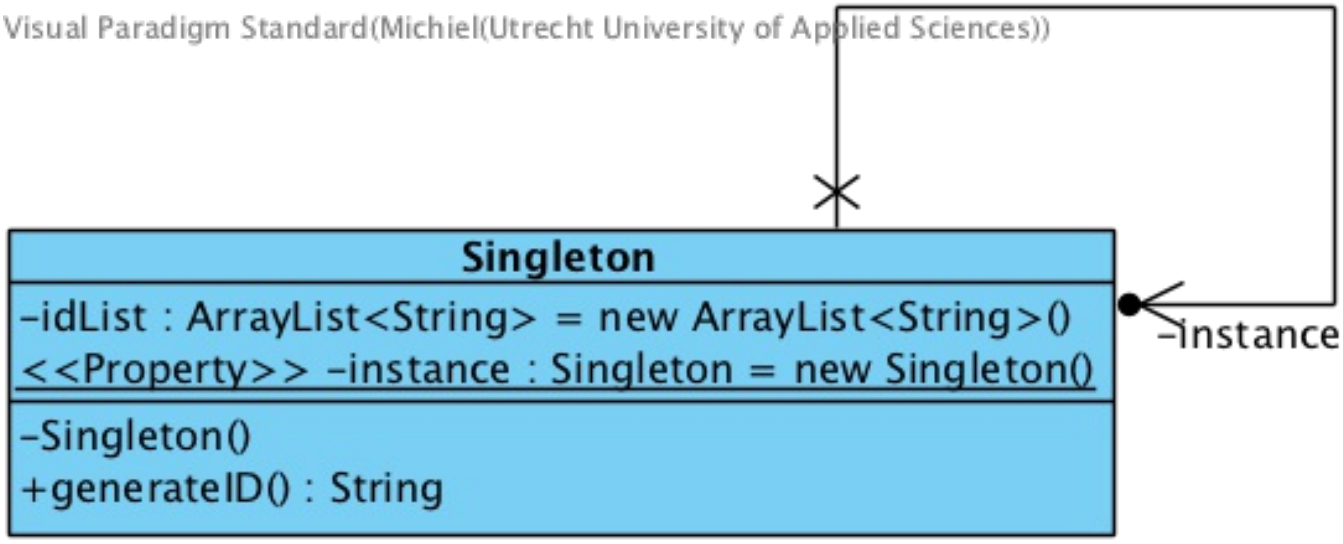
Design patterns

Creational patterns

Singleton

Het singleton pattern zorgt ervoor dat er niet meer dan één object van een klasse word aangemaakt. Deze instance is dan beschikbaar door de volledige code.

Een toepassing van de singleton is bijvoorbeeld het maken van unieke identificatienummers binnen een programma. Om er altijd zeker van te zijn dat elk identificatienummer uniek is, is het handig om dit door één enkel object te laten genereren. Dit is dan een singleton. (wikipedia)



Singleton.java

```
public class Singleton {

	private ArrayList<String> idList = new ArrayList<String>();

	private static Singleton instance = new Singleton();

	//Singleton-constructor moet private zijn zodat je niet meerdere instanties
aan kan maken
	private Singleton() {};

	public static Singleton getInstance() {
		return instance;
	}

	public String generateID() {
		String uniqueID = UUID.randomUUID().toString();

		if (idList.contains(uniqueID)) {
			System.out.println("id bestaat al.");
			return null;
		} else {
			return uniqueID;
		}
```

```
        }

}
```

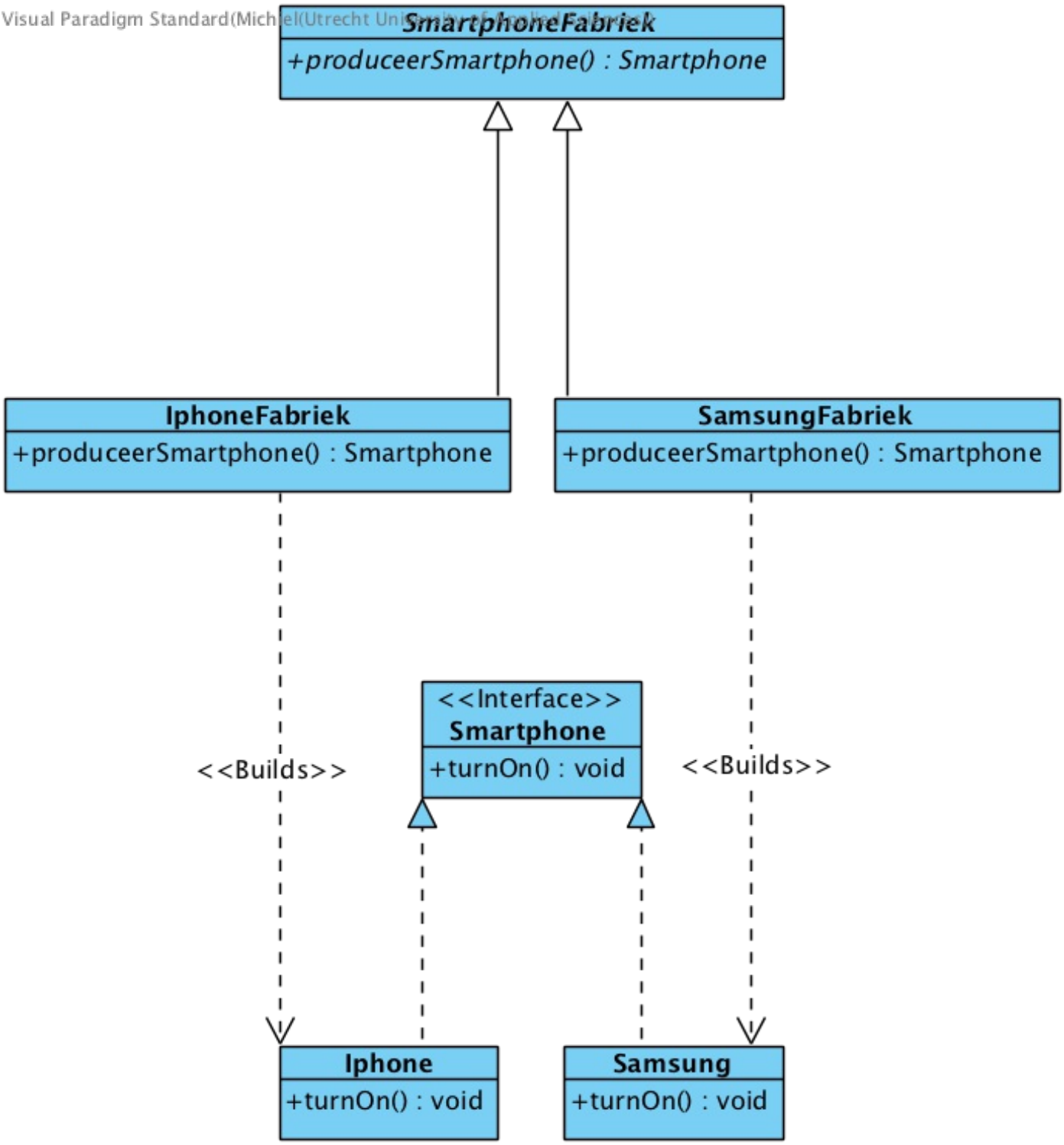
Main.java

```
public class Main {

        public static void main(String[] args) {

                //een instantie van singleton, geen nieuw object.
                Singleton s = Singleton.getInstance();

                System.out.println("ID1: "+s.generateID());
                System.out.println("ID2: "+s.generateID());
                System.out.println("ID3: "+s.generateID());

        }
}
```

Factory

Het factory pattern is een manier om objecten te instantiëren zonder exact vast te hoeven van welke klasse deze objecten zullen zijn. Er wordt een factory gemaakt die door subklassen geïmplementeerd kan worden. De klasse van het object dat door die methode geïnstantieerd wordt, implementeert een bepaalde interface. Elk van de subklassen kan vervolgens bepalen van welke klasse een object wordt aangemaakt, zolang deze klasse maar die interface implementeert.

Het doel van dit ontwerppatroon is het vereenvoudigen van het onderhoud van het programma. Als er nieuwe subklassen nodig zijn dan hoeft men alleen een nieuwe factory-methode te implementeren. (wikipedia)



Smartphone.java

```
public interface Smartphone {
        public void turnOn();
}
```

Iphone.java

```
public class Iphone implements Smartphone{

        @Override
        public void turnOn() {
```

```
                System.out.println("De iPhone werkt!");

        }
}
```

Samsung.java

```
public class Samsung implements Smartphone {

        @Override
        public void turnOn() {
                System.out.println("De Samsung werkt!");

        }
}
```

SmartphoneFabriek.java

```
public abstract class SmartphoneFabriek {

        public abstract Smartphone produceerSmartphone();

}
```

IphoneFabriek.java

```
public class IphoneFabriek extends SmartphoneFabriek{

        @Override
        public Smartphone produceerSmartphone() {
                return new Iphone();
        }

}
```

SamsungFabriek.java

```
public class SamsungFabriek extends SmartphoneFabriek {

        @Override
        public Smartphone produceerSmartphone() {
                return new Samsung();
        }

}
```

Main.java

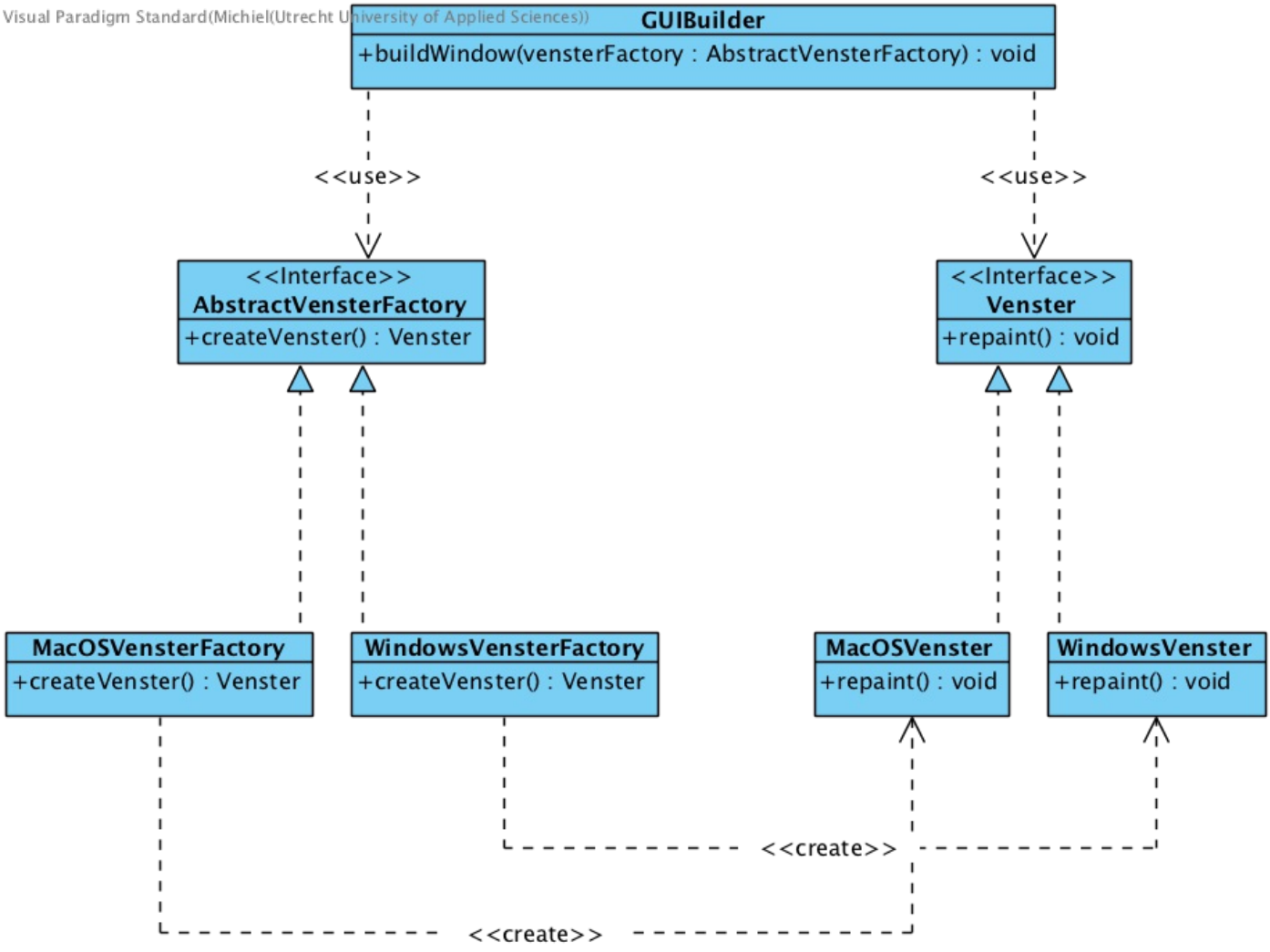
```
public class Main {

        public static void main(String[] args) {

                SmartphoneFabriek iphoneFabriek = new IphoneFabriek();
                Smartphone iphoneX = iphoneFabriek.produceerSmartphone();
                iphoneX.turnOn();
```

```
                SmartphoneFabriek samsungFabriek = new SamsungFabriek();
                Smartphone s9 = samsungFabriek.produceerSmartphone();
                s9.turnOn();

        }

}
```

Abstract Factory

Het Abstract Factory pattern werkt met een soort super-factory die dan weer andere factories creeërt. Een vaak gebruikt voorbeeld is het maken van specifieke venster voor Mac OS en Windows. De 'super-factory' is hier de AbstractVensterFactory. Deze creeërt weer de factories die het Windows of Mac venster kunnen gaan produceren.

Je kan dus tijdens runtime kijken welk besturingssysteem er draait en daarop je interface aanpassen.



AbstractVensterFactory.java

```
public interface AbstractVensterFactory {

        public Venster createVenster();

}
```

MacOSVensterFactory.java

```
public class MacOSVensterFactory implements AbstractVensterFactory{

	@Override
	public Venster createVenster() {
		MacOSVenster venster = new MacOSVenster();
		return venster;
	}

}
```

WindowsVensterFactory.java

```
public class WindowsVensterFactory implements AbstractVensterFactory{

	@Override
	public Venster createVenster() {
		WindowsVenster venster = new WindowsVenster();

		return venster;
	}

}
```

Venster.java

```
public interface Venster {

	public void repaint();

}
```

MacOSVenster.java

```
public class MacOSVenster implements Venster{

	@Override
	public void repaint() {
		// Knoppen aanpassen zodat het meer in de Mac OS omgeving past...
		System.out.println("Nieuw MAC OS -venster aangemaakt.");

	}

}
```

WindowsVenster.java

```
public class WindowsVenster implements Venster{

	@Override
	public void repaint() {
		// Knoppen aanpassen zodat het meer in de Windows omgeving past...
		System.out.println("Nieuw Windows-venster aangemaakt.");

	}

}
```

GUIBuilder.java

```
public class GUIBuilder {

        public void buildWindow(AbstractVensterFactory vensterFactory) {
                Venster venster = vensterFactory.createVenster();
                venster.repaint();
        }
}
```

Main.java

```
public class Main {

        public static String platform = "MACOS";

        public static void main(String[] args) {

                GUIBuilder builder = new GUIBuilder();
                AbstractVensterFactory vensterFactory = null;

                if(platform == "MACOS"){
                        vensterFactory  = new MacOSVensterFactory();
                } else {
                        vensterFactory  = new WindowsVensterFactory();
                }
                builder.buildWindow(vensterFactory);

        }

}
```

Abstract Factory

Het Abstract Factory pattern werkt met een soort super-factory die dan weer andere factories creeërt. Een vaak gebruikt voorbeeld is het maken van specifieke venster voor Mac OS en Windows. De 'super-factory' is hier de AbstractVensterFactory. Deze creeërt weer de factories die het Windows of Mac venster kunnen gaan produceren.

Je kan dus tijdens runtime kijken welk besturingssysteem er draait en daarop je interface aanpassen.