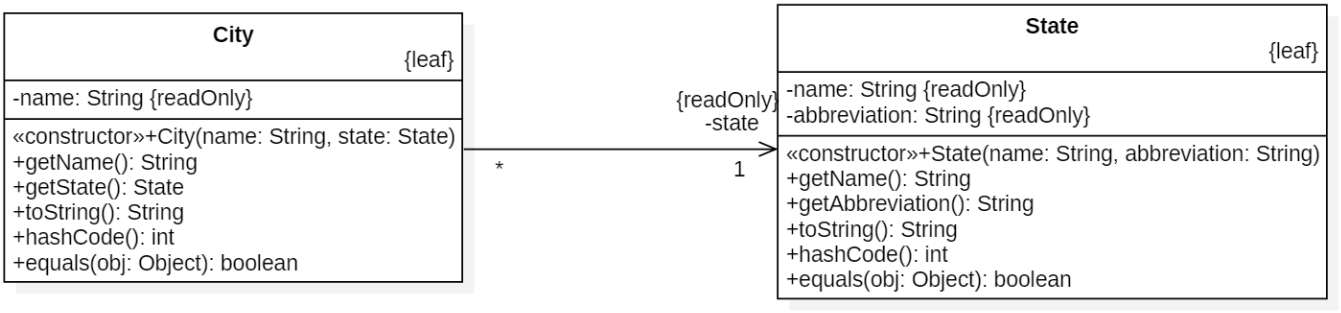
Roteiro 07 – Atividade 01/07

1. Observe o diagrama abaixo. Note que as duas classes (`City` e `State`) foram modeladas como imutáveis. Na prática, os objetos instanciados a partir destas classes é que serão imutáveis. Lembre-se que um objeto imutável é aquele que não pode ter seu estado (valor de qualquer um de seus atributos) modificado após sua instanciação.



   Para implementarmos uma classe imutável, ela deve seguir as seguintes regras:

   a) Não permitir redefinição (*override*) de seus métodos. Para garantir isso, utilizaremos o modificador `final` na classe. A notação UML para uma classe `final` é `{leaf}`.
   b) Todos os atributos devem ser privados (nenhuma novidade aqui) e `final` (não poderão ser alterados depois de inicializados). A notação UML para um atributo `final` é `{readOnly}`.
   c) Não fornecer operações que permitam alterar o valor dos atributos (estado) de um objeto após ele ter sido instanciado (ex: *setters*).
   d) Todos os atributos devem ser inicializados no construtor da classe.
   e) Se houver atributos que são referências para objetos mutáveis, não permitir que estes objetos sejam modificados. Para isso, utilizaremos uma técnica chamada **cópia defensiva**. Quando este atributo for acessado (por exemplo, por meio de um *getter*), o objeto retornará uma cópia idêntica de seu atributo e não a simples referência. Desta forma, quem fez a chamada não conseguirá alterar o atributo original do objeto, mantendo a imutabilidade.

2. Modele em seu diagrama e implemente a classe `State` (mesmo que Unidade Federativa-UF), considerando as regras para uma classe imutável. Note que você não precisa se preocupar com cópia defensiva aqui, uma vez que todos os atributos de `State` são imutáveis (`String`). Veja que no trecho de código abaixo, já estamos atendendo às regras (a) e (b).

```
public final class State {
    private final String name;
    private final String abbreviation;
```

   Não esqueça de preencher também o javadoc. Você pode utilizar o recurso de inserção de código do NetBeans para agilizar a implementação (Alt + INS).

3. Na sua implementação, você lembrou de realizar as validações? Isso é crítico para que um objeto imutável seja inconsistente. Uma vez que, depois de instanciado, um objeto imutável não poderá ser alterado, é fundamental que ele seja validado no momento de sua criação. Lembre-se também que a classe deve ter um único construtor.

```
/**
 * @param name o nome do estado da União
 * @param abbreviation a sigla do estado da União
 */
public State(String name, String abbreviation) {
    StringValidator val = new StringValidator();
    // name e abbreviaton precisam ter, no mínimo, 1 palavra
    val.minWordsCount(name, "Name", 1);
    val.maxWordsCount(abbreviation, "Abbreviation", 1);
    this.name = name;
    this.abbreviation = abbreviation;
}
```

Implementação das validações.

Roteiro 07 – Atividade 01/07

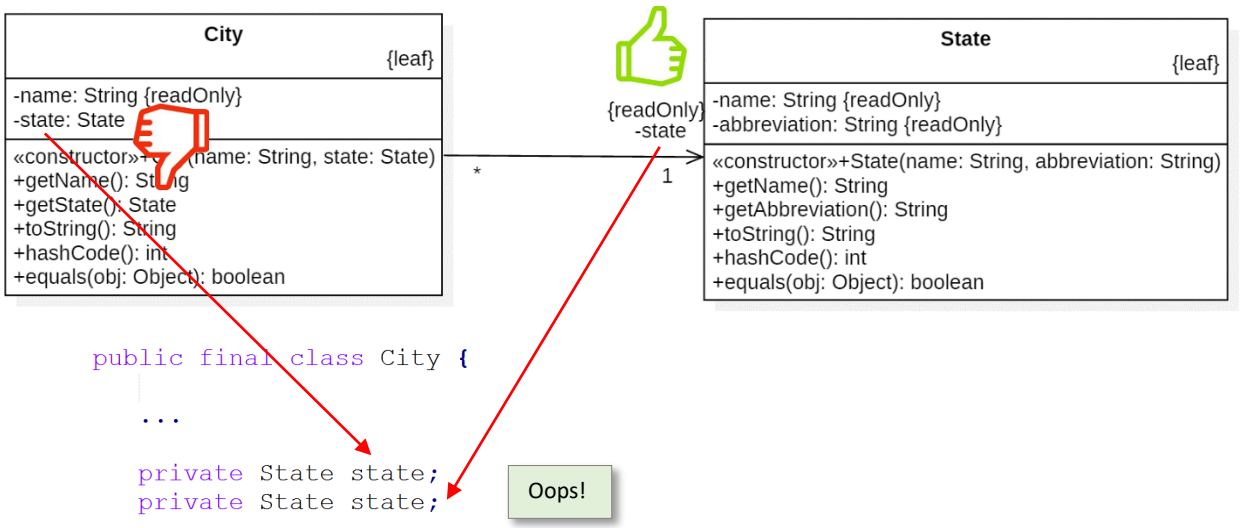
4. Realize testes para verificar se sua classe está funcionando. Alguns exemplos:

```
State s1 = new State("Santa Catarina", "SC");
System.out.println(s1.toString());

State s2 = new State("Santa Catarina", "SC");
System.out.println(s2.toString());

System.out.println("TRUE  ==> " + s1.equals(s2));
System.out.println("TRUE  ==> " + s2.equals(s1));
System.out.println("FALSE ==> " + (s1 == s2));

// exceção aqui...
State s3 = new State("Santa Catarina", "");

// exceção aqui...
State s4 = new State("", "SC");
```

Não são a mesma referência, mas são iguais (têm o mesmo estado).

Validação deve garantir que estes objetos não sejam instanciados.

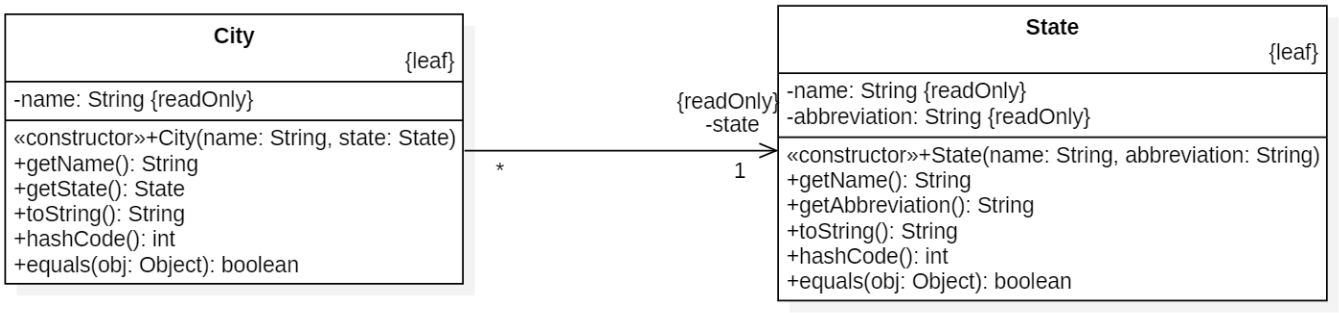
5. Uma última coisa sobre o diagrama do passo 1. Repare que o atributo `state` não foi representado no compartimento dos atributos da classe `City`. O desenho está correto porque o atributo já está representado no relacionamento, por meio do nome do papel assumido por `State`. Nunca utilize as duas representações simultaneamente (no compartimento dos atributos e por meio de um relacionamento), pois a interpretação será que existem dois atributos. No diagrama abaixo, esta situação é apresentada.



Sempre dê preferência a representação dos atributos por meio de relacionamentos (nomes de papel/*rolenames*) com outras classes (com exceção de classes da API Java ou de alguma biblioteca/framework conhecida).

Roteiro 07 – Atividade 02/07

1. Antes de você modelar e implementar a classe `City`, vamos analisar novamente o diagrama apresentado na atividade 01.



   A partir dele, podemos extrair algumas regras de negócio:

   - UM objeto `City` pode referenciar (apontar para) UM único objeto `State`.
   - UM mesmo objeto `State` pode ser referenciado por VÁRIOS objetos `City`.
   - UM objeto `State` não sabe quem o referencia (a navegabilidade é somente de `City` para `State`).
   - O objeto `State` não pode ser modificado após a instanciação de um objeto `City` (o relacionamento de associação indica que ele é privado e pode ser apenas lido – propriedade *readOnly*).

   Para implementar a classe City, também podemos extrair as seguintes orientações a partir do diagrama:

   - A classe `City` deve ser `final` (ela tem a propriedade *Leaf*).
   - O atributo que representa a referência para `State` deve ser chamado de `state` (veja o nome do papel no relacionamento).
   - O atributo `state` deve ser `private final` (veja a propriedade *readOnly* e a visibilidade "-" indicadas no nome do papel).
   - O atributo `name` deve ser `private final` (propriedade *readOnly*).

   Se você teve dificuldades em extrair as informações anteriores apenas observando o diagrama de classe, revise os roteiros anteriores, os slides sobre orientação a objetos disponibilizados no material didático e suas notas do semestre anterior.

2. Agora sim, atualize seu diagrama e implemente a classe `City`. Como o atributo `state` é imutável (atividade anterior), não precisaremos realizar cópia defensiva (o chamador não conseguirá modificar o valor de `state`). Não esqueça do javadoc e das validações (`state` não pode ser `null` e `name` tem que ter, pelo menos, uma palavra).

3. Teste a classe implementada.

```
State sl = new State("Santa Catarina", "SC");
City cl = new City("Florianópolis", sl);
System.out.println(cl.toString());
```

```
run:
br.univali.kob.poo1.aula07a.City {
  name = Florianópolis
  state = br.univali.kob.poo1.aula07a.State {
  name = Santa Catarina
  abbreviation = SC
}
}

BUILD SUCCESSFUL (total time: 0 seconds)
```

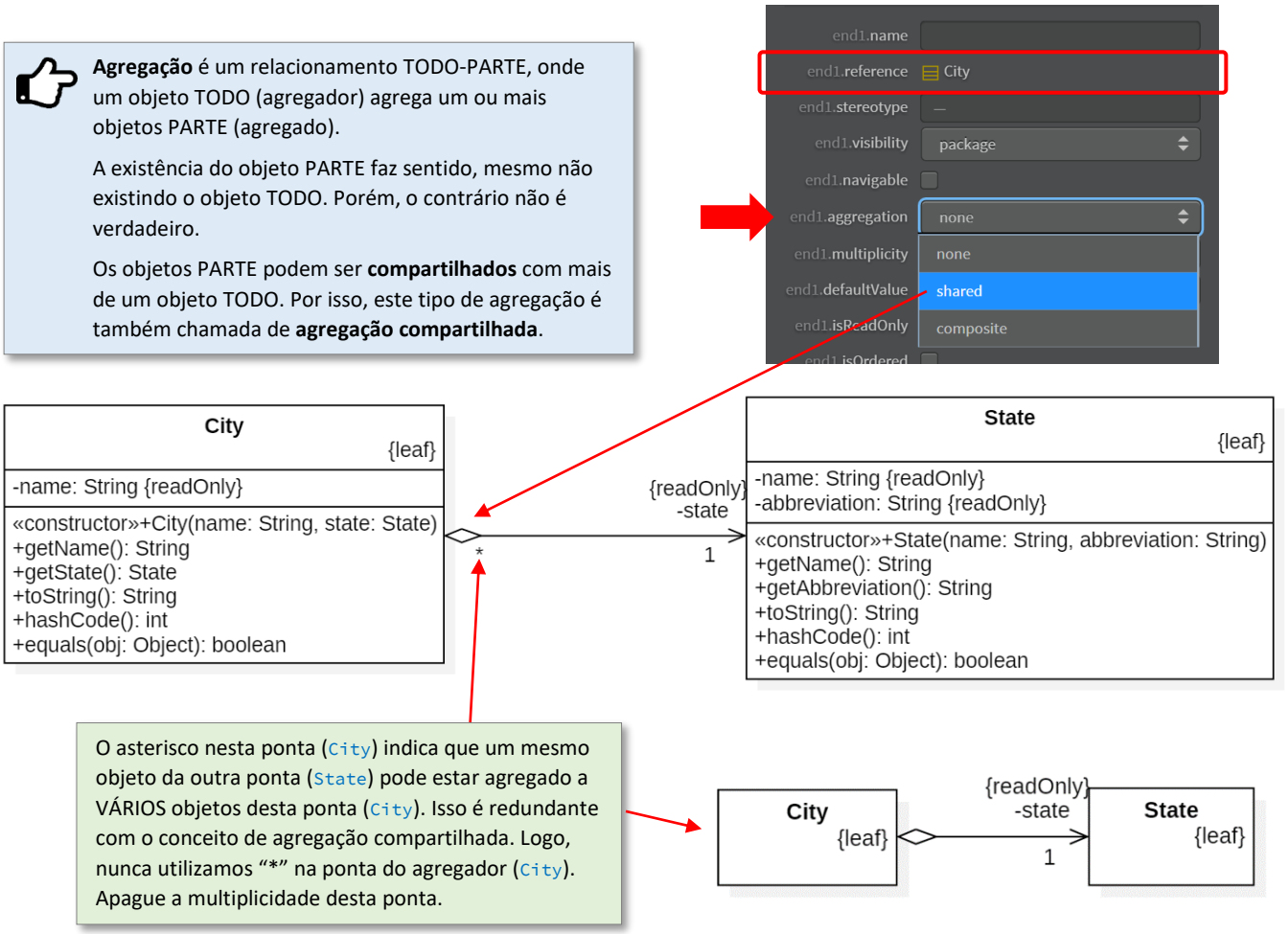
state

Roteiro 07 – Atividade 03/07

1. Voltando ao nosso diagrama inicial, vamos discutir agora o relacionamento entre as classes[1]. Inicialmente, foi utilizado o relacionamento de associação. Entretanto, vamos ponderar sobre as seguintes afirmações:

   - A classe `State` é independente: ela não depende de outras classes da aplicação (não consideraremos classes da API Java nesta análise).
   - Um objeto `State` pode existir na aplicação sem que nenhum outro objeto tenha sido instanciado: podemos, por exemplo, ter uma funcionalidade para cadastrar Unidades da Federação.
   - A classe `City` depende da classe `State`: modificações em `State` podem afetar o comportamento de um objeto `City`.
   - Um objeto `City` deve, obrigatoriamente, referenciar um objeto `State`: não faz sentido termos uma cidade sem que ela tenha uma Unidade da Federação. Logo, um objeto `State` é parte integrante de um objeto `City`.
   - Um mesmo objeto `State` pode ser compartilhado por vários objetos `City`.
   - Um objeto `State` não pode ser destruído enquanto houver, pelo menos, um objeto `City` o referenciando.
   - Quando um objeto `City` é destruído, o objeto `State` referenciado continua existindo.

   A partir desta análise, podemos concluir que o relacionamento mais adequado seria o de **agregação compartilhada (*shared*)**.

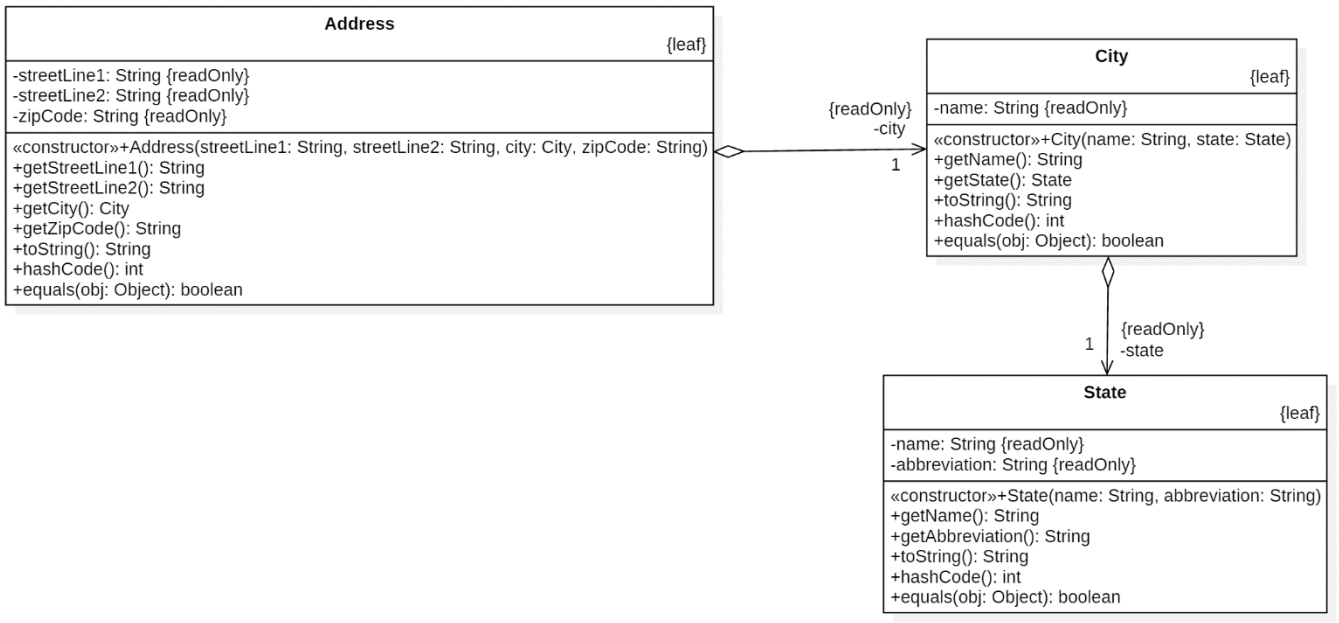


   A sua implementação não precisa ser modificada, pois já garantimos que um objeto `City` não pode ser instanciado sem que um objeto `State` exista. Por outro lado, não há como destruir um objeto `State` que esteja sendo referenciado por algum objeto `City` (o *garbage collector* só destrói objetos que não tem mais referência).

[1] Na prática, o relacionamento acontecerá entre os objetos da classe.

Roteiro 07 – Atividade 04/07

1. Agora, vamos incluir mais uma classe: `Address`. Um endereço agrega uma cidade, pois a informação sobre a cidade é parte de um endereço.



   Para as validações, considere que apenas a segunda linha do endereço pode ser `null`. A primeira linha corresponde ao nome do logradouro (rua, avenida, servidão, etc.) e ao número. A segunda linha corresponde ao complemento (por exemplo, “Apto 101”). O atributo `zipCode` corresponde ao código do CEP.

2. Modele, implemente e teste `Address`. Considere também que ela é imutável.

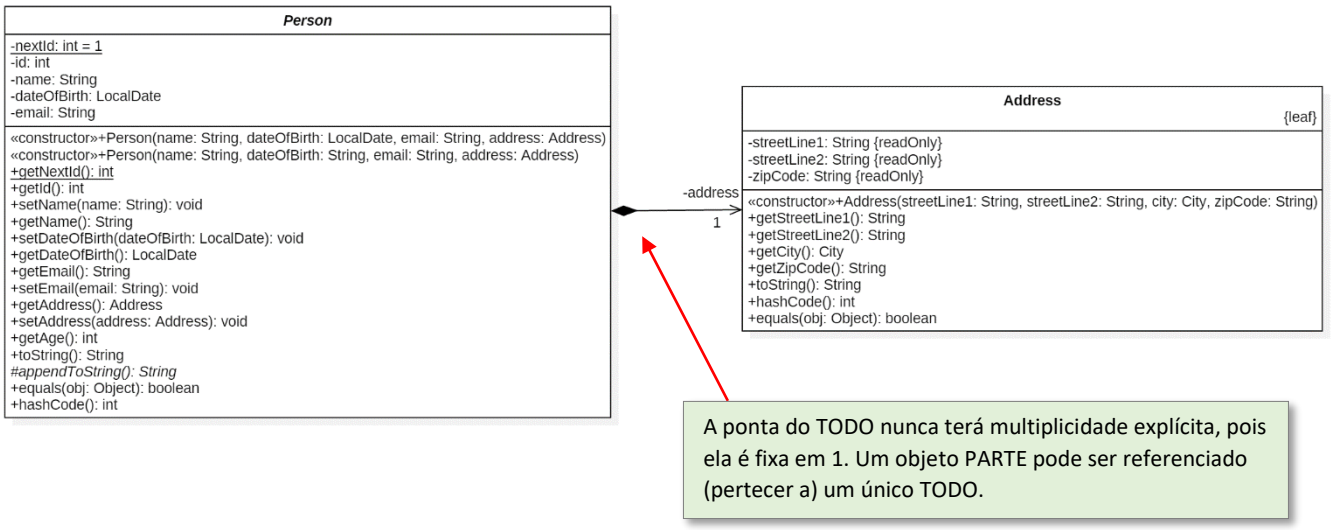
```
State sl = new State("Santa Catarina", "SC");
City cl = new City("Florianópolis", sl);
Address al = new Address("Rua Floriano, 2012", null, cl, "88015200");
System.out.println(al.toString());
```

Roteiro 07 – Atividade 05/07

1. Agora, vamos considerar que uma pessoa deve ter um endereço. Este relacionamento é mais forte do que uma agregação (compartilhada). Embora ainda seja um relacionamento TODO-PARTE, devemos considerar algumas regras adicionais:

   a) Não sentindo termos um objeto `Address` (PARTE) sem alguém ou alguma coisa (TODO) que o referencie; logo, quando destruirmos um objeto `Person` (TODO), o objeto `Address` (PARTE) deveria ser também destruído.
   b) Também não faz sentido compartilhar o objeto `Address` (PARTE) com outros objetos `Person` (TODO), pois cada pessoa tem seu próprio endereço; logo, um objeto `Address` (PARTE) pode pertencer a um e somente um objeto `Person` (TODO).

   Uma agregação que atende às regras acima é chamada de **agregação de composição** (ou somente **composição**). No diagrama abaixo, podemos ver a notação para uma composição. Repare que ela é similar à uma agregação (compartilhada), mas o losango é preenchido com uma cor sólida (preta).



   A partir deste diagrama, podemos afirmar que um objeto `Person` é composto de um objeto `Address`. O método `setAddress` utiliza a mesma lógica dos *setters* vistos até aqui (é preciso validar antes de atribuir o valor). Entretanto, para atendermos às regras da composição, o método `getAddress` deve retornar uma cópia. Como o objeto `Address` é imutável, esta solução não seria necessária. Mesmo passando a referência, o objeto chamador nunca conseguirá deixar o objeto chamado inconsistente. Mas para exemplificar como seria uma cópia defensiva neste caso, a implementação de `getAddress` ficaria como a imagem abaixo

```java
/**
 * @return uma cópia do endereço da pessoa
 */
@Override
public Address getAddress() {
    return new Address(address.getStreetLine1(), address.getStreetLine2(),
            address.getCity(), address.getZipCode());
}
```

Estamos retornando outro objeto, mas como o mesmo estado.

2. Modele, implemente e teste os ajustes na classe `Person`. Lembre-se que você precisará modificar também os construtores além das operações `equals`, `hashCode` e `toString`. Atualize também o javadoc. Depois, atualize os construtores das classes `Student`, `Employee` e `Professor`.

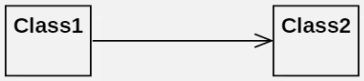
Roteiro 07 – Atividade 05/07

Orientações para decidir sobre qual relacionamento é mais adequado

Dependência

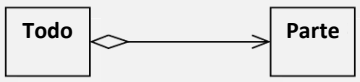
- A Class1 é dependente de Class2 quando a implementação de Class1 utiliza de alguma forma Class2; sempre unidirecional.
- Class1 é considerada o cliente e Class2 o provedor.
- A simples chamada a uma operação da Class2 ou a declaração de uma variável Class2 já configura dependência de Class1 para Class2.
- Qualquer mudança em Class2 pode ter impacto em Class1.
- Pode ser utilizado com outros elementos de notação da UML (ex: *packages*, *components, interfaces*).

Associação



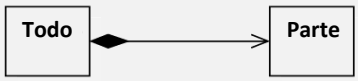
- Relacionamento n-ário entre objetos (uni ou multidirecional).
- Simétrico (se um objeto Class1 está associado a um objeto Class2, então um o objeto Class2 está associado ao objeto Class1).
- A ligação tem duração maior do que a execução de uma única operação.
- O relacionamento entre os objetos não faz com que eles dependam um do outro para existir
- Se o objeto A for destruído, o B continua existindo e vice-versa

Agregação
agregação compartilhada



- Relacionamento binário entre objetos (TODO-PARTE fraco); uni ou bidirecional.
- Assimétrico (se um objeto A é parte de um objeto B, então o objeto B não pode ser parte do objeto A).
- Um objeto todo não faz sentido sem seu(s) objeto(s) parte.
- Um objeto parte faz sentido sem seu objeto todo.
- O objeto parte pode ser compartilhado entre vários objetos todo (ligação por referência).
- Quando o objeto todo é destruído, o objeto parte continua existindo.
- Um objeto parte só pode ser destruído se ele não fizer parte de algum objeto todo.

Composição
agregação de composição



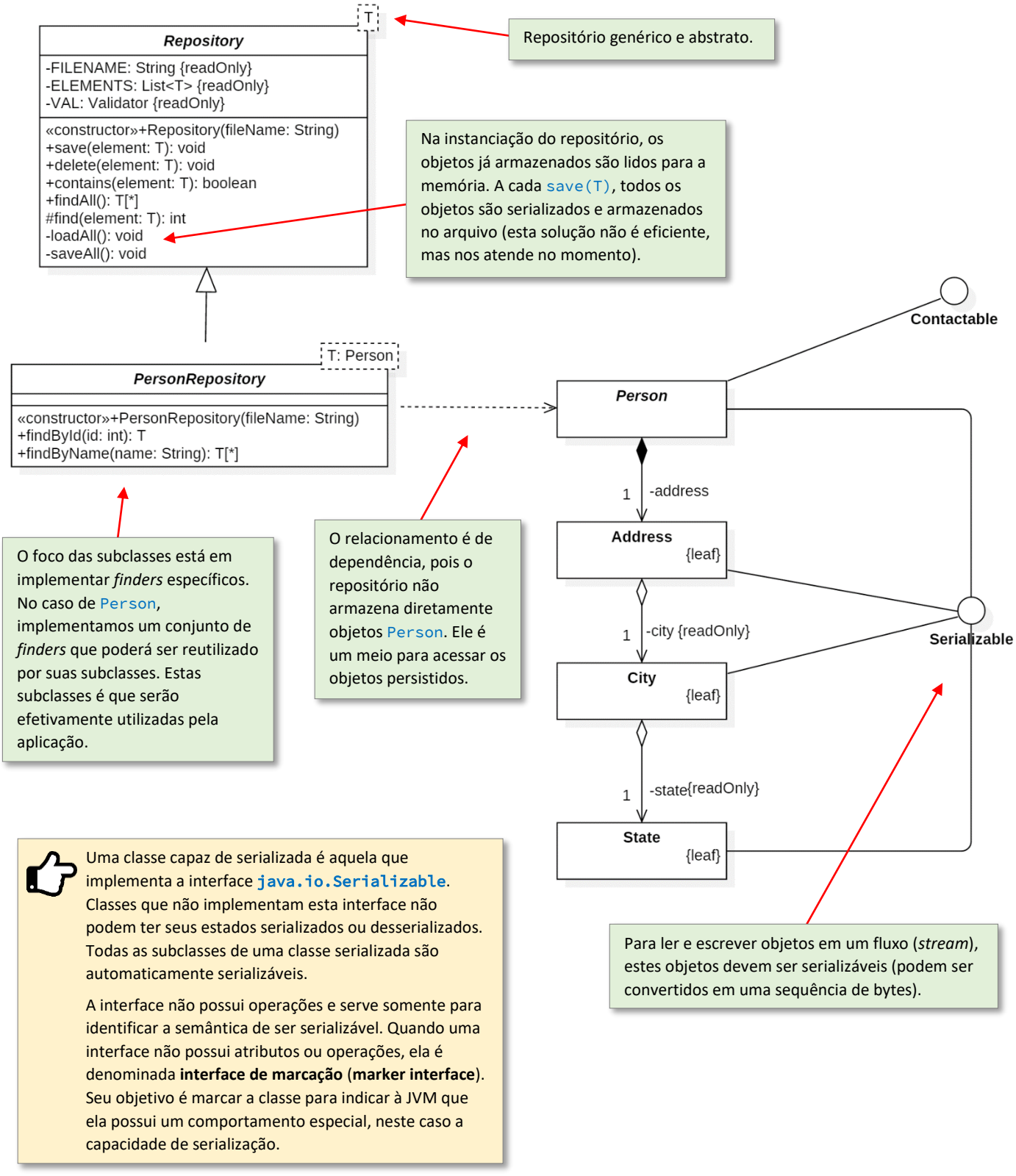
- Relacionamento binário entre objetos (TODO-PARTE forte); uni ou bidirecional.
- Assimétrico (se um objeto A é parte de um objeto B, então o objeto B não pode ser parte do objeto A).
- Um objeto todo não faz sentido sem seu(s) objeto(s) parte.
- Um objeto parte não faz sentido sem seu objeto todo.
- Um objeto parte é exclusivo de seu objeto todo (ligação por valor).
- Quando o objeto todo é destruído, o objeto parte também é destruído.

Observe que associação, agregação e composição também demonstram dependência entre classes. Se um objeto de uma classe possui uma referência para um objeto de outra classe, a primeira classe depende da segunda.

Roteiro 07 – Atividade 06/07

1. Nesta atividade, iremos criar repositórios para nossos objetos de negócio. Os objetos serão persistidos em arquivos binários (fluxos de bytes). Abra e consulte o arquivo **java.io - Fluxos (streams) e Arquivos (Marcello Thiry).pdf**, disponibilizado com este roteiro.

2. Agora, abra os arquivos **Repository.java** e **PersonRepository.java**. As classes presentes nestes arquivos estão modeladas no diagrama abaixo:



Roteiro 07 – Atividade 06/07

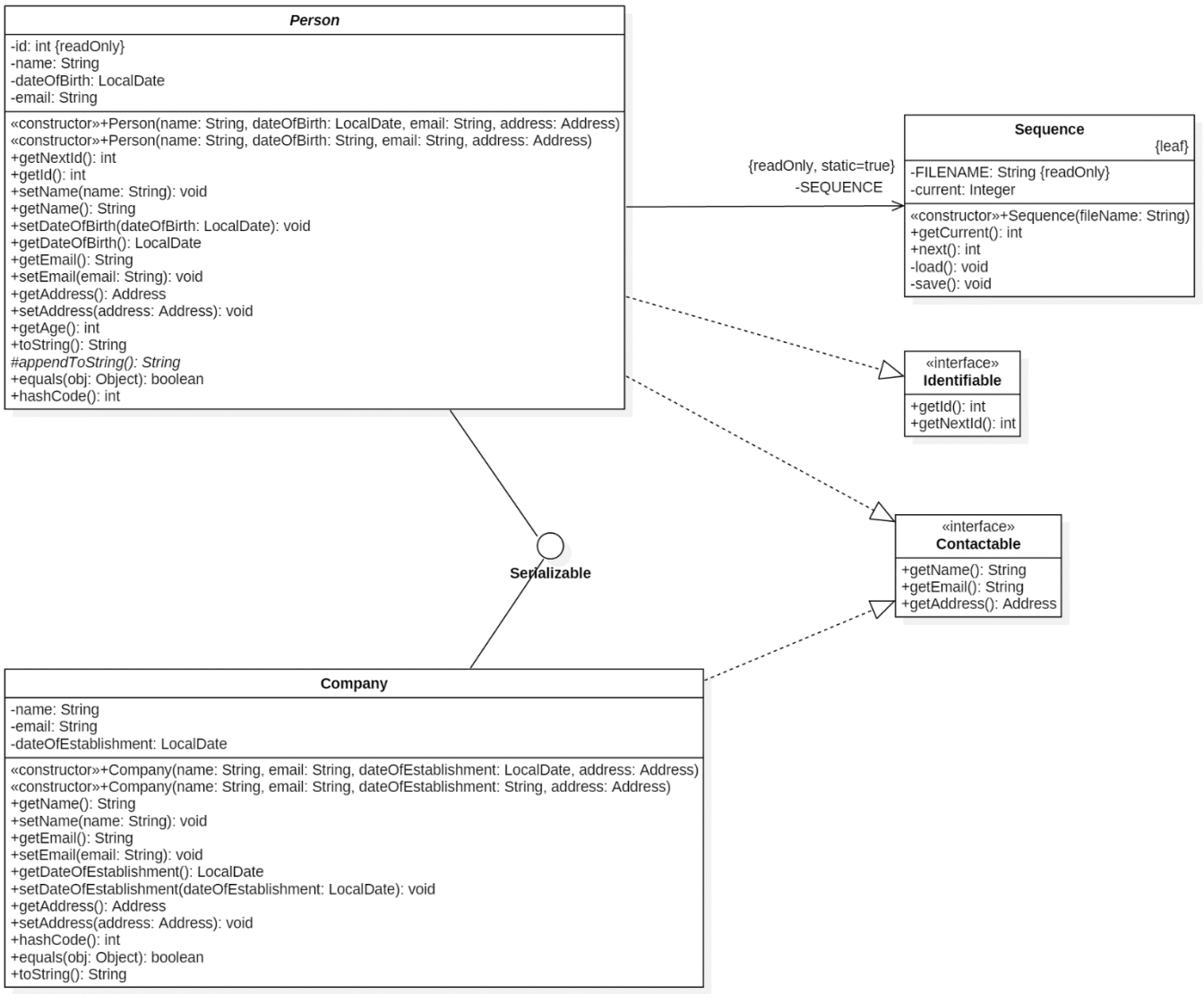
3. Modele e implemente as classes de Repositório. Garanta que todas as classes a serem persistidas implementem a interface `Serializable`.

4. Modele e implemente um repositório para: `Student`, `Employee`, `Professor` e `Company`. Na persistência, vamos considerar o nome da classe persistida com a extensão ".DAT". Por exemplo, passaremos para o construtor de `StudentRepository` o `fileName` "**C:\\Users\\marcello\\Documents\\Student.DAT**".

5. Implemente testes para confirmar que tudo está funcionando conforme o esperado.

Roteiro 07 – Atividade 07/07

1. Agora que implementamos persistência de nossos objetos, temos um novo problema. Atualmente, a implementação do ID é por meio de um atributo de classe, o qual é inicializado no momento em que a JVM carrega as classes para a memória. Logo, sempre que executamos nossos programas, o ID inicial volta a ser 1. Isso gerará números repetidos, quebrando nossa regra de negócio. Para resolver este problema, faremos algumas melhorias em nossa solução. Inicialmente, separaremos a responsabilidade de identificar um próximo ID da classe Person. Esta responsabilidade passará a ser de uma classe chamada `Sequence`. Esta classe será construída passando o nome do arquivo onde será persistido o valor atual de ID para uma determinada classe. Modele o diagrama abaixo (você precisa entendê-lo antes de implementá-lo).



Algumas observações sobre o diagrama:

- A classe `Sequence` utiliza a classe *wrapper* `Integer`. Lembre-se que escrevemos e lemos objetos.
- A implementação da escrita e leitura em arquivo de `Sequence` não precisa ser *bufferizada*, pois sempre será escrito e lido um único objeto.
- O nome do arquivo passado no construtor de `Sequence` segue a mesma regra do nome passado para a persistência nos repositórios, trocando apenas a extensão por ".SEQ". Como a classe `Person` implementa o controle para todas as suas subclasses, teremos "**Person.SEQ**".
- A operação `getAddress` foi adicionada na interface `Contactable`.
- Foi criada a interface `Identifiable`, a qual é implementada por `Person`.

Roteiro 07 – Atividade 07/07

2. Como as operações de leitura e escrita de objetos podem disparar várias exceções checadas, iremos criar uma exceção não checada que encapsulará uma exceção checada, caso ela aconteça. Isso simplificará a implementação e evitará várias exceções definidas na cláusula `throws`.

```java
/**
 * Exceção disparada quando houver algum problema de leitura ou escrita
 * no arquivo de sequência.
 *
 * @author Marcello Thiry
 */
public class InvalidSequenceException extends IllegalStateException {

    /**
     * Caminho completo para o arquivo de sequência.
     */
    private final String fileName;
    /**
     * Exceção checada encapsulada.
     */
    private final Exception ex;

    /**
     * Encapsula exceção checada na leitura/escrita de um arquivo de sequência.
     *
     * @param fileName o caminho completo para o arquivo de sequência
     * @param ex a exceção checada encapsulada
     */
    public InvalidSequenceException(String fileName, Exception ex) {
        super(ex.getMessage() + " - Filename: \"" + fileName + "\".", ex);
        this.fileName = fileName;
        this.ex = ex;
    }

    /**
     * @return o caminho completo para o arquivo de sequência
     */
    public String getFileName() {
        return fileName;
    }

    /**
     * @return a exceção checada encapsulada
     */
    public Exception getException() {
        return ex;
    }

}
```

Exceção não checada. Foi considerado que a exceção ocorre quando o chamador tentar quebrar a consistência do estado de um objeto `Sequence`.

Veja que criamos uma nova exceção que encapsula a original. Essa técnica é chamada **encadeamento de exceções (*exception chain*)**.

3. Veja como utilizar a exceção criada no trecho de código abaixo (`Sequence`).

```java
/**
 * Carrega o valor corrente da sequência do arquivo para a memória.
 *
 * @throws InvalidSequenceException se houver algum problema com o arquivo
 */
private void load() {
    // não precisa usar um buffer
    try (ObjectInput input = new ObjectInputStream(new FileInputStream(FILENAME));) {
        current = (Integer)input.readObject();
    } catch (FileNotFoundException ex) {
        // se caminho\arquivo ainda não existe
        current = 0;
    } catch (Exception ex) {
        // qualquer outra exceção será encapsulada
        throw new InvalidSequenceException(FILENAME, ex);
    }
}
```

Encadeamento de exceções (*exception chain*).