A - Bases réutilisations :

1. Version 1 simplifié

a)

Une affectation polymorphique est l'affectation d'une valeur d'un type ST à une variable de type statique T lorsque ST est une sous classe de T.

héros = h;

Cette instruction entraîne une affectation polymorphique car dans la fonction main, nous avons injecté une valeur de type d'objet Pacman qui va affecter directement à une variable de type statique Héros qui est une sur-classe de Pacman et Pacman est une sous-classe de Héros.	Cette ligne entraîne une affectation polymorphique car nous avons injecté une valeur de type Pacman dans le paramètre de la méthode injecterHéros(Héros h) et dans cette méthode, cette valeur va être affecté à une variable de type statique Héros. Or, Pacman est une sous-classe de Héros et héros est une variable de type statique Héros.

obstacles.add(o);

Cette instruction entraîne directement une affectation polymorphique car dans la fonction main, nous avons affecté une valeur de type d'objet Fantôme qui va affecter dans une variable de type liste qui est censé contenir des objets de type statique Obstacle. Or, Fantôme est une sous-classe de la classe Obstacle. Donc une fois l'objet Fantôme ajouté, il devient aussi polymorphique.	Cette ligne entraine une affectation polymorphique car nous avons injecter une valeur de type Fantôme dans le paramètre de la méthode injecterObstacle(Obstacle o). Or, Fantôme est une sous-classe de Obstacle et obstacles est une liste de type statique Obstacle. On peut dire qu'une fois l'objet Fantôme ajouté dans la liste, il devient aussi polymorphique.

b)

Dans le contexte d'un framework, nous pouvons constater que le lien entre affectation polymorphique et injection de dépendance est la réutilisation des objets des autres classes. Cela permet de réaliser les extensions d'un framework plus facilement. Ces extensions sont les classes qui implémentent la classe qui contient la définition du framework. Les extensions vont spécialiser les méthodes obligatoires du framework. Le framework va appeler les méthodes qui sont le point d'extension du framework.	Dans le contexte d'un framework, il existe un lien entre affectation polymorphique et injection de dépendance. C'est la réutilisation des objets. Nous pouvons donc grâce à cela réaliser les extensions à travers les classes qui implémentent d'une même interface ou une classe abstraite qui contiennent la définition du framework. Les méthodes du framework seront spécialisé en fonction de l'objet affecté dans le framework.

Par exemple :

```
Heros heros;

public void injecterHeros(Heros h){
      //injection dépendance
      //affectation polymorphique
      heros = h;
}

h.rencontrer(f);//un point d'extension
```

2.

a)

Les variables receveurs sont this et o.
Le type statique de this est Héros.
Le type statique de o est Obstacle.

b)

rencontrer(Obstacle o);

paramétré par composition car l'affichage dépend du type d'objet référencé de o pour traiter le comportement de o.toString(). Cette méthode toString() va appeler la méthode toString() implémenté dans les sous-classes de Obstacle.	paramétré par composition car la méthode o.toString() est varié en fonction de type d'objet référencé o qui est connu au moment de l'exécution. Cette méthode est spécialisé dans les sous-classes de la classe Obstacle.

paramétré par spécialisation car la méthode utilise aussi l'identificateur (this) référençant au receveur courant (par exemple Pacman) pour traiter l'affichage de this.toString().	paramétré par spécialisation car la méthode utilise l'identificateur this, qui référencie au receveur courant, pour l'appelle à la méthode this.toString().

Cette méthode est donc paramétré par composition et paramétré par spécialisation

c)

Nous devons créer une nouvelle classe héros qui sera une sous-classe de Héros. Dans cette sous-classe, nous allons redéfinir la méthode toString(). Nous allons aussi ajouter la méthode rencontrer(ObstacleSauter) dans la classe Héros et redéfinir cette méthode dans cette nouvelle sous-classe. Cette méthode devrait faire sauter Pacman vers une autre case dès qu'elle est déclenché. Ensuite, nous devons créer la nouvelle classe ObstacleSauter qui sera une sous classe de Obstacle. Dans cette sous classe, nous allons redéfinir la méthode toString() aussi.	Nous allons ajouter une nouvelle sous-classe de Héros, par exemple PacmanBis. Ensuite, nous redéfinir la méthode toString() ainsi que les méthodes nécessaires. Nous allons implémenter la méthode rencontrer(PacmanBis) afin de pouvoir redéfinir dans la classe PacmanBis. Ensuite, nous définissons une nouvelle sous-classe de la classe Obstacle dans laquelle elle redéfini les méthodes nécessaires comme toString().

```
Public class PacmanSauter extends Heros{

      public String toString(){
          return "un PacmanSauter";
      }

      public void rencontrer(Sauteur s){
          //changer la case de pacman?
          System.out.println(this.toString() + “a rencontré un” +
s.toString();
      }
}
```

```java
public class Sauteur extends Obstacle{

	public String toString(){
	    return "Sauteur";
	}
}
```

B - Framework version 2 - Spécialisation de méthodes

1.

Nous privilégions l'affectation polymorphique car nous allons cette méthode en commun dans tous les sous-classes de Héros. L'utilisation de instanceOf n'est pas une meilleur choix car nous n'avons pas de la sous-classe qui n'implémente pas cette méthode rencontrer().	Nous avons choisi de définir plusieurs méthodes rencontrer() car dans notre contexte, nous allons implémenter plusieurs type de héros. Tous les sous-classes de Héros auront toujours la méthode rencontrer() ce qui privilège le choix d'utiliser l'affectation polymorphique car l'utilisation de Instanceof sera utile seulement si on défini cette méthode pour une seule sous-classe.

2.

a)

La méthode invoquée par l'appel h.rencontrer(f) est celle de la classe Pacman car nous avons une fonction h.rencontrer(f) dans la classe Héros. Cela veut dire que celle de Pacman est une redéfinition et dans la ligne 1, nous avons une affectation polymorphique avec la valeur de type dynamique est Pacman, qui est une sous-classe de Héros, et la variable h de type statique Héros. De plus, La méthode rencontrer(Fantome f) est défini dans la classe Héros et redéfini dans la classe Pacman. Il s'agit alors une redéfinition. De plus, en java, seulement le simple dispatch est géré ce qui veut dire que à l'exécution la méthode la plus	L'envois de message h.rencontrer(f) du listing 4 invoque la méthode défini dans la classe Pacman. Nous pouvons constater dans la ligne 1 du listing 4. Il s'agit une affectation polymorphique avec la variable du type statique Héros et la valeur référencé est du type dynamique Pacman ainsi que Pacman est une sous-classe de Héros. Nous constatons que la méthode rencontrer(Fantome f) est définie dans la classe Héros et est redéfinie dans la classe Pacman. De plus, en java, la résolution dynamique des liens n'est réalisé que pour les méthodes redéfinies ce qui veut dire que à l'exécution la méthode la plus spécialisée sera invoquée par l'appel de méthode. C'est le cas de cette envois de message.

spécialisée sera choisi à invoquer. On peut donc dire que l'envois de message h.rencontrer(f) invoquée la méthode de la classe Pacman.	

b)

Dans le cas où on n'ait pas défini la méthode rencontrer(Fantôme) sur la classe Héros, c'est la méthode rencontrer(Obstacle o) de la classe Héros qui sera appelée car la méthode rencontrer(Fantome f) de la classe Pacman n'est plus une redéfinition mais c'est plutôt une surcharge de rencontrer(Obstacle o). Or, Java ne gère pas le multiple dispatch et comme la variable h est de type statique Héros. C'est donc la méthode rencontrer(Obstacle) de la classe Héros est invoquée par l'envois de message h.rencontrer(f);	Si la méthode rencontrer(Fantôme) n'a pas été défini dans la classe Héros. La méthode rencontrer(Fantome) de Pacman s'agit alors une surcharge. Or, dans Java, le multiple dispatch n'est pas géré. Nous pouvons donc dire que la méthode invoquée par l'appel de méthode h.rencontrer(f) est la méthode rencontrer(Obstacle) de la classe Héros car nous avons la variable h de type statique est Héros.

C - Test de logiciel

1)

Utilisation des asserts pour tester si la fonction move() change la valeur de x et de y du Fantôme. On initialise ElementJeu e1. On déclenche la fonction move(). On utilise la méthode AssertEqual sur la distance de Fantôme et de e1 pour voir si c'est égal.
exemple :
`@Test`
`public void testMove(){`
`  Element e1 = Fantome.getElementJeu();`
`  Fantome.move();`
`  AssertNotEquals(distanceBetween(e1, Fantome.getElementJeu()), 0);`
`}`

Nous allons tester la fonction move() pour voir s'il ya une changement de la valeur X et Y

du Fantome.

- Initialise la position du Fantome
- Envoi de message move()
- Tester si la position du Fantome est toujours la même.

2)

Utilisation AssertTrue pour tester si l'état de Héros est invincible et la distance entre Fantome et Pacman est plus grand qu'avant.

```
@Test
public void testMove_Invincible(){
	Element e1 = Fantome.getElementJeu();
	Element e2 = Pacman.getElementJeu();
	AssertTrue(Pacman.estInvincible() == true);
	Fantome.move();
	//ici on test si la distance actuel de fantome est plus grand par
rapport à Pacman est plus grand que la distance précédente de Fantome
par rapport à Pacman.
	AssertTrue( distanceBetween(e2, Fantome.getElementJeu()) >=
distanceBetween(e2, e1) );
}
```

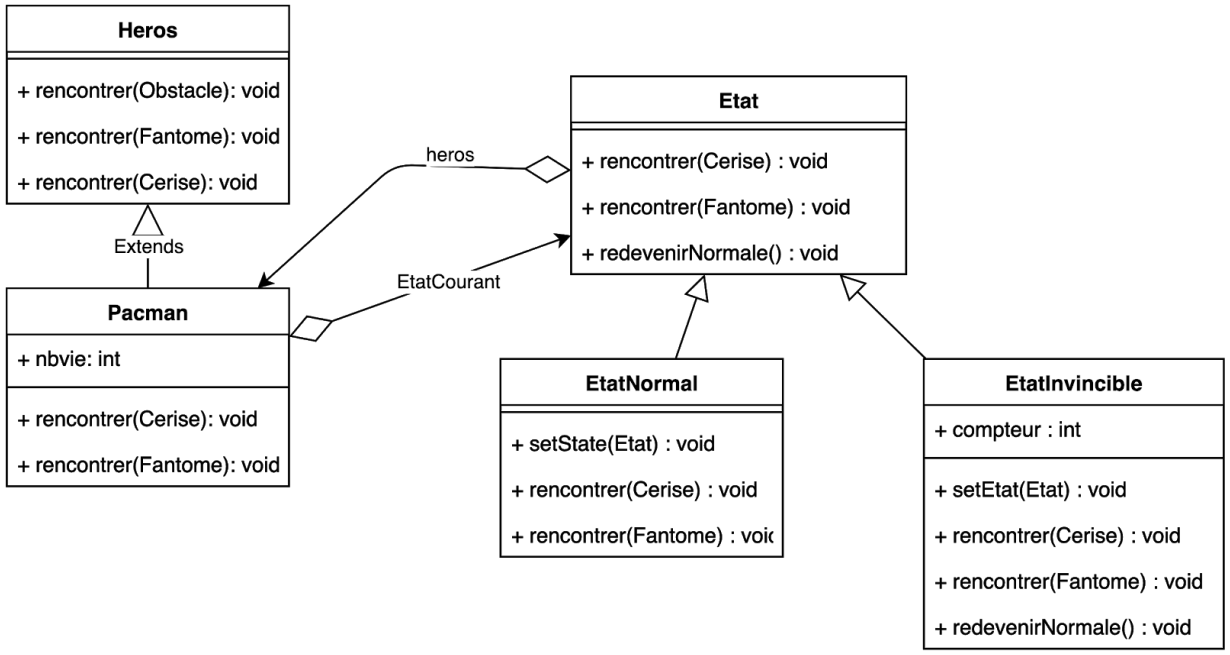
Nous allons tester quand l'état de Héros est invincible, la position entre Fantome et Pacman est plus grande ou plus petite après l'appel de move();

- Initialise la position de Fantome et Pacman
- Test si Pacman est invincible
- Envois de message move() au Fantome
- Tester si la position de Pacman et Fantome est plus grande(loin) ou plus petite(proche)

3) La méthode getX() sera appelé via la méthode distanceBetween(ElementJeu e1, ElementJeu e2){...} afin de comparer la distance entre les élements du jeu e1 et e2.

D - Schémas de conception - Gérer les changements de comportement du Framework

1)



2)

```
public class Pacman extends Héros {
	private Etat etatCourant;
	public Pacman(){
		this.etatCourant = new etatNormale(this);
	}

	public void rencontrer(Fantôme f){
		this.etatCourant.rencontrer(f);
	}

	public void rencontrer(Cerise c){
		this.etatCourant.rencontrer(c);
	}
}

public abstract class Etat{
	Pacman p;

	public Etat (Pacman p){
		this.p = p;
	}

	public abstract void rencontrer(Fantôme f);
	public abstract void rencontrer(Cerise c);
	public abstract void redevenirNormale();
}

Public class EtatNormale extends Etat{
	public EtatNormale(Pacman p){
		super(p);
	}
	public void rencontrer(Fantome f){
		this.p.nbvie--;
	}

	public void rencontrer(Cerise c){
		this.p.etatCourant = new EtatInvincible(p);
	}

	public void redevenirNormale(){
		//throws erreur ?
	}
}
```

```
Public class EtatInvincible extends Etat{
	//declaration compteur du temps

	public EtatInvincible(Pacman p){
		super(p);
	}

	public void rencontrer(Fantome f){
		f.envoyerPrison();
	}

	public void rencontrer(Cerise c){
		//prolonge temps invincible
	}

	public void redevenirNormal(){
		this.p.etatCourant = new EtatNormale();
	}
}
```

E - Framework version 3 : une version réutilisable réaliste du moteur de jeu

1)

Car java ne gère pas le multiple dispatch, du coup quand r est passé en paramètre à h.rencontrer(r), seul le type statique de o Obstacle est reconnu, et donc rencontrer(Obstacle) de la classe Héros est appelé. De plus celle de Pacman n'est pas une redéfinition mais une surcharge.	En java, le multiple dispatch n'est pas géré. C'est la raison pour laquelle l'appel de méthode h.rencontrer(r); invoquée la méthode rencontrer(...) de la classe Héros car la méthode rencontrer() de Pacman n'est qu'une surcharge ce qui n'est pas géré par la résolution dynamique des liens en Java.

2)

Nous allons définir la méthode rencontrer() dans la classe Obstacle et ses sous-classes afin d'appliquer le double-dispatch (= le patron de conception	Nous allons implémenter la méthode rencontrer(...) dans la classe Obstacle et ses sous-classes en utilisant le mécanisme double-dispatch, autrement dit Visiteur.

utilisant le patron Visiteur (double-dispatch) pour implémenter partiellement multiple dispatch) qui consiste d'appeler les méthodes du type de l'objet dynamique.	

```
Public class Obstacle {
	//this = type d'obstacle

	public void rencontrer(Héros h){h.rencontrer(this);}
}
public class Fantome extends Obstacle{
	//...
	public void rencontrer(Héros h){h.rencontrer(this);}
}
```

Nous allons appeler r.rencontrer(h); d'où r est le fantôme ou le cerise.