Modelling Dynamics (1): UML Statecharts

CS3 / SEOC1

Note 10

Sequence and Collaboration diagrams show how objects interact to meet some system requirement. But they don't show how the system decides *what is the right thing to do. State Diagrams give us the means to control these decisions.*

State Diagrams

- Based on the statechart notation introduced by Harel (also called HiGraph).
- These are finite state machines (recall CS2) with some extra mechanism to capture the meaning of transitions.
- Sequence and Collaboration diagrams correspond to scenarios and are decision free. There may be many Sequence or Collaboration diagrams for one Use Case. The choice of how to react (i.e. which scenario is appropriate) is based on the state.
- Each state is like a “mode of operation” for the object we are considering.

A `Bed` Class

```
public class Bed{
   public Ward thisWard;
      Bed(Ward w){
      thisWard = w;
   }
   protected boolean isOccupied = false;
   public boolean allocate() {
      if isOccupied {
        return(false);
      } else
        isOccupied = true;
        return(thisWard.allocated(self));
   }
   public boolean discharge_patient() {
      if isOccupied {
        isOccupied = false;
        return(thisWard.freed(self));
      } else
        return(false);
}  }
```

Simple State Diagrams

- We construct a state diagram for each class
- They illustrate how the state of the class changes on receiving a message.
- *States* correspond to some property of the attributes of objects of the class. There are usually a finite number of states.
- In the class `Bed` the state diagram has two states and transitions when messages of the class are received.
- We name the states and document them.
- We label the transitions with *events* that cause changes in state. The events we consider here are receiving a message but there are others.
- We designate an initial state in the stateset with a *start marker*.

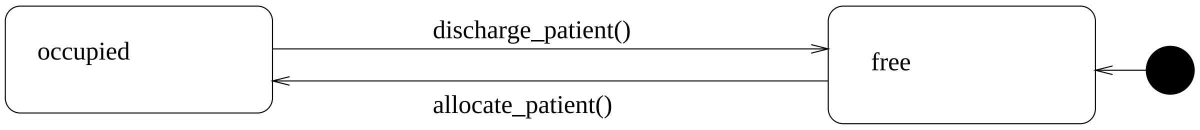


Figure 1: State diagram for `Bed` class.

- state names are chosen so as to be *informative*
- the state of class `Bed` is determined by the value of the attribute `isOccupied`
- only the *expected* state changes are shown
  - Why? Is this wise?

Extending State Diagrams

- Transitions are now labelled with event/action pairs that indicate the response corresponding to a particular event.
- We can also augment states with actions that show how an object of a class reacts to receiving a message *(entry actions)*
- ...and how an object of a class reacts on sending a message *(exit actions)*.
- Transitions can be *guarded*. This ensures that some condition must hold in order for the transitions to take place.

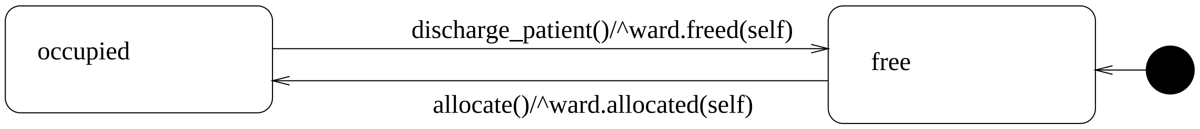

occupied
discharge_patient()/^ward.freed(self)
free
allocate()/^ward.allocated(self)

Figure 2: State diagram of class **`Bed`**, with actions.

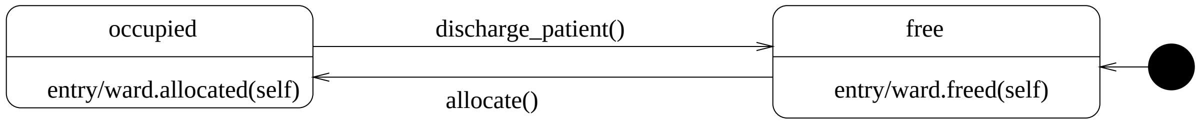


Figure 3: State diagram of class **`Bed`**, with **`entry`** actions.

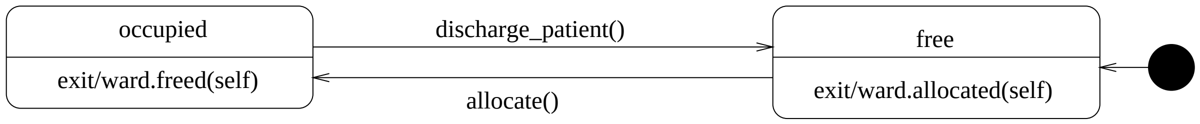


Figure 4: State diagram of class **`Bed`**, with **`exit`** actions.

A `Ward` Class

```
import java.util.*;
public class Ward{
   private String name;
   private Nurse seniorSister;
      Ward(String n, Nurse s){
      name = n;
      seniorSister = s;
   }

   protected HashSet allBeds =
               new HashSet();
   public boolean allocated(Bed b) {
         return( allBeds.remove(b) );
   }
   public boolean freed(Copy b) {
         return( allBeds.add(b) );
   }
}
```

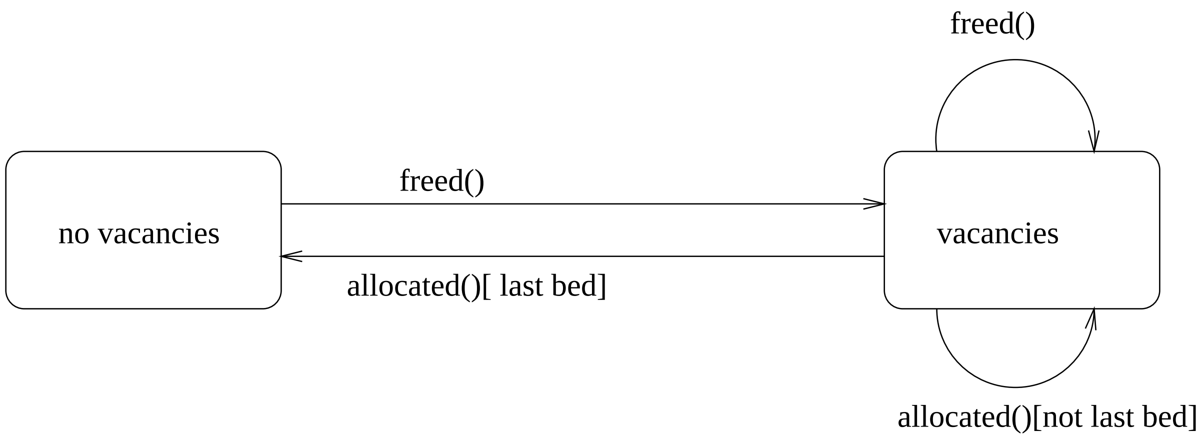


Figure 4: State diagram of class `Ward`.

- the correspondence between attributes and states is indirect
- such subtleties are one reason why state diagrams should be *documented*
- as before, only *expected* state changes are shown

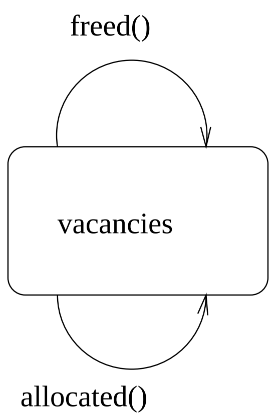


Figure 5: Alternative state diagram of class `Ward`.

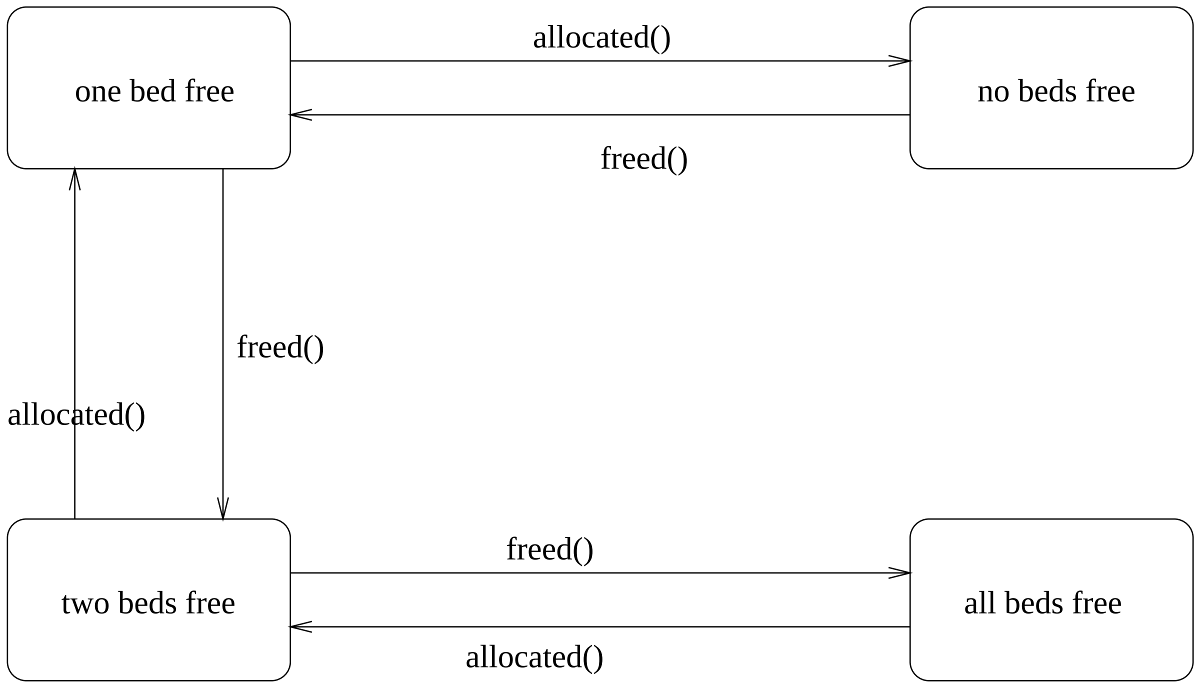


Figure 6: and another state diagram of class `Ward`.

Nested States

- the line `include/<<nested diagram name>>` indicates a compound state
- start and end markers are compulsory for compound states
- compound states enter at their start marker
- after the end marker there is an implicit “completion” event
- (thus: transitions out of a compound state *may* be unlabelled)
- **Decision diamonds:** mutually exclusive guards

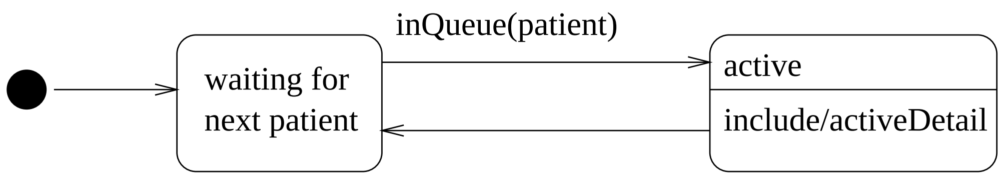


Figure 7: State diagram for class `Nurse`.

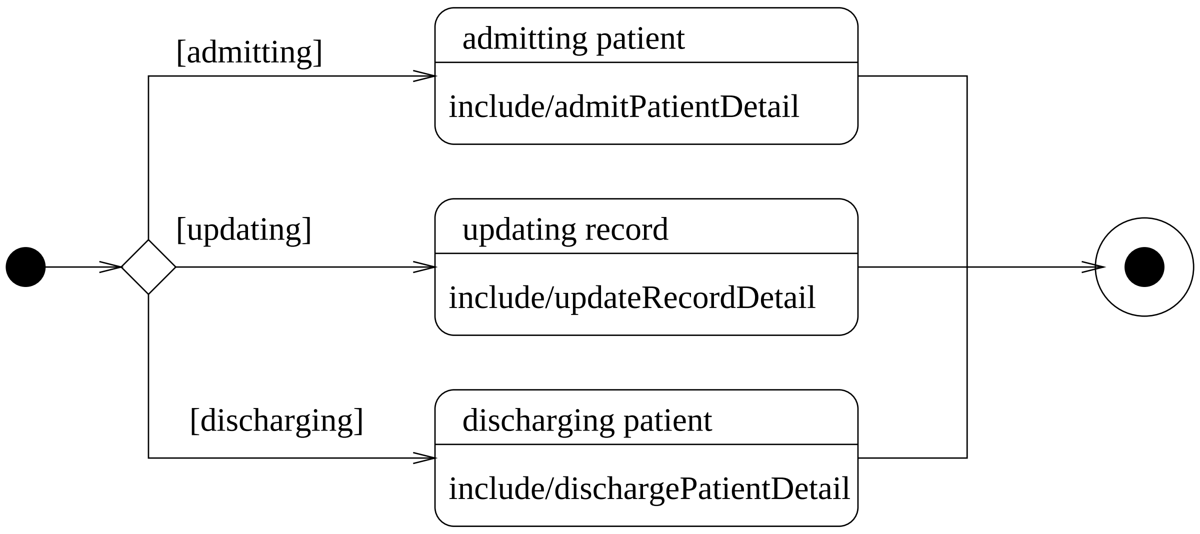
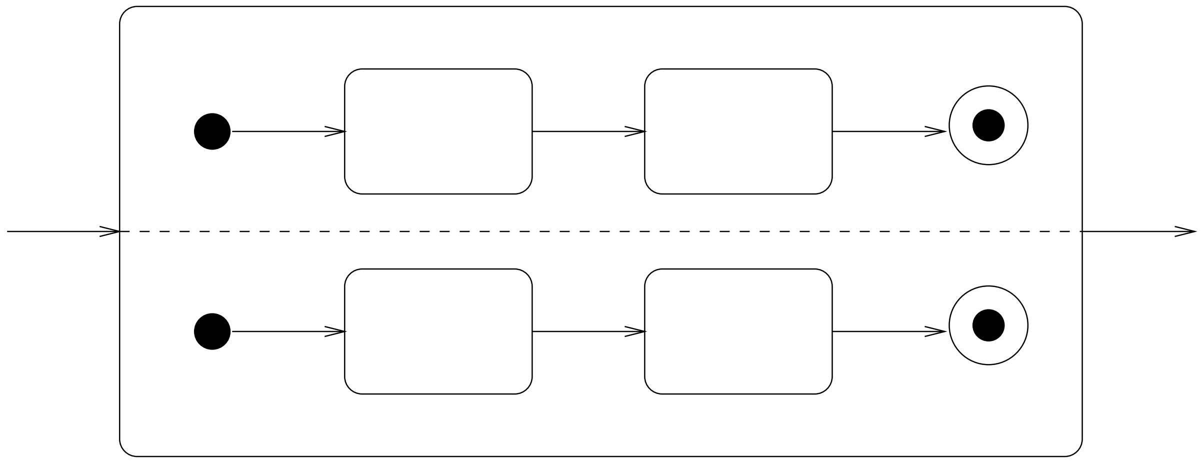


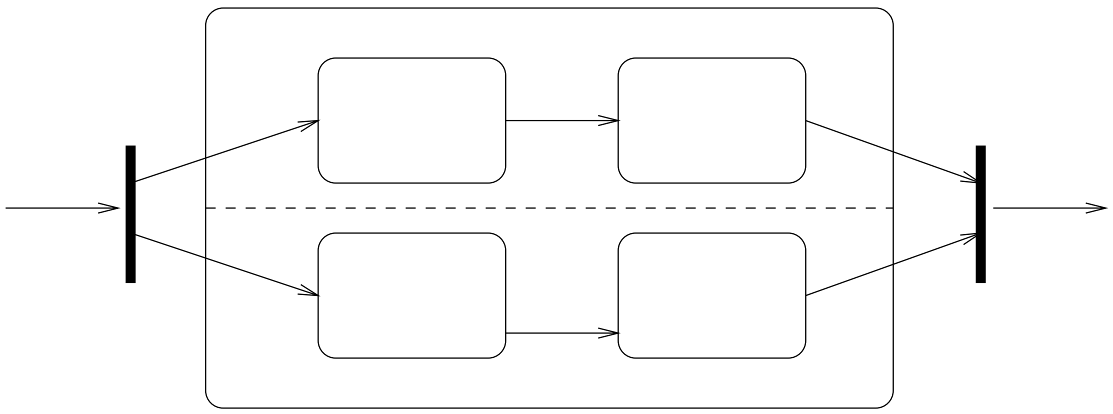
Figure 8: Nested state diagram `activeDetail` for class `Nurse`'s `active` state.

Concurrency Within States

- a nested state may containing independently executing regions
- regions marked by a horizontal dashed line
- Activity diagrams using *synchronization bars* to express joining and forking of subtasks (loose sense of concurrency)
- State diagrams may also use these
- Question: *what if we had transitions from one concurrent region to another?*



(a) State with internal concurrency



(b) Equivalent state with external synchronisation

Figure 9: State diagrams with concurrency.

Advice on Designing Classes with State Diagrams

- Keep the state diagram simple.

  State diagrams, with all of this extra notation, can very quickly become *extremely* complex and confusing. At all times, you should follow the aesthetic rule:

  Less is More

- If the state diagram gets too complex consider splitting it into smaller classes.
- Document states thoroughly
- Check consistency with the other views of the dynamics.
- Think about compound state changes in a collaboration or sequence.

Some (open) questions

- What are the benefits of having state in a system?
- What are the costs of having state in a system?
- Every state should have an edge for every message in the class – is this the right view?
- How does this description of state relate to design by contract?
- How would you check that a Java implementation was consistent with a state diagram?
- How does this differ with the treatment of state in programming languages? What does this say about the difference between modelling and programming?