma);
```

6. Se você implementou tudo conforme o que você aprendeu até agora, a exceção `StackOverflowException` deve ter sido disparada quando você executou o programa. Caso contrário, você deve estar pulando etapas e não há como garantir o que está acontecendo.

7. O defeito está na forma adotada para a redefinição de `toString`. Sua classe `Professor` deve estar chamando `classes.toString()` e sua classe `Class` deve chamar `professors.toString()`. O problema é o ciclo infinito: a) um professor concatena a lista de suas turmas; b) cada turma concatena a lista de seus professores; c) cada professor concatena a lista de suas turmas; etc. Uma forma de resolver este problema é alterar a implementação do `toString` em uma das classes. O exemplo abaixo mostra uma solução em `Class`.

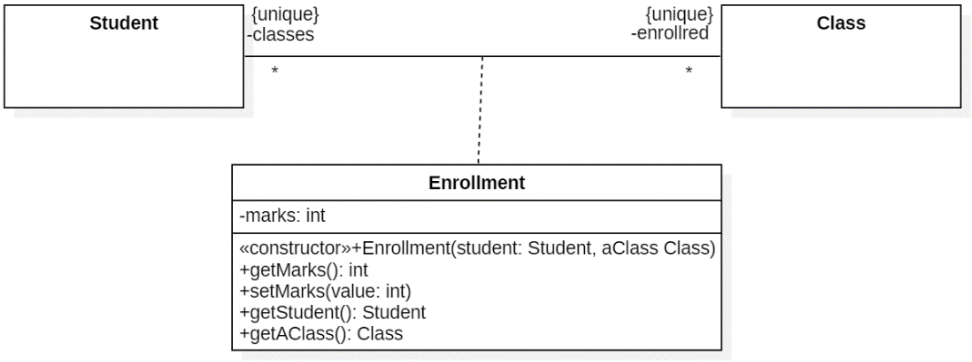
```
@Override
public String toString() {
    StringBuilder output = new StringBuilder();
    output.append(this.getClass().getName() + " {" + AppConfig.NEW_LINE);
    output.append("  year = " + year + AppConfig.NEW_LINE);
    output.append("  semester = " + semester + AppConfig.NEW_LINE);
    output.append("  course = " + course.getCode() + " - " + course.getName() + AppConfig.NEW_LINE);
    output.append("  professors = " + AppConfig.NEW_LINE);
    for (Professor professor : professors) {
        output.append("[" + professor.getId() + " - " + professor.getName() + "] " + AppConfig.NEW_LINE);
    }
    output.append("}" + AppConfig.NEW_LINE);
    return output.toString();
}
```

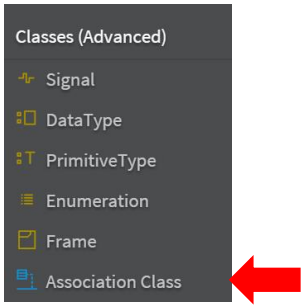
Os professores são concatenados manualmente ao invés de apenas chamar o `toString` de `professors`.

Roteiro 08 – Atividade 02/07

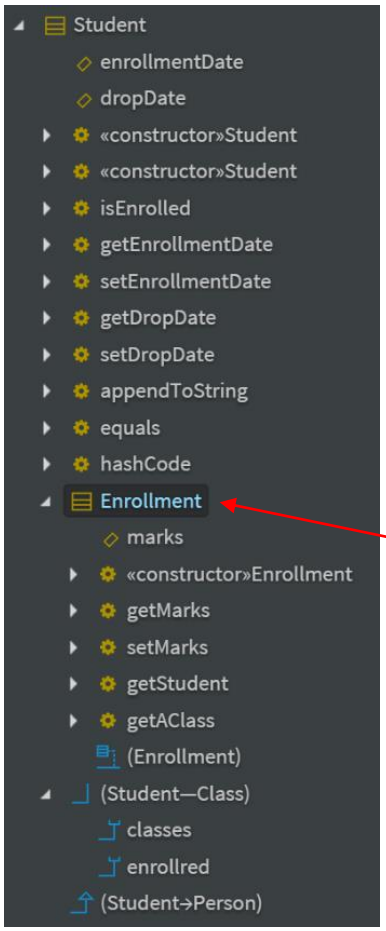
1. Nesta atividade, iremos adicionar o relacionamento entre estudantes e turmas. Um estudante pode estar matriculado em várias turmas e uma turma pode estar associada com vários estudantes. Pelo que aprendemos até agora, poderíamos modelar algo como (não modele ainda):

Entretanto, considere a nota (*marks*) de um aluno em uma respectiva turma. Esta informação não é nem da classe `Student` nem da classe `Class`, mas do relacionamento entre elas. Um estudante tem uma nota em cada turma que ele está matriculado. A solução para esta situação é utilizarmos uma classe de associação, a qual representação a ligação entre `Student` e `Class`. Por ser uma classe, podemos modelar o atributo `marks` nela.





Na ferramenta StarUML, a classe `Enrollment` será colocada dentro da classe `Student` ou da classe `Class` (dependendo de qual delas você clicou primeiro para desenhar a associação).

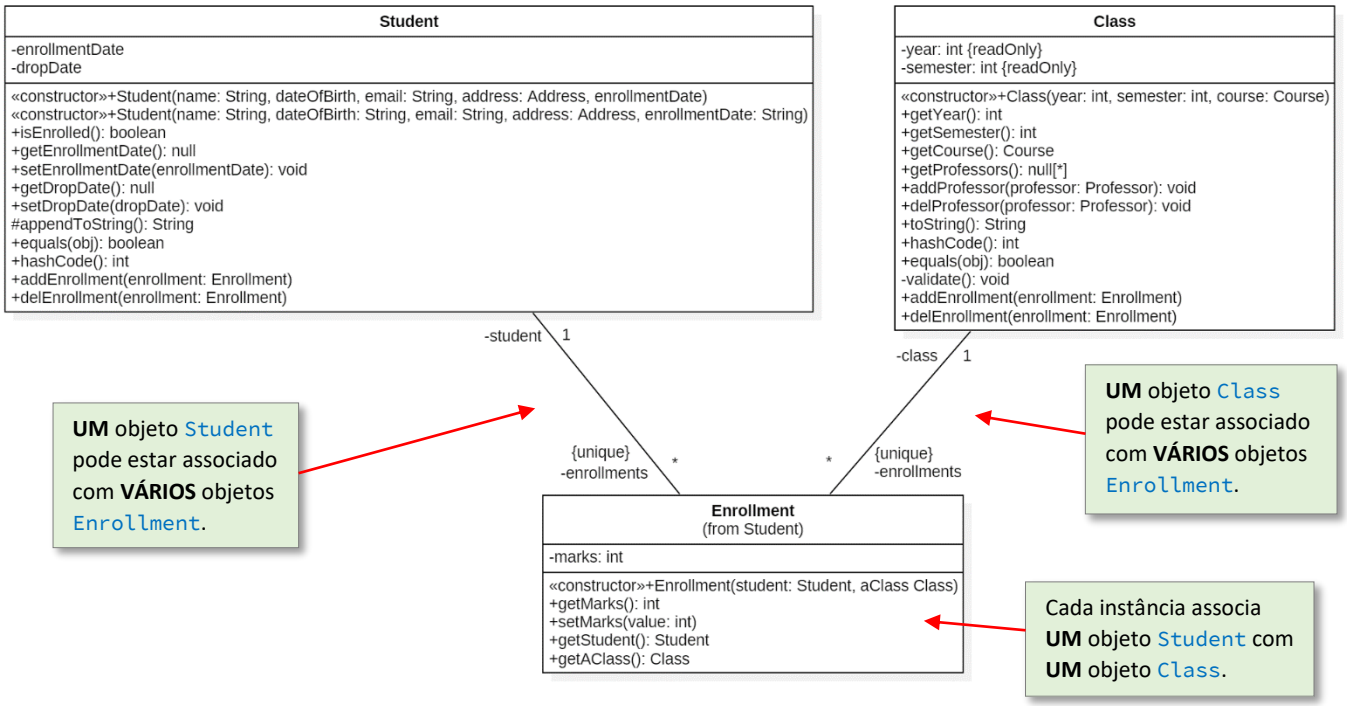


A class `Enrollment` está dentro da classe `Student` (*Model Explorer*).

Uma **classe de associação** é a representação de uma associação entre classes, quando precisamos adicionar informações na associação. Ela é tanto uma classe quanto uma associação. No nosso exemplo, nota não pertence nem à classe `Student` nem à classe Class. É uma informação exclusiva da relação entre elas, representada por uma matrícula (classe `Enrollment`). Desta forma, devemos utilizar uma classe de associação somente quando a associação precisa de atributos para defini-la.

Roteiro 08 – Atividade 02/07

A representação anterior é uma visão lógica do relacionamento, pois não mostra exatamente como ela será implementada fisicamente. A representação física seria como a imagem abaixo.



2. A implementação passa a ser um relacionamento UM para VÁRIOS ao invés de VÁRIOS para VÁRIOS. A implementação da classe de associação é trivial. Os atributos `student` e `aClass` podem ainda ser definidos como final. Depois que um aluno foi matriculado em uma disciplina, não faz sentido alterar estas informações.

```
/**
 * @param student o estudante matriculado
 * @param courseClass a turma na qual o estudante está matriculado
 */
public Enrollment(Student student, Class courseClass) {
    this.student = student;
    this.courseClass = courseClass;
    validateState();
}
...
/**
 * Valida o estado desta matrícula.
 */
private void validateState() {
    Validator val = new Validator();
    val.notNull(student, "student");
    val.notNull(courseClass, "courseClass");
}
```

Não esqueça que é necessário **validar os dados**.

No caso do atributo `marks`, considere que a nota pode variar entre 0 e 100. Você pode utilizar a nossa classe `ComparableValidator` para realizar esta validação. Você lembrou de utilizá-la para validar o semestre na classe `Course`?

```
public void setMarks(int marks) {
    new ComparableValidator().range(marks, "marks", 0, 100);
    this.marks = marks;
}
```

Roteiro 08 – Atividade 02/07

3. Precisamos implementar a associação em cada uma das pontas: `Student` e `Class`.

```java
public class Class implements Serializable {

    /**
     * Ano em que a turma foi ministrada.
     */
    private final int year;
    /**
     * Semestre em que a turma foi ministrada.
     */
    private final int semester;
    /**
     * Disciplina ministrada pela turma.
     */
    private final Course course;
    /**
     * Professores que ministram a turma.
     */
    private List<Professor> professors = new ArrayList<>();
    /**
     * Matrículas dos estudantes desta turma.
     */
    private List<Enrollment> enrollments = new ArrayList<>();
```

```java
public class Student extends Person {

    /**
     * Data na qual o estudante foi admitido (primeira matrícula).
     */
    private LocalDate enrollmentDate;
    /**
     * Data na qual o estudante se desligou da instituição.
     */
    private LocalDate dropDate;
    /**
     * Matrículas (em turmas) onde o estudante está ou esteve matriculado.
     */
    private List<Enrollment> enrollments = new ArrayList<>();
```

Cada ponta mantém sua lista de objetos `Enrollment`. Mas, é necessário manter as listas sincronizadas (consistência).

4. No trecho de código abaixo, são apresentados os métodos da turma (`Class`). que permitem adicionar ou remover uma estudante (`Student`). Os métodos em `Student` seguem a mesma lógica, mas podemos adicionar uma operação *getter* a mais, recuperando todas as matrículas dado um determinado ano e semestre.

```java
/**
 * @return todas as matrículas dos estudantes desta turma
 */
public List<Enrollment> getEnrollments() {
    return enrollments;
}
```

Retorna todas as matrículas da turma. Precisamos retornar o relacionamento, pois precisaremos acessar depois suas informações exclusivas. No nosso caso, a nota.

```java
/**
 * @param student o estudante matriculado nesta turma
 * @return a matrícula do estudante nesta turma, null se a o
 *          estudante não está/esteve matriculado na turma
 */
public Enrollment getEnrollment(Student student) {
    Enrollment enrollment = null; // apenas para compilar
    // código omitido
    return enrollment;
}
```

Retorna a matrícula de um estudante específico.

```java
/**
 * Matricula um estudante em uma turma.
 *
 * @param enrollment a matrícula a ser adicionada
 */
public void addEnrollment(Enrollment enrollment) {
    validateEnrollment(enrollment);
    if (!enrollments.contains(enrollment)) {
        enrollments.add(enrollment);
        enrollment.getStudent().addEnrollment(enrollment);
    }
}
```

Validação: a matrícula é mesmo desta turma?

```java
private void validateEnrollment(Enrollment enrollment) {
    if (enrollment.getStudent() != this) {
        throw new IllegalArgumentException("Expected student " +
                this + " but other have been sent " + enrollment.getStudent());
    }
}
```

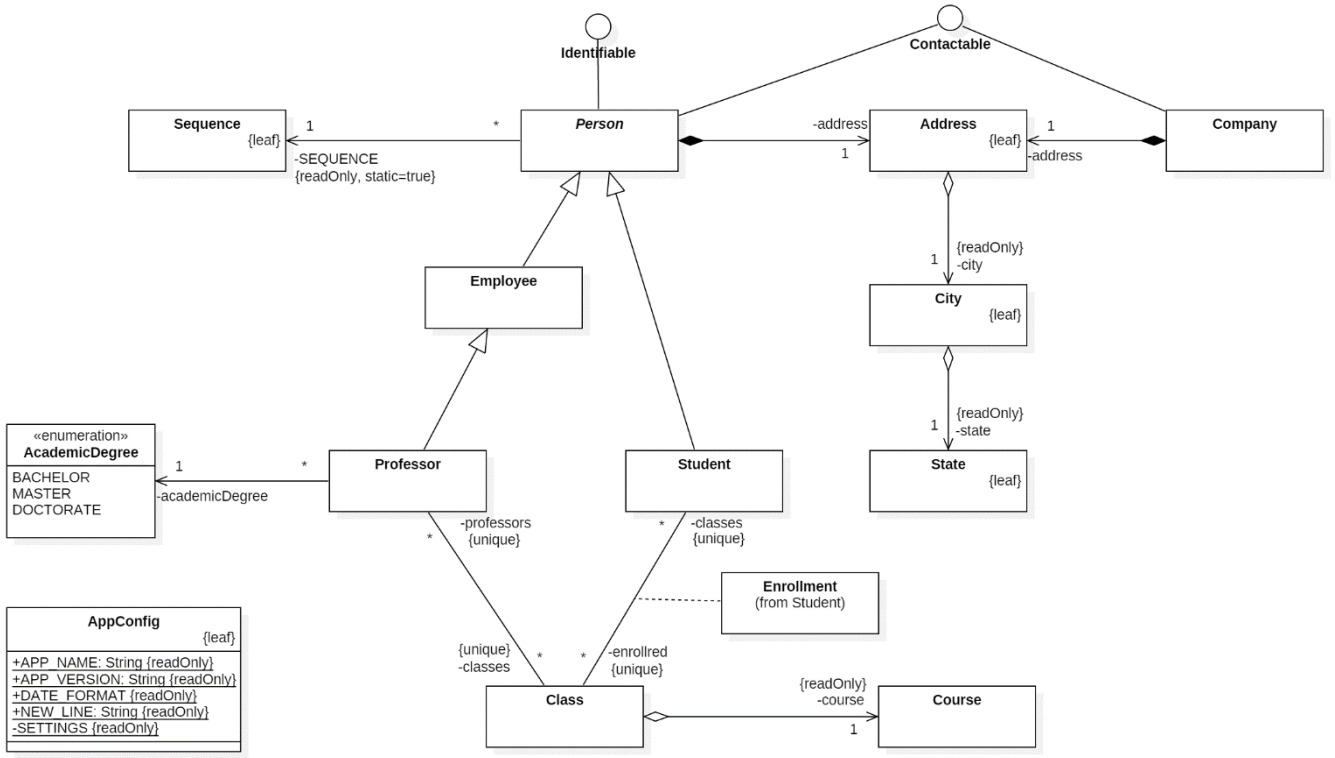
Esta condição evita repetição e também funciona como ponto de parada (evitar o ciclo infinito).

```java
/**
 * Remove a matricula de um estudante em uma turma.
 *
 * @param enrollment a matrícula a ser adicionada
 */
public void delEnrollment(Enrollment enrollment) {
    validateEnrollment(enrollment);
    if (enrollments.contains(enrollment)) {
        enrollments.remove(enrollment);
        enrollment.getStudent().delEnrollment(enrollment);
    }
}
```

5. Altere sua modelagem e implemente as alterações. Utilize o arquivo **EnrollmentTest.java** fornecido com este roteiro para realizar os testes. Se você seguiu as orientações, ele deve compilar com seu código sem problemas.

Roteiro 08 – Atividade 03/07

1. Se você implementou todos os roteiros e manteve seus projetos atualizados, sua modelagem deve estar como a imagem abaixo.



Não foram representadas aqui as classes que implementam os repositórios, exceções e a interface `Serializable` para evitar a poluição visual do diagrama. Na prática, nunca modelamos um sistema em um único diagrama. A medida que a quantidade de elementos no diagrama aumenta, a legibilidade é reduzida. Por exemplo, imagine se tivéssemos configurado o diagrama acima para mostrar também atributos e operações. Seria muito difícil de acompanhar a solução adotada e praticamente impossível evitar o cruzamento de relacionamentos.

Poderíamos desenhar um diagrama detalhado para mostrar a abstração `Person` (`Person`, `Identifiable`, `Contactable`, `Sequence`, `Address`, `City`, `State`) e outro para `Class` (`Class`, `Course`, `Professor`, `Student`, `Enrollment`). Diagramas focados em determinados aspectos da aplicação ajudam a quebrar um problema maior em partes menores, tratando a complexidade. Entretanto, tome cuidado para não criar uma explosão desnecessária de diagramas. A quantidade de elementos e de diagramas deve ser orientada pela legibilidade e agilidade no entendimento. Em alguns casos, algumas classes poderão não mostrar todos os detalhes, mas podem aparecer para mostrar relacionamentos e aumentar o entendimento.

A partir do diagrama apresentado, podemos também identificar quais repositórios são necessários:

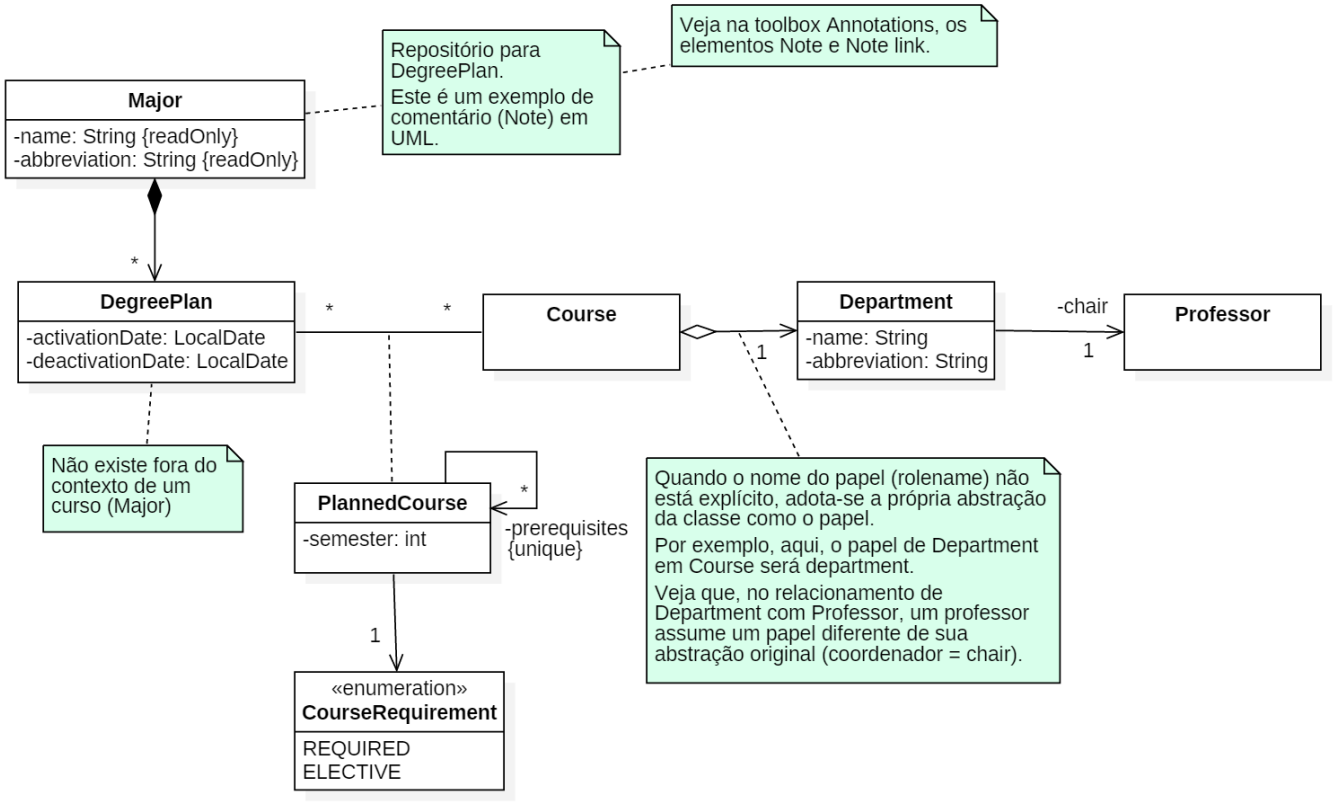
- EmployeeRepository
- ProfessorRepository
- StudentRepository
- CompanyRepository
- ClassRepository
- CourseRepository
- CityRepository
- StateRepository

O que estes repositórios têm em comum? O fato de que eles são entidades de negócio (existem fisicamente em um sistema acadêmico) e que não são composições de outras abstrações. Você deve ter notado que `Address`

Roteiro 08 – Atividade 03/07

não tem repositório. Isso acontece porque objetos `Address` são parte (composição) de `Person` e não existem fora desta classe. A classe Person, por sua vez, é abstrata e não existe fisicamente em sistema acadêmico (negócio). Classes de associação, conforme a nossa implementação, não existem sozinhas e são gerenciadas pelas classes relacionadas. Por exemplo, a classe de associação `Enrollment` é gerenciada pelas classes `Class` e `Student`.

2. Agora, estude a proposta de continuação da nossa modelagem. Modele e complemente o diagrama. Depois, implemente todas as classes, incluindo ajustes que forem necessários. Realize testes para garantir que sua implementação está funcionando.



3. Crie uma ou mais classes de teste para demonstrar as seguintes funcionalidades implementadas (você pode utilizar como base a classe de teste fornecida na atividade anterior):
   - Criar um curso de Bacharelado em Ciência da Computação.
   - Criar um departamento de Ciência da Computação com as disciplinas:
     - ALG – Algoritmos
     - C1 – Cálculo
     - C2 – Cálculo
     - ED – Estrutura de Dados
     - POO – Programação Orientada a Objetos
     - ES1 – Engenharia de Software
     - ES2 – Engenharia de Software
     - BD1 – Banco de Dados
     - BD2 – Banco de Dados
     - MD – Matemática Discreta
     - TOP – Tópicos em Computação
     - TCC1 – Trabalho de Conclusão de Curso
     - TCC2 – Trabalho de Conclusão de Curso

Roteiro 08 – Atividade 03/07

- Crie uma grade curricular para o curso criado, conforme a tabela abaixo:

Disciplina (Course)	Sem	Pré-requisitos
ALG – Algoritmos	1	-
C1 – Cálculo	1	-
C2 – Cálculo	2	C1
ED – Estrutura de Dados	2	ALG
POO – Programação Orientada a Objetos	2	ALG
ES1 – Engenharia de Software	3	ED, POO
ES2 – Engenharia de Software	4	ES1
BD1 – Banco de Dados	3	ED, POO
BD2 – Banco de Dados	4	BD1
MD – Matemática Discreta	3	ALG
TOP – Tópicos em Computação	4	-
TCC1 – Trabalho de Conclusão de Curso	5	ES2, BD2
TCC2 – Trabalho de Conclusão de Curso	6	TCC1

- A grade curricular deve ser capaz de calcular a quantidade de créditos total (somatório dos créditos das disciplinas que a compõe).
- Crie uma lista de Professores e Estudantes (para facilitar o acompanhamento dos resultados, inicie o nome dos professores sempre com Prof.).
- Imprima a grade curricular criada para verificar se tudo está ok. Ordene a grade pelo semestre e pelo código da disciplina. Utilize uma classe anônima (`Comparator`) de modo similar ao que já foi utilizado em roteiro anterior.
- Crie turmas para cada uma das disciplinas.
    - Este passo pode ser sincronizado com o próximo, onde, inicialmente seriam criadas as turmas para os calouros. Depois, seriam simulados os outros semestres de modo incremental, juntamente com as matrículas.
- Simule a matrícula de estudantes nestas turmas.
    - Você pode considerar inicialmente que todos os estudantes são calouros.
    - Depois, simule novas matrículas, verificando se as validações de pré-requisitos estão funcionando.
    - Os testes devem mostrar as salas criadas no semestre.
- Imprima o “diário” de cada turma criada.
- Utilize a classe HashMap para os dados de teste (veja uma ideia na próxima página).

Roteiro 08 – Atividade 03/07

```
/**
 * Cria base de professores.
 */
private void createProfessors() {
    professors.put("bruce", new Professor("Prof. Bruce Wayne", "02/05/1996", "wayne@com", address, "01/03/2017", 40, "20.34",
    professors.put("strange", new Professor("Prof. Stephen Vincent Strange", "23/09/1994", "frost@com", address, "31/07/2016"
}

/**
 * Cria base de estudantes.
 */
private void createStudents() {
    students.put("orin", new Student("Orin Curry", "02/05/1996", "wayne@com", address, "01/03/2017"));
    students.put("susan", new Student("Susan Kent-Barr", "23/09/1994", "frost@com", address, "31/07/2016"));
    students.put("emma", new Student("Emma Grace Frost", "23/09/1994", "susan@com", address, "31/07/2016"));
}

/**
 * Cria base de departamentos.
 */
private void createDepartments() {
    departments.put("DCC", new Department("Ciência da Computação", "DCC"));
}

/**
 * Cria base de disciplinas.
 */
private void createCourses() {
    Department cc = departments.get("DCC");
    courses.put("ALG" , new Course("ALG" , "Algoritmos", "Ementa Algoritmos", 4, cc));
    courses.put("C1"  , new Course("C1"  , "Cálculo", "Ementa Cálculo 1", 4, cc));
    courses.put("C2"  , new Course("C2"  , "Cálculo", "Ementa Cálculo 2", 4, cc));
    courses.put("ED"  , new Course("ED"  , "Estrutura de Dados", "Ementa Estrutura de Dados", 4, cc));
    courses.put("POO" , new Course("POO" , "Programação Orientada a Objetos", "Ementa Programação Orientada a Objetos", 4, cc
    courses.put("ES1" , new Course("ES1" , "Engenharia de Software", "Ementa Engenharia de Software 1", 4, cc));
    courses.put("ES2" , new Course("ES2" , "Engenharia de Software", "Ementa Engenharia de Software 2", 4, cc));
    courses.put("BD1" , new Course("BD1" , "Banco de Dados", "Ementa Banco de Dados 1", 4, cc));
    courses.put("BD2" , new Course("BD2" , "Banco de Dados", "Ementa Banco de Dados 2", 4, cc));
    courses.put("MD"  , new Course("MD"  , "Matemática Discreta", "Ementa Matemática Discreta", 4, cc));
    courses.put("TOP" , new Course("TOP" , "Tópicos em Computação", "Ementa Tópicos em Computação", 4, cc));
    courses.put("TCC1", new Course("TCC1", "Trabalho de Conclusão de Curso", "Ementa Trabalho de Conclusão de Curso 1", 4, cc
    courses.put("TCC2", new Course("TCC2", "Trabalho de Conclusão de Curso", "Ementa Trabalho de Conclusão de Curso 2", 4, cc
}

/**
 * Cria base de grades.
 */
private void createPlans() {
    plans.put("CC2017", new DegreePlan(LocalDate.parse("01/01/2017", AppConfig.DATE_FORMAT)));
}

/**
 * Cria base de relacionamentos entre disciplina e grade.
 */
private void createPlannedCourses() {
    List<PlannedCourse> prereq;
    plannedCourses.put("ALG_CC2017" , new PlannedCourse(courses.get("ALG") , plans.get("CC2017"), 1, null));
```