Die objektorientierte Mühle – Marc-Oliver Pahl

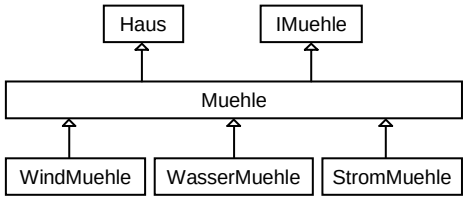
Es war einmal ein König, der machte Urlaub außerhalb seines Königreiches. Dort sah er eine tolle Einrichtung: Ein Haus, zu dem die Bauern ihr Korn fuhren und von dem die Bäcker Mehl abholten - eine Mühle.

Da du der Chefentwickler des Hofes bist, wendet sich der König –frisch erholt aus dem Urlaub zurück- an dich: „Ich will auch Mühlen in meinem Königreich und die sollen Folgendes können: Man muss an eine Mühle Korn *anliefern* können, diese muss das Korn dann *mahlen* und schließlich soll man Mehlsäcke *abholen* können."

Da du gerade in Info 1 etwas über Interfaces gehört hast, fasst du das Ganze gleich mal in ein solches ab:

```
public interface IMuehle {
      public Id anliefern(Korn korn);
   public void mahlen() throws MahlException;
   public MehlSack abholen();
}
```

Außerdem schickst du Kundschafter los, die herausfinden sollen, wie denn so eine Mühle mahlt. Nach einer Weile kehren die Kundschafter zurück und erzählen dir, was sie gesehen haben. Es fällt dir auf, dass bei allen Mühlen das Korn zuerst eingelagert wird. Anschließend wird es gemahlen und schließlich in Säcke verpackt wieder in ein Lager geschafft, von wo es die Bäcker dann abholen. Alle Mühlen haben also eine gemeinsame Grundfunktionalität. Weiterhin gibt es drei Formen von Mühlen: Solche, die mit Wind funktionieren (WindMuehle), solche, die mit Wasser funktionieren (WasserMuehle) und solche, die mit Strom funktionieren (StromMuehle). Du erinnerst dich daran, dass das Konzept des objektorientierten Programmierens dir hier hilfreich sein könnte, denn da kannst du ja gerade Klassen erstellen, die Funktionalität bieten und von diesen ableiten.
Du machst dich also sofort an die Arbeit und konzipierst ein UML-Diagramm:



Die Mühle hat zunächst einmal also die Funktionalität eines Hauses und dann implementiert sie natürlich das Interface IMuehle, wie der König es uns aufgetragen hat. Die einzelnen Arten von Mühlen (WindMuehle, WasserMuehle, StromMuehle) erweitern jeweils unseren Grundtyp der Mühle (Muehle).
Was ein Haus kann haben wir schon vor einiger Zeit implementiert, denn Häuser hat es schon viele in unserem Königreich. Darum müssen wir uns also nicht mehr kümmern, wir müssen nur noch die Zusatzfunktionalität von IMuehle einbauen. Machen wir uns also an die Arbeit. Unsere Klasse Muehle wird so beginnen:

```
public abstract class Muehle extends Haus implements IMuehle
```

Sie wird also IMuehle implementieren und Haus erweitern. Wie die mahlen-Methode des Interface realisiert wird, hängt von der Art der Mühle ab. Daher werden wir die Methode in unserer Muehle Klasse nicht implementieren. Damit wir dem Interface genügen, muss sie aber vorhanden sein. Wir werden sie also als abstrakte Methode einfügen:

```
public abstract void mahlen()
```

Daher muss die gesamte Klasse abstract sein.

In die Muehle Klasse wollen wir Funktionalität zum Einlagern einbauen. Wir erinnern uns daran, dass wir in der Vorlesung etwas von Queues gehört haben und diese hier gut verwenden könnten... Wir legen also zwei Queues an, eine für das Kornlager und eine für das Mehlsacklager:

```
protected LagerQueue kornlager;
protected LagerQueue mehllager;
```

Auf die Queues wollen wir von unseren speziellen Mühlentypen natürlich auch zugreifen können, deshalb deklarieren wir sie als protected. Die LagerQueue hat drei Methode push(Object obj), um Dinge in die Queue zu stellen, Object pop(), um das nächste Element (FiFo) von der Queue zu nehmen und Object top(), um das nächste Element auf der Queue zu betrachten, ohne es von der Queue zu nehmen. (Der genaue Code findet sich im Anhang).

Machen wir uns daran, das Interface zu implementiern. Zuerst das Anliefern. Beim Anliefern soll im wesentlichen das Korn eingelagert werden. Dazu geben wir ihm eine eindeutige Id (wir erinnern uns...) und stellen diese in unsere Kornqueue:

```
public Id anliefern(Korn korn){
      Id theId = new Id();
      kornlager.push(theId);
      return theId;
}
```

Dem Anlieferer geben wir die Id zurück, denn diese wird später auch auf unsere Mehlsäcke gedruckt, so dass die Produktionskette zurückverfolgt werden kann...
Die mahlen() Methode wollen wir erst in der nächsten Hierarchiestufe realisieren, bleibt also noch die abholen() Methode. Dabei soll ein Mehlsack mit der entsprechenden Id ausgegeben werden:

```
public MehlSack abholen(){
      return (MehlSack) mehllager.pop();
}
```

Damit wir unsere Mühlen auseinander halten können, wollen wir ihnen Namen geben. Diese Funktionalität haben wir glücklicherweise schon in unserer Hausklasse (setName), so dass wir sie im Konstruktor einfach aufrufen können:

```
public Muehle(String derName){
      kornlager = new LagerQueue();
      mehllager = new LagerQueue();
      setName(derName);
}
```

Damit ist schon sehr viel getan. In mahlen() muss jetzt nur noch irgendwie aus dem Korn das Mehl werden. Dazu müssen wir Korn aus dem Kornlager holen...

```
Id inBearbeitungId = (Id) kornlager.pop();
```

...es mahlen –wie genau das geht interessiert uns hier nicht...

```
// mahlen
```

...und das Ergebnis schließlich in einem neuen Sack, den wir mit der Id versehen, in das Mehllager stecken:

```
mehllager.push(new MehlSack(inBearbeitungId));
```

Und damit haben wir schon alles implementiert. Da es beim mahlen auch zu Problemen kommen kann haben wir schon im Interface eine Exception vorgesehen, von der wir für unsere verschiedenen Mühlentypen entsprechende Exceptions ableiten. Für die Windmühle beispielsweise:

```
public class KeinWindException extends MahlException {}
```

Die komplette mahlen() Methoden sieht dann am Beispiel der Windmühle so aus:

```
public void mahlen() throws KeinWindException{
      while(kornlager.top() != null){
            if (!wind) throw new KeinWindException();
            Id inBearbeitungId = (Id) kornlager.pop();
            // mahlen mit Wasser
            mehllager.push(new MehlSack(inBearbeitungId));
      }
}
```

Die äußere Schleife lässt die Mühle so lange arbeiten, bis kein Korn mehr im Lager ist. Sobald kein Wind mehr vorhanden ist, wird die Exception geworfen und die Verarbeitung stoppt...

Entsprechend sind die Implementierungen der anderen Mühlentypen.

Das tolle daran, dass wir den objektorientierten Ansatz gewählt haben ist, dass wir jetzt ganz viele Mühlen in unser Königreich bauen können, indem wir einfach immer neue Instanzen erzeugen:

```
IMuehle[] meineMuehlen = new IMuehle[4];
meineMuehlen[0] = new WindMuehle("WindMühle A");
meineMuehlen[1] = new WindMuehle("WindMühle B");
meineMuehlen[2] = new StromMuehle("StromMühle");
meineMuehlen[3] = new WasserMuehle("WasserMühle");
```

Alle Mühlen, egal von welchem speziellen Typ sie sind, haben von Muehle geerbt. Das heißt wir können in jedem Fall alle Methoden von Muehle verwenden. Insbesondere heißt das, dass wir alle Mühlen gleich ansprechen können. Vor allem für den Bauern und den Bäcker ist dies sehr positiv, denn wenn sie umziehen müssen sie sich nicht neu lernen, wie die Mühle funktioniert ;)

Wir können also mal den Betrieb simulieren:

Anliefern...

```
for(int i=0; i<10; i++){
      // 10x wird Korn an zufällig gewählte Mühlen geliefert.
      meineMuehlen[(int) (4*Math.random())].anliefern(new Korn());
}
```

Mahlen...

```
// alle Mühlen mahlen lassen
try{
      for(int i=0; i<4; i++) meineMuehlen[i].mahlen();
}catch(MahlException e){
      // irgendetwas sinnvolles tun
};
```

Hierbei müssen wir natürlich unsere Exceptions abfangen. Da alle vom Obertyp MahlException sind können wir diesen für alle Mühlentypen abfangen...

Abholen...

```
// Lager von allen Mühlen leeren
MehlSack sack;
for(int i=0; i<4; i++)
      while ((sack = meineMuehlen[i].abholen()) != null);
```

Der Wert von `(sack = meineMuehlen[i].abholen())` ist hierbei der von sack. Die While-Schleife läuft so lange, wie abholen uns noch Objekte gibt (solange die Mahllager-Queue noch Objekte enthält).

Nachdem wir wieder einmal ganze Arbeit geleistet haben, ist der König so begeistert, dass er uns erstmal in den wohlverdienten Urlaub schickt...

Die kompletten Sourcen:

IMuehle.java

```
/**
 * Eine Mühle muss grundsätzlich erstmal Korn annehmen
 * können und irgendwann muss man sein gemahlenes Korn,
 * also das Mehl, abholen können.
 */
public interface IMuehle {
	/**
	 * Methode die der Bauer aufruft, der Korn anliefert.
	 * @param korn
	 * @return Id (eindeutige AbholId)
	 */
	public Id anliefern(Korn korn);

	/**
	 * Wandelt Korn in Mehl.
	 * @throws MahlException
	 */
	public void mahlen() throws MahlException;

	/**
	 * @return Korn (das gemalene Korn)
	 */
	public MehlSack abholen();
}
```

Haus.java

```
public class Haus {
	private String name;

	public Haus(){
		name = "kein Name";
	}

	public Haus(String derName){
		name = derName;
	}

	public void setName(String derName){
		name = derName;
	}

	public String getName(){
		return name;
	}
}
```

Muehle.java

```java
/**
 * Muehle stellt grundlegende Funktionalität einer
 * Mühle zur Verfügung:
 * - Die Lagerhaltung
 *
 * mahlen() bleibt weiterhin abstract
 */
public abstract class Muehle extends Haus implements IMuehle {
	protected LagerQueue kornlager;
	protected LagerQueue mehllager;

	public Muehle(){
		kornlager = new LagerQueue();
		mehllager = new LagerQueue();
	}

	public Muehle(String derName){
		this();
		setName(derName);
	}

	public Id anliefern(Korn korn){
		Id theId = new Id();
		kornlager.push(theId);
		System.out.println(getName()+": Korn angeliefert. Folgende Id vergeben: "+theId.getId());
		return theId;
	}

	public abstract void mahlen() throws MahlException;

	public MehlSack abholen(){
		MehlSack res = (MehlSack) mehllager.pop();
		if (res != null) System.out.println(getName()+": Mehlsack mit der Id "+res.getId().getId()+"
abgeholt.");
		else System.out.println(getName()+": Lager leer");
		return res;
	}
}
```

WindMuehle.java

```java
public class WindMuehle extends Muehle {
	private boolean wind = true;

	public WindMuehle(){
		super();
	}

	public WindMuehle(String name){
		super(name);
	}

	public void mahlen() throws KeinWindException{
		while(kornlager.top() != null){
			if (!wind) throw new KeinWindException();
			Id inBearbeitungId = (Id) kornlager.pop();
			// mahlen mit Wasser
			mehllager.push(new MehlSack(inBearbeitungId));
		}
	}
}
```

WasserMuehle.java

```java
public class WasserMuehle extends Muehle {
	private boolean wasser = true;

	public WasserMuehle(){
		super();
	}

	public WasserMuehle(String name){
		super(name);
	}

	public void mahlen() throws KeinWasserException{
		while(kornlager.top() != null){
			if (!wasser) throw new KeinWasserException();
			Id inBearbeitungId = (Id) kornlager.pop();
			// mahlen mit Wind
			mehllager.push(new MehlSack(inBearbeitungId));
		}
	}
}
```

StromMuehle.java

```java
public class WindMuehle extends Muehle {
	private boolean wind = true;

	public WindMuehle(){
		super();
	}

	public WindMuehle(String name){
		super(name);
	}

	public void mahlen() throws KeinWindException{
		while(kornlager.top() != null){
			if (!wind) throw new KeinWindException();
			Id inBearbeitungId = (Id) kornlager.pop();
			// mahlen mit Wasser
			mehllager.push(new MehlSack(inBearbeitungId));
		}
	}
}
```

Hilfsklassen

Korn.java

```java
/* Diese Klasse repräsentiert Korn.
 *
 * Für das Beispiel wird keinerlei Funktionalität
 * benötigt. Deshalb kann die Klasse auc nichts.
 *
 */
public class Korn {

}
```

MehlSack.java

```java
/* Diese Klasse repräsentiert Mehl.
 *
 * Für das Beispiel wird nur die Funktionalität einer
 * Id benötigt, um zu identifizieren, von wo das Korn kommt.
 *
 */
public class MehlSack {
	private Id id;
	public MehlSack(Id meineId){
		id = meineId;
	}
	public Id getId(){
		return id;
	}
}
```

Id.java

```java
/* Diese Klasse beinhaltet eine Id, wie wir sie schon mehrfach
 * gesehen haben.
 */
public class Id {
	private static int newUniqueId=0;

	private int myId;

	/**
	 * Erstellt ein neues Objekt vom Typ Id mit
	 * einer eindeutigen Id.
	 */
	public Id(){
		myId = newUniqueId++;
	}

	/**
	 * Gibt die Id zurück.
	 * @return die Id
	 */
	public int getId(){
		return myId;
	}
}
```

IObjectQueue.java

```java
/**
 * Interface für eine Queue (FiFo).
 */
public interface IObjectQueue {
	/**
	 * Fügt ein Objekt in die Queue ein.
	 * @param obj
	 */
	public void push(Object obj);
	/**
	 * Gibt das als erstes hinzugefügte Objekt hinzu und
	 * nimmt es von der Queue.
	 * @return
	 */
	public Object pop();
	/**
	 * Gibt das als erstes hinzugefügte Objekt hinzu.
	 * @return
	 */
	public Object top();
}
```

LagerQueue.java

```java
/**
 * Implementiert eine Queue auf Basis einer doppelt
 * verketteten Liste.
 */
public class LagerQueue implements IObjectQueue {
	protected class Node{
		protected Object nodeData;
		protected Node nextNode;

		Node(Node next){
			nextNode = next;
		}

		Node(Node next, Object data){
			nextNode = next;
			nodeData = data;
		}
```

```
	void setNextNode(Node next){
		nextNode = next;
	}

	void setData(Object data){
		nodeData = data;
	}

	Node getNextNode(){
		return nextNode;
	}

	Object getData(){
		return nodeData;
	}
}


protected Node head;

/* (non-Javadoc)
 * @see IObjectQueue#push(java.lang.Object)
 */
public void push(Object obj){
	if (head == null) head = new Node(null, obj);
	else {
		Node actNode = head;
		while(actNode.getNextNode() != null){
			actNode = actNode.getNextNode();
		}
		// Jetzt ist actNode der letzte Knoten
		actNode.setNextNode(new Node(null, obj));
		// Jetzt ist der neue Knoten hinten angehängt
	}
}
```

```
/* (non-Javadoc)
        * @see IObjectQueue#pop()
        */
       public Object pop(){
             // kein Knoten da
             if (head == null) return null;

             Node res = head;
             head = res.getNextNode();
             // ehemaliger Frontknoten ausgekettet
             return res.getData();
             // Der ehemalig head wird gabagecollectet
       }

       /* (non-Javadoc)
        * @see IObjectQueue#top()
        */
       public Object top(){
             if (head == null) return null;
             else return head.getData();
       }
}
```

Die Exceptions

```
public class MahlException extends Exception {}

public class KeinWindException extends MahlException {}

public class KeinWasserException extends MahlException {}

public class KeinStromException extends MahlException {}
```

Die MainKlasse

```java
/**
 * Testet unsere Mühlen.
 */
public class Main {

	public static void main(String[] args) {
		IMuehle[] meineMuehlen = new IMuehle[4];
		meineMuehlen[0] = new WindMuehle("WindMühle A");
		meineMuehlen[1] = new WindMuehle("WindMühle B");
		meineMuehlen[2] = new StromMuehle("StromMühle");
		meineMuehlen[3] = new WasserMuehle("WasserMühle");

		for(int i=0; i<10; i++){
			// 10x wird Korn an zufällig gewählte Mühlen geliefert.
			meineMuehlen[(int) (4*Math.random())].anliefern(new Korn());
		}

		// alle Mühlen mahlen lassen
		try{
			for(int i=0; i<4; i++) meineMuehlen[i].mahlen();
		}catch(MahlException e){
			// irgendetwas sinnvolles tun
		};

		// Lager von allen Mühlen leeren
		MehlSack sack;
		for(int i=0; i<4; i++)
			while ((sack = meineMuehlen[i].abholen()) != null);
	}
}
```