Task 3- Design Patterns in Practice 2

Schwerpunkt des Projekts

Wie in den Anforderungen beschrieben wird, haben wir den Hauptfokus auf die Verwendung von Design Pattern gelegt.

Bis auf Strategy Pattern wurden alle Pattern in der Implementierung sinngemäß eingesetzt.

Im Projekt wurden folgende Pattern eingesetzt, die dann in AR5 mit Code-Beispielen und UML-Klassendiagramm ausführlicher beschrieben werden.

Iterator Pattern
Composite Pattern
Proxy Pattern
Observer Pattern
Abstract Factory Pattern
Decorator
Factory Method

Projektübersicht

Dia App soll als eine Back-Office-App für die Erstellung von Klienten, Invoices und alle gebundenen Use-Case verwendet werden. Wir können sowohl Daten via Web-Service (FastBill API) oder unsere Testdaten verwenden um die Funktionalität der Applikation zu veranschaulichen.

Additional Requirements (AR)

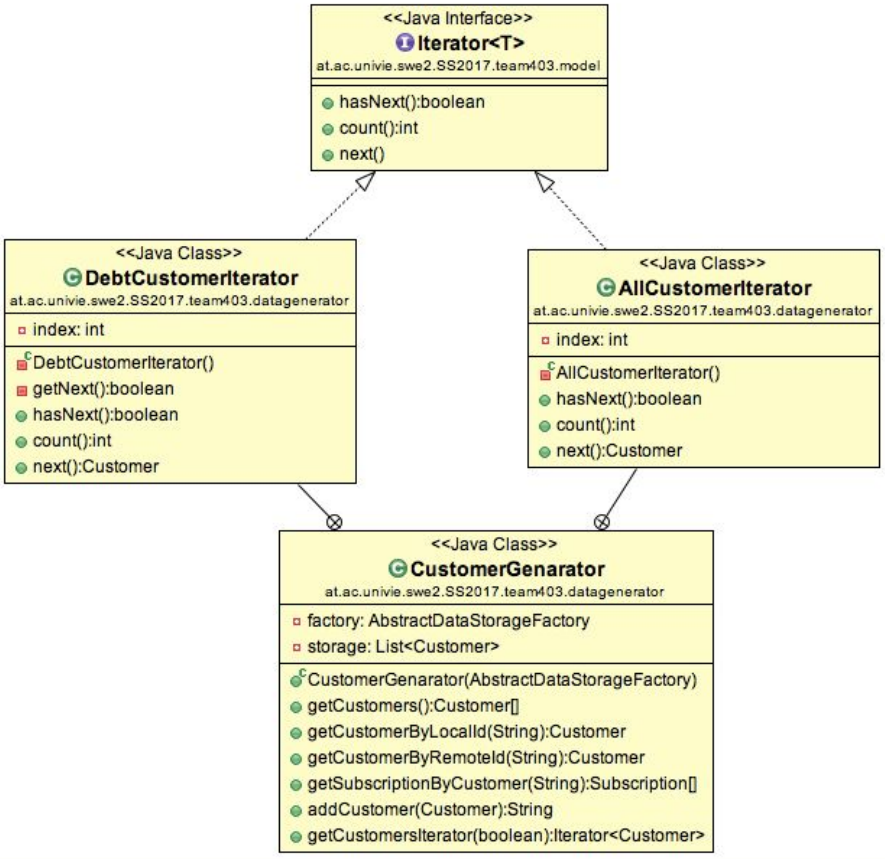
AR5:

Folgende Pattern wurden im Rahmen dieses Projektes in der Implementierung eingesetzt.

1. Iterator Pattern
2. Composite Pattern
3. Factory Method
4. Proxy Pattern
5. Observer Pattern
6. Abstract Factory Pattern
7. Decorator

1. Iterator Pattern:

Iterator Pattern wurden in den Klassen CustomerGenarator und FastBillCustomerStorage eingesetzt.


<<Java Interface>>
Iterator<T>
at.ac.univie.swe2.SS2017.team403.model
hasNext():boolean
count():int
next()
<<Java Class>>
DebtCustomerIterator
at.ac.univie.swe2.SS2017.team403.datagenerator
index: int
DebtCustomerIterator()
getNext():boolean
hasNext():boolean
count():int
next():Customer
<<Java Class>>
AllCustomerIterator
at.ac.univie.swe2.SS2017.team403.datagenerator
index: int
AllCustomerIterator()
hasNext():boolean
count():int
next():Customer
<<Java Class>>
CustomerGenarator
at.ac.univie.swe2.SS2017.team403.datagenerator
factory: AbstractDataStorageFactory
storage: List<Customer>
CustomerGenarator(AbstractDataStorageFactory)
getCustomers():Customer[]
getCustomerByLocalId(String):Customer
getCustomerByRemoteId(String):Customer
getSubscriptionByCustomer(String):Subscription[]
addCustomer(Customer):String
getCustomersIterator(boolean):Iterator<Customer>

```java
    /**
     *This class is used to access the elements of the class customer
     *in sequential manner without any need to know its underlying representation.
     */
    private class AllCustomerIterator implements Iterator<Customer> {
        private int index = 0;

        @Override
        public boolean hasNext() {
            return (index < storage.size());
        }

        @Override
        public int count() {
            return storage.size();
        }

        @Override
        public Customer next() {
            if(this.hasNext()) {
                return storage.get(index++);
            } else {
                return null;
            }
        }
    }

    /**
     *This class is used to access the elements of the class customer who has debt
     *in sequential manner without any need to know its underlying representation.
     */
    private class DebtCustomerIterator implements Iterator<Customer> {
        private int index = 0;

        private boolean getNext() {
            while (index < storage.size()) {
                for (Subscription subscription : storage.get(index).getSubscriptions()) {
                    for (Invoice invoice : subscription.getInvoices()) {
                        if (invoice.isUnpaid()) {
                            return true;
                        }
                    }
                }
                ++index;
            }
            return false;
        }


        @Override
        public boolean hasNext() {
            return (index < storage.size());
        }

        @Override
        public int count() {
            return -1;
        }

        @Override
        public Customer next() {
            if(this.getNext()) {
                return storage.get(index);
            } else {
                return null;
            }
        }
    }

    @Override
    public Iterator<Customer> getCustomersIterator(boolean onlyWithDebt) {
        if (onlyWithDebt) {
            return new DebtCustomerIterator();
        } else {
            return new AllCustomerIterator();
        }
    }
}
```

2. Composite Pattern:

Das Interface Billing wird in den Klassen in den Klassen Subscription, Customer und BackOfficeSystem implementiert. Wenn wir Billing bei BackOfficeSystem aufrufen, dann wird es für jeder Customer aufgerufen, jeder Aufruf bei Customer loest billing bei jeden seinen

Subscriptions aus. Damit können wir einzelne Elemente abrechnen oder die vom gesamten System

Codeabschnitt aus Billing.java

```java
package at.ac.univie.swe2.SS2017.team403.model;

/**
 *
 * Composite Pattern with Customer and Subscription
 */
public interface Billing {
    /**
     *
     * @returntrue, when the Object is changed
     */
    public boolean billing();
    /**
     *
     * @param payments an array of the payments
     * @return true, when the Object is changed
     */
    public boolean billing(Payment[] payments);
}

```

Codeabschnitt aus Customer.java

```java
    @Override
    public boolean billing() {
        return this.billing(null);
    }

    @Override
    public boolean billing(Payment[] payments) {
        boolean changed = false;
        for(Subscription subscription : getSubscriptions()) {
            if (subscription.billing(payments))
                changed = true;
        }

        return changed;
    }

```

Codeabschnitt aus BackOfficeSystem.java

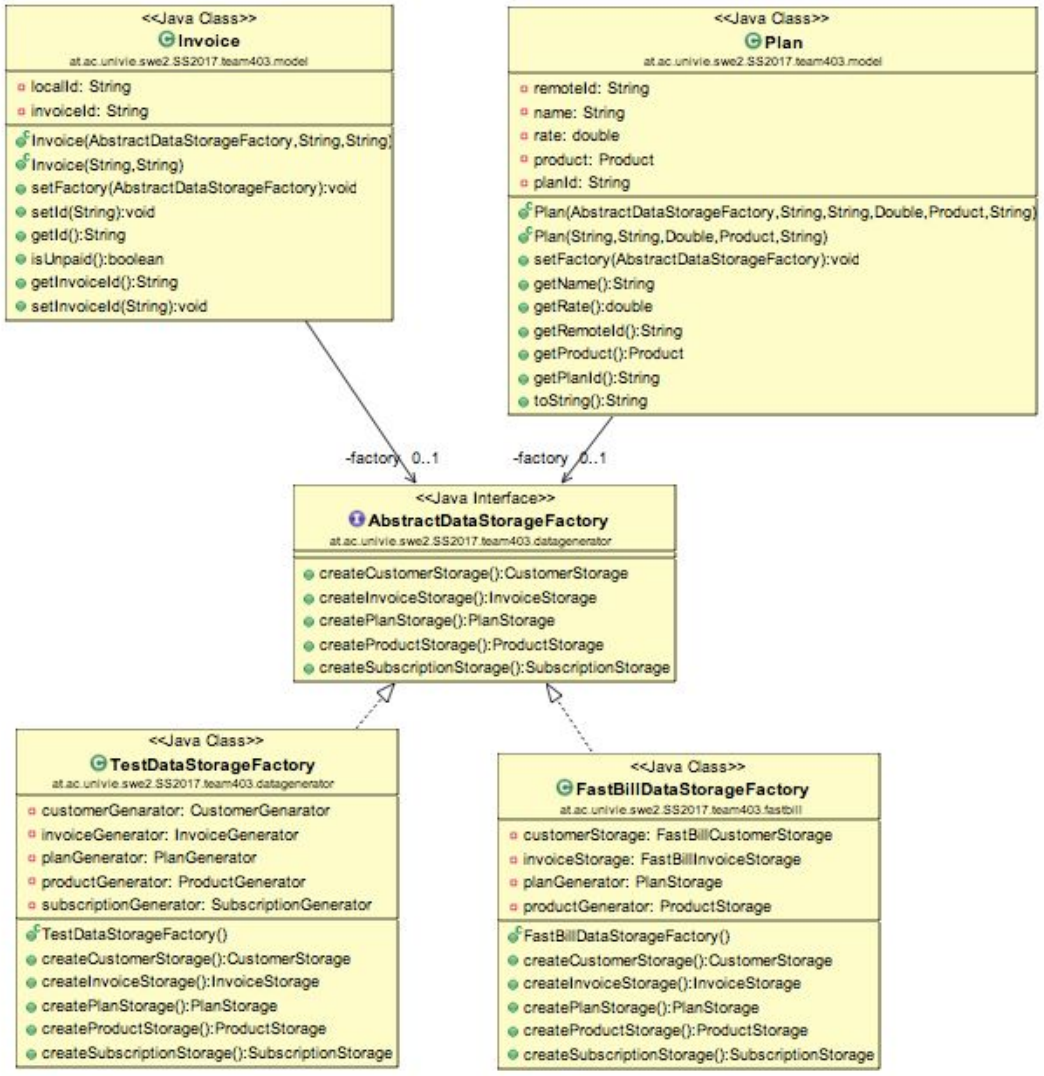
```
167
168     @Override
169     public boolean billing() {
170         return this.billing(null);
171     }
172
173     @Override
174     public boolean billing(Payment[] payments) {
175         boolean changed = false;
176         for (Customer customer : this.getCustomers()) {
177             if (customer.billing(payments)) {
178                 changed = true;
179
180                 for (CustomerListener l : listeners)
181                     l.afterCustomerChanged(customer);
182             }
183         }
184
185         return changed;
186     }
187
```

3. Factory Method Pattern

Diese Art von Pattern wurden in der Klasse FastBillCustomerStorage eingesetzt.
Codeabschnitt aus FastBillCustomerStorage.java

```
285
286     @Override
287     public Iterator<Customer> getCustomersIterator(boolean onlyWithDebt) {
288         IteratorCreator<Customer> creator;
289         if (onlyWithDebt) {
290             creator = new DebtCustomerIteratorCreator();
291         } else {
292             creator = new AllCustomerIteratorCreator();
293         }
294
295         return creator.factoryMethod();
296     }
297
298     private class DebtCustomerIteratorCreator extends IteratorCreator<Customer> {
299         @Override
300         public Iterator<Customer> factoryMethod() {
301             return new DebtCustomerIterator();
302         }
303     }
304
305     private class AllCustomerIteratorCreator extends IteratorCreator<Customer> {
306         @Override
307         public Iterator<Customer> factoryMethod() {
308             return new AllCustomerIterator();
309         }
310     }
311 }
312
```

4. Abstract Factory Pattern



Codeabschnitt aus AbstractDataStorageFactory

```java
package at.ac.univie.swe2.SS2017.team403.datagenerator;

import at.ac.univie.swe2.SS2017.team403.model.CustomerStorage;

/**
 * This factory-interface is used to create storages.
 * Abstract Factory Pattern
 */
public interface AbstractDataStorageFactory {
    public CustomerStorage createCustomerStorage();
    public InvoiceStorage createInvoiceStorage();
    public PlanStorage createPlanStorage();
    public ProductStorage createProductStorage();
    public SubscriptionStorage createSubscriptionStorage();
}

```

Implementiert wurde das Interface AbstractDataStorageFactory in TestDataStorageFactory und FastBillDataStorageFactory.

5. Proxy Pattern

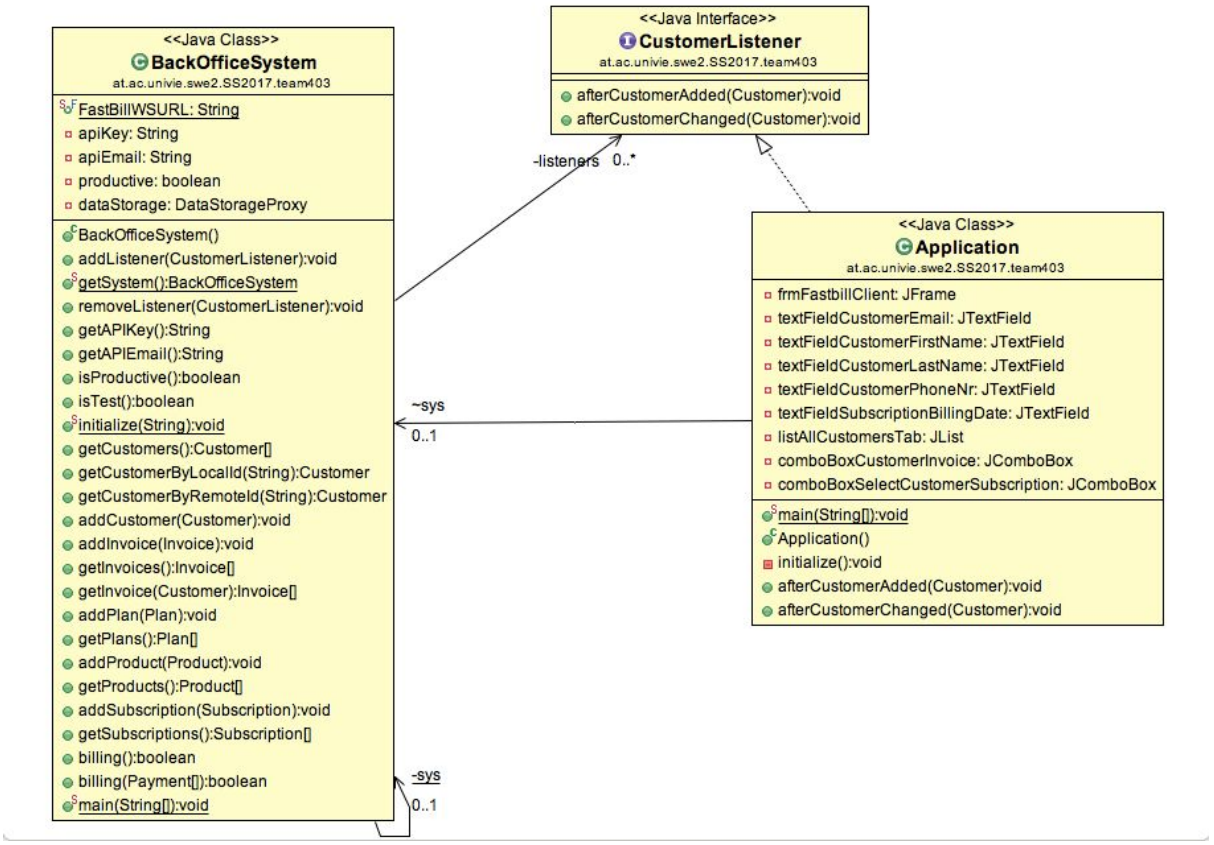
Proxy Pattern wurden in DataStrogeProxy implementiert.

Codeabschnitt aus DataStorageProxy.java

```java
package at.ac.univie.swe2.SS2017.team403;

import at.ac.univie.swe2.SS2017.team403.datagenerator.AbstractDataStorageFactory;

/**
 * This class is used to handle all Storages.
 * Proxy Pattern
 */
public class DataStorageProxy {
    private AbstractDataStorageFactory factory;

    public DataStorageProxy(boolean productive) {
        if (productive) {
            factory = new FastBillDataStorageFactory();
        } else {
            factory = new TestDataStorageFactory();
        }
    }

    public CustomerStorage getCustomerStorage() {
        return factory.createCustomerStorage();
    }

    public InvoiceStorage getInvoiceStorage() {
        return factory.createInvoiceStorage();
    }

    public PlanStorage getPlanStorage() {
        return factory.createPlanStorage();
    }

    public SubscriptionStorage getSubscriptionStorage() {
        return factory.createSubscriptionStorage();
    }

    public ProductStorage getProductStorage() {
        return factory.createProductStorage();
    }

}
```

6. Observer Pattern

BackOfficeSystem agiert als **Observable** und schickt die Events zu **Observer** Objects.
GUI Application agiert als **Observer**



Codezeilen aus CustomerListener.java

```java
package at.ac.univie.swe2.SS2017.team403;

import at.ac.univie.swe2.SS2017.team403.model.Customer;

/**
 *
 * Observer Pattern
 */
public interface CustomerListener {
    void afterCustomerAdded(Customer customer);
    void afterCustomerChanged(Customer customer);
}

```

<u>Codeabschnutt aus BackOfficeSystem:</u>

```java

    public void addCustomer(Customer customer) throws IllegalArgumentException {
        dataStorage.getCustomerStorage().addCustomer(customer);

        for (CustomerListener l : listeners)
            l.afterCustomerAdded(customer);
    }

```

7. Decorator

Customer implementiert CustomerReportNotifier, der ermoeglicht eine EmailReport zu Customer zu schicken
Wir haben dazu CustomerReportSMSNotifier implements CustomerReportNotifier
, mit dem kann man SMS verschicken.

Code Zeilen

```
public class CustomerReportSMSNotifier implements CustomerReportNotifier {

	private Customer customer;

	public CustomerReportSMSNotifier(Customer customer) {
		this.customer = customer;
	}

	@Override
	public void send() {
		// send sms
		System.out.println("send SMS to " + customer.getPhone());
	}

}
```

von BackOfficeSystem.java

```
		for (Customer customer : system.getCustomers()) {
			System.out.println(customer);

			CustomerReportNotifier notifier = customer;

			if (customer.getPhone() != null && customer.getPhone().length() > 0 )
{
				notifier = new CustomerReportSMSNotifier(customer);
			}

			notifier.send();
		}
```