UML & State Diagram Modelling

Part II

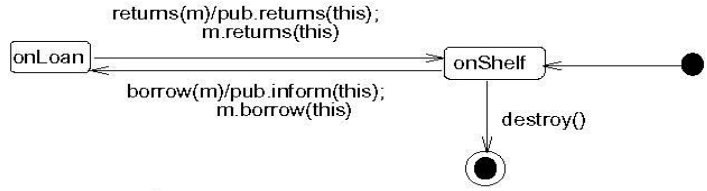
CSCU9P5 – Software Engineering I

Generating Code

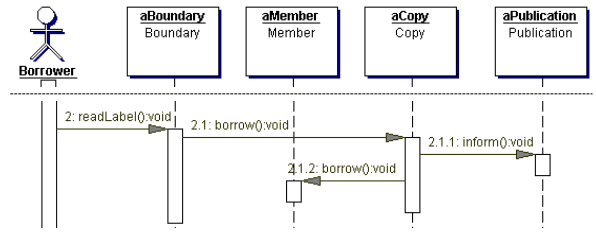
- The **state diagrams** have defined the behaviour of an object of class `Copy`.
- It is therefore possible to generate the corresponding Java or C++ code.
- How do we do this?
- Instance (global) variables come from class diagram attributes and associations, and sequence diagram messages *sent*
  - The state is determined from these
  - …either directly or indirectly
- Methods come from class diagram operations, sequence diagram messages *received,* and state diagram transitions
  - Bodies from sequence diagram activations & state diagram actions
  - Plus optional guards and validity checks

Copy object

- Recall the state diagram that we saw for a **Copy** object:



- And an extract from the sequence diagram for borrowing:



Generating Code

```
public class Copy {
   // Attributes
   // Parent Publication
   private Publication pub;

   // Define the possible states
   private final static int onShelf = 0;
   private final static int onLoan = 1;
   private int currentState;

   public Copy(Publication p) {   // Constructor
      pub = p;
      currentState = onShelf;
   }
```

For this class:
The state happens to be explicitly held as an attribute

Generating Code

```
public void borrow(Member m) {
   if (currentState == onShelf) {
      pub.inform(this);
      m.borrow(this);
      currentState = onLoan;
   }
}
```

- State diagram implies the *pre-condition* for **borrow** is:

  currentState == onShelf

- Following **Design by Contract**, an unexpected message is just ignored
  - In state **onLoan**: the **borrow** method "ignores" the call
  - In a more strict interpretation: **borrow** *assumes* the pre-condition is true and the **if** statement is not required
- Alternatively, in **Defensive Programming**:
  - An **else** part would give error response of some kind

Generating Code

```
public void returns(Member m) {
   if (currentState == onLoan) {
      pub.returns(this);
      m.returns(this);
      currentState = onShelf;
   }
}
} // end Copy
```

- State diagram implies the *pre-condition* for **returns** is:

  currentState == onLoan

- We have DbC:
  - In state **onShelf**, the **returns** method ignores the call
  - In a more strict interpretation:
    returns *assumes* the pre-condition is true and the **if** is not required
- In DP: an **else** part would give error response

Generating Code

- Note that the information in the generated code comes **from several different** diagrams.
- A drawback of the code is that it is **not really very readable** by humans.
- However, we can regard the UML diagrams as our **source program** and the (automatically) generated Java or C++ as **machine code**.
- Idea is that the UML is at a higher more **abstract level** and so we are better **able to reason** at that level of abstraction about the overall design.
- It should therefore be **easier to be confident** that the system, and hence the corresponding "generated" code, satisfy the requirements.

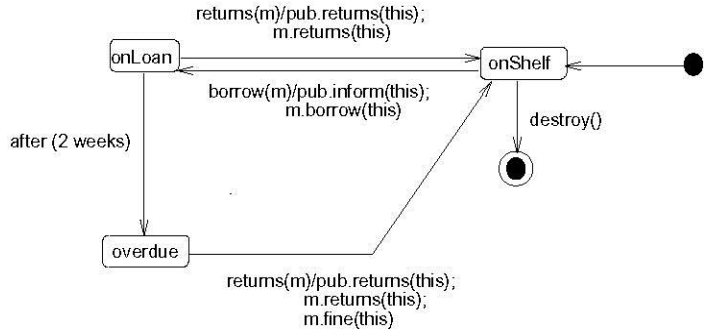
Other triggering events

- So far, in our state diagrams, **transitions** have been caused by a **method** of the object being **called** – **"call events"**
  - The object can then, as an action, **call** the methods of other objects…
  - …which may in turn cause them to undergo state transitions
- We also have **"change events"**:
  - A change event represents the satisfaction of some condition (changing from false to true) – a new state is entered
  - Typically due to a **change in the values of attributes**
  - This is modelled by a **transition** labelled with an event of the form:

```
when(x == 10)
```

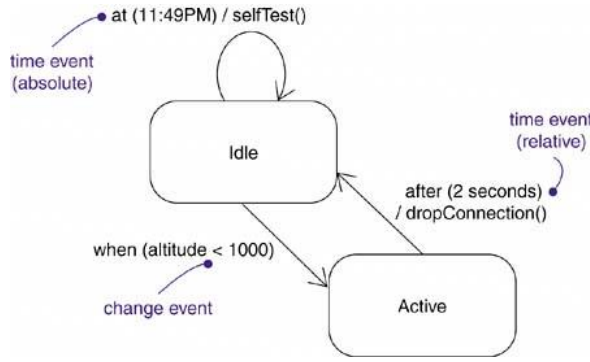
Time

- State diagrams can also include **time events:** "at" and "after"
- Suppose that our `Copy` object became overdue after two weeks and so moved into the `overdue` state. The state diagram could become:



Example: change and time events

- A nice example: looks like part of an auto-pilot design
  - From: The Unified Modeling Language User Guide, 2nd edition, Grady Booch, James Rumbaugh, Ivar Jacobson Addison Wesley, 2005



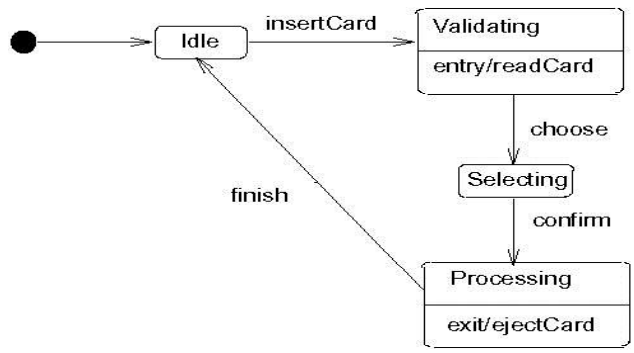
(seen at *https://docs.google.com/file/d/0B_ihlJjXUoTeYnIxbk9WckdvLVE/edit Oct 2018)*

State diagrams - Substates

- State diagrams **can become very large** - with lots of states
  - Hence difficult to read
  - This is especially true when we include error handling
- There is also the case where the same transition can occur from many different states
  - Typically from many source states to the same target
  - This can cause a large number of arrows to be required

ATM

- Consider, for instance, a possible state diagram for our **ATM** example:

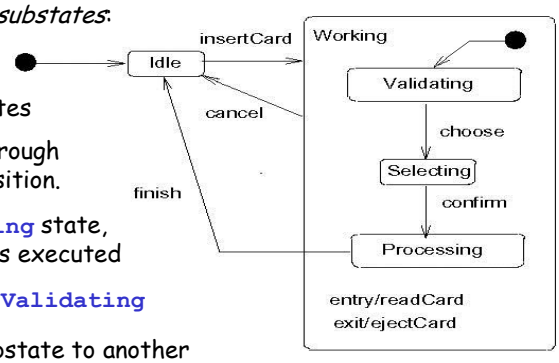


ATM

- The user may want to **cancel** a transaction at **any time**. This can be modelled by a having a `cancel` transition from the `Validation`, `Selection` and `Processing` states, i.e. an arrow from each of these states to the `Idle` state.
- In each case, we want the `ejectCard` action to be associated with the transition.
- That would greatly complicate the diagram.

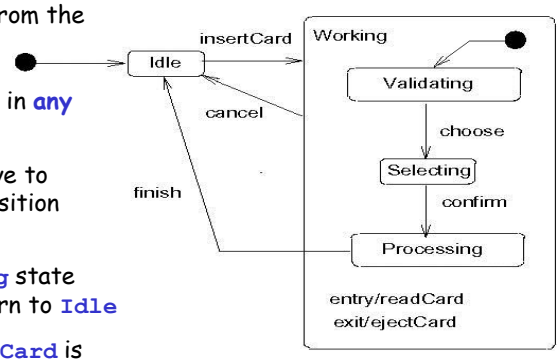
Substates

- The solution is to use *substates*:
- The `Working` state has a series of substates
- We enter `Working` through the `insertCard` transition.
- On entry to the `Working` state, the action `readCard` is executed
- The initial substate is `Validating`
- We move from one substate to another in the normal way
  - and the `finish` transition will cause us to return to the `Idle` state
- When that occurs, the action `ejectCard` is executed



Substates

- Showing a transition from the **Working** state means that it can occur when we are in **any** of its substates
- We therefore only have to show the **cancel** transition once.
- It causes the **Working** state to be left and we return to **Idle**
- The exit action **ejectCard** is executed when we leave **Working**
- It does not matter what substate we are in or which transition caused us to leave



Use of state diagrams

- As state diagrams are based on **finite state automata**, they are amenable to **mathematical reasoning**.
- It is therefore (in principle) possible to **prove** that the object has certain properties.
- For instance, considering again event driven modelling, one might wish to check whether the set of transitions defining response to external stimuli is redundant, minimal, exhaustive, …, correct, …
- In practice this can be very challenging
  - It is an on-going research topic
  - But some tools have been built

End of state diagrams section