

# CPE 345: Modeling and Simulation

## Lecture 3

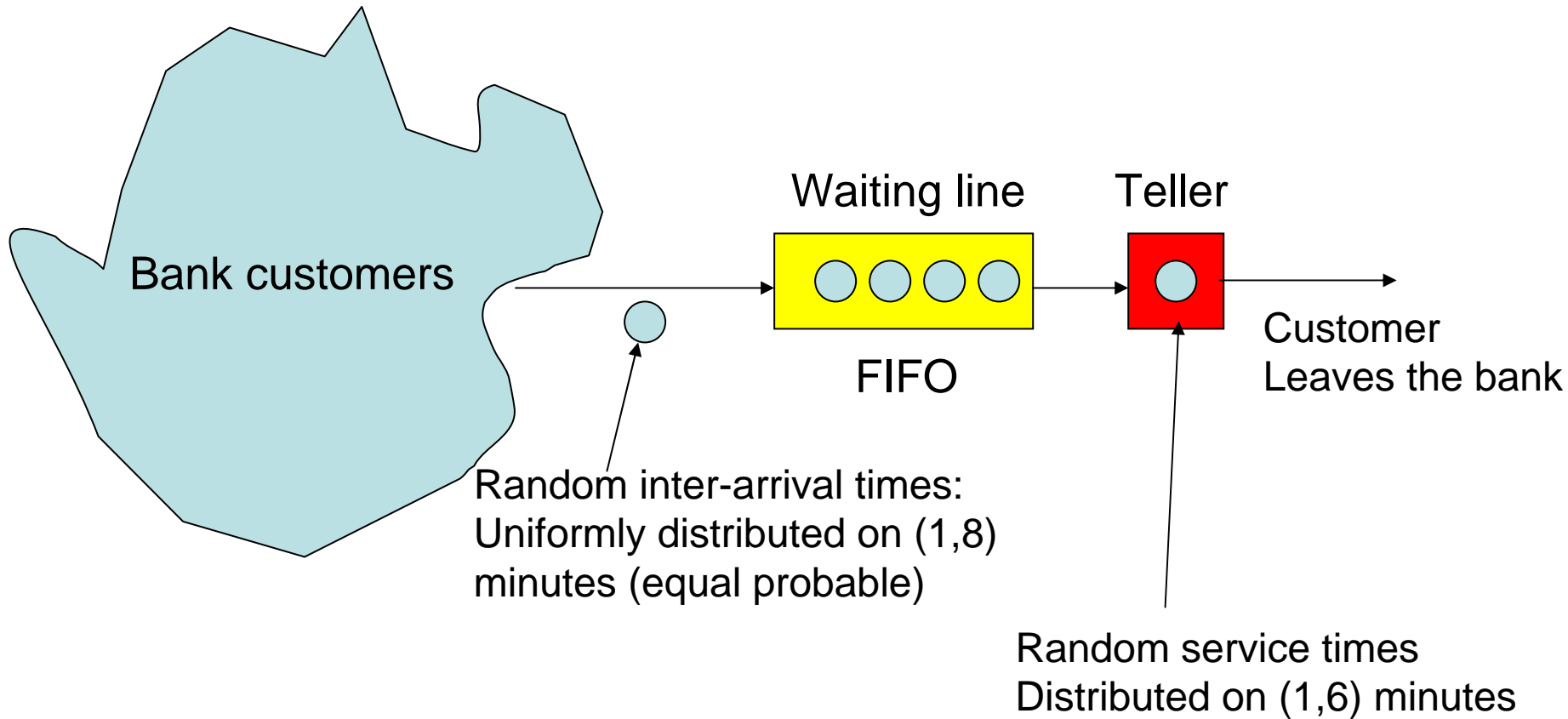
# Today's topics

- Discrete event simulation
  - Updated grading policy
  - General principles (chapter 3)
  - Implementing a FIFO queue in OMNET++:  
Introduction to OMNET++ programming

# Updated grading policy

- Homework: 20%
- Midterm 40%
- Final (project) 40%
- Make-up final (final project → 10%, make-up final 30%)
  
- Final (project): 10 minutes presentation in class by the project leader (last day of classes) and report submission.

# Recall our simple bank teller example



Objective: simulate arrivals and service for 20 customers

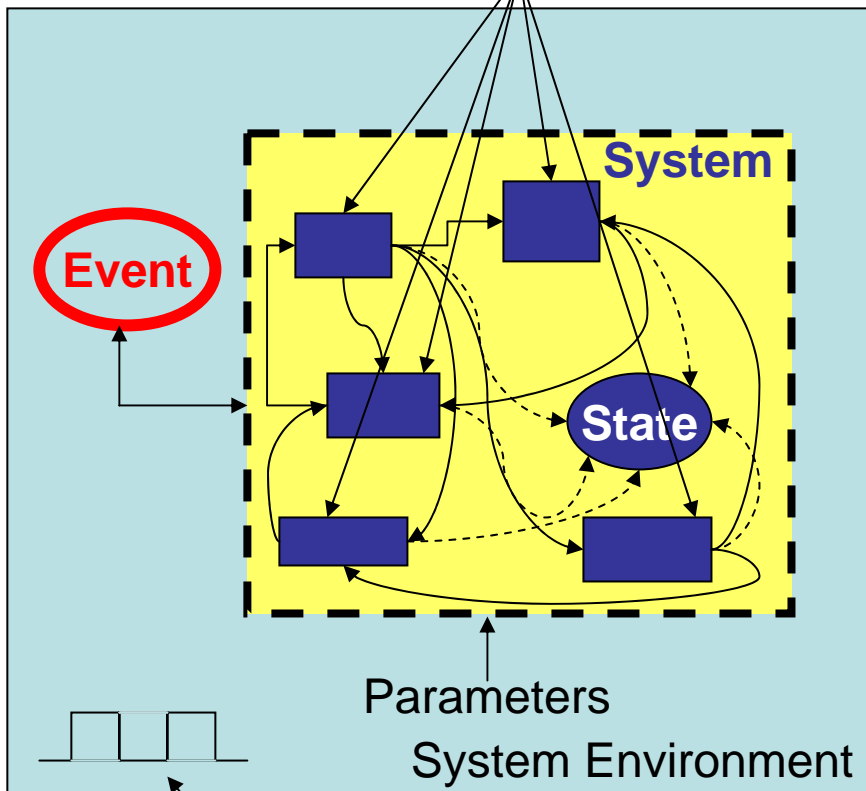
# Chronological ordering of events

Event Type	Customer number	Clock time
Arrival	1	0
Departure	1	2
Arrival	2	2
Departure	2	3
Arrival	3	6
Arrival	4	7
Departure	3	9
Arrival	5	9
Departure	4	11
Departure	5	12
Arrival	6	15
Departure	6	19
.....	.....	.....

What do we need to advance the simulation time and guarantee that all events are executed in correct chronological order ?

# Concepts in Discrete-Event Simulation

## Entities



We have defined:

**Entity**  
**Attribute**  
**Activity**  
**State of system**  
**Event**  
**System Environment**

We measure **delay**:

- a duration of time of unspecified length, which is not known until it ends

Clock: a system variable representing simulated time

**Event notice:** Record of an event to occur at some present or future time, along with the associated data

**Event list:** List of event notices (Future Event List: FEL)

**List:** a collection of associated entities, ordered in some logical fashion  
- e.g. Queues (order can be FIFO, LIFO, etc)

# The two bank tellers example: revisited

- System state:
  - $Q(t)$  – number of customers waiting in the queue at time  $t$
  - $S_1(t)$  – state of teller 1: busy/idle
  - $S_2(t)$  – state of teller 2: busy/idle

- Entities

- Teller 1
- Teller 2
- Arrivals generator (it is actually external to the system)

- Activities

- Teller 1 service time
- Teller 2 service time
- Inter-arrival time

the duration of the activity not affected by the occurrence of other events; it is based on the modeler's specifications: **unconditional wait**

- Delay

- Customer waiting time until service started or completed

Determined by the system conditions (not specified by the modeler ahead of time): **conditional wait**

# The two bank tellers example: events and discrete event simulation

- Events are determined by the completion of an activity – **primary events: place event notice in FEL**
  - Service completion teller1
  - Service completion teller2
  - Inter-arrival time ends: new arrival occurs
- Delay completion – **secondary events – not represented in FEL**
- **Other events:** end of simulation
  - Run for a preset time (e.g. 1000 seconds of simulator time)
  - Run for a preset number of events (e.g. 1000 customers have been served)
  - Run until an specific internal event occurs (e.g. until convergence of some simulated algorithm)
  - Run until an external event occurs (e.g. stop by the user to modify experiment or to collect data)
- **Discrete event simulation:** produces a sequence of **system snapshots** (system images) which represent the evolution of the system through time.

# System snapshot example

Clock	System state	Entities and Attributes	Set1	Set2	FEL	Cumulative statistics and counters
t	(x,y,z,...)		Queue membership		(3,t1) (1,t2)	
t1	(x-1,y,z,...)				(1,t2)	
....						

For our example:

state variable x (e.g. customers waiting)

y (e.g. teller 1 busy)

z = (e.g. teller 2 busy)

Type 1 event: arrival

Type 2 event: departure teller1

Type 3 event: departure teller 2

Future event list: {(3,t1),(1,t2)}:

type 3 event will occur at t=t1 (e.g. teller 2 service completion at t1)

type 1 event will occur at t=t2 (e.g. customer arrival at t2)

# Event-scheduling/time advance algorithm

<b>Clock</b>	<b>System state</b>	<b>.....</b>	<b>FEL</b>	<b>.....</b>
<b>t</b>	<b>(5,1,1)</b>		<b>(3,t1)</b> <b>(1,t2)</b> <b>(1,t3)</b> <b>...</b> <b>(2, tn)</b>	

1. Remove event notice for imminent event (at  $t=t_1$ )
2. Advance clock to imminent event time
3. Execute imminent event
  - update system state, change entity attributes, set membership (as needed)
4. Generate future events (if needed) and place them in FEL, ranked by time of occurrence
5. Update cumulative statistics and counters

<b>Clock</b>	<b>System state</b>		<b>FEL</b>	
<b>t1</b>	<b>(4,1,1)</b>		<b>(1,t2)</b> <b>(3,t*)</b> <b>(1,t3)</b> <b>...</b> <b>(2, tn)</b>	

# FEL implementation

- Actions:
  - Delete imminent event
  - Removal of some events (cancellation for events)
    - e.g. in wireless scenario, calls may be dropped, then, the end of service event for the dropped call must be cancelled
  - Addition of new events
    - e.g. after arrival of customer, new future arrival is scheduled
  - **Maintain time ordering of events**
    - If unordered list, a complete search for the imminent event must be done before each clock advance – very inefficient
- Length and contents of FEL changes dynamically during simulation
- Efficient list processing and dynamic memory allocation is necessary (optional supplementary reading: section 3.2, pp. 85-92)

# Generating events

- At time 0, the system snapshot is defined by the initial conditions and the generation of the “exogenous” event
  - Exogenous event: is happening outside the system, but impinges on the system: e.g. arrival process at a queue
    - Note: in our OMNET++ simulator, the arrival generator will be considered as an entity in the system
  - At time 0, an arrival event is scheduled
- New event generation
  - Arrivals: bootstrapping – create external event stream
    - When the clock is advanced to the time of current arrival ( $t_1$ ), a future arrival event is generated: inter-arrival time is determined:  $a^*$ , and then time of next arrival is  $t^* = t_1 + a^*$ . The arrival notice is then registered in FEL:  $(1, t^*)$ .
  - Departures
    - When one customer completes service at current clock time  $t_2$  (e.g. for teller 1), if a customer waits for service, a new service time is generated for the new customer,  $s^*$ , and the new departure event is scheduled and registered in the FEL:  $(2, t^{**})$ ,  $t^{**} = t_2 + s^*$ .

Customer	Time between arrivals (min)		Customer	Customer	Service time (min)		Customer	Service time (min)
1	-		11	1	4		11	3
2	8		12	2	1		12	5
3	6		13	3	4		13	4
4	1		14	4	3		14	1
5	8		15	5	2		15	5
6	3		16	6	4		16	4
7	8		17	7	5		17	3
8	7		18	8	4		18	3
9	2		19	9	5		19	2
10	3		20	10	3		20	3

<b>t = 0</b>	<b>0</b>	<b>-</b>	<b>(0,1)</b>	<b>(1,8)</b> ↷ <b>(2,4)</b> <b>(2,4)</b> ↶ <b>(1,8)</b>
<b>t = 4</b>		<b>4</b>	<b>(0,0)</b>	<b>(1,8)</b>
<b>t = 8</b>	<b>8</b>		<b>(0,1)</b>	<b>(1, 14)</b> ↷ <b>(2,9)</b> <b>(2,9)</b> ↶ <b>(1,14)</b>
<b>t = 9</b>		<b>9</b>	<b>(0,0)</b>	<b>(1,14)</b>
<b>t = 14</b>	<b>14</b>		<b>(0,1)</b>	<b>(1, 15)</b> <b>(2,18)</b>

Clock	Arrival Time	Departure time	System state	FEL
t = 14	14	-	(0,1)	(1,15) (2,18)
t =15	15	-	(1,1)	(2,18) (1, 23)
.....				

Customer	Time between arrivals (min)		Customer	Time between arrivals (min)
1	-		11	1
2	8		12	1
3	6		13	5
4	1		14	6
5	8		15	3
6	3		16	8
7	8		17	1
8	7		18	2
9	2		19	4
10	3		20	5

# World Views

System model for different simulation packages is developed differently, depending on the “world view” employed

- **Event scheduling**
  - Focus on events and their effect on system state
- **Process-interaction**
  - The modeler thinks in terms of processes
  - A process is the life cycle of one entity in simulation, and consists of various events and activities
  - Intuitive appeal: analyst can describe the process flow in terms of high-level blocks, while the interaction between processes is handled automatically
  - Usually, many processes are simultaneously active, and the interaction among them may be quite complex
  - Suitable for object oriented programming languages (e.g. C++)
    - Entities and processes = objects

# World views- continuation

- Activity scanning

- Simulation runs in fixed time increments
- At each clock advance, conditions are checked, and if met, the activity can begin
- Repeated scanning for activity conditions: slow down for the computer simulation

- Hybrid activity-scanning/event-scheduling approach

- Two types of activities:
  - B activities
    - Primary events and unconditional activities
    - Can be scheduled in advance
  - C activities
    - Conditional events or activities
    - Cannot be scheduled in advance: depend on conditions being true
    - Scan for them at the end of a time advance

# Improved activity scanning: three phase approach

## Phase A

**Remove the imminent event from the FEL**  
**Advance the clock to its event type**  
**Remove all other events in FEL with the same event time**

## Phase B

**Execute all B-type events removed from FEL**

## Phase C

**Check conditions that trigger conditional activity**  
**Activate any C-type activity for which the conditions were met**

# OMNET++ world view

- Supports both **event scheduling** and **process interaction**
- Last class, we have discussed how to describe the network using the NED language
  - Fifo1 example:
    - The simple module definitions (e.g. FF1Generator, FF1PacketFifo, FF1BitFifo, FF1Sink), as well as the activity of the modules are implemented in C++ files (e.g. gen1.cpp, fifo1.cpp, sink1.cpp)
    - What kind of functionality should be provided by the C++ implementation of the modules?
      - Depends on the world view adopted:
        - » fifo1 – process based implementation
        - » fifo2 – event handling approach

# Implementation of a simple module

- Events occur inside simple modules
- User creates simple module classes by sub-classing the `cSimpleModule` class (part of OMNET++ library)
- Redefine some virtual functions:
  - `void initialize()`
  - `void activity()` - process interaction approach
  - `void handleMessage (cMessage *msg)` – handle events approach
  - `void finish()`

**Note: Modules written with `activity()` and `handleMessage(cMessage *msg)` can be freely mixed within a simulation model**

# Implementation of a simple module – cont.

- Declaration of the module class (subclassed from `cSimpleModule` )

**Example: `gen1.h` in `fifo1` sample program**

```
class FF1Generator : public cSimpleModule
```

```
{
```

```
    Module_Class_Members(FF1Generator,cSimpleModule,16384)
```

```
    virtual void activity();
```

```
    virtual void finish();
```

```
};
```

stack size



**Note: if `stack size = 0`, the simulation kernel interprets that the module uses `handleMessage()` function for event processing!**

- A module type registration (`Define_Module()` or `Define_Module_Like()` macro) – Example `gen1.cpp` in `fifo1` sample program

```
Define_Module( FF1Generator );
```

- Announces that you are using the class as a simple module type
- Associates the module class with an interface declared in NED

# Implementation of a simple module – cont.

- Implementation of the module class

```
void FF1Generator::activity()
{
    int num_messages = par("num_messages");
    cPar& ia_time = par("ia_time");
    cPar& msg_length = par("msg_length");
    for (int i=0; i<num_messages; i++)
    {
        char msgname[32];
        sprintf( msgname, "job-%d", i);
        ev << "Generating " << msgname << endl;
        cMessage *msg = new cMessage(
msgname );
        msg->setLength( (long) msg_length );
        msg->setTimestamp();
        send( msg, "out" );
        wait( (double) ia_time );
    }
}
```

```
void FF1Generator::finish()
{
    ev << "*** Module: " << fullPath()
<< "***" << endl;
    ev << "Stack allocated: " <<
stackSize() << " bytes";
    ev << " (includes " <<
ev.extraStackForEnvir() << " bytes
for environment)" << endl;
    ev << "Stack actually used: " <<
stackUsage() << " bytes" << endl;
}
```

# Activity() function

- Most important functions
  - receive()* – to receive messages (events)
  - wait()* – to suspend execution for some time (model time)
  - send()* family of functions – to send messages to other modules
  - scheduleAt()* – to schedule an event (module sends a message to itself)
  - cancelEvent()* – to delete event scheduled with *scheduleAt*
  - end()* – finish execution of this module
- *activity()* – contains an infinite loop, with at least a *wait()* or *receive()* call in its body
- *wait()* and *receive()* functions are special points in the activity function, because this is where
  - Simulation time elapses in the module
  - Other modules get a chance to execute

**Note:** modules written with *activity()* need starter messages to “boot”. These are inserted automatically into FEL by OMNET++ at the beginning of the simulation, even before the *initialize()* functions are called.

# initialize() and finish() with activity()

- Local variables of activity() are preserved across events (saved in the stack – 16Kb: usually a good choice)
  - Can be initialized at the top of activity() function
  - There is not so much need to use initialize()
- finish() – useful if you want to collect statistics
  - Cannot access the local variables for activity()
  - Must put the objects and the variables that contain the statistics into the module class
    - Still don't need initialize, since class members can be initialized at the top of the activity() function as well

# handleMessage() function

- is called for each event (message) that arrives at the module
- The function processes the message and returns immediately after that (passes the control back to the simulation kernel)
- No simulation time elapses with a call to handleMessage()
- Modules with handleMessage() are not started automatically
  - Need to schedule self-messages from the initialize() function, if the module needs to be started without first receiving messages from other modules
- The local variables of handleMessage() are destroyed once the function returns – no information can be preserved using its local variables
  - Data must be added to the module class and initialize using initialize()

**Note: you cannot use receive() and wait() functions in handleMessage()**

Example: fifo2 sample program

# Advantages/drawbacks activity() versus handleMessage()

- Advantages
  - Initialize() not needed
  - Process-style description – natural programming model
- Disadvantages
  - Memory overhead
  - Run-time overhead – switching between co-routines is slower than a simple function call
- When is handleMessage() a better choice?
  - For large simulations (several thousand modules)
    - Memory stack required by activity() – too much memory
  - Modules that maintain little or no state information (e.g. packet sinks)
  - Modules with a large state space and many arbitrary state transition possibilities (e.g. most communication protocols)

# initialize() and finish()

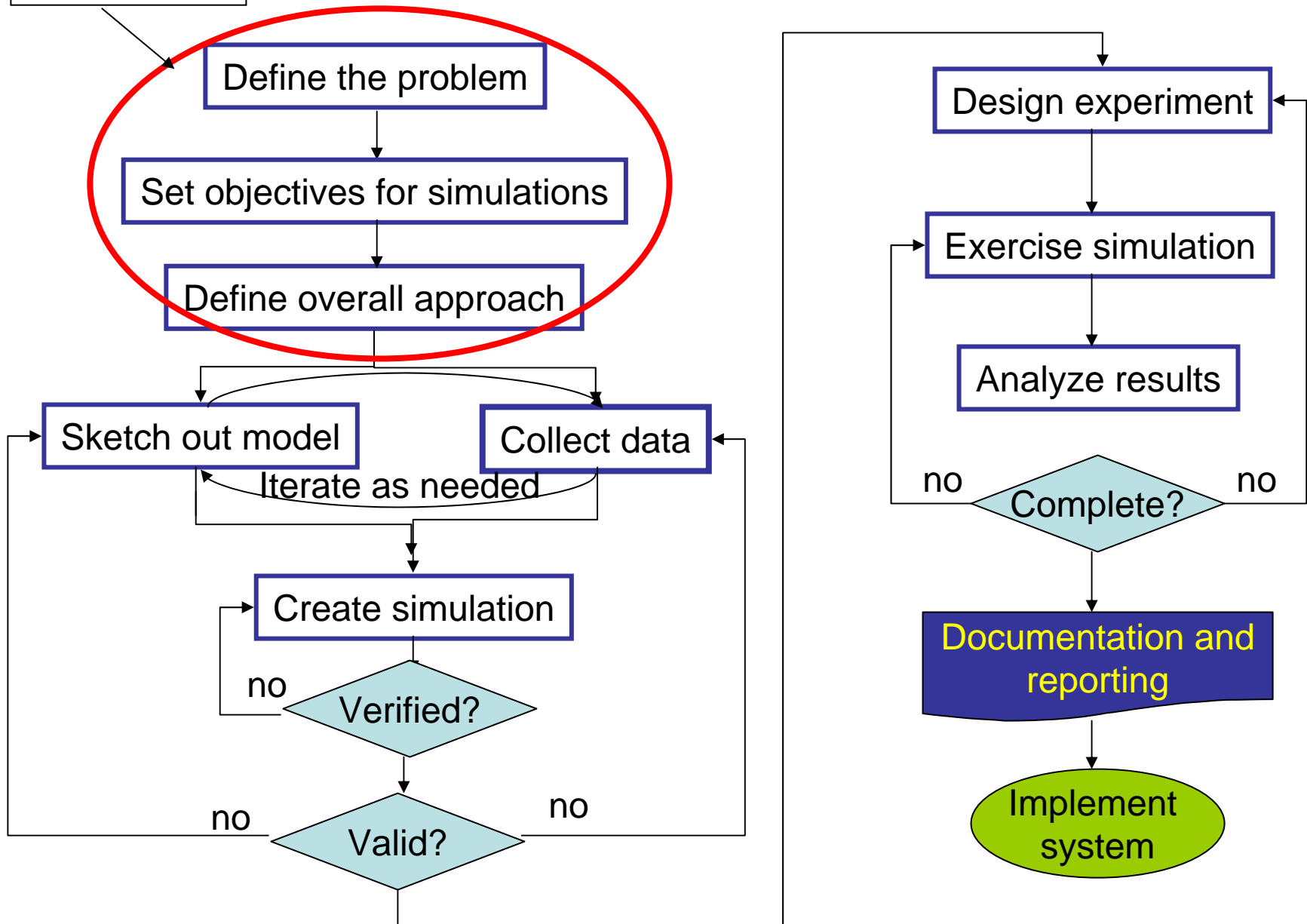
- initialize ()
  - provides place for any user setup code
  - simulation related code should not be put in constructor, some information about the network may be needed
  - invoked before the first event is processed, but after the initial events (starter messages) have been placed in the FEL by the simulation kernel
- finish()
  - records statistics after the simulation is complete
  - Only if the simulation ended normally
  - It is not always called – not a good place for cleanup code
  - Cleanup code should go into the destructor
  - In general, the cleanup is automatically implemented by OMNET++, no need for destructor implementation

# First phase project – due 2 weeks from now

- Choose your simulation topic (a list of topics will be provided)
- Formulate a simulation plan and present a proposal (1-2 pages)
  - Project title
  - What will you study
  - Objectives – *questions to be answered by your simulation*
  - Overall approach – *alternative solutions/algorithms to be studied*
  - Data gathering (observe physical system, or use references for modeling)
  - Project group members (3-5 members) – about 11 groups expected

We are here for the first step of the project

# Steps in a simulation study



# Homework –due next week in class

- Create a system snap shot table (using the event-scheduling/time advance algorithm) for the two-teller example we have discussed in lecture 2, based on the manual simulation table presented in class
- Implement using OMNET++, an arrivals generator for the inter-arrival distribution in the following table:

<b>Inter-arrival time (min)</b>	<b>Probability</b>	<b>Cumulative probability</b>
<b>1</b>	<b>0.20</b>	<b>0.20</b>
<b>2</b>	<b>0.10</b>	<b>0.30</b>
<b>3</b>	<b>0.20</b>	<b>0.50</b>
<b>4</b>	<b>0.35</b>	<b>0.85</b>
<b>5</b>	<b>0.10</b>	<b>0.95</b>
<b>6</b>	<b>0.05</b>	<b>1.00</b>

Determine the arrival times for the first 20 customers. Please include in your submission your listing for the code.