Diagramme d'états

1. Tamagotchi

On souhaite modéliser des tamagotchis ; on vous demande de réaliser le diagramme d'états de celui-ci. Souvenez-vous le tamagotchi … Petit animal virtuel japonais…

Le Tamagotchi en état normal n'a pas faim pendant un certain temps (appelé temps d'autonomie). Au bout de ce temps, le Tamagotchi a faim et il pleure. Pour lui donner à manger, l'utilisateur du Tamagotchi le met à table et le Tamagotchi s'arrête de pleurer. Un

Tamagotchi mange pendant un certain temps (appelé temps de restauration). Au bout de ce temps, il se remet à pleurer. Il pleure jusqu'à ce que l'utilisateur le sorte de table. Quand il sort de table, le Tamagotchi revient dans l'état normal... et ainsi de suite tant que le Tamagotchi ne meurt pas. Si le Tamagotchi pleure plus de 5 minutes d'affilée, il meurt.

On suppose que les actions réalisées par le Tamagotchi vers l'utilisateur sont : « avoir faim », « ne plus avoir faim », « mourir » et que les actions effectuées par l'utilisateur et reçues par le Tamagotchi sont « mettre à table » et « sortir de table ».

1. Dessiner un automate à 5 états modélisant le comportement du Tamagotchi. On utilisera les noms « pas faim pleure pas », « faim pleure », « à table pleure pas », « à table pleure » et « mort » pour ces 5 états. On fera figurer les événements nommés à la question précédente sur les transitions de l'automate. Une flèche vers le haut (/vers le bas) à côté d'un événement indiquera que l'événement est émis (/ reçu) par le Tamagotchi.

2. Quand l'utilisateur appuie sur le bouton « mettre à table » (resp. « sortir de table »), le Tamagotchi reçoit l'événement « être mis à table » (resp. « être sorti de table »). L'horloge se présente sous forme d'une instance de la classe `Horloge` avec une méthode `lancerTempo(int durée)`. Si le Tamagotchi appelle la méthode `lancerTempo(durée)` à un instant t, il recevra un événement « tempo écoulé » à l'instant t+durée. Ajouter les appels à la méthode `lancerTempo(int t)` et les événements « tempo écoulé » reçus par le Tamagotchi dans l'automate détaillé.

3. Le bip se présente sous forme d'une instance de la classe `Bip`. Si le Tamagotchi appelle la `déclencherPleurs()`, le bip commence à émettre des pleurs sans interruption. Si le Tamagotchi appelle la `arreterPleurs()`, le bip s'arrête. Les méthodes `lancerTempo(durée)`, `déclencherPleurs()` et `arreterPleurs()` s'exécutent instantanément. Au sens donné par UML, sont-elles des activités ou des actions ?

Rappel : une activité est quelque chose qui se déroule tant qu'un objet se trouve dans un certain état. (do) et une action est quelque chose déclenché par un évènement (entry, exit, on « event » …)

4. Préciser à nouveau l'automate en ajoutant les appels aux méthodes `déclencherPleurs()` et `arreterPleurs()` sous forme d'actions d'entrée ou de sortie.

2. Photocopieuse

Représentez le diagramme d'états du système décrit ci-dessous.

Un Système de Contrôle de machine photocopieuse.

On se propose de mettre au point un logiciel de contrôle destiné à être incorporé dans une machine photocopieuse pour donner le fonctionnement suivant :

La machine est allumée en appuyant sur le bouton ON/OFF, se met à chauffer pendant 30 secondes avant d'être prête à lire les commandes entrées par l'usager (nombre de copies, autres options d'impressions). La machine est complètement éteinte si le bouton ON/OFF est relâché. Lorsqu'on appuie sur le bouton START, la machine se met à produire des copies. Si le bac à papier est vide, une charge de papier est réclamée en affichant un message « charger papier ». Dès que le papier est mis, la production de copies continue son cours. Parfois, il peut arriver un blocage (papier coincé), le processus de production de copies s'arrête et le problème est signalé en affichant le message « problème à diagnostiquer ». Dès que le problème est réparé (manuellement) et que la machine ne détecte plus aucun blocage, elle reprend automatiquement son fonctionnement normal en s'apprêtant à recevoir de nouvelles commandes. À tout moment, le processus de production des copies peut être arrêté à l'aide du bouton STOP.

3. Système MonAuto

Voici un code de la classe `Réparation` provenant de la modélisation d'un système de gestion des réparations automobiles. Dans cette classe, un diagramme d'états a été implémenté.[1]

- Pouvez-vous fournir ce diagramme d'états de la réparation correspondant à cette implémentation. Afin de simplifier le diagramme, on vous demande de ne pas tenir compte des cas d'échec (pas de fin avec échec).
- Comment vous y prendriez-vous pour modéliser toutes les transitions menant à l'échec ? Répondez en français. NE MODIFIER PAS VOTRE DIAGRAMME !

```
package monAuto;
import java.util.GregorianCalendar;
import java.util.HashMap;
import java.util.Map;
import util.Util;

public class Réparation {
private static final int CREEE=0, EN_COURS=1, TERMINEE=2, EN_ORDRE=3, EXPORTEE=4, ARCHIVEE=5;
          private final Voiture voiture;
          private GregorianCalendar demande;
          private GregorianCalendar réception;
          private GregorianCalendar restitution;
          private Map<Pièce, Integer> pièces;
          private Map<Mécanicien, Integer> heuresPrestées;
          private int état;
```

[1] Dans le cadre du cours de Patterns en 3ème année, vous verrez comment implémenter proprement un diagramme d'états.

```
public Réparation(Voiture voiture) {
          Util.checkObjet(voiture);
         this.voiture = voiture;
         pièces = new HashMap<Pièce, Integer>();
         heuresPrestées = new HashMap<Mécanicien, Integer>();
         état = CREEE;
         demande = new GregorianCalendar();
}
public Voiture getVoiture() {
         return voiture;
}
public GregorianCalendar getDemande() {
         return demande;
}
public GregorianCalendar getRéception() {
         return réception;
}
public GregorianCalendar getRestitution() {
         return restitution;
}
public void ajouterPièce(Pièce pièce, int quantité) {
         Util.checkObjet(pièce);
         Util.checkPositive(quantité);
         if (état == CREEE) {
                   état = EN_COURS;
                   réception = new GregorianCalendar();
         }
         if (état != EN_COURS)
                   throw new IllegalStateException();
         if (pièces.containsKey(pièce))
                   pièces.put(pièce, quantité + pièces.get(pièce));
         else
                   pièces.put(pièce, quantité);
}
public void supprimerPièce(Pièce pièce, int quantité) {
         Util.checkObjet(pièce);
         Util.checkPositive(quantité);
         if (état != EN_COURS)
                   throw new IllegalStateException();
         if (!pièces.containsKey(pièce))
                   throw new IllegalArgumentException(
                             "pièce non enregsitrée pour cette réparation");

         if (pièces.get(pièce) > quantité)
                   pièces.put(pièce, pièces.get(pièce) - quantité);
         else if (pièces.get(pièce) == quantité) {
                   pièces.remove(pièce);
         } else
                   throw new IllegalArgumentException(
                             "impossible de supprimer une quantité plus " +
                        "grande que celle enregistrée pour la réparation");
}
public int prix(Pièce pièce){
         Integer nombre = pièces.get(pièce);
```

```
                if (nombre == null)
                        return 0;
                return nombre*pièce.getPrix();
        }
        public int prixTotalPièces(){
                int somme = 0;
                for (Pièce pièce : pièces.keySet()){
                        somme += prix(pièce);
                }
                return somme;
        }
       public void saisirHeuresTravail(Mécanicien mécanicien, int quantité) {
                Util.checkObjet(mécanicien);
                Util.checkPositive(quantité);
                if (état == CREEE) {
                        état = EN_COURS;
                        réception = new GregorianCalendar();
                }
                if (état != EN_COURS)
                        throw new IllegalStateException();
                if (heuresPrestées.containsKey(mécanicien))
                        heuresPrestées.put(mécanicien, quantité +
                                                heuresPrestées.get(mécanicien));
                else  heuresPrestées.put(mécanicien, quantité);
        }
        public void terminerRéparation() {
                if (état != EN_COURS)
                        throw new IllegalStateException();
                état = TERMINEE;
        }
        public void enregistrerEssai(boolean OK) {
                if (état != TERMINEE)
                        throw new IllegalStateException();
                if (OK) {
                        état = EN_ORDRE;
                        restitution = new GregorianCalendar();
                } else
                        état = EN_COURS;
        }
        public boolean résultatEssai() {
                return état >= EN_ORDRE;
        }
        public void exporter() {
                if (état != EN_ORDRE)
                        throw new IllegalStateException();
        // envoyer la réparation à la comptabilité pour facturation
                état = EXPORTEE;
        }
        public void archiver() {
                if (état != EXPORTEE)
                        throw new IllegalStateException();
                état = ARCHIVEE;
        }
}
```