Today’s Agenda

- Class Diagrams (from last class)
- Communication diagrams
- Recognizing concepts from system state diagrams
- Concept-level state diagrams
- Concept-level sequence diagrams

Example: Stopwatch/clock

Reading: none (refer to previous readings on state diagrams)
Finding Concepts

…using the system state diagram.

1. List the events, actions, activities, variables, and states found in the system state diagram.

2. Group the above into cohesive units - some of these elements (particularly states) may have to be refined into multiple units.

3. You may find that concurrent states in the system state diagram are already cohesive units that can become concepts.

4. If you have used data abstraction – these are likely to become concepts.
Finding Concepts

...using the system state diagram.

- Every action and activity must be the responsibility of a concept – possibly these will be refined and be a shared responsibility among multiple concepts.

- Every environmental input must be received by some concept. It becomes an operation of the concept.

- Every variable must be owned by some concept.

- Every state must either be the state of a concept or decomposed into the states/attributes of multiple concepts.
Concept-level State Diagrams

Every “interesting” (i.e., non-trivial concept) needs a UML state diagram.

The syntax and semantics of concept-level state diagrams are the same as for system state diagrams.

However, we will adopt different conventions for the use of the state diagrams for these two different purposes.
Concept-level State Diagrams

- Events must be:
  - Operation invocations including parameters. Events input from the environment (i.e., changes in the environment) in the system state diagram become operation invocations of an object.
  - Outputs are invocations of operations of the environment – these don’t appear as operations on classes within our system.
  - Adopt a naming convention that makes it clear which events are from the environment (e.g., ext_in..., ext_out...)
  - (external/internal) change in a condition (e.g., when(temperature > 100 degrees))
  - (temporal) the occurrence of a specific date/time or passage of time (e.g., after (20 minutes), when(date=25/12/2005))
Concept-level State Diagrams

The events of concept-level state diagrams are all operation invocations of the operations of this class.

The state diagram captures:

- When such an operation can be called – i.e., when does such an operation call have an effect (as opposed to being ignored).

- The effect of such an operation invocation, which will be the actions of the transition):
  - Setting attributes of the object
  - Calling operations of other objects
  - External outputs (calling operations outside the system)
  - Returning values to the calling object
Concept-level State Diagrams

- “Get” operations do not have to be shown for an object → their effect is that of a looping transition that returns the value of an attribute.

- “Get” operations are used by other objects wishing to get the value of an object’s attributes.

- “Set” operations for an object are shown on its state diagram only if they affect the behaviour of the object.

- “Set” operations are used by other objects wishing to set the value of an object’s attributes.

- Avoid iterators
  
  Instead, use the postconditions of an operation to indicate the results of iteration.
Concept-level State Diagrams

- No activities should appear in concept-level state diagrams!
  - All activities should be decomposed into actions (i.e., non-interruptible units).

- All actions and events should be refined into any constituent parts.

- If you have many concurrent states in a concept-level state diagram, you should consider dividing the behaviour between more objects, although this is not always necessary.
What’s a trivial class?

i.e., when do I not need to draw a concept-level state diagram?

- A concept with only set/get operations.
- A concept with state-independent behaviour – for all events of interest, the object always responds the same way.

The object reacts the same way to all operations (or external events) all the time (i.e., the state diagram would be a flower diagram).
Example

On

Show Clock

Show 12hr clock

Show 24hr clock

mode

Show Timer

but3[inState(TimerRunning)]/lapTime=timer

Show Timer

but3

ShowLapTime

but2

mode

Light

but3/light

TimerNotRunning

but3[inState(ShowTimer)]/resetTimer(timer)

TimerRunning

but2[inState(ShowTimer)]/beep

but2

but3

but2

but3
Options

We will show two possible decompositions for this system state diagram and discuss the advantages and disadvantages of each:

1. Object controls display of its own data.  
   Includes hardware and software in the system.

2. Separate display object.  
   Includes only software.
Option #1: Watch Controller

- **off**
  - ext_in_off
  - ext_in_on
  - ext_in_mode/clock.mode; timer.mode
  - ext_in_but2/clock.but2; timer.but2
  - ext_in_but3/timer.but3; show_light

- **on**
  - ext_in_mode/clock.mode; timer.mode
Option #1: Clock

Clock

Showing Clock

12hr

after(1sec)/display=12hr(time)

24hr

after(1sec)/display=24hr(time)

Not Showing Clock

mode

but2
Option #1a: Timer

![Diagram showing the execution flow of different timer states and actions. The diagram includes transitions between states such as TimerNotRunning and TimerRunning, and actions like showing the counter or lap time, and responding to user inputs with beeps and counter resets.]

- **TimerNotRunning**
  - Transition triggered by `but2[istate(ShowTimer)]/beep`
  - Transition triggered by `but3[istate(ShowTimer)]/counter=0`

- **TimerRunning**
  - Transition triggered by `after(1sec)/inc(counter)`

- **ShowingCount**
  - Transition triggered by `but3[istate(ShowTimer)]/display=counter`

- **ShowingLapTime**
  - Transition triggered by `but3[istate(TimerRunning)]/lapTime=counter`

- **ShowingClock**
  - Transition triggered by `mode`

- **ShowTimer**
  - Action `entry/display=counter;`
Option #1b: Timer
Option #1

Light and Sound classes are trivial. Their behaviour does not depend on state (i.e., they always react the same way).

We do not have to draw a state diagram for these classes.

However in the section on this class in the SRS, we should show any preconditions, postconditions, and invariants of the operations in this class.
Option #2

Option #2

Timer
- counter : TIME_UNIT
- laptime : TIME_UNIT
- display : TIME_UNIT
- start_timer()
- stop_timer()
- reset_timer()

Clock
- time : TIME_UNIT

Mode
- ext_in_mode()
- ext_in_but2()
- ext_in_but3()
- ext_in_on()
- ext_in_off()

Display
- digits : TIME_UNIT

Light
- ext_out_light()

Sound
- ext_out_bEEP()

WatchUser
- make_sound
- makes_light

<<actor>>
Option #2: Mode

On

Clock

12hr clock

after(1sec)/display.set_digits(12hr(clock.get_time()))

24hr clock

after(1sec)/display.set_digits(24hr(clock.get_time()))

Stopwatch

after(1sec)/display.set_digits(timer.get_display())

ext_in_mode/timer.but2()

ext_in_mode/timer.but3()

Light

ext_in_mode/timer.but3/ext_out_show_light()
Option #2: Timer

Diagram:

- **Timer**
  - **ShowTimer**
    - action: after(1sec)/display=counter
  - **ShowLapTime**
    - action: but3[inState(TimerRunning)]/laptime=counter; display=laptime
  - **TimerNotRunning**
    - action: but2[inState(ShowTimer)]/ext_out_bEEP()
    - action: but3[inState(ShowTimer)]/counter=0
  - **TimerRunning**
    - action: after(1sec)/inc(counter)
The remaining objects (clock, display) are trivial, and do not require a state diagram.

**clock** – just has to update the clock every 1 second (essentially it has one looping transition. This behaviour can be described in a table.

**display** – just has a set operation – does not require any further description.
Operation Calls

There will likely be many objects in a class. How can we send a message (invoke an operation) to a particular object (i.e., instance of the class)?

```
p := Patron.create(patronName);
```

creates an object. Then we can do:

```
p.checkfine();
```
Operation Calls

Alternatively, we will assume that we can find an object with certain attribute values using a “find” operation.

\[ q := \text{Patron.find}(\text{patronName}); \]

The attributes provides must uniquely identify an object.

(In code, this would be a class-scope operation, but we will assume that such an operation exists for all objects.)
Return Values?

Operations sometimes need to return values.

The behaviour of get operations or create operations is obvious and does not need to be specified in a concept-level state diagram.

If calculating a return values takes several transitions in a concept-level state machine, you may assume that these are completed in time for the calling object to consider them to happen all at once.
Multiple Inputs

If an object receives an operation call while it is responding to another operation call or receives multiple calls at once, what happens?

You could:

- be able to process multiple inputs at once using concurrency
- use the synchrony hypothesis – i.e., assume the above won’t happen
- use the single input assumption – i.e., assume multiple inputs won’t happen at the same time
- assume each object has a queue of inputs – every input is put in the queue and in each step, the object responds to the next input in the queue.
Outputs in UML 2.0

In UML 2.0, they have adopted an SDL symbol to show an output (called a signal):

beep

As this is quite cumbersome, we will instead use the previously mentioned naming conventions to indicate outputs.
State Diagrams

- The system state diagram shows the behaviour of the entire system and its interaction with the environment.

- At the concept-level, there is one state diagram per class (describes the possible behaviours for each instance of the class). We don’t need state diagrams for the classes with simple behaviour.

- Concept-level state diagrams for all the objects of the system together describe how the system reacts to inputs from its environment (users, etc.).
Concept-level Sequence Diagrams

For each system state diagram in your SRS, show how the concepts work together to accomplish this behaviour in a concept-level sequence diagram.

There should be one lifeline for each participating object.

Communication diagrams can also be used.
Summary

- Class Diagrams (from last class)
- Communication diagrams
- Recognizing concepts from system state diagrams
- Concept-level state diagrams
- Concept-level sequence diagrams

Next Lecture: User interface requirements

Reading: none