UML & State Diagram Modelling

Part I

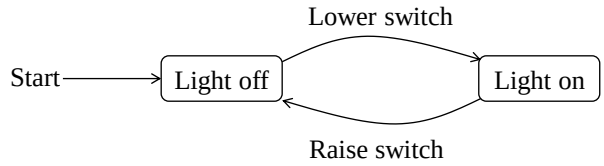
CSCU9P5 – Software Engineering I

State Diagrams

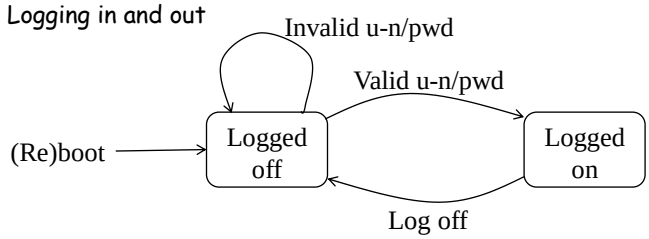
- In UML, we specify **the behaviour of the individual objects of a class** through **State Diagrams** (State Machine Diagrams).
- Based on the notion of **finite automaton** or **finite state machine**, UML State Diagrams are an extension of State charts [Harel '87,'88].
- Central notions: **state** and **state transformation**:
  - A system may assume different *"states"* at different times, and *change* its state according to *event occurrences*
    - Typically methods calls in an object oriented world
  - Different states may characterise
    - Different internal conditions/situations
    - So also different way in which the system can react to inputs
- Easiest to see from some simple examples – next slide

Simple examples

- First a non-computing example:
  - A light in a room, controlled by an up/down switch



- Second, a computing example:
  - Logging in and out



State Diagrams

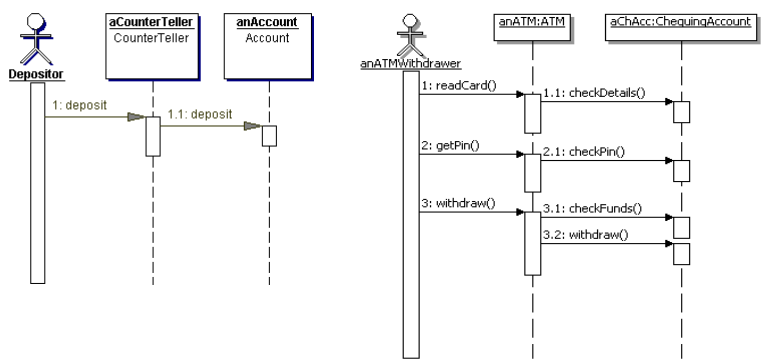
- A state diagram describes a system that, at any given time, is in a certain **state** and can take part in an **event**, such as having one of its methods called
  - State Diagrams specify **how an object reacts when each of its methods (operations) is called**
- As a consequence of that **event**, the object can:
  - move to a **new state**,
  - perform **actions**, such as
    - calling the methods of other objects, or
    - change the values of attributes
- We say that **the event triggers a transition**
- We **label** the transition with the name of the event
  - And possibly other information

State vs Sequence Diagrams

- A State Diagram is **different from** a Sequence diagram:
  - The former is a view on the "internal" behaviour of an object
  - The latter gives us information about how **multiple objects** can interact with each other and collaborate to carry out a task.
- State Diagrams do not need to be really detailed in terms of data-flow

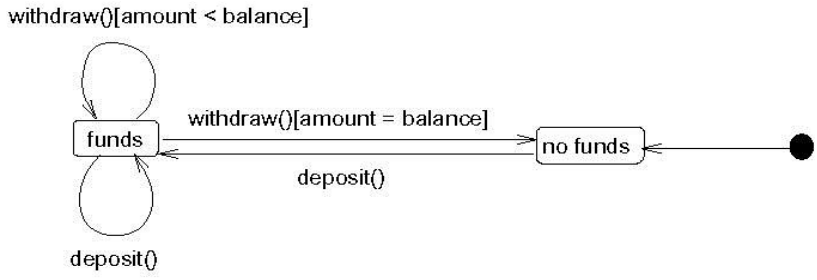
Example

- Let us look at this through the simple example of our **`Account`** class
- The **state** of an Account object can change when we deposit or withdraw money



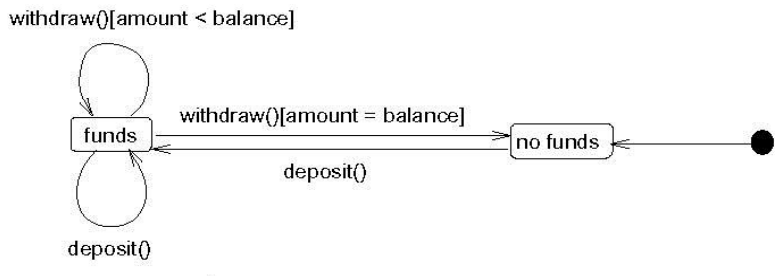
State diagrams: States

- *Every* combination of values in the attribute variable(s) could be considered as a *separate* state – "microstates"
  - Usually not practical/useful
  - Instead *aggregate* them to give *useful* modelling states
- Example: The states that an **`Account`** object can go through can be shown in a state diagram as follows



State diagrams: Start marker

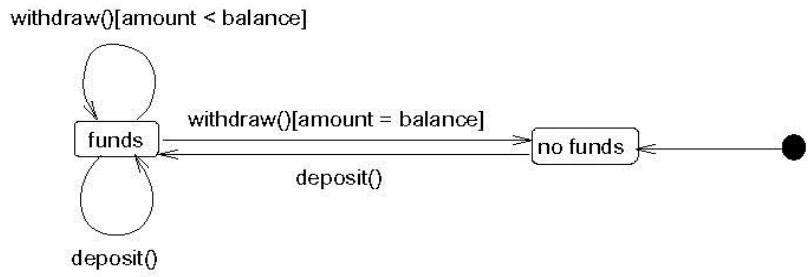
- The filled circle is the **start marker**
- It means that when a new object of class **`Account`** is created, it starts in the **`no funds`** state.



- Objects of class **`Account`** can accept calls of methods **`deposit`** and **`withdraw`**

State diagrams: Operations

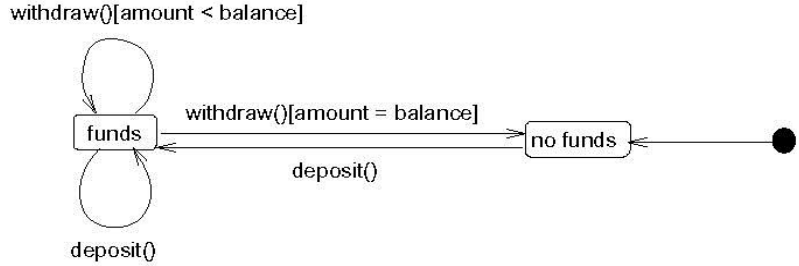
- The state diagram shows that when an object is in the **no funds** state, the only operation that can be invoked is **deposit**
- When in the state **no funds**, we cannot withdraw money



- When the object receives a call of **deposit**, the object moves to the **funds** state

State diagrams Operations

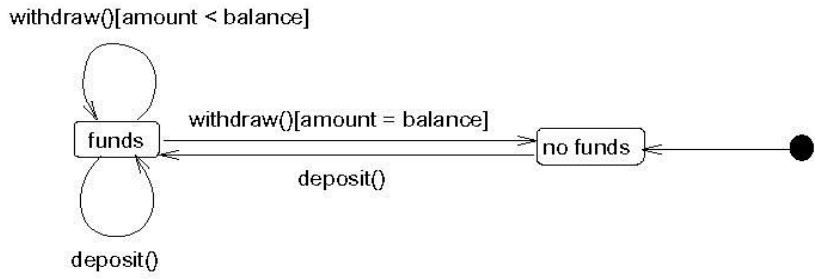
- When in the **funds** state, the object can receive more calls of **deposit**. These cause us to stay in the **funds** state



- The effect of **withdraw** depends on the current state of the object.
- It can only be called in the state **funds** and its effect depends on whether there are sufficient funds to satisfy the request

State diagrams: Operations

- There are three situations for **`withdraw`**:
  - **sufficient funds**: stay in **`funds`** state and decrement **`balance`** attribute



  - **exactly correct funds**: move to **`no funds`** state (and make **`balance`** attribute zero)
  - **insufficient funds**: we can regard this as an **error**, i.e. we should never receive a **`withdraw`** message in this circumstance.

State diagrams: Guards

- To keep things simple, we are only modelling **correct** situations, i.e. we are not showing error transitions in the diagram.
  - They may greatly complicate the diagram.
- We distinguish between different correct behaviours by putting **guards** on events,
  - e.g. **`amount < balance`**
- Note the connection with **pre-** and **post-conditions**
- In **Design by Contract** view, the pre-condition:

`amount<=balance`

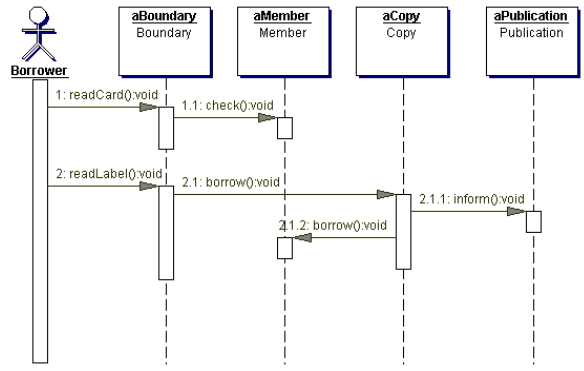
  - on the **`withdraw`** operation, does not require the **`Account`** class to specify behaviour when the pre-condition is not met
  - It is the responsibility of the client/caller to ensure that the pre-condition is satisfied.

State diagrams: Actions

- We can associate **actions** with each event.
- An action can involve:
  - calling a method,
  - changing an attribute such as **`balance`**, that is we can have assignments.
- However, *in the early stages*, we do not want to put too much detail into a state diagram; the emphasis is on getting the main structure right

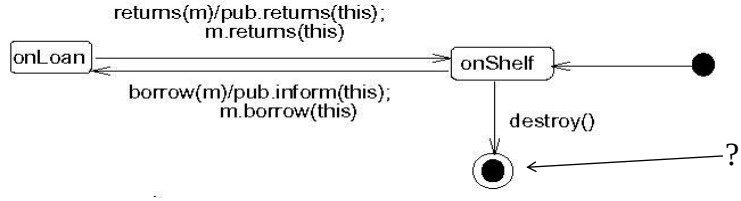
Library Example

Let us now devise a state diagram for the **Copy** object that we had in the **Library** case study.



Copy object

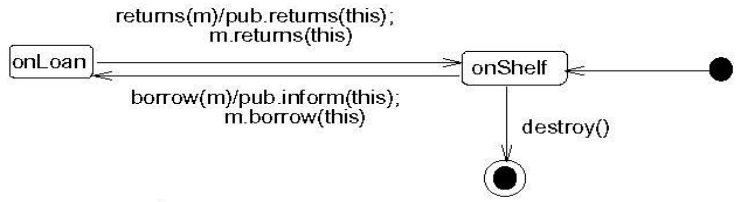
- When a **Copy** object is initially created, it is in state **onShelf** and may receive a **borrow** message



- As a result of receiving this message, the **Copy** object will call (see the Sequence Diagram):
  - the **inform** method of its associated **Publication** object and
  - the **borrow** method of the **Member** object that is doing the borrowing.
- We show this **action** by putting these calls after a '/' and separating them by ";"

Copy object

- The call of **borrow** has a reference to the borrowing **Member** object as the parameter **m**



- When the **Copy** object was created, the association with a **Publication** object was set up.
- That is shown in an implementation by the **Copy** object having a reference to a **Publication** object as the attribute pub.
- We then use **m** and **pub** in the actions of calling other object's methods.

Copy object

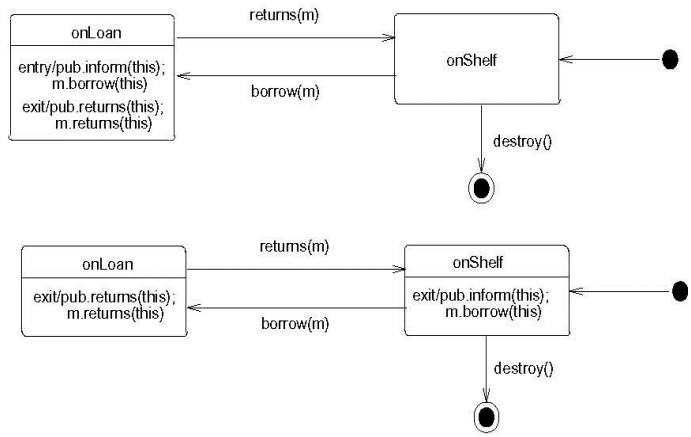
- Instead of writing the actions on the transitions, we can write the actions inside the state.
- This means that actions are not specific to a transition, but rather of a state
- More precisely, there is an implicit
  - `entry` event when a state is entered and
  - `exit` event when it is left

  And actions can be associated with these events

Copy object

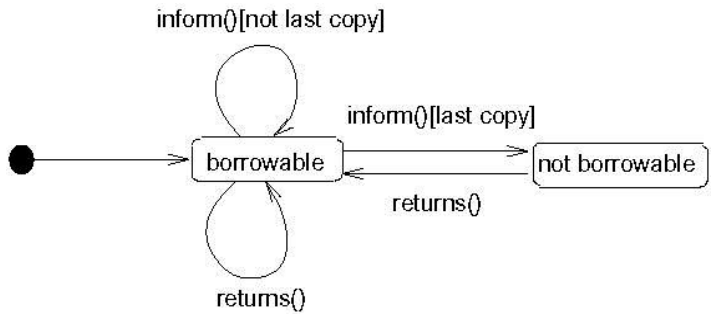
UML does not indicate what the syntax of the actions should be.

The State Diagram for `Copy` can be shown in either of the two following ways:



Class `Publication`

- Let us now look at a possible state diagram for **`Publication`**:



End of first part

Next: Generating code, time events, and substates