

Software Design
CSE 335, Spring 2006

OO design: Control and System Architecture

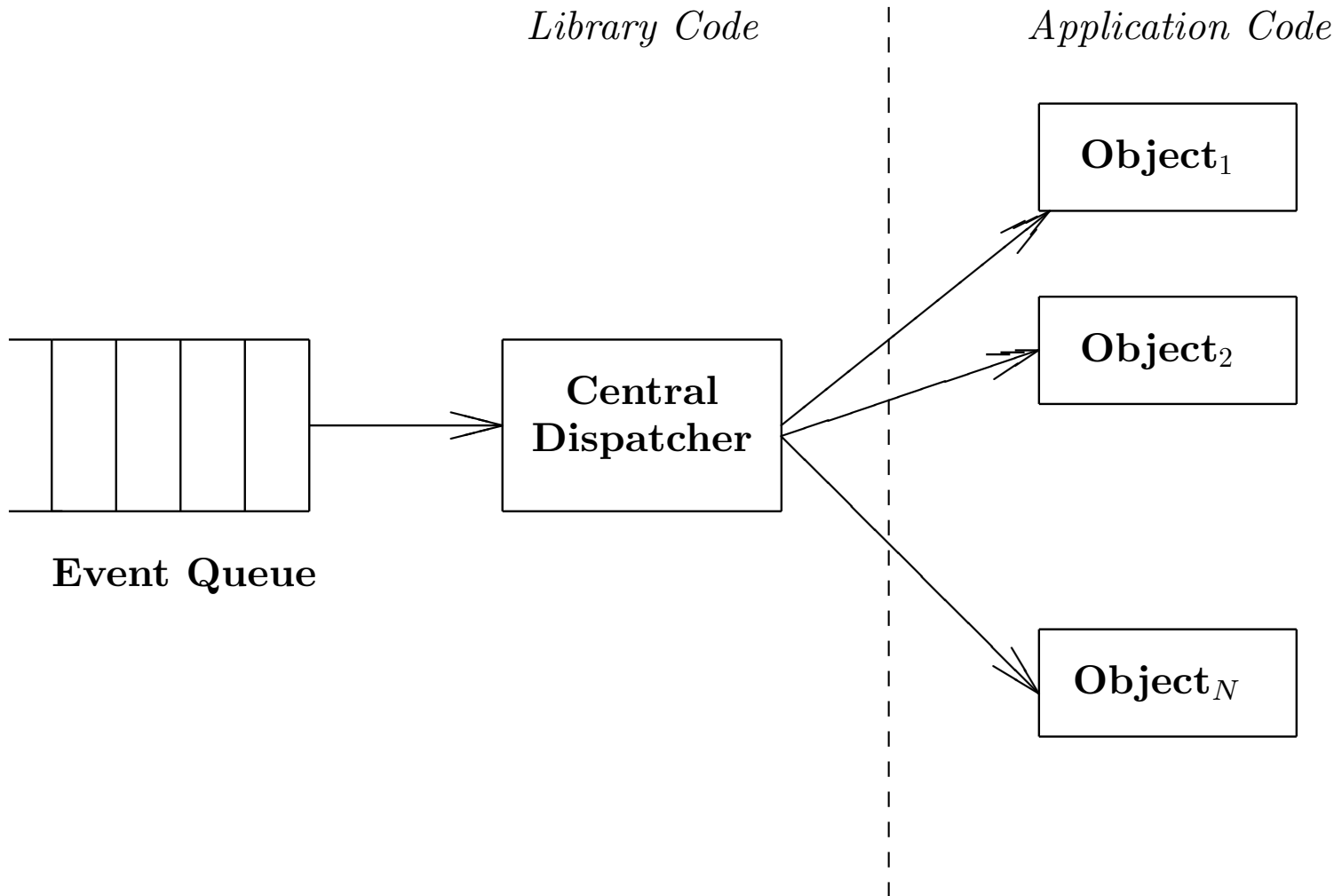
Stephen Wagner

Michigan State University

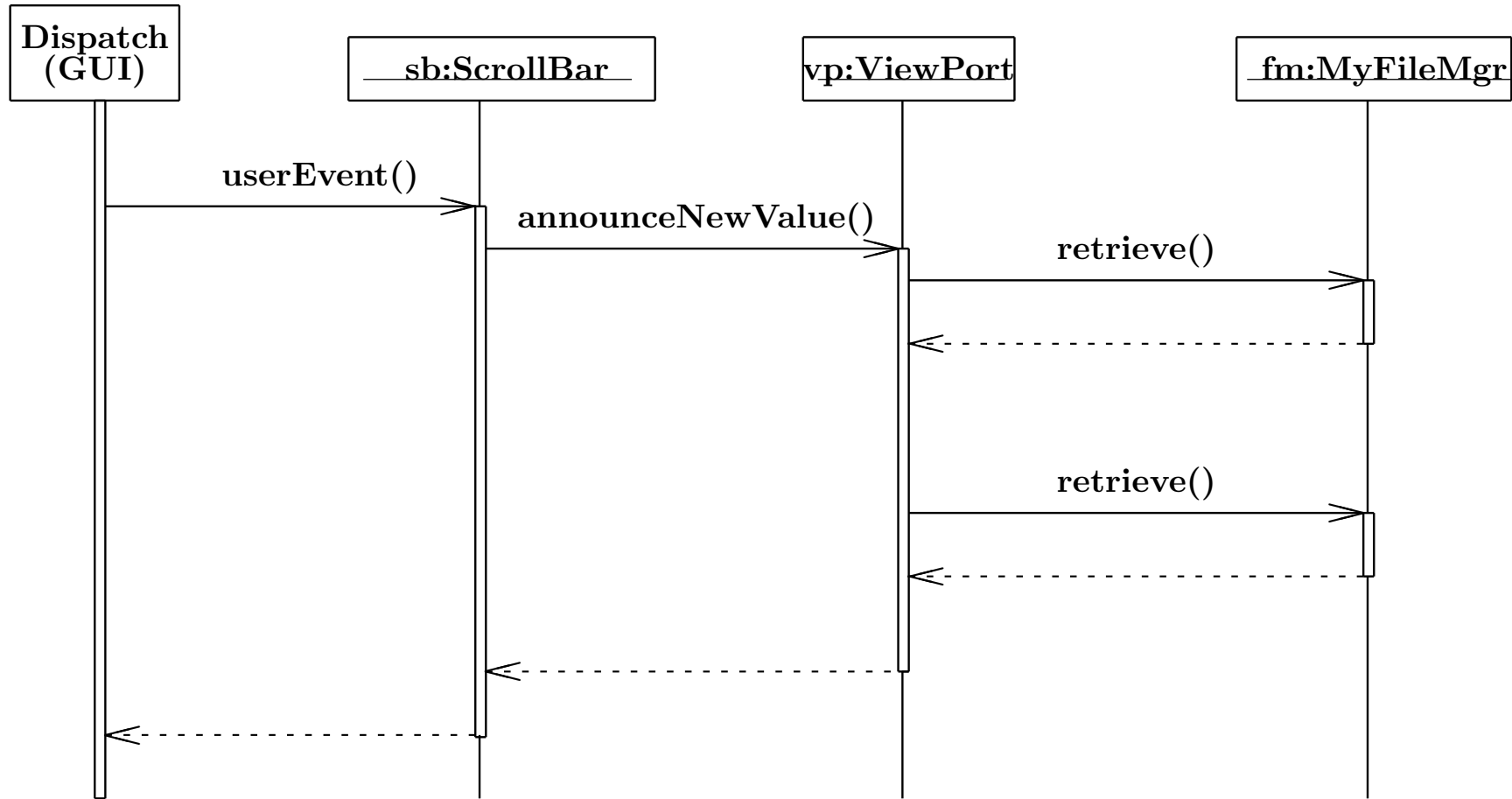
Software control

- Styles of control in a program
 1. Procedural:
 - Control resides in “application code”
 - Requests for resources block until resource available
 2. Event-driven:
 - Control resides in a central dispatcher
 - Application code does not explicitly request resources
 - Application methods are “called back” to handle *external events*
 3. Concurrent:
 - Multiple simultaneous *threads* of control
 - New problems to contend with: synchronization, mutual exclusion
 - Active vs. passive objects

Event-driven control



Scrollbar "call-back" scenario



Notes on the “Dispatch” object

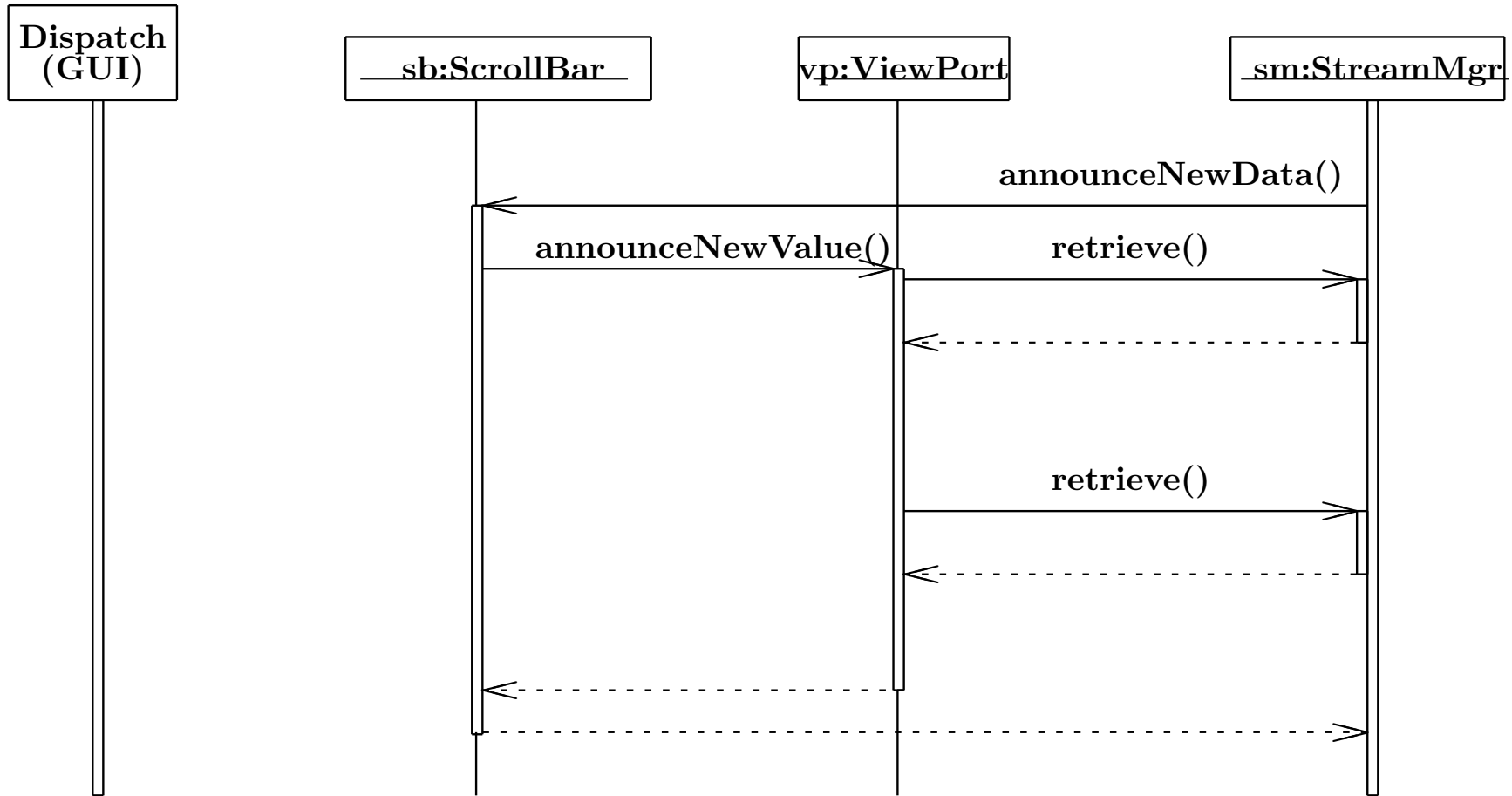
- The dispatch object is not created by our “application code”:
 - We got it when we linked in the fltk library
 - Function `main` *cedes control* to Dispatch once `main` has created and linked the “application objects”
 - Control ceded by

```
return Fl::run();
```
 - Dispatch is a loop that services device events until all windows have been destroyed
 - Dispatch is always active

Exercise

Suppose we want to replace the `FileManager` with a *StreamManager*, which continuously reads lines to display from a service over the network. This manager must notify scrollbar when a new line is read from the network.

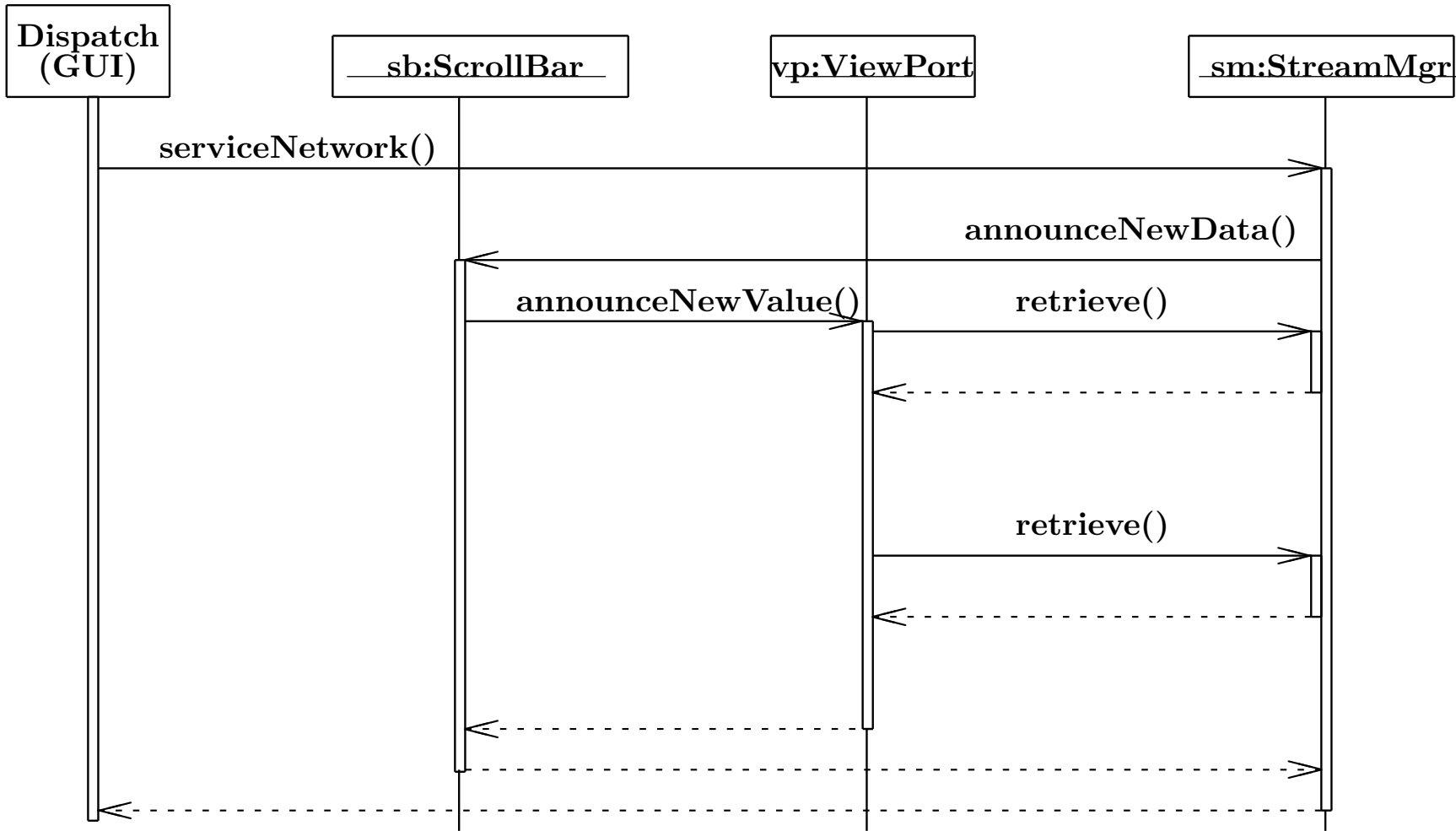
StreamManager Interaction



Observations

- Locus of control
 - Resides in object that is active throughout interaction
 - So far our designs have involved single locus of control
- Stream-manager sequence diagram involves two loci of control
 - One for “The GUI”
 - One for the stream manager
 - As yet we have not discussed a way to build applications with multiple loci of control
- How can we simulate multiple loci of control?

Solution



Modifications to main()

```
int main (void)
{
    StreamManager * = new StreamManager (...);
    ...

    // New event loop
    while (Fl::check()) {
        Fl::wait(interval);
        sm->serviceNetwork();
    }
    return 0;
}
```

How to simulate additional loci of control?

New concept: Active objects

- StreamManager is an example of an *active object*
 - Appears to be running in its own locus of control distinct from the main application
 - Often needed to monitor asynchronous events: e.g. user input, mouse motion, network traffic
- Design complexity increases when application involves multiple active objects
- How can this complexity be hidden?

Exercise

Develop a class called **GUIManager** that simplifies extending the event loop to cede time to active objects.

What other classes will be helpful?

ActiveObjectInterface

```
class ActiveObjectInterface
{
    public:
        virtual bool activate() = 0;
        virtual void terminate() = 0;
};
```

Class GUIManager

```
class GUIManager
{
    public:
        GUIManager();

        void setTimeoutInterval(float interval);
        void registerActiveObject(ActiveObjectInterface *);
        bool activate();
        void terminate();

    protected:
        float                                timeoutInterval;
        vector<ActiveObjectInterface *>      activeObjects;
};
```

Questions

1. What happens if the user clicks on scrollbar *during* the serviceNetwork transaction?
2. What happens to the responsiveness of the user interface if the call to serviceNetwork *blocks* for some perceivable amount of time?
3. How might we fix this?

Control implementation strategies

Event-driven sequential: Central dispatcher invokes registered objects to service asynchronous events

Time-sliced multiplexing:

- Scheduler activates each active object in sequence and periodically preempts one actor to schedule another
- Generally requires operating system support
- *Thread*: “Lightweight process”; essentially a sequence of code that can be multiplexed with other sequences of code within a sequential process

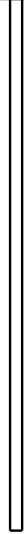
Multi-process: Active objects associated with operating system processes

Multi-process

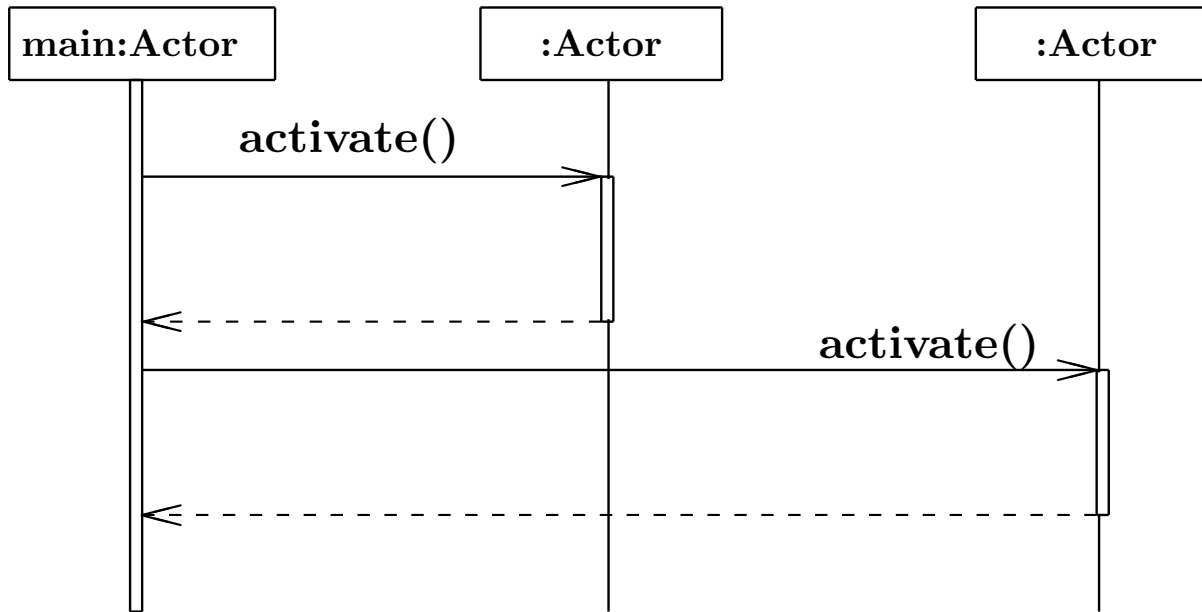
main:Actor

:Actor

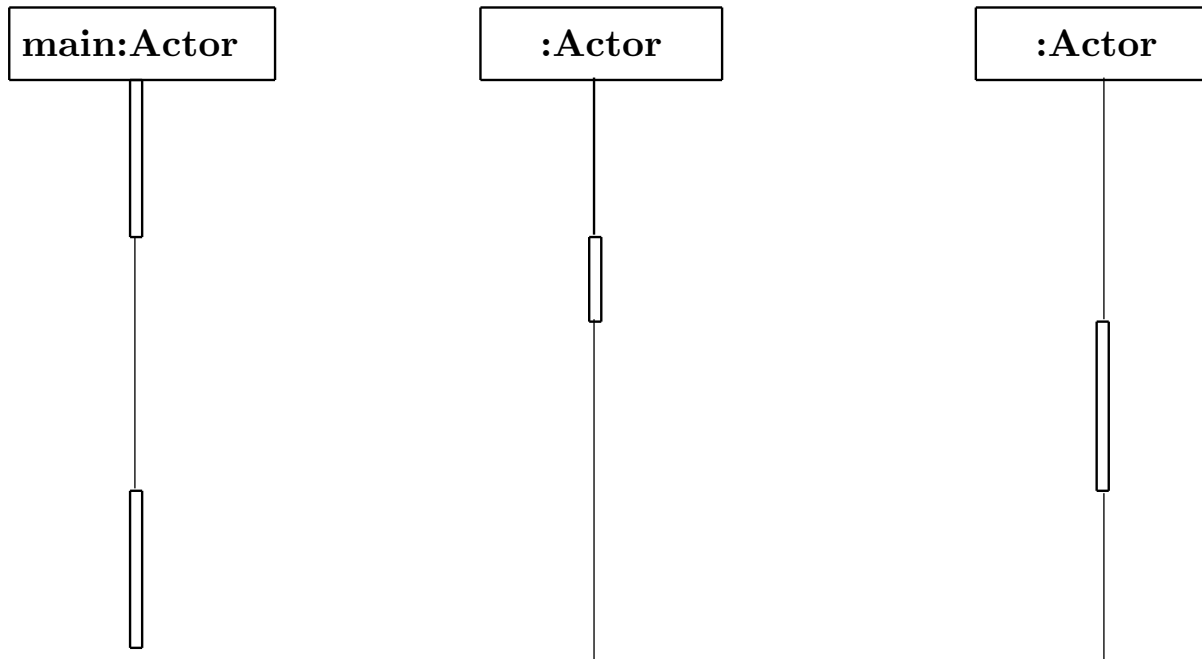
:Actor



Event Driven



Time-slice multiplexing



Design Method

After identifying the major interactions in your system:

- Identify the active objects
- Choose an implementation strategy for associating a locus of control with each object
 - Note: the choice of strategy may introduce new roles and collaborations into the problem
 - E.g., if we want each active object to be a concurrent process and we want them to be distributed over the network, then “sending a message” is no longer just a method invocation
- Use role-collaboration synthesis techniques we have already studied to develop concrete classes and configuration code

New Issues

- Event-driven sequential strategy is easy to reason about, but may exhibit noticeable time delays if activations require time complete
- Time-sliced multiplexing good alternative, but it incurs a whole new set of design complexities:
 - Activations do not run to completion
 - They may be interrupted and resumed at any time
 - This causes problems when multiple active objects *share* one or more passive objects.