Simulation

Object based simulation

Process based simulation

- Logically related events are collected to a single life cycle (instead of separate event routines)
  - Easier to manage subprocesses or -entities
- Several concurrent life cycles have to be managed
- Same life cycle/process may have several instances running simultaneously

Client process

- In wash machine example each client has a clear life cycle.
- Example can be modelled with one life cycle that is copied for each client.
- How to manage concurrent processes if this is not supported by the programming language?

Client process

- Life cycle is divided to phases (one event per phase), that can be refered to and stored for each instance of the process.
- Event list refers to the process instance and the phase (and time).
- Simulation main routine
  - Reads the event list.
  - Calls for a process instance to execute a given phase.

Client process

```
ClientProcess(me, Phase) //"me" stands for client in question
CASE Phase
Arrival {
  Car = new Client // Calls for next client
  Schedule(Car,"Arrival", ArrivalTimeDistribution())
  If (!Service.IsFull()) //if place in queue available
   me.NextPhase="Start"
   Service.Add(me)
   Service.Reserve()
  Else          // Lost client
   me.NextPhase="Departure"   }
Start  {
  Schedule(me,"End",ServiceTimeDistribution())
       }
```

Client process

```
End    {
  Service.Release()
  Service.Reserve()
  Schedule (me,”Departure”,0.)
       }
Departure {
  // Collect statistics
  me.Remove //destructor
           }
  ENDCASE
```

Service

```
//Service has methods like Add, TakeNext, Length,
   and IsFull for internal queue,  and
   Reserve/Release
Reserve()
ClientType :: Car
    {
 If(Free and Length()>0) {
   Car=TakeNext() //gives next from the queue
   Free = false
   Schedule(Car,Car.NextPhase,0.) //Start
   }
}
```

Analysis

- Traditional (i.e. non-object) languages require separate actions
  - To communicate the phase of execution
  - To communicate internal variables
  - To divide life cycle to explicit phases
  - To build conditional life cycles
- Programming is easier if these are inherent in the process instance -> Object

Object simulation

- Objects were invented to encapsulate process instances (SIMULA, 1967).
- Inheritance was needed to hide the control structures related to concurrent processes (threads).
- Common terms and methods for process phases and mutual communication.

States of process object

- Four possible states
  - Active (currently executed)
  - Scheduled
    - Event list has reference to future activation of the object
  - Passive (no future event scheduled)
    - Some other object has to schedule/activate this
  - Terminated
    - Can not be activated by any means

State changes

- Only active object can make state changes
  - To itself
    - Passivate (waits until some other activates it)
    - Hold (waits for given time)
    - Terminate (if the life cycle is over)
  - To others
    - Activate (wakes up a passive object (now or later))
    - Cancel (cancel scheduled activation)
    - Terminate (removes the entire process)

Example

- Wash machine can be modeled in many ways
- Different divisions to active objects (with own life cycle) and other entities (classes with methods for process objects to use).
  - Active clients, passive service resource and queue
  - Passive client and queue, active service with life cycle

Client life cycle

```
Client Car
Service Q
Car = new Client
Car.Activate(ArrivalTimeDistribution()) // next one
If (!Q.IsFull())
  Q.Reserve(*this) // reserve service after evt
                   // queuing up
  Hold(ServiceTimeDistribution()) // control shifts
  Q.Release()
// Collect statistics
Terminate // client dies and control shifts away
```

Service

```
Setup()         // Initialize empty queue etc

Reserve(Client Car)
If Free
   Free=false
else
   Queue.Add(Car)    // Wait in queue if service not free
   Car.Passivate()   // Shift control away

Release()
If(Length >0)
    Car = Queue.Next()
    Car.Activate(0.)   // Activate to use the service
else
   Free=true          // Service is set idle
```

”Main”

```
Q = New Service
Q.Setup
Car = New Client
Car.Activate(ArrivalTimeDistribution())
Hold(DurationOfSimulation)
// Report the results
// Terminate the clients at queue, remove queue
// Shift control to the actual main thread
```

- ”Main”, controller, is a process object with simulation process methods
- Is created in the actual main-thread

Analysis

- Concurrent processes needed
  - Use (of threads) is in the background
  - Simulation classes are inherited from the thread classes of the programming language
  - Cf class libraries of JavaSim and C++Sim
- Example does not work in practice
    - Dynamic clients create new clients
    - When first client-thread dies, the others get unstable
    - ”Permanent ” client-generator is needed
    - Also reserving services may need elaboration

Service based model

- Example can be modeled with two process instances
  - Client generator
  - Service process
- Clients and queue as ordinary classes (no life cycle/simulation methods)
- Modification of JavaSim ”Basic” example

Container harbor

- Several possible strategies to model the situation
  - Ships can be active processes or passive load containing only routing information
  - Harbor can be a collection of active services (docks), a single service with given capacity or a passive resource (with given capacity)
  - Dock can be an active service or passive resource

Container harbor

- Each strategy has its own pitfalls
  - How to manage passive ship to right harbor at right time
  - If harbor is active and dock a passive resource, where is the ship when it is unloaded
  - Even fully passive harbor-dock needs own structures (queues, capacity management)
- Common higher level constructs are useful

Higher level constructs

- Semaphore/resource
  - Construct for a critical reservable resource that enables queuing for it
    - Given (fixed) capacity
    - Internal queue for demands
    - Methods for reserving and releasing the capacity
    - Blocks the reserving process if capacity is not available, activates the (next) waiting process when capacity is released

Higher level constructs

- Bin/Stock
  - For storing things between two processes (provider/user)
    - Internal storage for things
    - Internal queue for users waiting (when storage empty)
    - Internal queue for providers (if stock with finite capacity) waiting at full stock

Higher level constructs

- Wait queues
  - Needed to handle asynchronous events
  - Queues for processes that can <u>wait</u> for some condition to become true
    - Certain time/event
    - Ending of a certain process
    - Some other <u>trigger</u>

Passive harbor

- Consider active ships (ProcessObject) and passive harbor resources
  - Harbor H as a semaphore/resource with capacity with two methods
    - H.Get (ProcessObject) and H.Release()
    - Get
      - enqueues ProcessObject internally
      - reserves the resource
      - passivates ProcessObject if resource not available
    - Release frees the resource and activates the next waiting ProcessObject

Harbor as semaphore

- Flow of a ship through sequence of harbours H[]
  - Assume H as JavaSim Semaphore

```
For (int i=0; i< N; i++)
      {
      Hold(traveltime(i));// travel to next harbour
      H[i].Get(this); //get the resource
      Hold(servicetime(i)); // actual unloading
      H[i].Release(); release the resource
      }
```

Harbor as resource

– Assume ship has methods Get and Release

```
For (int i=0; i< N; i++)
      {
      Hold(traveltime(i));// travel to next harbour
      Get(H[i]); //get the resource
      Hold(servicetime(i)); // actual unloading
      Release(H[i]); release the resource
      }
```

- Get passivates the process if H[i] is not available, Release activates the next in line

Harbor as resource

- Results to rather simple model structure
  - The flow of events is all in the client (ship) life cycle
    - Client based data collection is easy to arrange
    - Monitoring the resources requires extra work
    - The whole flow of events (with all variants) is to be modeled explicitly
    - Hierarchical subtasks/systems not available

Real examples

- Most simulation models have several components
  - Many services, queues, client streams
  - Life cycle of a single component is relatively easy to manage (class with parameters)
  - Mutual interactions between components have to modelled (routing tables and diagrams, visualisation, graphical editor)
  - In practice graphical classes are needed also

Links

- JavaSim
  - Essentially the SIMULA environment as an open Java implementation
  - Course examples mainly for newest version
    - https://github.com/nmcl/JavaSim
- Desmo-J
  - More elaborate Java-based environment containing event and object based approaches
    - http://desmoj.sourceforge.net/home.html
- SimPy (http://simpy.readthedocs.org/en/latest/index.html)
  - Python package of simulation objects but not using Simula-based terminology

Links

- JaamSim
  - Discrete event simulation environment with graphical user interface
  - Allows both event and process based modelling (+ drag and drop model building)
  - http://jaamsim.com/index.html
    - See >Downloads >programmer manual for the basic internal constructs
    - Makes no explicit use of SIMULA type constructs