Eric Michaud IPI- CDEV191

IMDD070 Design Patterns

Rendu des travaux e-learning

**Exercice : Jeu de rôle (modélisation UML diagramme de classe)**

Toutes les personnes ont un nom et un niveau

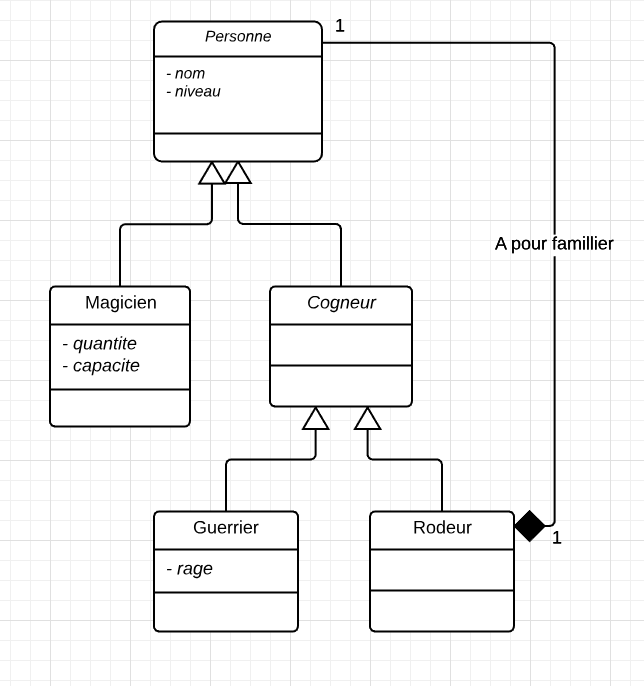
Deux grandes catégories : magiciens et cogneurs

Les magiciens ont besoin de « mana », ils disposent d’une certaine quantité et d’une capacité maximale

Les cogneurs sont soit des guerriers soit des rôdeurs.

Les guerriers doivent accumuler de la rage

Les rôdeurs sont toujours accompagnés d’un familier qui possède son propre nom et son propre niveau. Ils sont liés à la vie à la mort, si l’un meurt l’autre aussi

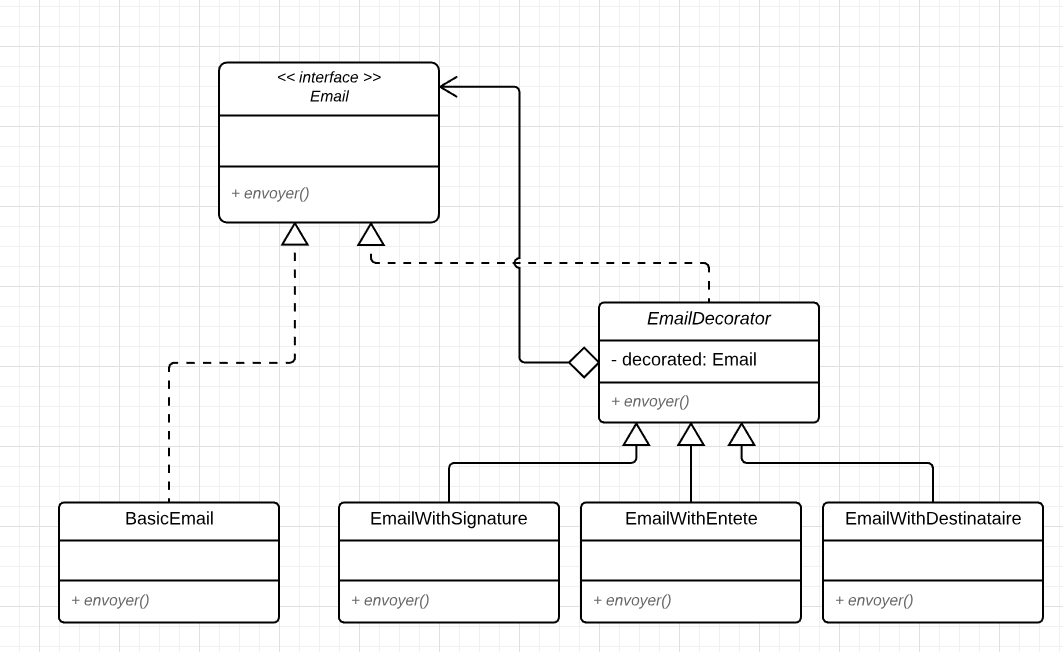


## Exercice Decorator pattern

Soit la classe Email,   
avec un constructeur prenant le paramètre "texte" de type String   
et la méthode envoyer() qui écrit dans la console le contenu de l'email  
On souhaite ajouter les fonctionnalités suivantes :

1. envoyer un texte avec signature ajoutée automatiquement
2. envoyer un texte avec un "Bonjour madame, monsieur" ajouté automatiquement
3. envoyer un texte avec une signature et un "Bonjour madame, monsieur" ajoutés automatiquement  
   Proposer une solution utilisant le Decorator Pattern : UML + code JAVA (ou pseudo code) + code pour produire l'envoi des 3 messages cités

### UML



### Code

**public interface** Email {  
 **public** voidenvoyer();  
}

**public abstract class** EmailDecorator **implements** Email {  
  
 **private** Email **decorated**;  
  
 **public** EmailDecorator(Email email){  
 **decorated** = email;  
 }  
  
 @Override  
 **public void** envoyer() {  
 **decorated**.envoyer();  
 }  
  
}

**public class** BaseEmail **implements** Email {  
  
 **private** String **message**;  
  
 **public** BaseEmail(String message) {  
 **this**.**message** = message;  
 }  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 **public void** setMessage(String message) {  
 **this**.**message** = message;  
 }  
  
 @Override  
 **public void** envoyer() {  
 System.***out***.println(**"email to sent :"** + **message**);  
 }  
  
}

**public class** EmailWithEntete **extends** EmailDecorator {  
  
 **public** EmailWithEntete(Email email) {  
 **super**(email);  
 }  
  
  
 @Override  
 **public void** envoyer(){  
 *//ajoute l'entête* System.***out***.println(**"Entète : Bonjour Madame, Monsieur"** );  
 *// puis traite le reste du mail* **super**.envoyer();  
 }  
}

**public class** EmailWithSignature **extends** EmailDecorator{  
 **public** EmailWithSignature (Email email) {  
 **super**(email);  
 }  
  
  
 @Override  
 **public void** envoyer(){  
 *// traite le début du mail* **super**.envoyer();  
 *// puis ajoute la signature* System.***out***.println(**"Signature: Mochi le Grogneur"**);  
 }  
}

**public class** Main {  
  
 **public static void** main(String args[]){  
  
 Email mail1=**new** BaseEmail(**"Contenu mail 1"**);

Email mail2=**new** EmailWithEntete(**new** BaseEmail(**"Contenu mail 2"**));

Email mail3=**new** EmailWithSignature(**new** BaseEmail(**"Contenu mail 3"**));

Email mail4=**new** EmailWithSignature(**new** EmailWithEntete(  
 **new** BaseEmail(**"Contenu mail 4"**)));

Email mail5=**new** EmailWithEntete(**new** EmailWithSignature(  
 **new** BaseEmail(**"Contenu mail 5"**)));

mail1.envoyer();  
System.***out***.println(**"--------------------------"**);  
mail2.envoyer();  
System.***out***.println(**"--------------------------"**);  
mail3.envoyer();  
System.***out***.println(**"--------------------------"**);  
mail4.envoyer();  
System.***out***.println(**"--------------------------"**);  
mail5.envoyer();

}  
}

email to sent :Contenu mail 1

--------------------------

Entète : Bonjour Madame, Monsieur

email to sent :Contenu mail 2

--------------------------

email to sent :Contenu mail 3

Signature: Mochi le Grogneur

--------------------------

Entète : Bonjour Madame, Monsieur

email to sent :Contenu mail 4

Signature: Mochi le Grogneur

--------------------------

Entète : Bonjour Madame, Monsieur

email to sent :Contenu mail 5

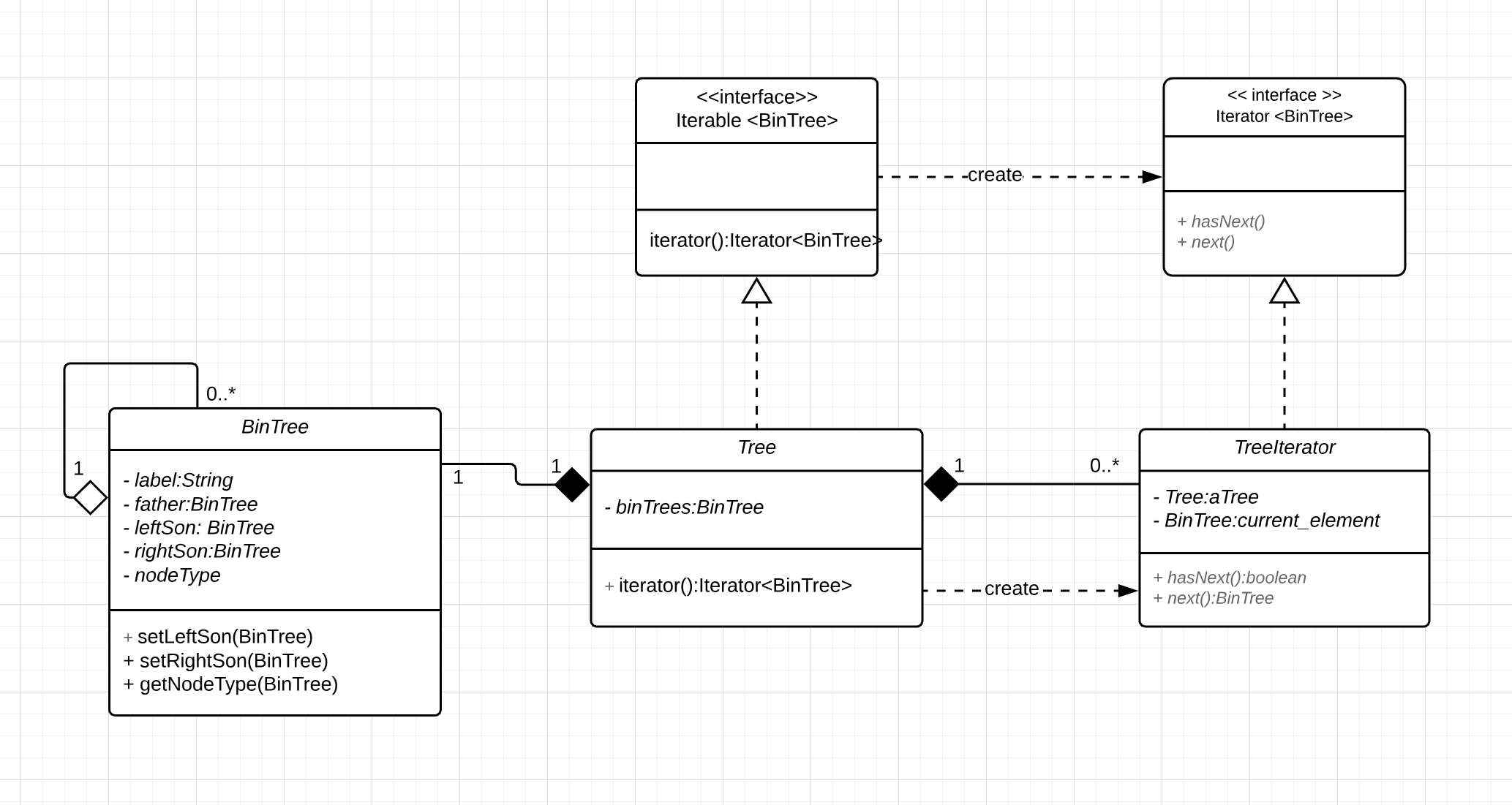
Signature: Mochi le Grogneur

Process finished with exit code 0

## Exercice Iterator pattern

Soit la classe ArbreBinaire possedant le constructure new ArbreBinaire(String etiquette)  
et les propriétés (get/set) filsGauche et filsDroit de type ArbreBinaire,   
et le getter de le propriété etiquette  
A l'aide du pattern iterator proposez une conception permettant d'itérer sur chaque noeud de l'arbre  
UML + code JAVA (ou pseudo code) + code pour créer un arbre et le parcourir, en le parcourant on écrira dans la console le nom de l'étiquette

### UML



### Code

public class Tree implements Iterable<BinTree> {  
  
 private BinTree binTrees;  
  
 public BinTree getBinTrees() {  
 return binTrees;  
 }  
  
 public Tree( String name ){  
  
 binTrees= new BinTree(name);  
   
 };  
  
  
 @Override  
 public Iterator<BinTree> iterator() {  
 return new TreeIterator(this);  
 }

}

public class TreeIterator implements Iterator<BinTree> {  
  
 private Tree tree;  
 private BinTree current\_node;  
 private BinTree next\_node;  
  
 public TreeIterator(Tree tree) {  
 this.tree = tree;  
 // start on first node, ie root  
 this.current\_node = null;  
 this.next\_node = tree.getBinTrees();  
 }  
  
 @Override  
 public boolean hasNext() {  
  
 if (current\_node==null && next\_node!=null) {  
 // that's first iteration no current node but next is root  
 return true;  
 }  
  
 if (current\_node.getLeftSon()==null) {  
 //node is a leaf  
 if (current\_node.getNodeType().equals(NodeType.LEFT)) {  
 //leaf is left, next is right brother  
 next\_node = current\_node.getFather().getRightSon();  
 return true;  
 } else {  
 //leaf is Right, need to climb tree  
 System.out.println("need to climb");  
 next\_node = climb(current\_node);  
 if (next\_node == null) {  
 System.out.println("has no next\_node");  
 return false;  
 }  
 return true;  
 }  
 }  
  
 //node is not a leaf, next is left son  
 next\_node = current\_node.getLeftSon();  
 return true;  
 }  
  
  
  
 @Override  
 public BinTree next() {  
 current\_node = next\_node;  
 return current\_node;  
 }

private BinTree climb(BinTree node) {  
 if (node.getNodeType().equals(NodeType.ROOT)) {  
 // we are on top of tree  
 // that's means there is no follower  
 return null;  
 }  
  
 if (node.getNodeType().equals(NodeType.LEFT)) {  
 // next of a left node is its right brother  
 return node.getFather().getRightSon();  
 }  
  
 // it's a right node need to climb Tree  
 System.out.println("more to climb");  
 return climb(node.getFather());  
 };  
  
}

public class BinTree {  
  
 private String label;  
 private BinTree rightSon;  
 private BinTree leftSon;  
 private BinTree father;  
 private NodeType nodeType;  
  
  
 public BinTree() {  
 this.label = "";  
 this.leftSon = null;  
 this.rightSon = null;  
 this.father = null;  
 this.nodeType = null;  
 }  
  
 public BinTree(String label) {  
 this.nodeType = ROOT;  
 this.father = null;  
 this.label = label;  
 this.leftSon=null;  
 this.rightSon=null;  
 }  
  
  
 public String getLabel() {  
 return label;  
 }  
  
 public void setLabel(String label) {  
 this.label = label;  
 }  
  
 public BinTree getRightSon() {  
 return rightSon;  
 }  
  
 public void setRightSon(String label) {  
 this.rightSon = new BinTree(label);  
 this.rightSon.setFather(this);  
 this.rightSon.setNodeType(NodeType.RIGHT);  
 return;  
 }  
  
 public BinTree getLeftSon() {  
 return leftSon;  
 }  
  
 public void setLeftSon(String label ) {  
 this.leftSon = new BinTree(label);  
 this.leftSon.setFather(this);  
 this.leftSon.setNodeType(NodeType.LEFT);  
 return;  
 }  
  
 public BinTree getFather() {  
 return father;  
 }  
  
 public void setFather(BinTree father) {  
 this.father = father;  
 }  
  
 public NodeType getNodeType() {  
 return nodeType;  
 }  
  
 public void setNodeType(NodeType nodeType) {  
 this.nodeType = nodeType;  
 }  
  
}

public class Main {  
  
 public static void main(String args[]){  
  
 Tree myTree = new Tree("General Sherman");  
 myTree.getBinTrees().setLeftSon(" level 1L");  
 myTree.getBinTrees().setRightSon(" level 1R");  
  
 myTree.getBinTrees().getLeftSon().setLeftSon(" level 2LL");  
 myTree.getBinTrees().getLeftSon().setRightSon(" level 2LR");  
 myTree.getBinTrees().getRightSon().setLeftSon(" level 2RL");  
 myTree.getBinTrees().getRightSon().setRightSon(" level 2RR");  
  
 myTree.getBinTrees().getLeftSon().getLeftSon().setLeftSon(" level 3LLL");  
 myTree.getBinTrees().getLeftSon().getLeftSon().setRightSon(" level 3LLR");  
 myTree.getBinTrees().getLeftSon().getRightSon().setLeftSon(" level 3LRL");  
 myTree.getBinTrees().getLeftSon().getRightSon().setRightSon(" level 3LRR");  
 myTree.getBinTrees().getRightSon().getLeftSon().setLeftSon(" level 3RLL");  
 myTree.getBinTrees().getRightSon().getLeftSon().setRightSon(" level 3RLR");  
 myTree.getBinTrees().getRightSon().getRightSon().setLeftSon(" level 3RRL");  
 myTree.getBinTrees().getRightSon().getRightSon().setRightSon(" level 3RRR");  
  
 Tree secondTree = new Tree("Sapin Président");  
 secondTree.getBinTrees().setLeftSon(" level 1L");  
 secondTree.getBinTrees().setRightSon(" level 1R");  
  
 secondTree.getBinTrees().getLeftSon().setLeftSon(" level 2LL");  
 secondTree.getBinTrees().getLeftSon().setRightSon(" level 2LR");  
 secondTree.getBinTrees().getRightSon().setLeftSon(" level 2RL");  
 secondTree.getBinTrees().getRightSon().setRightSon(" level 2RR");  
  
  
 // form 1  
 for (Iterator<BinTree> iterator = myTree.iterator();iterator.hasNext();) {  
 BinTree node=iterator.next();  
 System.out.println(node.getLabel());  
 if (node.getLabel().contains("level 3LLL")){  
 // test of a second iteration to same tree  
 // use form 2  
 for (BinTree autrenode : myTree)  
 {  
 System.out.println("second interation:" + autrenode.getLabel());  
 }  
 };  
  
 }  
 // test with a second tree  
 for (BinTree node:secondTree){  
 System.out.println(node.getLabel());  
 }  
  
 }  
}

## Exercice Civilization (modélisation UML diagramme de classe)

Modéliser les villes dans un jeu similaire à Civilization :   
Chaque ville possède:   
Un nom   
Un nombre d’habitants   
Des coordonnées (x,y)

Des bâtiments qui possèdent les propriétés: un nom de type unique (bibliothèque, baraquement, …)

Une file de bâtiments à construire

On souhaite disposer des méthodes suivantes:  
ajouter une ville   
retrouver une ville par ses coordonnées   
pour une ville, itérer sur l’ensemble de ses bâtiments.

### UML

