CS 535 Object-Oriented Programming & Design
Fall Semester, 2011
Doc 18 How do orcs move
Nov 10 2011

Common Manager Behavior

A project is behind schedule

So to get back on schedule they hire more people

The Result

The project will be even later

Parameters of any Project

Time

- How much time we have for the project

Scope (Size)

- Features of the project
- How much work is to be done

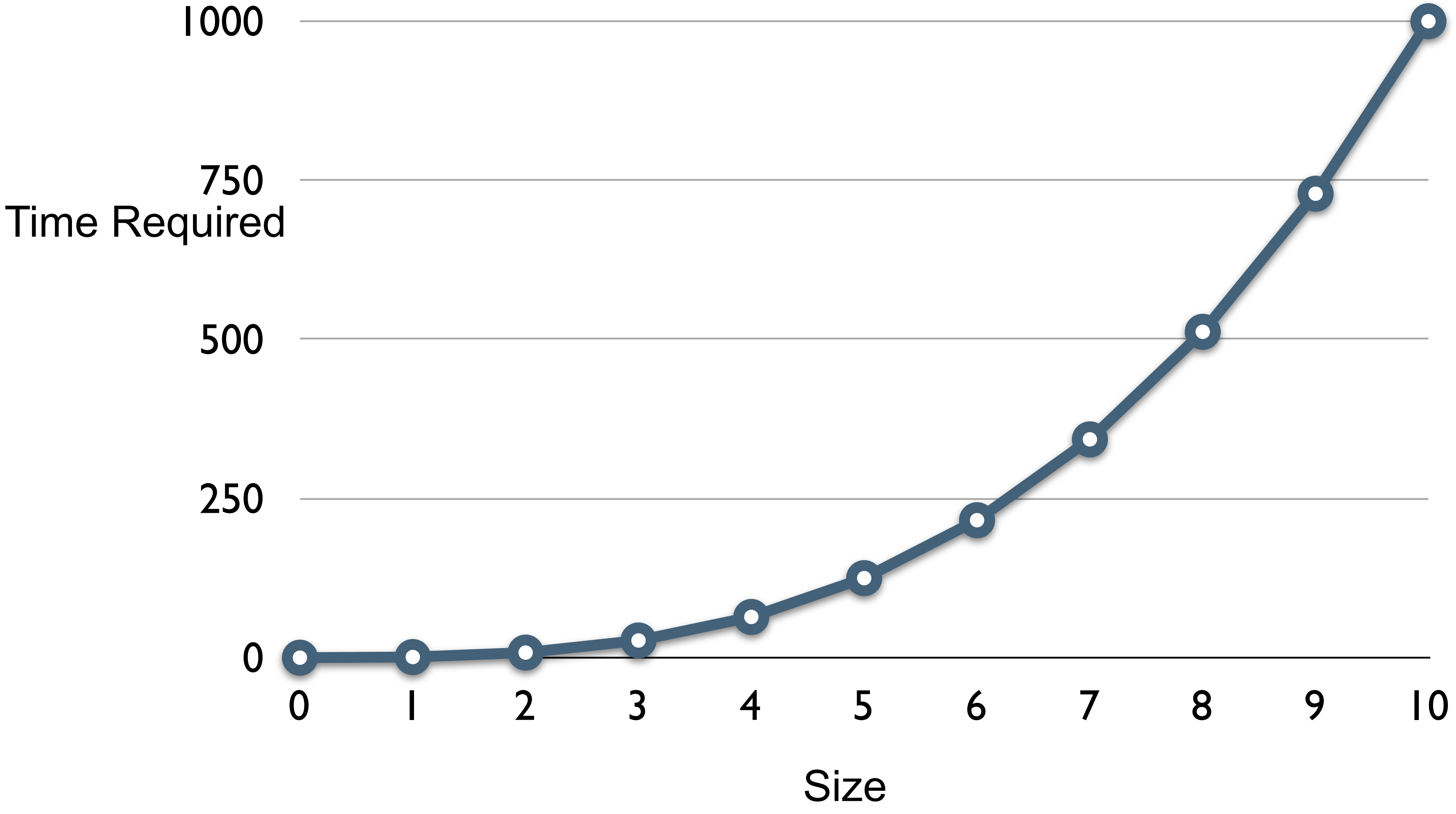
Quality

- The quality of work

Cost

- How many people work
- Tools used

Non-linear Relationships



So

Doubling size of project more that doubles the amount of work

Doubling the team does not halve the time

Why adding people slows down projects

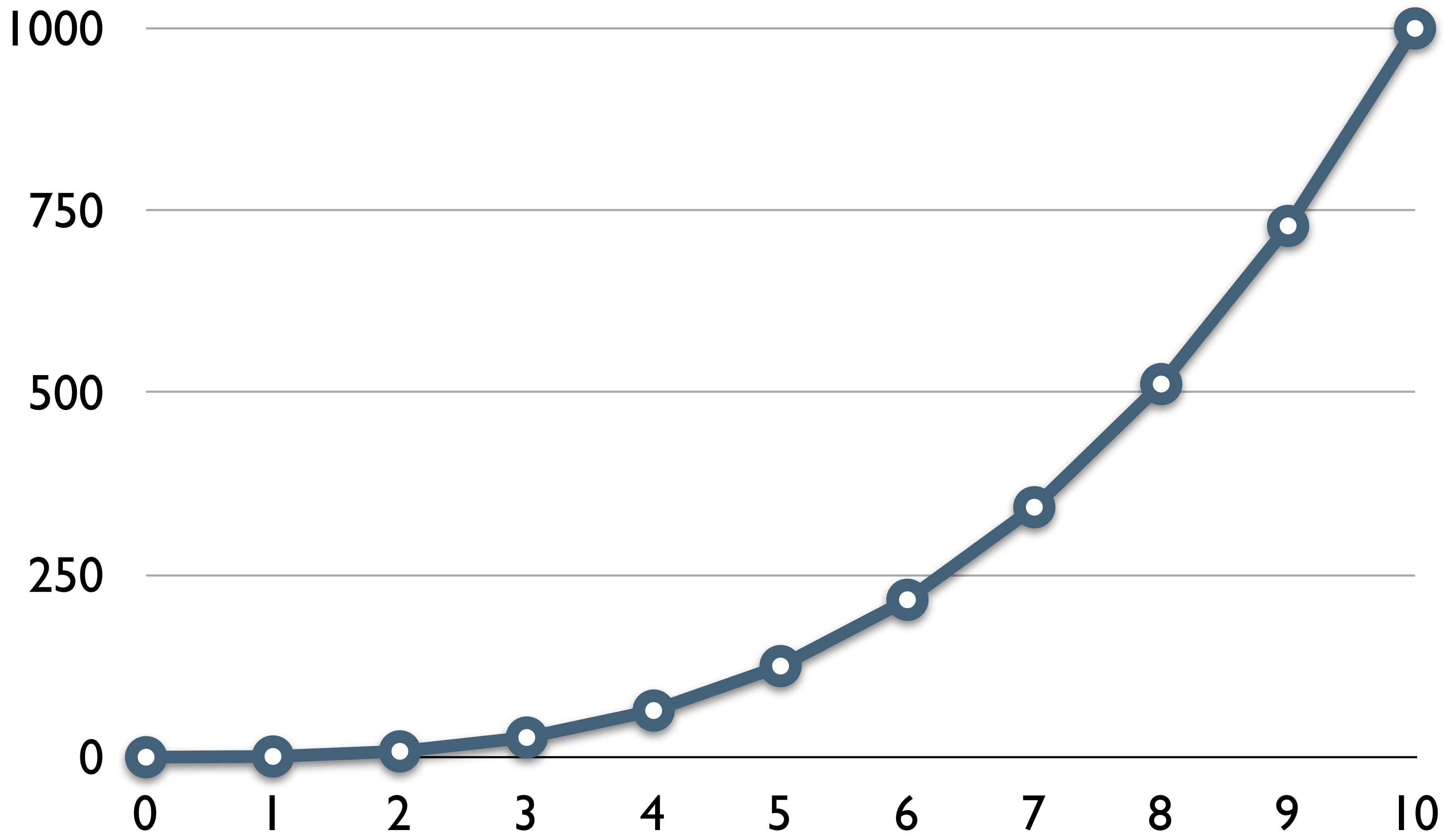
Existing people need to help bring new people up to speed

- So get less work done

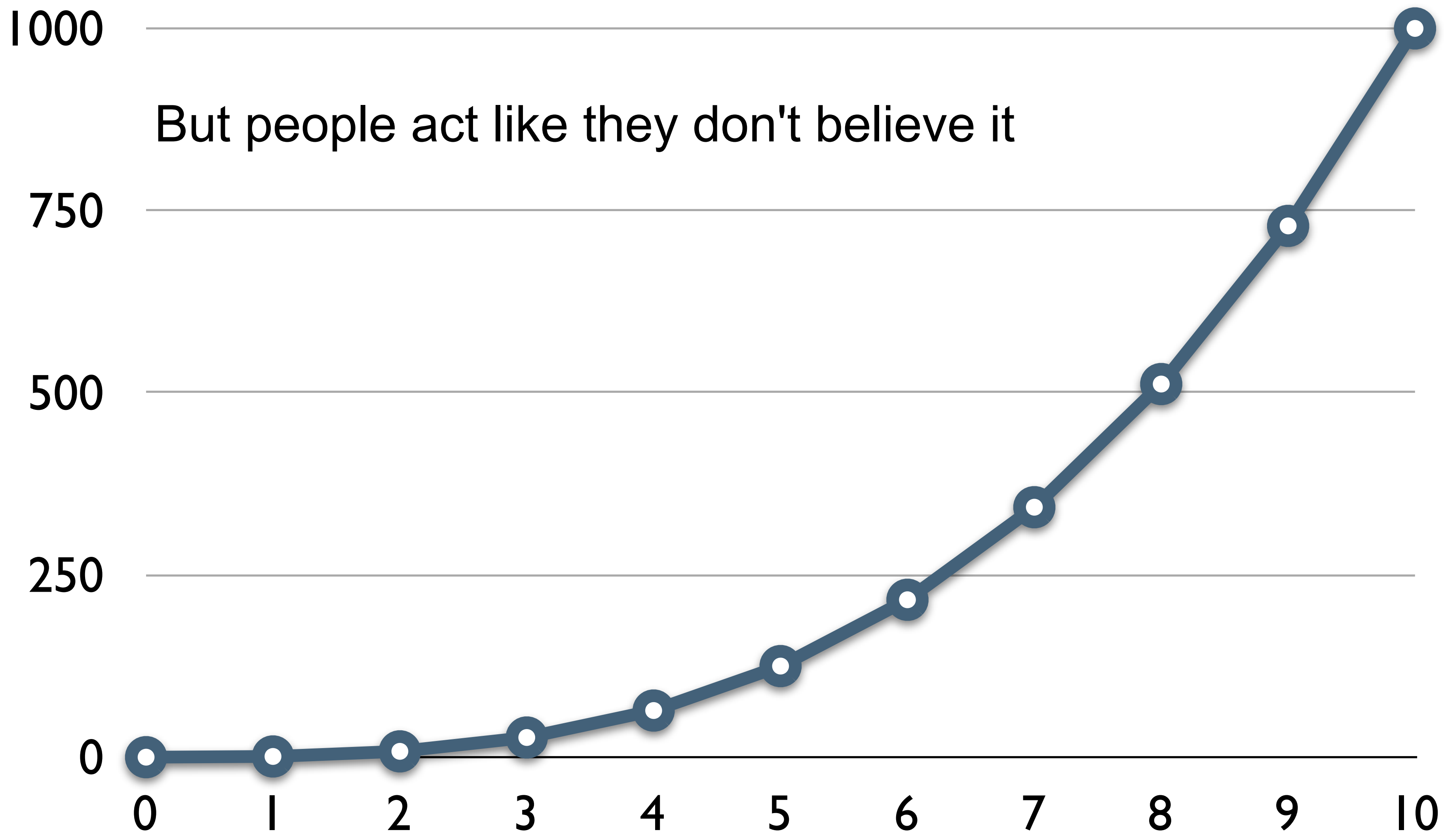
More people on team makes it harder to communicate

- More meetings
- More documents
- Less work

Small is better



Small is better



But people act like they don't believe it

Survey

1/2 way done with project

Need make orcs move independent of player

But have never done that before so don't know how

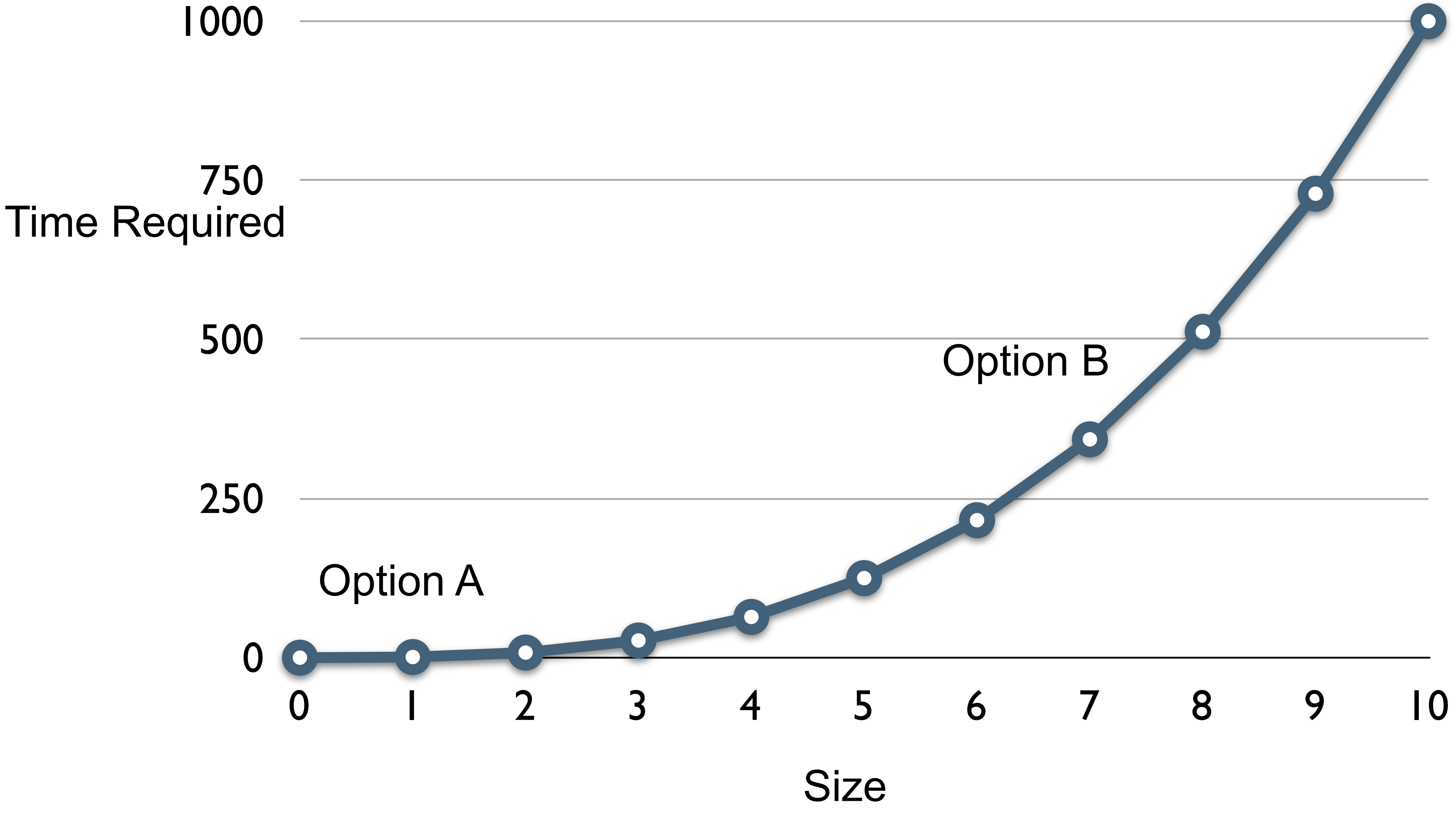
Option A

Start new project
to explore how to do it

Option B

Using existing project to
explore how to do it

Which is better



Technical Spikes

How do orcs move?

Parsing commands

- What did the user just type?

What is a program?

How detect near things?

Goal - How to make Orcs Move

- Spike
    - Simple Clock app

Timer

```
| timer count |
count := 0.
timer := Timer every: 0.2 seconds
            do:
                [Transcript
                    show: count printString;
                    cr;
                    flush.
                count := count + 1].
3 seconds wait.
timer := nil
```

But timer goes way when code done

Need to keep a reference that continues

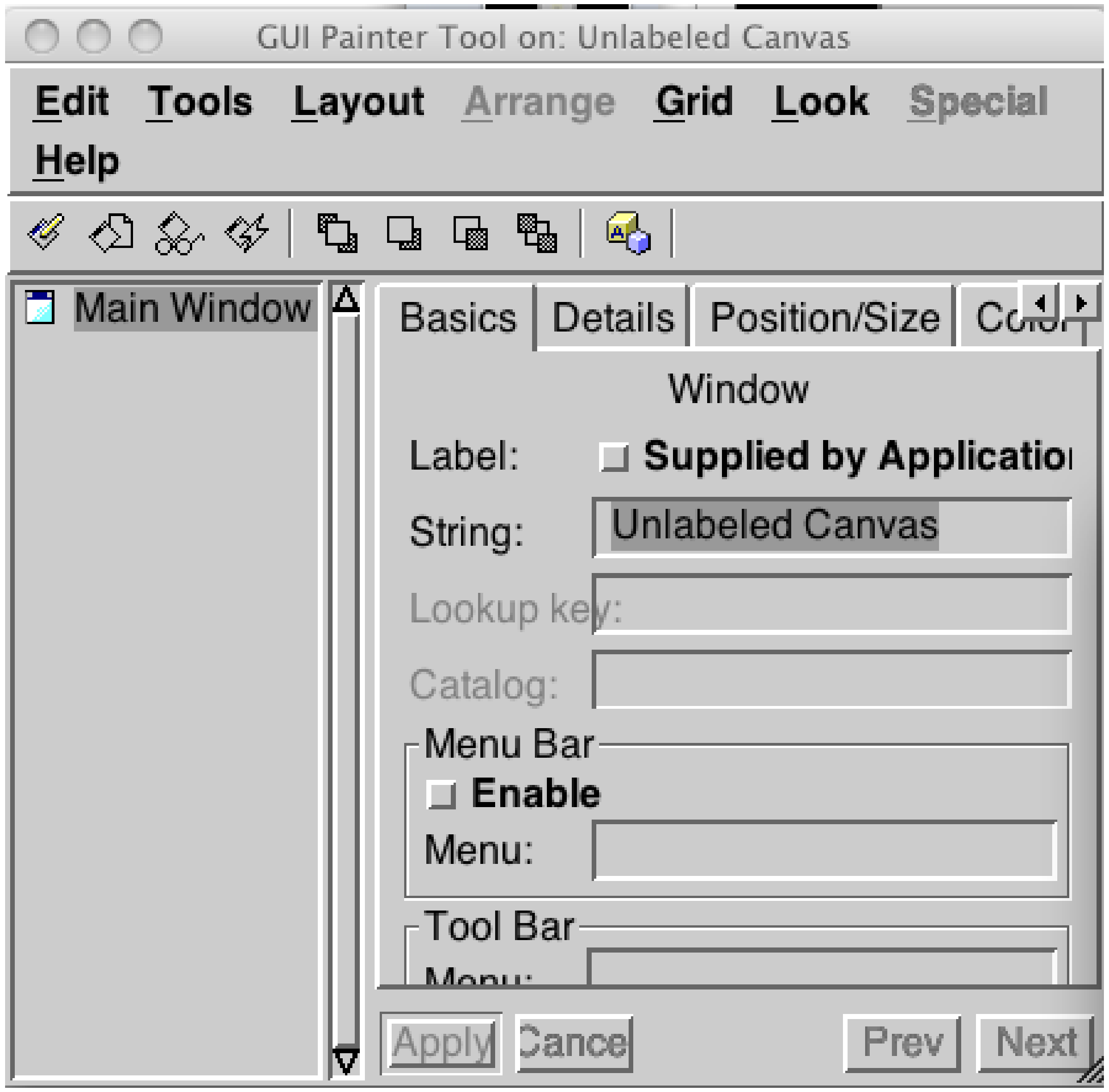
Some GUI review

Gui Builder

Buttons

Text

UI Painter Windows


GUI Painter Tool on: Unlabeled Canvas
Edit Tools Layout Arrange Grid Look Special
Help
Main Window
Basics Details Position/Size Color
Apply Cancel Prev Next

Window

Label: □ Supplied by Application

String: Unlabeled Canvas

Lookup key:

Catalog:

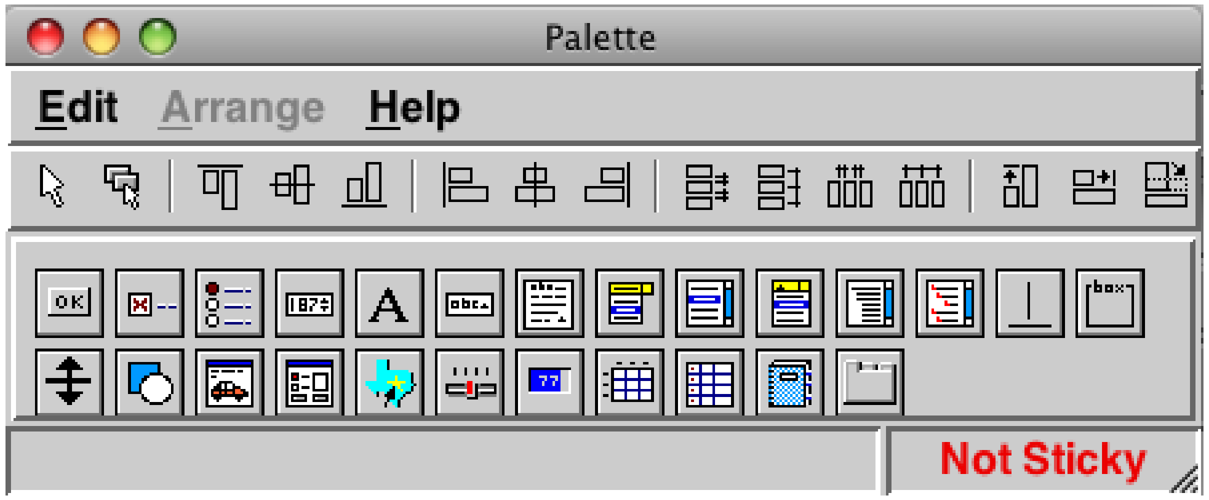
Menu Bar

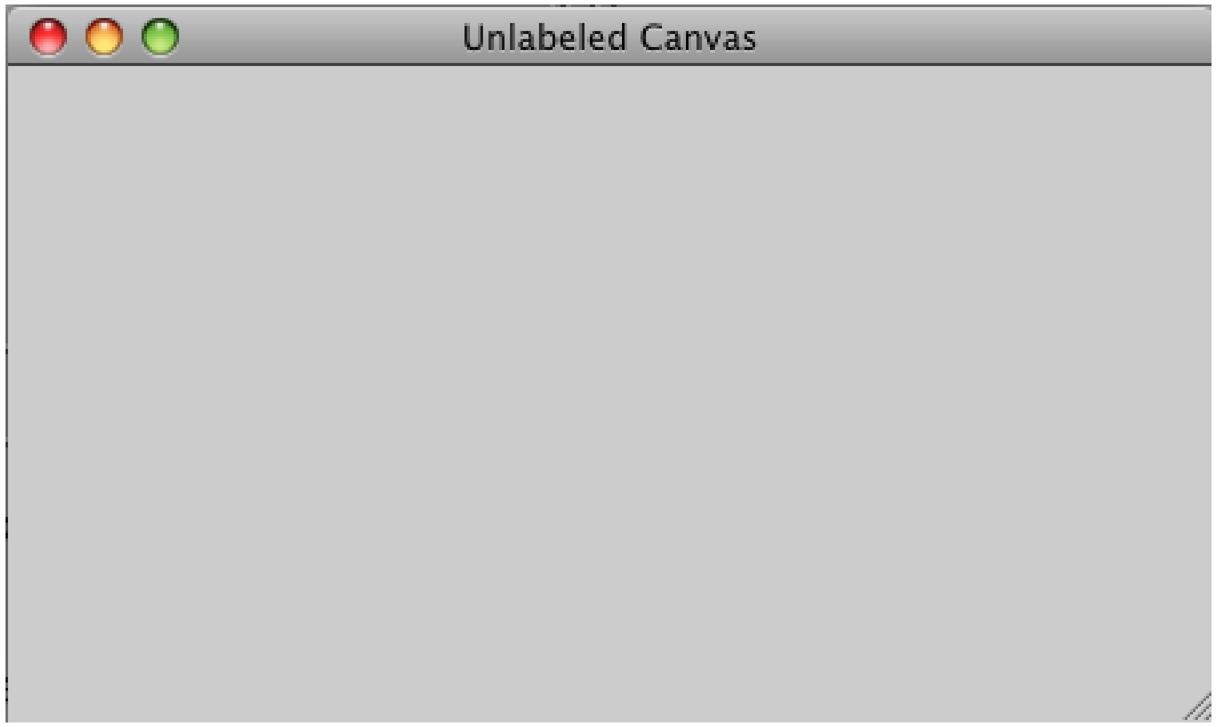
□ Enable

Menu:

Tool Bar

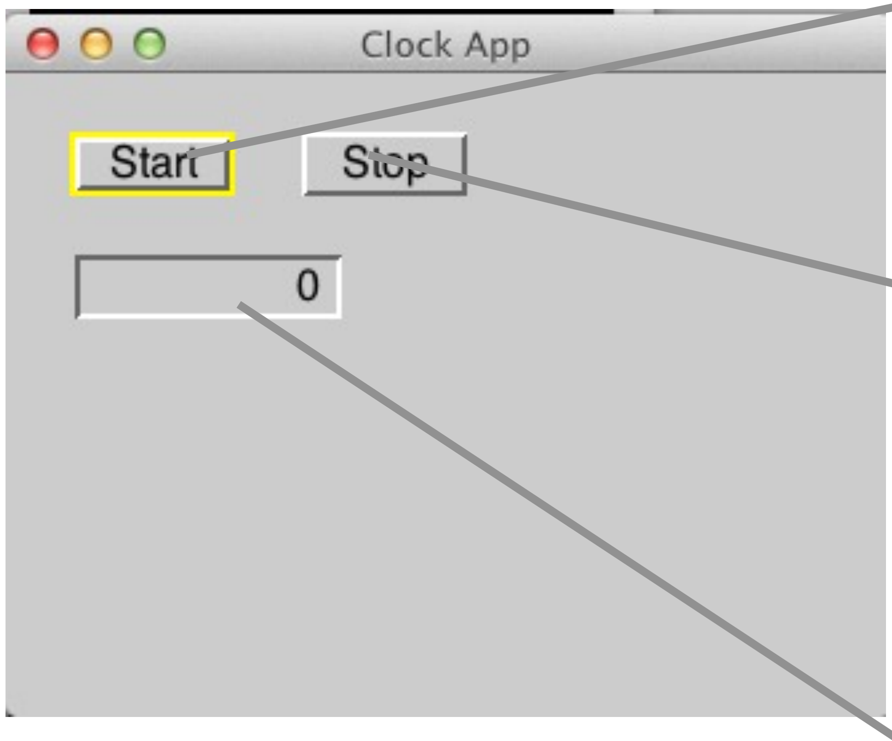
Menu:





Palette - Widgets that we can put in the window
Unlabeled Canvas - Window we are constructing
GUI Painter Tool - Details about the widgets in our new window

The App



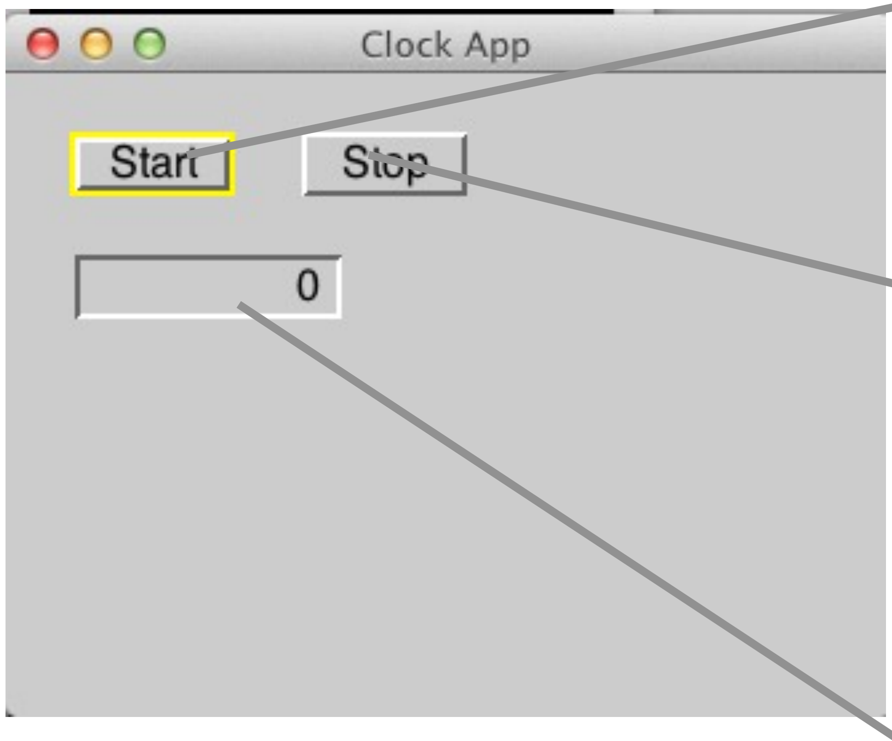
```
startTimer
      clock startAfter: 0 seconds
```

```
stopTimer
      clock stop
```

```
timeDisplay
      ^timeDisplay isNil
            ifTrue:
                  [timeDisplay := 0 asValue]
            ifFalse:
                  [timeDisplay]
```

```
initialize
      time := 0.
      clock := Timer new.
      clock
            period: 1 seconds;
      block:
                  [time := time + 1.
                  timeDisplay value: time]
```

How does this work?



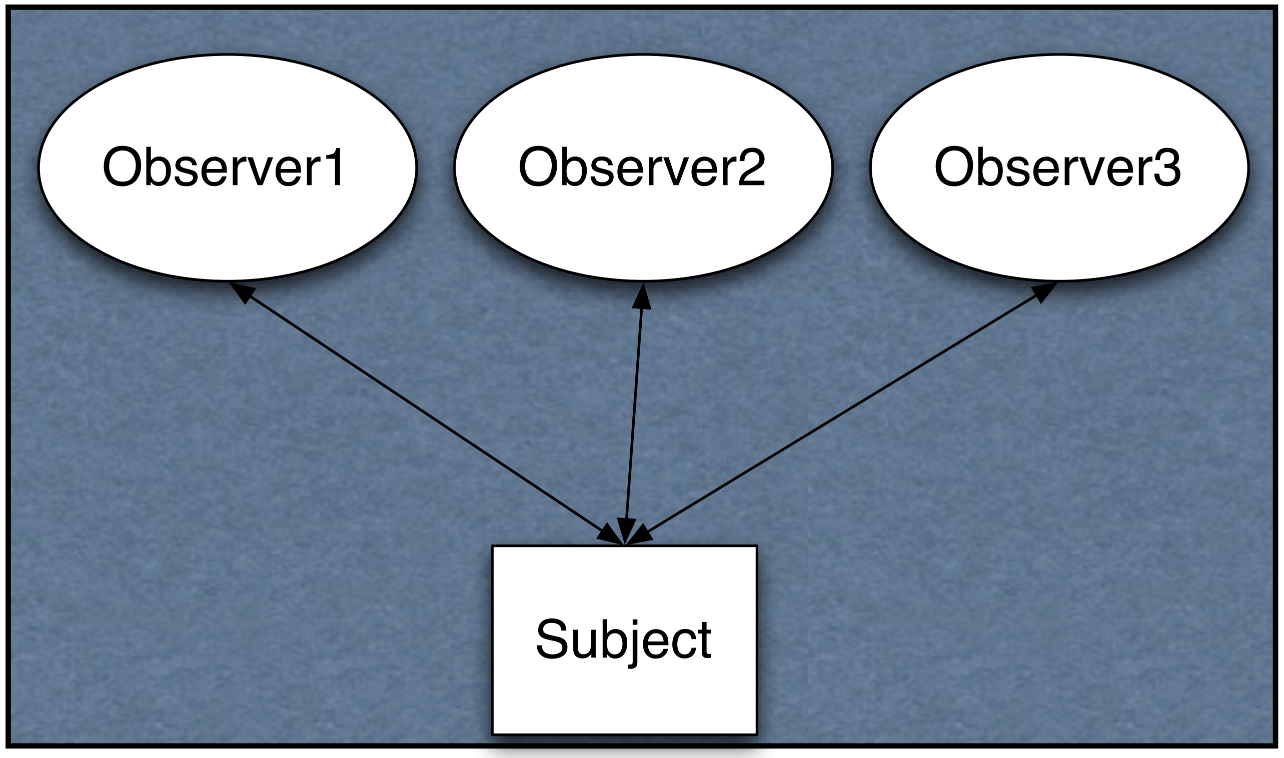
```
startTimer
    clock startAfter: 0 seconds
```

```
stopTimer
    clock stop
```

```
initialize
    time := 0.
    clock := Timer new.
    clock
        period: 1 seconds;
    block:
            [time := time + 1.
            timeDisplay value: time]
```

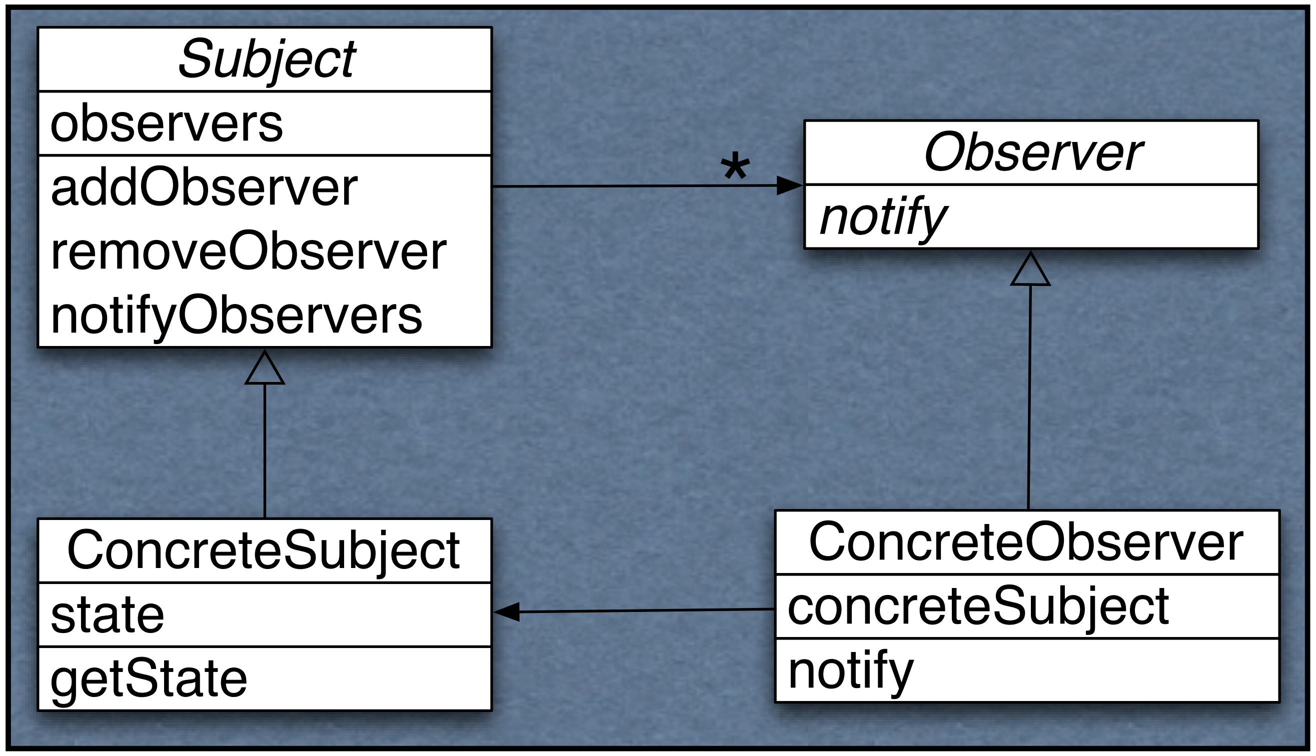
```
timeDisplay
    ^timeDisplay isNil
        ifTrue:
            [timeDisplay := 0 asValue]
        ifFalse:
            [timeDisplay]
```

Observer



Subject notifies all observers when it changes

Keeping it Flexible



```
Subject>>notifyObservers
    observers do: [:each | each notify]
```

ValueHolder

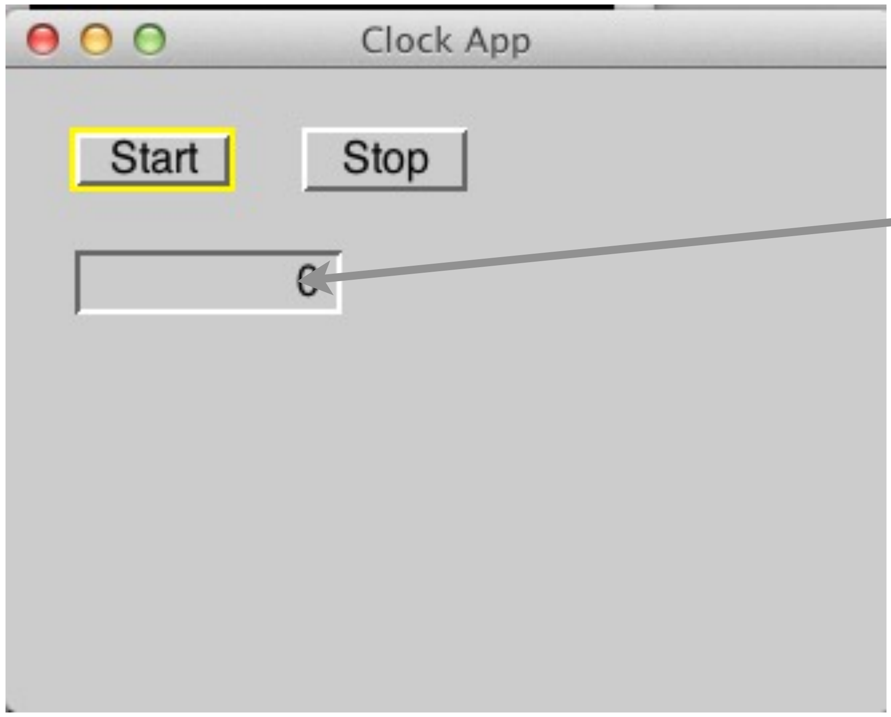
A subject

When value changes it notifies observers

foo asValue
- Returns ValueHolder on foo

valueHolder value: newValue
- Changes the value
- Notifies observers

How does this work?



Observer

Subject

```
initialize
      time := 0.
      clock := Timer new.
      clock
            period: 1 seconds;
      block:
                  [time := time + 1.
                  timeDisplay value: time]
```

```
timeDisplay
      ^timeDisplay isNil
            ifTrue:
                  [timeDisplay := 0 asValue]
            ifFalse:
                  [timeDisplay]
```

Subject Changed

Coupling

Measure of the interdependence among modules

"Unnecessary object coupling needlessly decreases the reusability of the coupled objects "

"Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more of the coupled objects"

Coupling and Transcript

```
Smalltalk.CS535 defineClass: #Customer
   superclass: #{Core.Object}
   instanceVariableNames: 'name phone id '
```

```
Customer>>display
   Transcript
      show: 'Customer(';
      print: name;
      show: ', ';
      print: phone;
      show: ', ';
      print: id;
      show: ')'
```

```
foo := Customer new.
...
foo display.
```

Separate display device from Customer

```
Customer>>printOn: aStream
   aStream
      print: 'Customer(';
      print: name;
      print: ', ';
      print: phone;
      print: ', ';
      print: id;
      print: ')'
```

```
foo := Customer new.
...
Transcript
     show: foo printString.

bar := 'bar' asFilename writeStream.
bar
     nextPutAll: foo printString
```

By separating the output device from the class we gain flexibility on where the output goes.

Model-View-Controller (MVC)

Model

Encapsulates

- Domain information
- Core data and functionality

Independent of

- Specific output representations
- Input behavior

View

Display data to the user

Obtains data from the model

Multiple views of the model are possible

Controller

Handles input

- Mouse movements and clicks
- Keyboard events

Each view has it's own controller

Programmers commonly don't see controllers

The Controller Mess

Smalltalk 80 created the MVC pattern

Considered very good

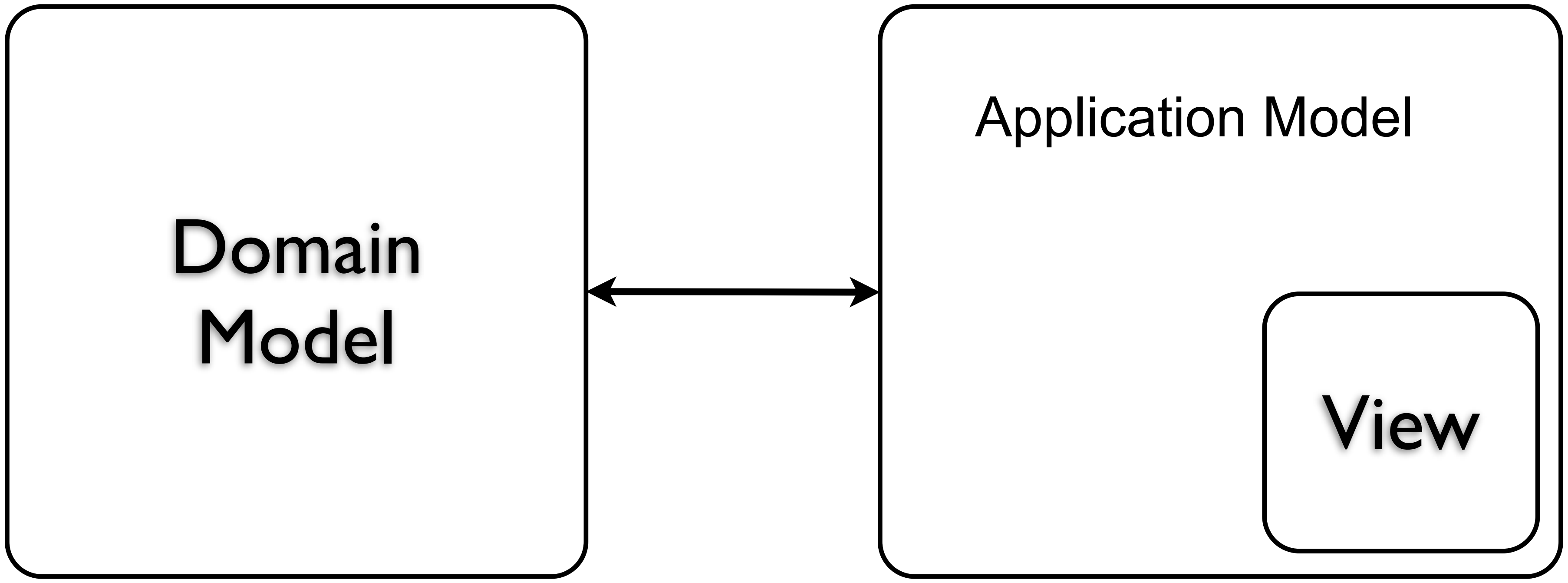
But Smalltalk found controller

- Painful
- Always did same thing

So Smalltalk hid the controller

But everyone wants to copy Smalltalk's MVC

Smalltalk Uses Application Model



Application Model

- Presentation of domain to user
- GUI + logic to present data from domain

Application Model == Controller

What all systems now call Controller is really Application model

Presentation of domain to user

GUI + logic to present data from domain

The Controller Trap

Controller ends up doing all the work

Domain logic ends up in controller

Clock App

Model

ButtonExample

View

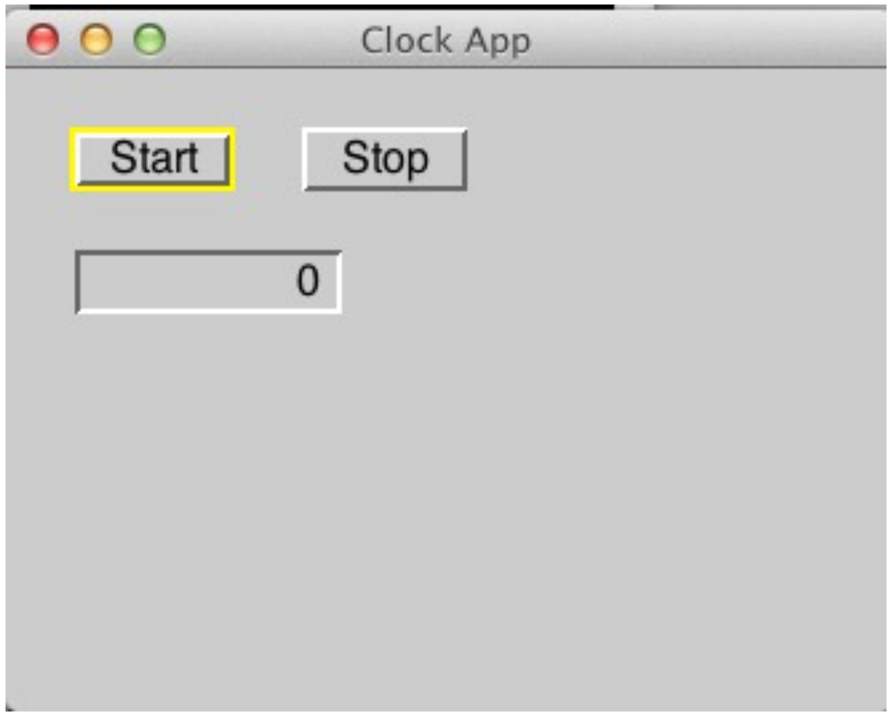
Created dynamically from window spec

Controller

Hidden

Clock App

View



Application Model Logic

```
startTimer
    clock startAfter: 0 seconds
```

```
stopTimer
    clock stop
```

```
timeDisplay
    ^timeDisplay isNil
        ifTrue:
            [timeDisplay := 0
asValue]
        ifFalse:
            [timeDisplay]
```

Clock App - Where is the Domain Model?

```
initialize
      time := 0.
      clock := Timer new.
      clock
            period: 1 seconds;
      block:
                  [time := time + 1.
                  timeDisplay value: time]
```

time + clock = Domain Model

But Application Model contains
code to make domain model work

Domain logic is in application model

So who cares?

Domain Logic in controller

- Can't reuse domain model - missing logic

Controller becomes more complex

- Does two different things

So create Domain Object - Clock

```
Smalltalk defineClass: #Clock
    superclass: #{Core.Object}
    instanceVariableNames: 'count timer '
```

Class Method

```
period: aDuration
    ^super new setPeriod: aDuration
```

Instance Methods

```
setPeriod: aDuration
    count := 0.
    timer := Timer new.
    timer period: aDuration.
    timer block: [timer := timer + 1]
```

```
start
    timer startAfter: 0 seconds
```

```
stop
    timer stop
```

```
time
    ^count
```

But how to know when to display new time

Three solutions

- Clock block
- Classic Subject-Observer
- Announcements

Clock Block

Give Clock object a block

Clock executes block when timer goes off

Block updates text view with new time

So create Domain Object - Clock

```
Smalltalk defineClass: #Clock
    superclass: #{Core.Object}
    instanceVariableNames: 'count timer operation'
```

Class Method

```
period: aDuration operation: aBlock
    ^super new
        setPeriod: aDuration operation: aBlock
```

Instance Methods

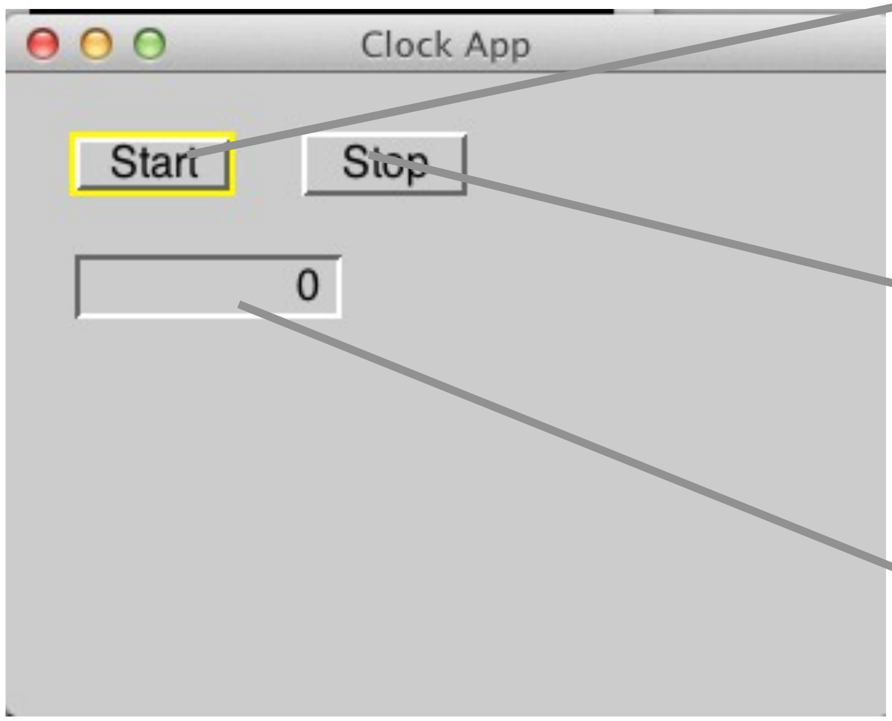
```
setPeriod: aDuration operation: aBlock
    count := 0.
    operation := aBlock.
    timer := Timer new.
    timer period: aDuration.
    timer block:
            [count := count + 1.
            operation value: count]
```

```
start
    timer startAfter: 0 seconds
```

```
stop
    timer stop
```

```
time
    ^count
```

New Clock App



```
startTimer
    clock start
```

```
stopTimer
    clock stop
```

```
timeDisplay
    ^timeDisplay isNil
        ifTrue:
            [timeDisplay := 0 asValue]
        ifFalse:
            [timeDisplay]
```

```
initialize

    clock := Clock period: 1 seconds
                operation: [:time | timeDisplay value: time]
```

Advantage of Using Clock Domain Object

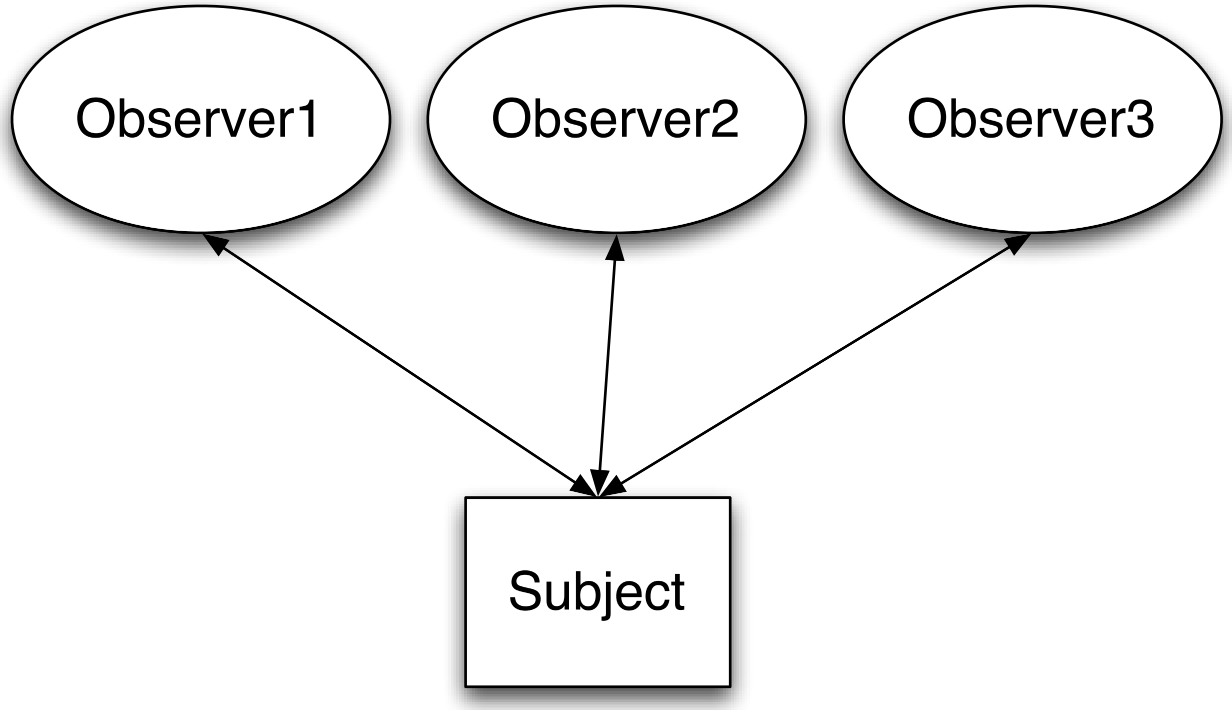
We can use Clock in other settings

(like to tell Orcs when to move)

Disadvantage

Clock can only notify one thing

Solution - Observer pattern



Subject notifies all observers when it changes

Make Clock a subject so it can have many observers

Classic Observer pattern

To add an observer subject

subject addDependent: anObserver

All classes in Smalltalk act as subject

How subject starts notification

self changed.

How observer registers with subject

subject addDependent: theObserver

After "self changed" subject sends message

"update: " to all Observers

This is the basics, there are a few more options in Smalltalk.

Clock as Subject

```
Smalltalk defineClass: #Clock
    superclass: #{Core.Object}
    instanceVariableNames: 'count timer '
```

Class Method

```
period: aDuration
    ^super new
        setPeriod: aDuration
```

Instance Methods

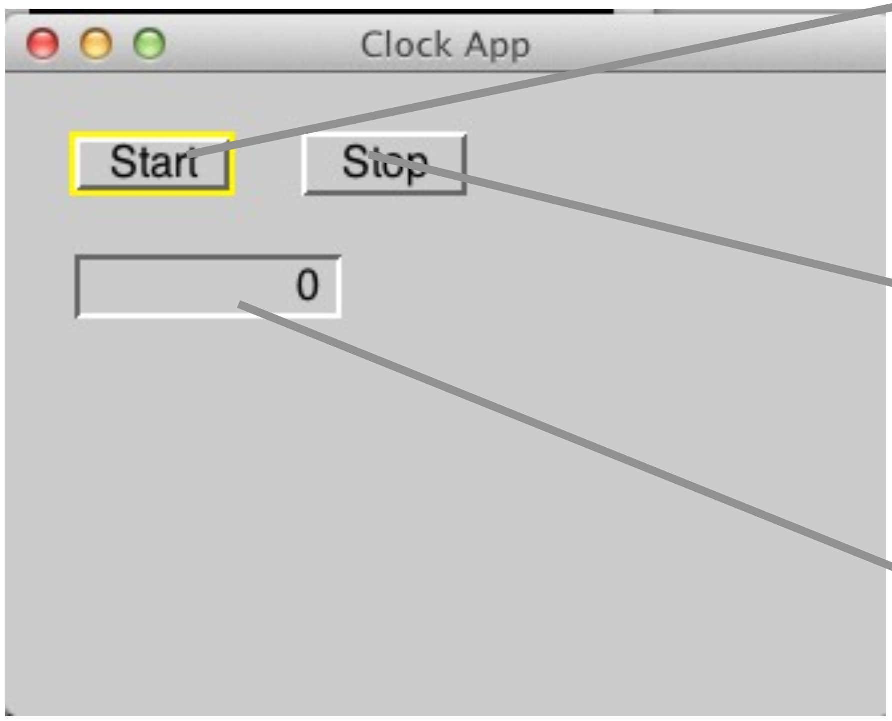
```
setPeriod: aDuration
    count := 0.
    timer := Timer new.
    timer period: aDuration.
    timer block:
            [count := count + 1.
            self changed]
```

```
start
    timer startAfter: 0 seconds
```

```
stop
    timer stop
```

```
time
    ^count
```

Clock App with Clock subject



```
startTimer
    clock start
```

```
stopTimer
    clock stop
```

```
timeDisplay
    ^timeDisplay isNil
        ifTrue:
            [timeDisplay := 0
asValue]
        ifFalse:
            [timeDisplay]
```

```
initialize
    clock := Clock period: 1 seconds.
    clock addDependent: self
```

update: aSymbol

```
update: aSymbol
    timeDisplay value: clock time
```

Advantages of using Subject

Clock can have multiple observers

So clock could tell multiple orcs to move

Disadvantage

Each observer needs to implement "update:"

- Update method needs to know
  - what to do
  - How to get data from subject

Announcements

Observer pattern

Specify which method subject calls on observer

How subject starts notification

    self announce: AnnouncmentType

How observer registers with subject

    subject when: AnnouncementType send: #methodName to: subject

After "self announce" subject sends

    What ever method indicated to observer

Clock as Subject

```
Smalltalk defineClass: #Clock
      superclass: #{Core.Announcer}
      instanceVariableNames: 'count timer '
```

Class Method

```
period: aDuration
      ^super new
         setPeriod: aDuration
```

Instance Methods

```
setPeriod: aDuration
      count := 0.
      timer := Timer new.
      timer period: aDuration.
      timer block:
                  [count := count + 1.
                  self announce:
ClockClick]
```

```
start
      timer startAfter: 0 seconds
```

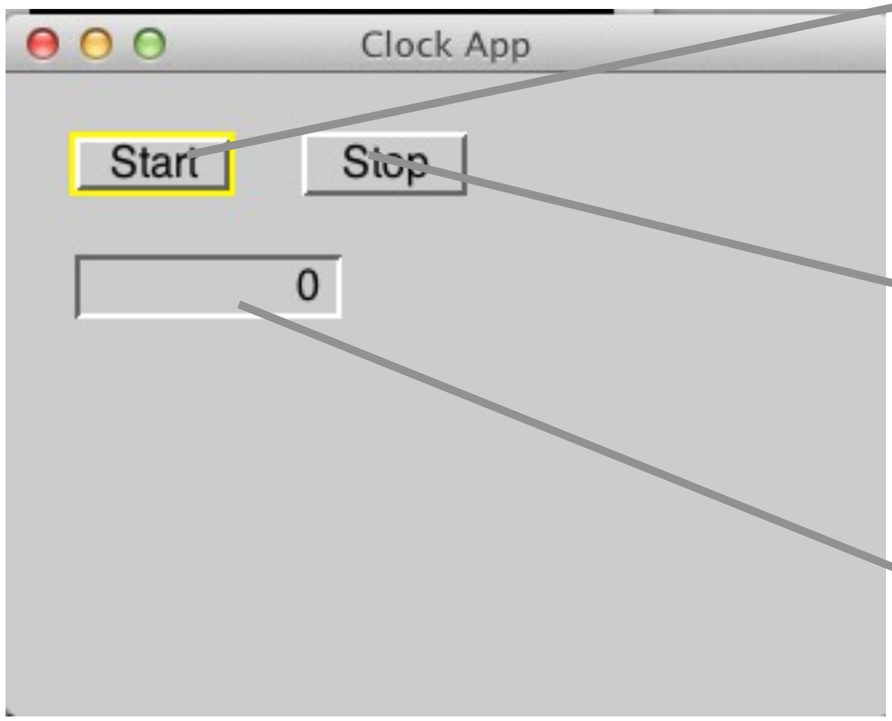
```
stop
      timer stop
```

```
time
      ^count
```

ClockClick

```
Smalltalk defineClass: #ClockClick
    superclass: #{Core.Announcement}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''
```

Clock App with Clock & Announce



```
startTimer
      clock start
```

```
stopTimer
      clock stop
```

```
timeDisplay
      ^timeDisplay isNil
            ifTrue:
                  [timeDisplay := 0
asValue]
            ifFalse:
                  [timeDisplay]
```

```
initialize
      clock := Clock period: 1 seconds.
      clock when: ClockClick
            send: #updateTimeDisplay to: self
```

```
updateTimeDisplay
      timeDisplay value: clock time
```

Options

Can send data in Announcement

Multiple parameters possible

Subject can send different types of announcements

Observers can do different things to different types announcements