Introduction to Software Engineering

Assignment 8

Dimitri Tabatadze, 2024-12-04

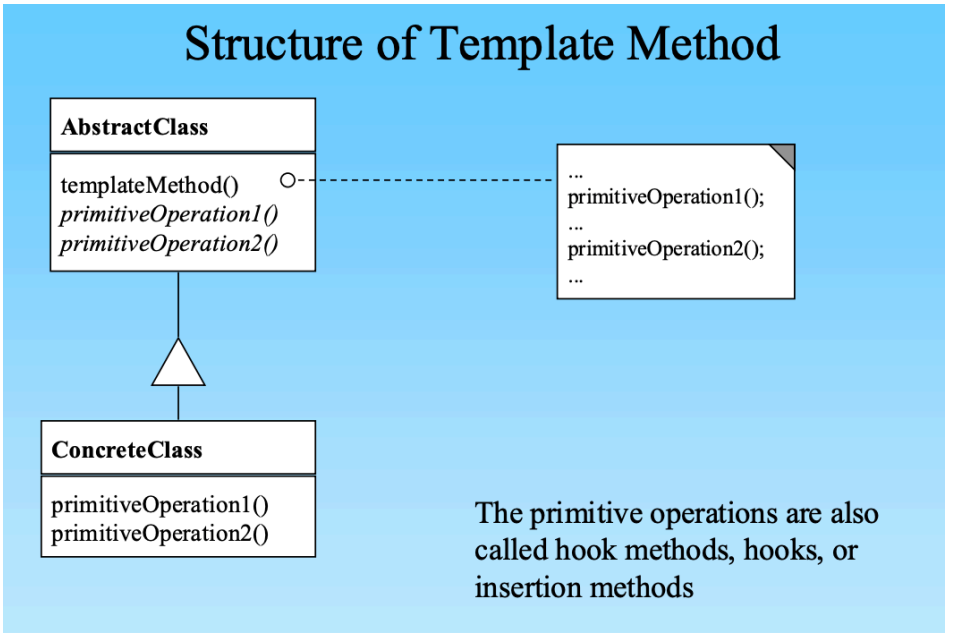
1

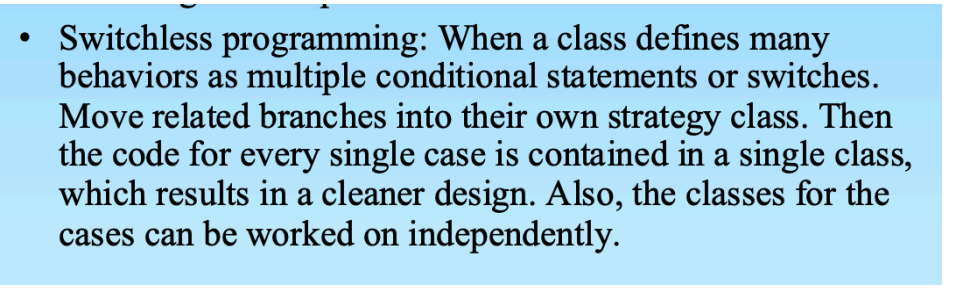
Explain why Template Method is an instance of “switch-less programming”. Use UML and proper English (complete, correct sentences) in your explanation.

Solution

The *Tmplate Method* is a way to have multiple implementations of the same method deffered to different subclasses. In switchless programming we want to remove the case splitting from the code that doesn’t necessarily need to know about it for cleannes sake. The Template Mehtod allows us exacly that.

Note: I don’t konw what I would use UML here for. I think my explaination is enough. Here’s some screenshots though:





2

This problem is about implementing the design pattern *Observer* in correct Java. (This problem was an exam question, so try to do it from memory, without looking at the pattern.) Use the conventions for the implementation of UML class diagrams presented in class. Include constructors.

a) Complete the following classes:

```java
public abstract class Subject { ... }

public class ConcreteSubject extends Subject {
  State subjectState;

  public State getState() {
    return subjectState;
  }

  ...
}

public class ConcreteObserver extends Observer {
  State observerState;

  ...
}
```

b) The above assumes the *pull* model. Discuss what needs to be changed for a *push* model.

c) Discuss the advantages and disadvantages of the *push* model vs. the *pull* model.

Solution

a)

```java
public abstract class Subject {
  private List<Observer> observers;

  Subject() {
    this.observers = new ArrayList<Observer>();
  }

  public void attach(Observer o) {
    this.observers.add(o)
```

```
  }

  public void detach(Observer o) {
    for (int i = 0; i < this.observers.length; i++) {
      this.observers.remove(i);
      break;
    }
  }

  public void notify() {
    for (Observer o : this.observers) {
      o.update();
    }
  }

  public abstract State getState();
  public abstract void setState(State newState);
}

public class ConcreteSubject extends Subject {
  State subjectState;

  ConcreteSubject() {
    super();
    this.subjectState = new State();
  }

  public State getState() {
    return subjectState;
  }

  public void setState(State newState) {
    this.subjectState = newState;
  }
}

public abstract class Observer {
  private Subject subject;

  Observer(Subject subject) {
    this.subject = subject;
  }

  public abstract void update();
```

```
54 }
55
56 public class ConcreteObserver extends Observer {
57   State observerState;
58
59   ConcreteObserver(Subject subject) {
60     super(subject);
61   }
62
63   public void update() {
64     observerState = this.subject.getState();
65   }
66 }
```

b) In the *pull* model, the observer fetches all data from the subject whereas in the *push* model, the subject passes the data in the update method.

To change to the push model, we need to change two functions: `notify` and `update`.

```java
1  public abstract class Subject {
2    ...
3
4    public void notify() {
5      for (Observer o : this.observers) {
6        o.update(this.getState());
7      }
8    }
9
10   ...
11 }
```

and

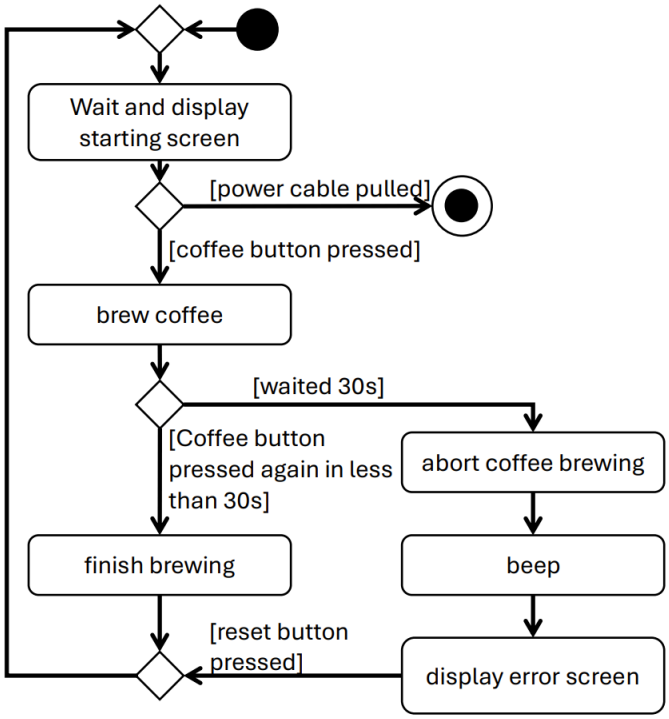
```java
1 public class ConcreteObserver extends Observer {
2   ...
3
4   public void update(State newState) {
5     observerState = newState;
6   }
7 }
```

c)

Push	*Pull*
asynchronous friendly	may skip an update asynchronously
overnotification	can skip updates

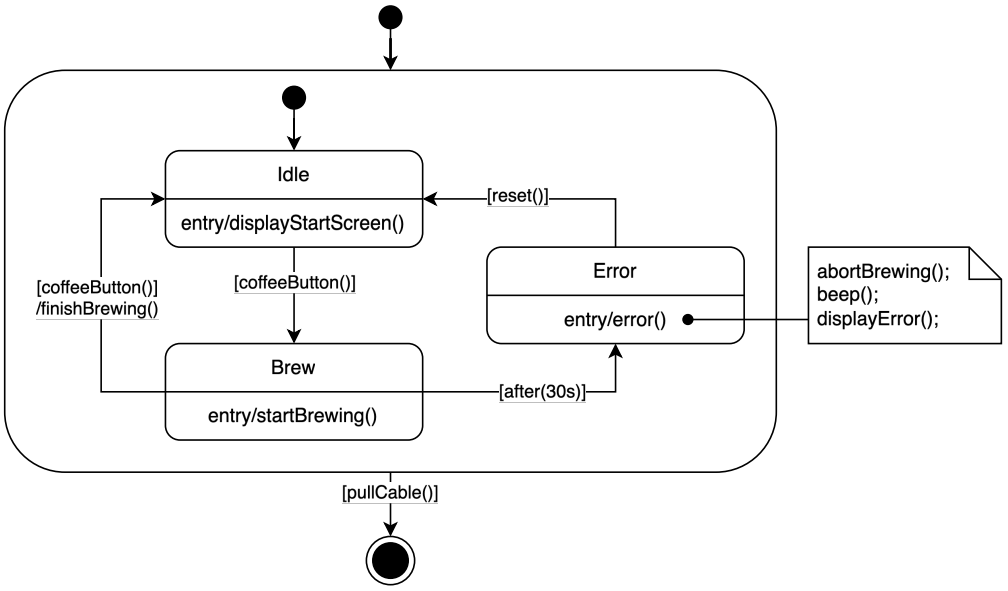
3

The coffee machine of your team is not working right. The main problem seems to be that one must press the coffee button twice, otherwise the brewing process stops. Your boss who needs coffee to work has drawn an activity diagram and now asks you to fix the machine. To get a better view of what happens when, you decide to figure out what the states of the machine are. The activity diagram is below. (SS19 ü4)



Convert the UML activity diagram above into a UML state chart. Model the states of the machine as accurately as possible. Your UML state chart should have as few states as possible. Provide the state transitions with events and operations and use entry, exit and continuous actions as well as conditions if necessary.

Solution

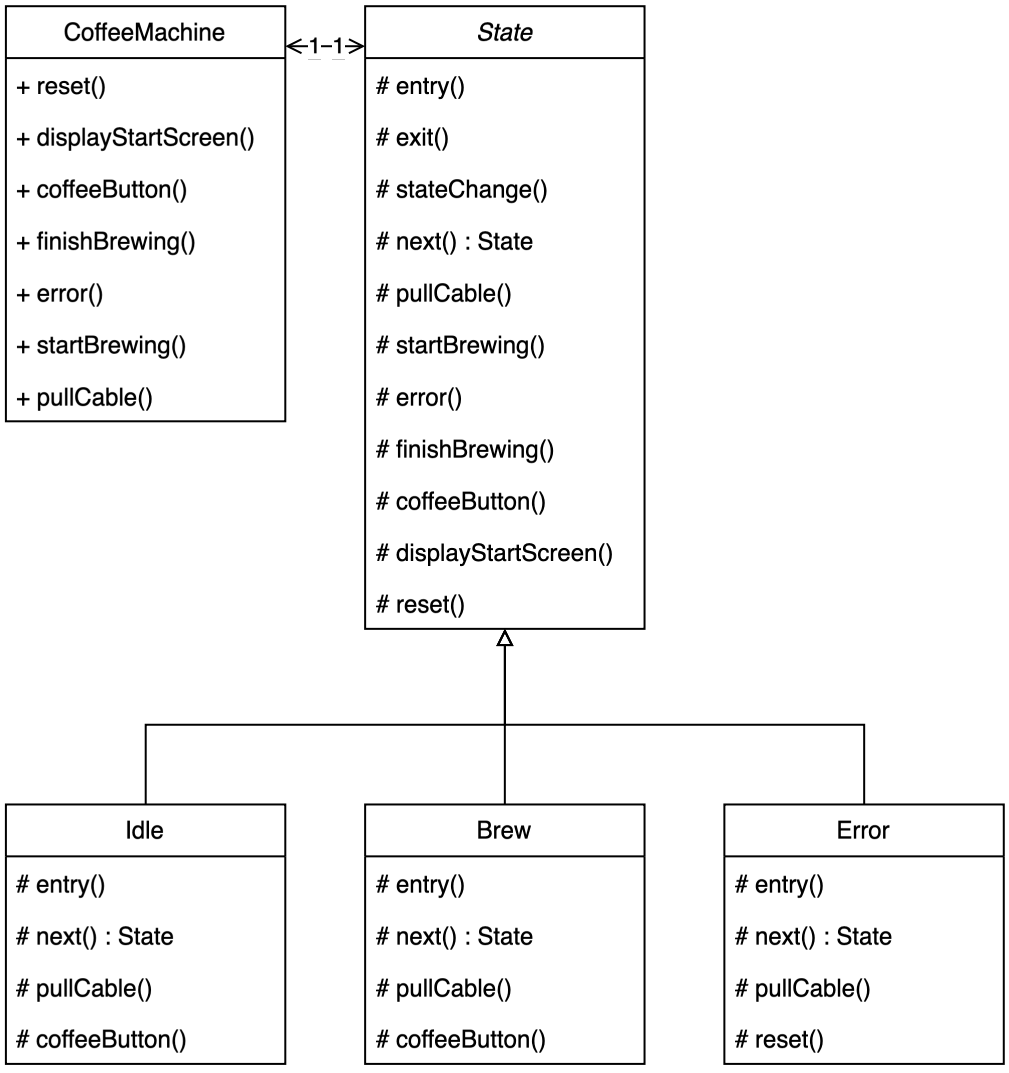


> Note: The power cable can probably be pulled while being in any state, so I considered it correct to do that even though the given activity diagram only assumed that the cable could get pulled while idle.

4

Convert the state chart you produced in the previous problem into a UML class diagram according to the state machine design pattern presented in class.

Solution


CoffeeMachine
+ reset()
+ displayStartScreen()
+ coffeeButton()
+ finishBrewing()
+ error()
+ startBrewing()
+ pullCable()
1–1
State
entry()
exit()
stateChange()
next() : State
pullCable()
startBrewing()
error()
finishBrewing()
coffeeButton()
displayStartScreen()
reset()
Idle
entry()
next() : State
pullCable()
coffeeButton()
Brew
entry()
next() : State
pullCable()
coffeeButton()
Error
entry()
next() : State
pullCable()
reset()