Development with the PDCA Framework

1.1 Overview

At first, the Plan-Do-Check-Act management framework is introduced in the context of object oriented programming. Therefore, it is translated to a programming framework represented by generic and abstract Java[1] classes and interfaces. The intended use of the framework is to solve management problems (e.g. financial management) with a well-known pattern and a good fit to the plan-do-check-act cycle, which is the foundation in various management domains.

Generic PDCA Framework	
<<Plan>>	`public abstract class PlanProcess<T> implements Runnable {`
T: plan configuration as parameter container	`protected PlanningRules<T> planningRules;`
	`public abstract void plan();`
	`}`
<<Plan Rules>>	`public interface PlanningRules<T> {`
T: plan configuration	`public T applyPlanningRules();`
	`}`
<<Do>>	`public abstract class DoProcess implements Runnable {`
	`protected DoRules doRules;`
	`public abstract void operate();`
	`}`
<<Do Rules>>	`public interface DoRules {`
	`public void applyDoRules();`
	`}`
<<Check>>	`public abstract class CheckProcess<T> implements Runnable {`
T: type which is measured and compared with the objective.	`public CheckingRules checkingRules;`
	`public ObjectiveSetting<T> objectiveSetting;`
	`public MeasuredPerformanceValue<T> performanceValue;`
	`public abstract Deviation<T>`
	`getCheckResult(ObjectiveSetting<T> objective,`
	`MeasuredPerformanceValue<T> performanceMeasureValue);`
	`public void setPerformanceValue(MeasuredPerformanceValue<T>`
	`performanceValue) {`
	`this.performanceValue = performanceValue;`
	`}`
	`}`
<<Control Rules>>	`public interface CheckingRules {`
	`public void applyCheckingRules();`
	`}`
<<Deviation>>	`public abstract class Deviation<T> {`
T: type of deviation	`private Long id;`
	`protected T value;`
	`public Deviation(T value) {`

```java
      super();
      this.value = value;
   }
   public T getValue() {
      return value;
   }
   public Long getId() {
      return id;
   }
}
```

<<Act>>
T: type of corrective act output
K: type of deviation

```java
public abstract class ActProcess<T, K> implements Runnable {

   public CorrectiveActRules correctiveActRules;
   public AdaptiveActRules adaptiveActRules;

   public abstract CorrectiveActOutput<T> act(Deviation<K>
deviation);

}
```

<<Control Rules>>
Corrective Act Rule

```java
public interface CorrectiveActRules {
    public void applyActRules();
}
```

<<Control Rules>>
Adaptive Act Rule
(not used in the current single closed loop mgmt. process)

```java
public interface AdaptiveActRules {
   public void applyActRules();
}
```

<<Control Input>>

T: type of corrective act output

(e.g. CPPI: the part of risky asset investment, which is one input for the do-process)

```java
public abstract class CorrectiveActOutput<T> {

   protected T value;

   public CorrectiveActOutput(T value) {
      super();
      this.value = value;
   }
   public T getValue() {
      return value;
   }
}
```

<<Measure Rules>>
T: type which is measured

```java
public interface MeasureRules<T> {
   public MeasuredPerformanceValue<T> measure();
}
```

<<Performance>>

(e.g. CPPI: Total Shareholder Return)

```java
public abstract class MeasuredPerformanceValue<T> {
   protected T value;
   public T getValue() {
      return value;
   }
   public MeasuredPerformanceValue(T value) {
      super();
      this.value = value;
   }
}
```

Artifact 1 - Generic PDCA Programming Framework

Although this framework defines multiple abstract classes to implement the "Runnable" interface, there is also the possibility to use a second version **without**[2] an implementation of it. Especially in an enterprise context, where the process is not executed by calling the run() method but rather by receiving a message, it is recommended to avoid "Runnable" (e.g., in the onMessage(Message msg) method of the "MessageListener" interface from JavaEE[3])

1.2 Example usage of the framework

The PDCA programming framework may now be used to solve the CPPI strategy implementation task. According to the component diagrams the classes are mainly realized within the component CPPIStrategy, while CPPIDataService contains data access objects for requesting and inserting the CPPI model values. For simplicity reasons, the following section focuses exemplarily on the <<Act>> process implementation and the usage of the framework. Java Enterprise (JEE)[3] is the main technology used in the code snippets.

CPPIActProcessMDB.java

```java
@MessageDriven(mappedName="jms/ActQueue")
public class CPPIActProcessMDB implements MessageListener {

    private CPPICorrectiveActRules correctiveActRules;

    @EJB(name ="CPPIDataService")
    private ICPPIDataService dataServiceBean;

    //JMS connection factory
    @Resource
    private ConnectionFactory connectionFactory;

    //JMS session
    private Session session = null;

    //queue of next process (<<DO>>) for performing the investment
    @Resource(mappedName = "jms/doQueue")
    private Queue doQueue;

    //message producer (for sending messages to the doQueue)
    private MessageProducer doProcessMessageProducer;

    @PostConstruct
    public void init() {
        try {
            connection = connectionFactory.createConnection();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            doProcessMessageProducer = session.createProducer(doQueue);
            connection.start();
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onMessage(Message msg) {
   try {
        //receive the information to perform the act process for a certain customer
       Object receivedObj = ((ObjectMessage) msg).getObject();
       if (receivedObj instanceof StartActProcessDTO) {
            StartActProcessDTO dto = (StartActProcessDTO) receivedObj;

            //retrieve customer from the message DTO
            Customer customer = dto.getCustomer();

            //retrieve deviation result from preceeding check process
            CPPIDeviation deviation = dataServiceBean.
                    getCurrentDeviationForCustomer(customer.getId());

            //retrieve current CPPI values by customer
            CPPIValues values = dataServiceBean.
                    getCurrentCPPIValuesForCustomer(customer.getId())

            //actual realization of "act"
            ActProcess actProcess = new ActProcess(customer);

            //applying the act rule within act(...)
            BigDecimal exposure = actProcess.act(deviation).getValue();

            //update results
            values.setExposure(exposure);
            values.setReserveasset(
                    values.getPortfolio()
                            .subtract(values.getExposure()));
            values.setPartRiskyAsset(
                    values.getExposure()
                            .divide(values.getPortfolio(),4,RoundingMode.HALF_UP));
            values.setPartRisklessAsset(
                    values.getReserveasset()
                            .divide(values.getPortfolio(),4,RoundingMode.HALF_UP));

            //persist results
            dataServiceBean.storeNewCPPIValuesForCustomer(values, customer);

            //asynchronous call (send message) to the do process
            // in order to undertake the investment for the given customer
           ObjectMessage doMessage =
                    session.createObjectMessage(new StartDoProcessDTO(customer));

            doProcessMessageProducer.send(doMessage);
        }
   } catch (JMSException ex) {
       System.out.println("Message transport was not successful.");
   }

}
```

Artifact 2 - CPPIActProcess Message Driven Bean

CPPIActProcess.java

```java
public class CPPIActProcess extends ActProcess<BigDecimal, BigDecimal> {

    private CPPIValues cppiValues;

    public CPPIActProcess(CPPIValues cppiValues) {
        this.cppiValues = cppiValues;
    }

    @Override
    public CorrectiveActOutput<BigDecimal> act(Deviation<BigDecimal> deviation) {
        CPPICorrectiveActRules correctiveActRules = new CPPICorrectiveActRules();
        correctiveActRules.setDeviation(deviation);
        correctiveActRules.setLeverage(cppiValues.getLeverage());
        correctiveActRules.setPortfolio(cppiValues.getPortfolio());

correctiveActRules.setMaxRiskyFraction(cppiValues.getConf().getMaxRiskyFraction());

        //actual computation of the risky part of the investment (exposure)
        correctiveActRules.applyActRules();
        return correctiveActRules.getCorrectiveActOutput();
    }
}
```

Artifact 3 - CPPIActProcess (act realization)

As it can be seen in Artifact 2 and Artifact 3 the implementation of the act process is split up to two classes. The first one is a message driven bean and receives messages for performing the act process for a certain customer. This class (=CPPIActProcessMDB) is in charge of retrieving the necessary parameters from the CPPIDataService, which then queries the database. These parameters get passed to CPPIProcess class, where the actual implementation of the act method is located. The act rule is applied and the result is returned to the message driven bean, which persists it and sends a message to the do process in order to perform the investment accordingly.

1.3 Comments

The implementation of the PDCA management framework with messaging, data services and container management (e.g. dependency injection) allows the developer to create an application which is modular, scalable and suitable for an integration within an enterprise environment. (Nevertheless, the framework could also have been used in a simplified and local environment.) Due to the guidance of the framework with its’ encapsulation of the rules and the processes, the interchangeability of those objects is ensured, which makes it easy to replace some of them with other algorithmic variants.

Clarifications and Simplifications for the Assignment

[1] You are provided with the Java PDCA framework in TUWEL. Integrate the Java files in your project as a package and derive your concrete PDCA classes from them. Since you are allowed to use other programming languages too, you should transform the framework classes to classes in the desired language.

[2] Probably it will be more suitable for your enterprise-like implementation to remove the “implements Runnable” clause from the respective classes. You are not allowed to perform other manipulations to the provided framework interfaces and abstract classes (except package changes).

[3] The Java EE example, presented above, does not consider the CPPIDataService as a separate microservice, rather it is invoked as an EJB within the same container. Therefore, the code snippet is not 100% correct, considering the requirements. So please avoid to just copy-and-paste it. It has to be embedded in a proper microservice environment (e.g., with Spring Boot), where each component is a separate process.