Programming Project #2

Beers-Bars-Drinkers

Discrete Event-Driven Simulation

An Application of Priority Queues

Due Date: Monday March 18, 2019

Discrete Event-Driven Simulation

- One of the most common applications of priority queues is in the creation of simulations.
- A discrete event-driven simulation is one popular simulation technique.
  - Objects in the simulation model represent objects in the real world and are programmed to react as much as possible as the real objects would react.
- A priority queue is used to store a representation of "events" waiting to happen.
  - This queue is ordered based on the time the event should occur, so the smallest element will be the next event to be modeled (Min Heap).
  - As an event occurs, it can spawn other events.
  - These subsequent events are placed into the queue as well.
  - Execution continues until all events have occurred or until a preset time for simulation is exceeded.

Programming Project: Discrete Event-Driven Simulation

This project introduces Object-Oriented Frameworks and features Inheritance and Polymorphism

- A framework is like the skeleton of an application
  - → It provides the structure for a solution, but none of the application-specific behavior.
- Our framework for simulations describes the features common to all discrete event-driven simulations, namely:
  - ① the concept of a priority queue of pending events, and
  - ② a loop that repeatedly removes the next event from the queue and executes it.
- But note that the framework has no knowledge of the specific simulation being constructed.

The Software Gurus Bar

- Imagine that you are thinking of opening a bar in San Mateo!
  - You need to decide how large the bar should be, how many seats you should have, and so on.
  - If you plan too small, customers will be turned away when there is insufficient space, and you will lose potential profits.
  - On the other hand, if you plan too large, most of the seats will be unused, and you will be paying useless rent on the space and hence losing profits.
  - So you need to choose approximately the right number, but how do you decide?

The Software Gurus Bar

- To create a simulation, you first examine similar operations in comparable locations and form a model that includes, among other factors,

  1) an estimation of the number of customers you can expect to arrive in any period of time,

  2) the length of time they will take to decide on an order, and

  3) the length of time they will stay after having been served.

  Based on this, you can design a simulation.

The Software Gurus Bar
"Where everybody knows your name!"

The Software Gurus Bar

- To see how we might create a simulation of our Software Gurus Bar, consider a typical scenario:

  A group of customers arrive at the bar.

  From our measurements of similar bars, we derive a probability that indicates how frequently this occurs.

  - For example, suppose that we assume that groups will consist of from one to five people, selected uniformly over that range.

    In actual simulation, the distribution would seldom be uniform.

    For example, groups of size two, three and four might predominate, with groups of one and five being less frequent.

    We will start with uniform distributions and later you will modify appropriately.

The Software Gurus Bar

- These groups will arrive at time spaced 2 to 5 minutes apart, again selected uniformly.

  Once they arrive, a group will either be seated or, seeing that there are no seats available, leave.

  If seated, they will take from 2 to 10 minutes to order and then will remain from 30 to 60 minutes in the bar.

- We know that every customer will order from three kinds of beer:
  - local beer, import beer, or special beer.

    The bar makes a profit of

    $2 on each local beer,

    $3 on each imported beer and

    $4 on each special beer.

Object of Simulation – the Bar

- The primary object in the simulation is the bar itself.

  It might seem odd to provide "behavior" for this inanimate object, but we can think of the bar as a useful abstraction for the servers and managers who work in the bar.

- The bar manages two data items:
  - the number of available seats and
  - the amount of profit generated.

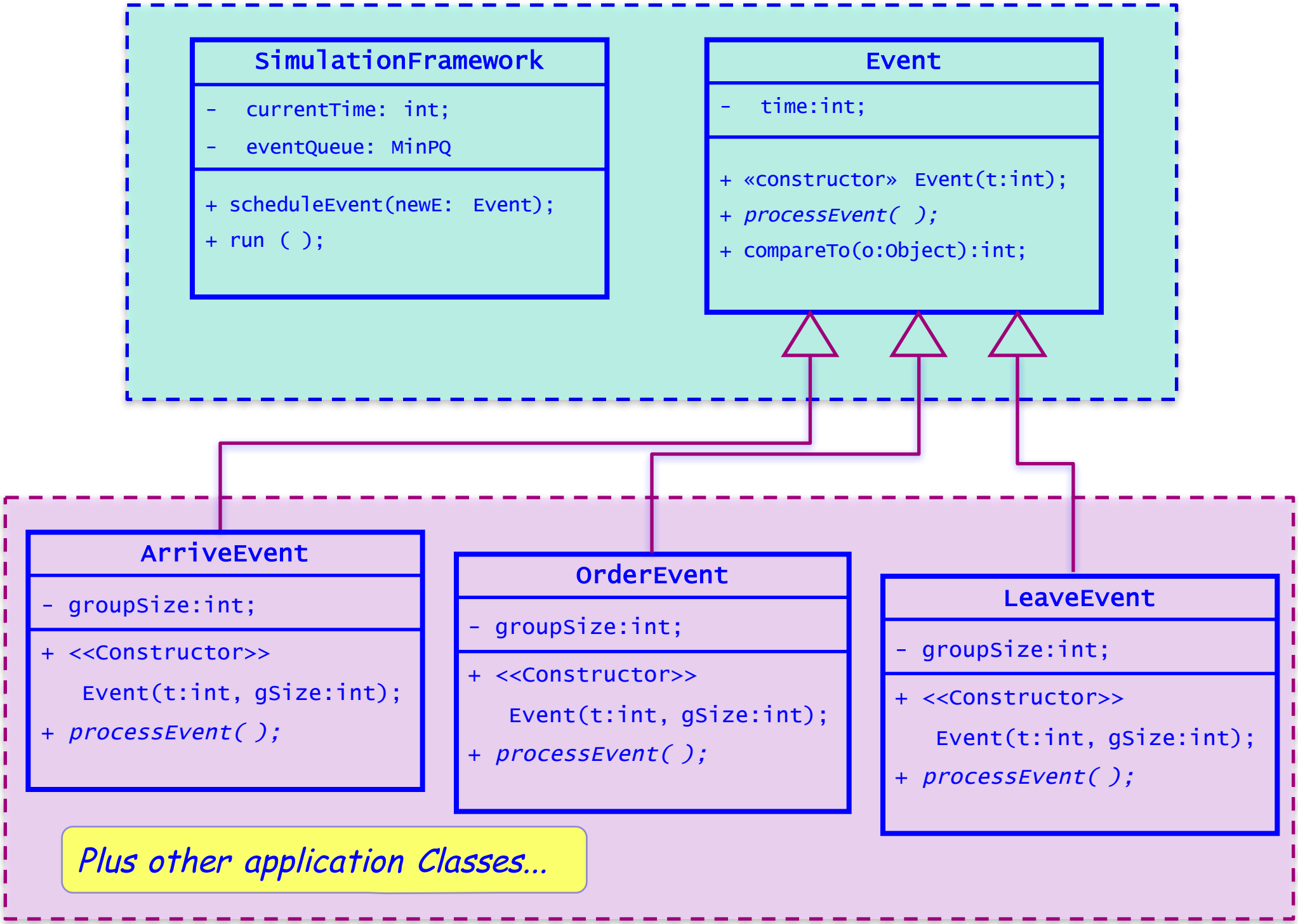
Object of Simulation – the Bar

- The behavior of the bar can be described by the following list:
  - When a customer group arrives, the size of the group is compared to the number of available seats.

    If insufficient seats are available, the group leaves.

    Otherwise, the group is seated and the number of seats is decreased.
  - When a customer orders and is served, the amount of profit is updated.
  - When a customer group leaves, the seats are released for another customer group.

- A partial description of the `SoftwareGurusBar` class is given below:


SimulationFramework
- currentTime: int;
- eventQueue: MinPQ
+ scheduleEvent(newE: Event);
+ run ( );
Event
- time:int;
+ «constructor» Event(t:int);
+ processEvent( );
+ compareTo(o:Object):int;
ArriveEvent
- groupSize:int;
+ <<Constructor>>
Event(t:int, gSize:int);
+ processEvent( );
OrderEvent
- groupSize:int;
+ <<Constructor>>
Event(t:int, gSize:int);
+ processEvent( );
LeaveEvent
- groupSize:int;
+ <<Constructor>>
Event(t:int, gSize:int);
+ processEvent( );
Plus other application Classes...

class SoftwareGurusBar

```
# include <iostream.h>
# include <PriorityQueue>
# include <vector>
# include "event.h"

class randomInteger {
   public:
        unsigned int operator ( ) (unsigned int);
} randomizer;

unsigned int randomInteger::operator ( ) (unsigned int max)
{
   // rand return random integer
   // convert to unsigned to make positive
   // take remainder to put in range
   unsigned int rval = rand( );
   return rval % max;
}

unsigned int randBetween(int low, int high) {
   return low + randomizer(high - low);
}
```

class SoftwareGurusBar (cont.)

```
class SoftwareGurusBar {
   public:
      // try with 50 chairs, then try with 40, 60, ...
      // in order to find out "optimal" profit prospects
      SoftwareGurusBar( )
         : freeChairs(50), profit(0.0) { }

      bool canSeat (unsigned int numberOfPeople);   // slide 14
      void order(unsigned int beerType);            // slide 15
      void leave(unsigned int numberOfPeople);      // slide 15

      unsigned int freeChairs;
      double profit;
};

SoftwareGurusBar  theBar;

SimulationFramework  theSimulation;
```

class SoftwareGurusBar (cont.)

```
bool SoftwareGurusBar::canSeat (unsigned int numberOfPeople)
{
   // if sufficient room, then seat customers
   cout << "Time: " << theSimulation.currentTime;
   cout << " Group of " << numberOfPeople << " customers arrives";
   if (numberOfPeople < freeChairs) {
            cout << " Group is seated" << endl;
            freeChairs -= numberOfPeople;
            return true;
         }
   else {
            cout << " No room, group leaves" << endl;
            return false;
         }
}
```

class SoftwareGurusBar (cont.)

```
void SoftwareGurusBar::order (unsigned int beerType)
{
   // serve beer
   cout << "Time: " << theSimulation.currentTime;
   cout << " serviced order for " << beerType << endl;
   // update profit based on this beerType
   // add your code here ...
}


void SoftwareGurusBar::leave (unsigned int numberOfPeople)
{
   // people leave, free up chairs
   cout << "Time: " << theSimulation.currentTime;
   cout << " group of size " << numberOfPeople << " leaves" << endl;
   freeChairs += numberOfPeople;
}
```

class SoftwareGurusBar (cont.)

```
void main( ) {
   // load priority queue with initial Arrive Events then run simulation
   unsigned int t = 0;

  // load 4 hours (240 minutes) of Arrive Events
   while (t < 240) {
      // new group every 2-5 minutes
      t += randBetween(2,5);

      // group size ranges from 1 to 5
      theSimulation.scheduleEvent(new ArriveEvent(t, randBetween(1,5)));
    }

    // then run simulation and print profits
    theSimulation.run( );
    cout << "Total profits " << theBar.profit << endl;
}
```

class ArriveEvent

```
class ArriveEvent : public Event {
   public: ArriveEvent (unsigned int time, unsigned int gs)
                : Event(time), groupSize(gs) { }
        virtual void processEvent ( );

   protected:
        unsigned int groupSize;
};



void ArriveEvent::processEvent( )
{
   if (theBar.canSeat(groupSize))
   // place an order within 2 & 10 minutes
      theSimulation.scheduleEvent (
         new OrderEvent(theSimulation.currentTime + randBetween(2,10),
                        groupSize));
}
```

class OrderEvent

```
class OrderEvent : public Event {
   public: OrderEvent (unsigned int time, unsigned int gs)
                : Event(time), groupSize(gs) { }
        virtual void processEvent ( );

   protected:
        unsigned int groupSize;
};



void orderEvent::processEvent( )
{
   // each member of the group orders a beer (type 1,2,3)
   for (int i = 0; i < groupSize; i++)
      theBar.order(randBetween(1,3));
   int t = theSimulation.currentTime + randBetween(15,35);
   // schedule a LeaveEvent for this group of drinkers
   // all the group leaves together
   // add your code here ...
};
```

class LeaveEvent

```
class LeaveEvent : public Event {
   public:
        LeaveEvent (unsigned int time, unsigned int gs)
                : Event(time), groupSize(gs) { }
        virtual void processEvent ( );

   protected:
        unsigned int groupSize;
};



void leaveEvent::processEvent ( )
{
  theBar.leave(groupSize);
}
```

The Software Gurus Bar

- The method `randBetween(low, high)` returns a random integer between the two given endpoints low & high.
- The methods `canSeat`, `order`, and `leave` manipulate the profits and the number of chairs.
- The execution of the simulation is controled by the simulation framework and the inner classes `ArriveEvent`, `OrderEvent`, and `LeaveEvent` - which are described next.

Frameworks: Reusable Subsystems

- *A framework* is reusable software that implements a generic solution to a generalized problem.
  - It provides common facilities applicable to different application programs.

- *Principle: Applications that do different, but related, things tend to have quite similar designs*

Frameworks to Promote Reuse

- A framework is intrinsically *incomplete*
  - Certain classes or methods are used by the framework, but are missing *(slots)*
  - Some functionality is optional
    - Allowance is made for developer to provide it *(hooks)*
  - Developers use the *services* that the framework provides
    - Taken together the services are called the Application Program Interface *(API)*

Object-Oriented Frameworks

- In the object oriented paradigm, a framework is composed of a library of classes.
  - The API is defined by the set of all public methods of these classes.
  - Some of the classes will normally be abstract

Frameworks

- A framework is like the skeleton of an application
  - It provides the structure for a solution, but none of the application-specific behavior.
- Our framework for simulations describes the features common to all discrete event-driven simulations, namely:
  1. the concept of a priority queue of pending events, and
  2. a loop that repeatedly removes the next event from the queue and executes it.
- But note that the framework has no knowledge of the specific simulation being constructed.

Frameworks

- To create a working application, certain key classes provided by the framework are used to create subclasses. These subclasses can provide the application-specific behavior.

  In our simulation framework, the key class is `Event`. Three new types of events are created, corresponding to groups of drinkers

  ① arriving at the bar,

  ② drinkers ordering for beer, and

  ③ drinkers leaving the bar.

- Other frameworks you may have already seen are Java's AWT & Swing.
  - → To create a new application, you use inheritance to subclass from `Frame` (or `JFrame`), adding application specific behavior.
- Frameworks can be created for any task that is repeated with a wide range of variations but that has a core of common functionality.

A Framework for Simulation

- Rather than simply code a simulation of this one problem, we will generalize the problem and first produce a generic framework for simulations. At the heart of a simulation is the concept of event.
- An event will be represented by an instance of class `Event`. The only value held by the class will be the `time` the event is to occur. The method `processEvent` will be invoked to "execute" the event when the appropriate time is reached.
- The simulation queue will need to maintain a collection of various types of events. Each form of event will be represented by different derived classes of class `Event`. Not all events will have the same type, although they will all be derived from class `Event`.

A Framework for Simulation

- We are now ready to define the class `SimulationFramework`, which provides the basic structure for simulation activities.
  - The class `SimulationFramework` provides three functions.
    - The first is used to insert a new event into the priority queue,
    - the second runs the simulation,
    - and the third returns the simulation time, which is being held in a private data field.

class SimulationFramework (event.h)

```
class SimulationFramework {
   public:
      SimulationFramework ( ) : eventQueue( ), currentTime(0) { }

      void scheduleEvent (Event * newEvent)
      {
         // insert newEvent into eventQueue (Priority Queue)
         // Priority Queue is based on MinHeap using newEvent's time
         eventQueue.insert (newEvent);
      }

      void run( );

      unsigned int currentTime;

   protected:
      priority_queue <vector<Event *>, eventComparison> eventQueue;
};
```

class SimulationFramework (event.h)

```
void simulation::run( )
{
   // execute events until event queue becomes empty
   while (! eventQueue.empty( )) {
      // copy & remove min-priority element (min time) from eventQueue
      Event * nextEvent = eventQueue.peak( );
      eventQueue.deleteMin( );

      // update simulation's current time
      currentTime = nextEvent->time;

      // process nextEvent
      nextEvent->processEvent( );    // what do you see here???

      // cleanup nextEvent object
      delete nextEvent;
   }
}
```

Priority Queue API

Requirement: Generic items are Comparable.

Records with keys (priorities) that can be compared

Key must be Comparable

```
public class MinPQ <Key extends Comparable<Key>>
```

	MinPQ( )	*create an empty priority queue*
	MinPQ(Key[ ] a)	*create a priority queue with given keys*
void	insert(Key v)	*insert a key into the priority queue*
Key	delMin( )	*return and remove the element with the smallest key*
boolean	isEmpty( )	*is the priority queue empty?*
Key	min( )	*return the element with the smallest key*
int	size( )	*number of entries in the priority queue*

class Event & class eventComparison (event.h)

```
class Event {
   public:
      // constructor requires time of event
      Event (unsigned int t) : time(t) { }

      // time is a public data field
      unsigned int time;

      // execute event by invoking this method
      virtual void processEvent( ) { }
};


class eventComparison {
   public:
      bool operator ( ) (Event * left, Event * right)
      {
         return left->time > right->time;
      }
};
```

What is left for you to do?

- The following tasks are left for you to complete this project:
  - Implement the Priority Queue ADT based on the MinHeap Data Structure
  - Understand the given code and complete the "tiny" sections of code marked with (left for you)
  - Get it to run
  - Modify the simulation so it uses the weighted discrete random number-generated function described on the next slide.
    - Select reasonable numbers of weights for the type of beer ordered by the drinkers and for the sizes of the groups of drinkers
    - Compare a run of the resulting program to a run of the given program (based on uniform distribution)
- Chalenge: Applying good object-oriented design principles, add a fourth event type corresponding to re-ordering more beer
  - Entire group remains in bar for another round with a diminishing probability
  - Make sure this does not infinitely spawn re-order events
- Warning: please don't leave things till last minute!

Weighted Probability

- One alternative to the use of uniform distribution is the idea of a weighted discrete probability. Suppose that we observe a real bar and note that 15% of the time they will order local beer, 60% of the time they will order imported beer, and 25% of the time they will order special beer. This is certainly a far different distribution from the uniform distribution we used above (33% of each type of beer). In order to simulate this new behavior, we can add a new method to our class random.
  - Add a method named `weightedProbability` to the class **`SimulationFramework`**. This method will take as argument a vector of unsigned integer values. For example, to generate the preceding distribution, you will pass to the `weightedProbability` method a vector of three elements containing 15, 60, and 25.
  - The method first sums the values in the array, resulting in a maximum value. In this case, the value would be 100. A random number between 1 and this maximum value is then generated.
  - The method then decides in which category the number belongs. This can be discovered by "scanning" through the values. In our example, if the number is less than 15, the method should return 0; if less than or equal to 75 (15 + 60), return 1; otherwise return 2.

Cheers...

Making your way in the world today takes everything you've got.

Taking a break from all your worries, sure would help a lot.

Wouldn't you like to get away?

Sometimes you want to go

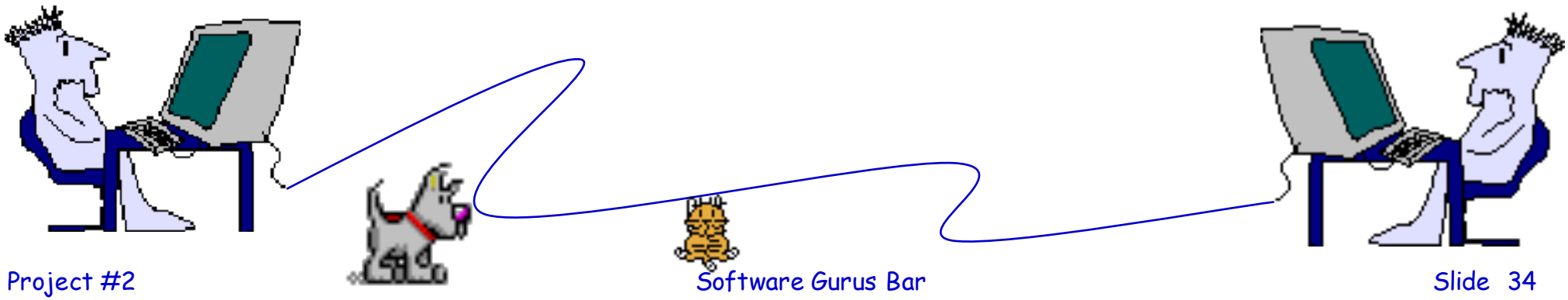
Where everybody knows your name,
and they're always glad you came.

You wanna be where you can see, our troubles are all the same

You wanna be where everybody knows Your name.

You wanna go where people know, people are all the same,

You wanna go where everybody knows your name.




EST. 1895