SWEN30006 Software Modelling and Design
Workshop 6: State Machines

School of Computing and Information Systems
University of Melbourne
Semester 2, 2020

Completion: You *should* complete the exercises ***as a team***. To receive a workshop mark, you **need** to demonstrate your active participation during the workshop. Your tutor will observe your participation for you to be eligible to receive a mark for this workshop. The active participation include (but not limit to) present your solutions, help your teammates, and actively engage with the team. Attending the workshop with only passive participation will *not* be given a mark. See the Workshop Overview under Subject Information on the LMS for more details.

Requisite Knowledge and Tools

To complete this workshop you will need to be familiar with the following terms and concepts:

- Domain models and class diagrams
- Sequence diagrams
- UML state machine diagram notation
- Implementing state machines

Introduction

State machines are a form of dynamic modelling that is principally concerned with the different states that a given piece of software can be in, along with the allowable transitions into and out of those states. State machines can be applied at a number of different levels, from low level class design to high level systems architecture.

This week you will be creating state machines from system descriptions. You will start by creating a domain model and then move into developing a state machines to represent this model. We will then finish by implementing the state machine in code.

Background Knowledge

A UML state machine diagram is designed to show all the states that a given system/object can be in and the valid transitions between these states. The *State* of a system is the condition of an object/system **at a given point in time**. The choice here is what level of detail we wish to show with our state machine. Because state machines are generally used as a tool to add understanding to an existing model (e.g., use

case or sequence diagrams), it is important that the level of detail in the state machine diagram matches the corresponding model.

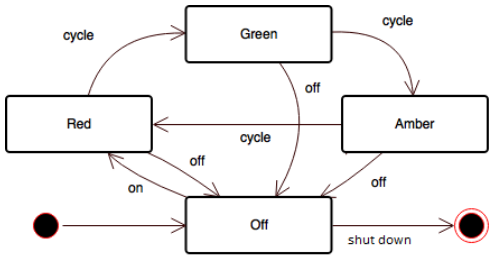

Figure 1: An example state machine for a traffic light

A *State* may also represent a range of values. For example, in Figure 1 we could consider the colour of a traffic light to be a state. A state machine model should show the full lifecycle of the object/system of interest. We indicate the initial state (denoted by the target of the transition from the solid circle) and we can include final states denoted by a solid circle with a concentric circle around it. In the our example of traffic light, the final state is only reached when the traffic light has been shut down. We represent states in our state machine diagram using rounded rectangles. The name of the state should be descriptive and use terms from within the problem domain. We then represent transitions between states with connecting arrows, the direction of which indicates the direction of the allowable transition. Transitions represent events that occur within a system, and are generally one of three types:

1. *External Event*: An event that is caused by something outside our system. For example, receiving touch inputs from a user.
2. *Internal Event*: An event caused by something inside our system. For example, the notification component of the observer pattern.
3. *Temporal Event*: An event caused by a specific date or time, generally controlled by checks against a system clock.

Determining the events that occur within a system and the transitions between states that they create is key to creating usable and realistic state machine models. Though this can be tricky at first, state machine models can be very useful especially when trying to deliver secure and safe software. State machines often provide a perspective on system design that helps developers to identify the necessary guards to put in place when implementing a system. They help us to identify what precautions need to be put in place to prevent invalid state transitions and ensure system correctness and safety.

The System Under Discussion: A credit card management system

You recently joined a software company and have been assigned to work on a team which is developing a credit card management system. The use cases for the account management functionality of this system have already been extracted. From these use cases we can extract a list of actions which should be allowed by the software. These actions are:

- When a customer is onboarded (i.e., an application is approved and a credit card is issued), the associated account is pending until the customer activates the credit card.
- When the credit card is activated, the account status becomes active.
- If a customer reports that the credit card is lost or stolen, the account will be pending and a new card is issued.

- If the credit card usage exceeds the credit limit, the account will be suspended. The account can become active again when its available funds are greater than zero.
- If a customer fails to pay a bill on time, the account will be in default.
- If a customer whose account in in default pays all outstanding bills, the account is once again active.
- If a customer does not respond appropriately while the account is in default, the account will be closed.
- If an account is closed, the customer must go through the same process as a new customer (i.e., submit a new application).
- If a customer does not have outstanding bills, but the credit card was not used for more than 6 months, the account is inactive. The account will become active again when the customer contact the credit card company to resume the account.
- If an account is inactive for more than 6 months, the account will be automatically closed.

Part 1 Building Models

Task 1.1 Creating a State Machine

Using the actions listed described in the previous section, build a state machine model for an account. You should consider what states an account can be in and the valid transitions between these states. You may have to make some assumptions or inferences.

Discussion: What are the types of the events in your model? Are they external, internal, or temporal?

Task 1.2 Creating A Nested State

You may have previously identified a state that represents the default status of an account when the payment of that account is overdue. At a high level this is correct, but often within such a billing system there are a number of different states that an account with an overdue payment can be in. The credit card provider can use these different states to manage the customer's debt. In the software you are building, the following rules apply for a credit card account in default:

- When a customer fails to pay their bill on time but does not have a history of missed payments, the customer is given a grace period where they can make the payment without incurring additional fees or penalties.
- If a customer has a history of missed payments, or fails to make a payment within the grace period, they are charged a $20 fee and offered a payment plan.
- If a customer accepts a payment plan, they are classified as having a healthy debt.
- If a customer fails to respond to the payment plan offer within a reasonable time period or refuses the payment plan, they are classified as having an unhealthy debt.
- If a customer with a healthy debt misses a payment they will be reclassified as having an unhealthy debt.
- Any account classified as having an unhealthy debt will eventually be referred to collections. Collections are handled by a third party agency.
- If a collection agency is used on an account which is in default, the debt will be written-off or collected; either way, the account will be closed.

- If a customer pays their full overdue bill at any point prior to their account being referred to collections, any restrictions on their account are removed and they are once again active.

Using these rules, build a state machine model for the sub-states that an account can be in when it is in default. Again, compare and discuss your state machine with your neighbour's. Once you are confident with your solution, incorporate these sub-states into your first diagram using UML state machine syntax for nested states.

Part 2 Implementing State Machines

Task 2.1 Creating a Design Model

Create a design class diagram for your state machine. You should create the appropriate classes to represent an account, methods to trigger transitions, and enumeration to represent states.

Task 2.2 Writing Code

Using your class diagram, implement your state machine in Java. Make sure you include the nested states. Add an appropriate `main` method to demonstrate that your implementation allows only valid state transitions. This method should show examples of all the elements of a state machine, including states, transitions, events, guards, and actions. You will need to be able to explain how your implementation matches your state machine model.