**Q1**

a) A template pattern

b) A strategy pattern

c) Modify name of Classes *HtmlFormattedList* and *LatexFormattedList* to *HtmlLanguage* and *LaTeXLanguage* and make modfications on them. Also create an Interface *MarkupLanguage* and make changes in class *FormattedList* too.

**HtmlLanguage Class**

package Q1;  
  
public class HtmlLanguage implements MarkUpLanguage {  
  
 @Override  
 public String formatHeader() {  
 return "<ul>";  
 }  
  
 @Override  
 public String formatItem(String item) {  
 return " <li>" + item + "</li>";  
 }  
 @Override  
 public String formatFooter() {  
 return "</ul>";  
 }  
}

**LaTeXLanguage Class**

package Q1;  
  
public class LaTeXLanguage implements MarkUpLanguage {  
  
 @Override  
 public String formatHeader() {  
 return "\\begin{itemize}";  
 }  
  
 @Override  
 public String formatItem(String item) {  
 return " \\item " + item;  
 }  
  
 @Override  
 public String formatFooter() {  
 return "\\end{itemize}";  
 }  
}

**MarkUpLanguage Interface**

package Q1;  
  
public interface MarkUpLanguage {  
 String formatHeader();  
 String formatItem(String item);  
 String formatFooter();  
}

**FormattedList Class**

package Q1;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class FormattedList {  
  
 private List<String> content = new ArrayList<>();  
 private MarkUpLanguage language;  
  
 public FormattedList(MarkUpLanguage language, String... items) {  
 this.language = language;  
 content.addAll(Arrays.*asList*(items));  
 }  
  
 public void add(String item) {  
 content.add(item);  
 }  
  
 public void print() {  
 System.*out*.println(language.formatHeader());  
 for (String item : content) {  
 System.*out*.println(language.formatItem(item));  
 }  
 System.*out*.println(language.formatFooter());  
 }  
}

d) The strategy pattern is much more preferred than the template pattern.

· The latter one introduces tight coupling between the parent and child classes, due to the use of inheritance.

· Although duplication is removed using both patterns, when using the latter one, it is not possible to have separated independent reusable components.

· The strategy pattern avoids this problem by using composition, introducing more flexibility due to the looser coupling created between components.

**Q2**

a) Builder pattern

b), c) Create class *HttpRequestBuilder* and test class *HttRequestTest*

**HttpRequestBuilder Class**

package Q2;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class HttpRequestBuilder {  
 private String url = "";  
 private HttpRequest.Method method = null;  
 private List<String> params = new ArrayList<>();  
 private List<String> headers = new ArrayList<>();  
 private String body = "";  
  
 public HttpRequestBuilder withUrl(String url) {  
 this.url = url;  
 return this;  
 }  
  
 public HttpRequestBuilder withMethod(HttpRequest.Method method) {  
 this.method = method;  
 return this;  
 }  
  
 public HttpRequestBuilder withParam(String param) {  
 this.params.add(param);  
 return this;  
 }  
  
 public HttpRequestBuilder withHeader(String header) {  
 this.headers.add(header);  
 return this;  
 }  
  
 public HttpRequestBuilder withBody(String body) {  
 this.body = body;  
 return this;  
 }  
  
 public HttpRequest build() {  
 return new HttpRequest(url, method, params, headers, body);  
 }  
}

**HttpRequestTest Class**

package Q2;  
  
import org.junit.Test;  
  
import static org.hamcrest.CoreMatchers.*is*;  
import static org.hamcrest.MatcherAssert.*assertThat*;  
import static org.junit.Assert.*assertTrue*;  
  
public class HttpRequestTest {  
  
 private static final String *URL* = "http://exams.imperial.ac.uk/575";  
 private static final String *BODY* = "mark=100";  
 private static final String *HEADER* = "Date=02-05-2019";  
  
 @Test  
 public void canBuildHttpRequestUsingBuilder(){  
 HttpRequest request = new HttpRequestBuilder()  
 .withUrl(*URL*)  
 .withBody(*BODY*)  
 .withHeader(*HEADER*)  
 .build();  
  
 *assertThat*(request.getUrl(), *is*(*URL*));  
 *assertThat*(request.getBody(), *is*(*BODY*));  
 *assertTrue*(request.getHeaders().contains(*HEADER*));  
 }  
  
}

**Q3**

a), b), c) Create class *CardChecker* and *CardCheckerTest* and Interface *Observer*

**CardChecker Class**

package Q3;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class CardChecker {  
 List<Observer> listOfObservers = new ArrayList<>();  
  
 public void validate(String cardNumber){  
 if(cardNumber.length() != 12 || !cardNumber.matches("\\d+")){  
 for(Observer obs : listOfObservers) {  
 obs.alert(cardNumber);  
 }  
 }  
 }  
  
 public void addObserver(Observer obs){  
 listOfObservers.add(obs);  
 }  
  
 public void removeObserver(Observer obs){  
 listOfObservers.remove(obs);  
 }  
}

**Observer Interface**

package Q3;  
  
public interface Observer {  
 void alert(String cardNumber);  
}

**CardCheckerTest Class**

package Q3;  
  
import org.jmock.Expectations;  
import org.jmock.integration.junit4.JUnitRuleMockery;  
import org.junit.Before;  
import org.junit.Rule;  
import org.junit.Test;  
  
public class CardCheckerTest {  
  
 private static final String *CARD\_NUMBER\_1* = "111122223333";  
 private static final String *CARD\_NUMBER\_2* = "1111";  
 private static final String *CARD\_NUMBER\_3* = "2222";  
  
 CardChecker card = new CardChecker();  
  
  
 @Rule  
 public JUnitRuleMockery context = new JUnitRuleMockery();  
 Observer observerOne = context.mock(Observer.class, "observerOne");;  
 Observer observerTwo = context.mock(Observer.class,"observerTwo");;  
  
  
 @Before  
 public void initialize(){  
 card.addObserver(observerOne);  
 card.addObserver(observerTwo);  
 }  
  
 @Test  
 public void ValidateFirstCreditCard(){  
  
 context.checking(new Expectations(){{  
 never(observerOne);

never(observerTwo);  
 }});  
 card.validate(*CARD\_NUMBER\_1*);  
 }  
  
 @Test  
 public void ValidateSecondCreditCard(){  
  
 context.checking(new Expectations(){{  
 exactly(1).of(observerOne).alert(*CARD\_NUMBER\_2*);  
 exactly(1).of(observerTwo).alert(*CARD\_NUMBER\_2*);  
 }});  
 card.validate(*CARD\_NUMBER\_2*);  
 }  
  
 @Test  
 public void ValidateThirdCreditCardButRemovingOneObserverBefore(){  
  
 card.removeObserver(observerTwo);  
 context.checking(new Expectations(){{  
 exactly(1).of(observerOne).alert(*CARD\_NUMBER\_3*);  
 never(observerTwo);  
 }});  
 card.validate(*CARD\_NUMBER\_3*);  
 }  
  
}