

3-3

A Brief Introduction to Design Patterns

Object-Oriented Patterns

Topics

- **Brief introduction to patterns**
- **Example Patterns from Gamma et al.**
 - The Observer Pattern
 - Mediator Pattern
- **Exercise**

Object-Oriented Patterns

Topics

- **Brief introduction to patterns**
- **Example Patterns from Gamma et al.**
 - The Observer Pattern
 - Mediator Pattern
- **Exercise**

Design Patterns

- **Aim to**
 - record experience in designing object-oriented software in a form that people can use effectively
- **Are**
 - problem description and illustrated solution
 - solution is in terms of co-operating classes and objects
 - a design vocabulary that is not as fine-grained as programming language vocabulary

The Patterns of Gamma et al.

- **Design Patterns that record good solutions to OO problems that occur often**
- **Patterns capture design expertise**
- **Gamma's Patterns deal with general OO design problems and proven solutions**
- **Patterns solve small, typical, construction problems not large architecture problems**
- **Not domain specific**

Gamma's Patterns (Essential Elements)

- **Pattern Name**: -forms basic design vocabulary
“Finding good names has been one of the hardest parts of forming the pattern catalog”
- **Problem**: describes when to apply pattern, the problem context and special conditions that need to apply to use the pattern
- **Solution**: describes the solution elements, their responsibilities, relationships, collaborations. The solution is abstract -not implementation dependent
- **Consequences**: Examines the trade-offs in applying the pattern (e.g. decoupling vs. performance). Provides guide-lines to evaluate the pattern
- **Sample Code**: C++ or Smalltalk so show possible implementations

Gamma's Patterns (Names and Classification)

Purpose

	Creational	Structural	Behavioural
Class	Factory	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Gamma's Design Patterns - Granularity

“One person's pattern is another's idiom”

- **No patterns for linked lists, stacks, hash tables, ...**
- **No domain specific patterns**
- **Not for entire applications or sub-systems (not arch.)**
- **Assumes OO language features found in
Smalltalk, C++, JAVA
So no patterns for inheritance, polymorphism, ...**
- **A pattern for Smalltalk may be trivial in C++, or vice versa**

Object-Oriented Patterns

Topics

- **Brief introduction to patterns**
- **Example Patterns from Gamma et al.**

-The Observer Pattern

-Mediator Pattern

- **Exercise**

Example: Observer Pattern

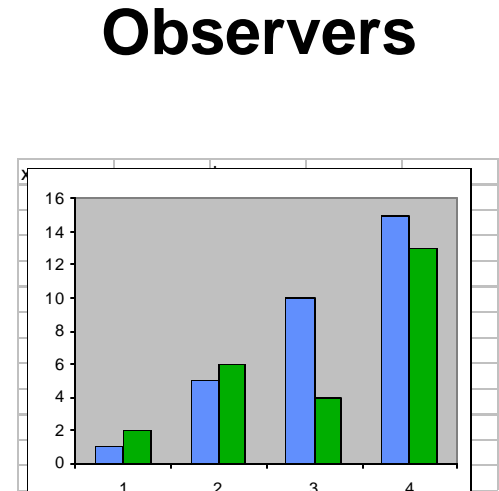
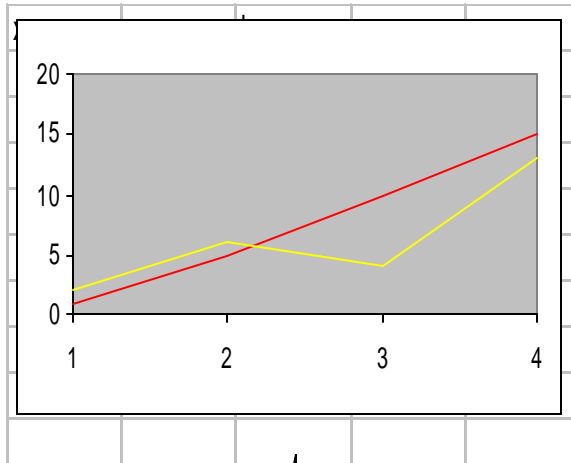
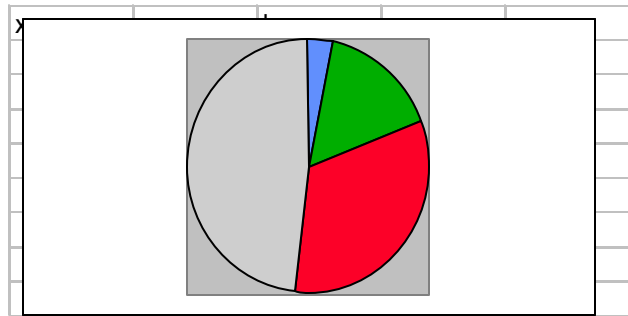
Observer Pattern:

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

(Notice the obvious usefulness to GUI applications)

Also known as: Dependents, Publish-Subscribe

Observers



Observers

Subject

x	y
1	2
5	6
10	4
15	13

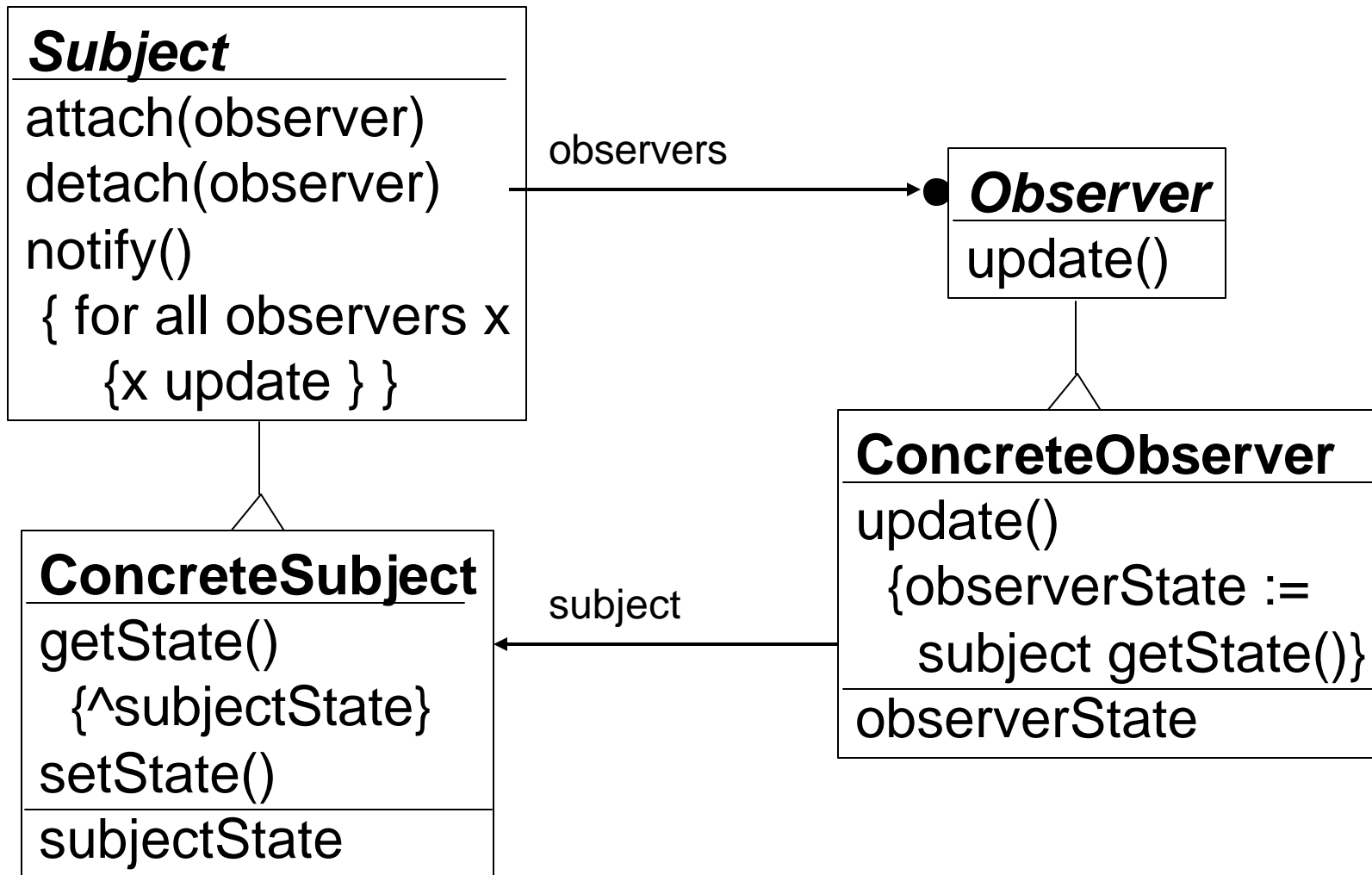
Requests,
Modifications

Change
Notification

Subjects and Observers

- **Subjects will notify their observers (dependents) whenever the subject changes state; subjects don't know or care who the observers are**
- **Observers will register interest in the subject, and query the subject for state information when notified of a state change**

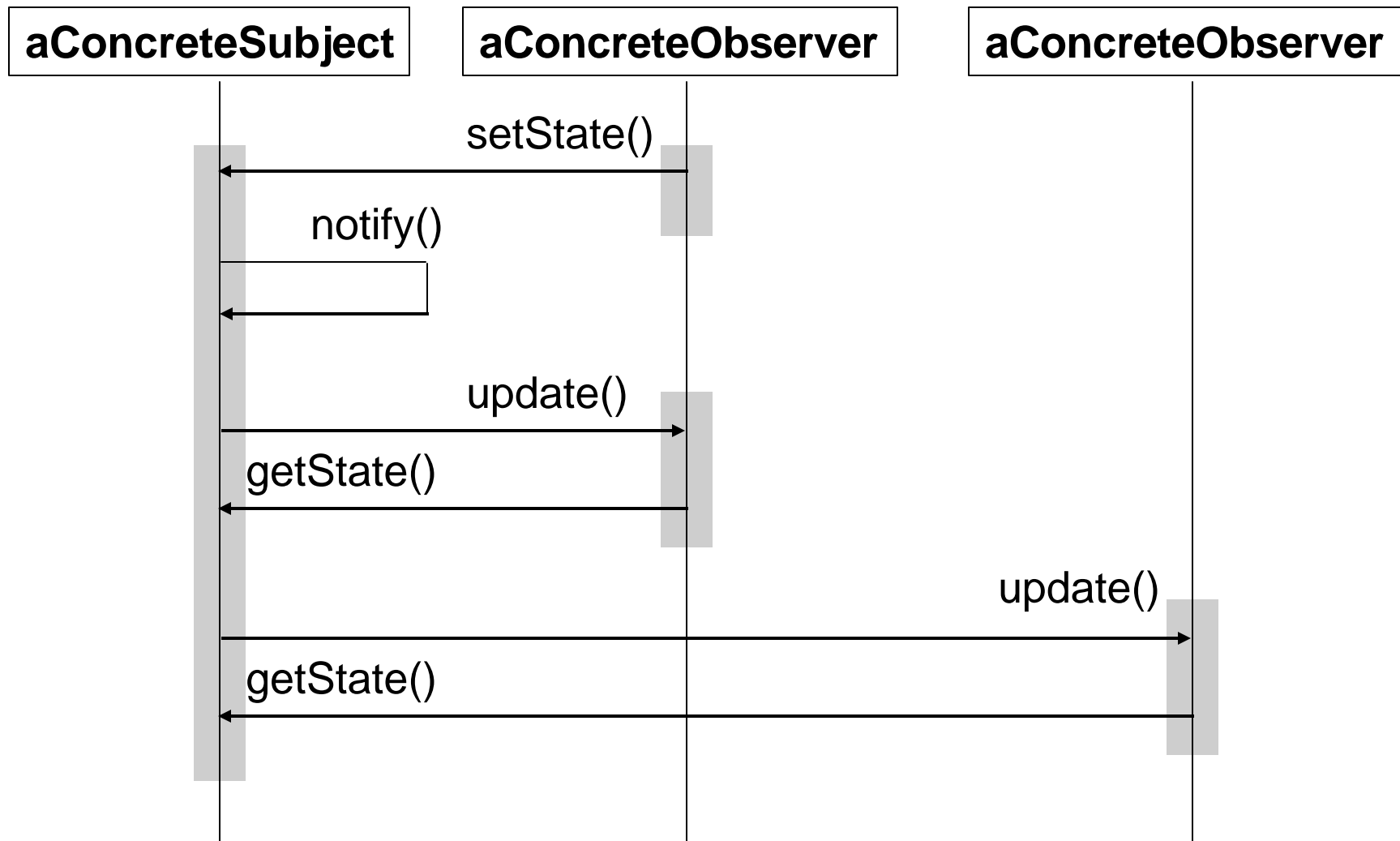
Observer Pattern (OMT Structure)



Observer Pattern -Participants

- **Subject**
 - knows it has some number of observers
- **Observer**
 - defines an updating interface for objects that should be notified of changes in a subject
- **Concrete Subject**
 - stores state of interest
 - notifies observers whenever a change occurs that could leave observer inconsistent
- **Concrete Observer**
 - maintains reference to concrete subject
 - stores state at should be consistent with subjects
 - implements the Observer updating interface to keep its state consistent with the subjects

Observer Pattern Collaborations



Observer Pattern -Consequences

- **Subjects and observers can be varied or reused independently of each other**
- **Abstract coupling between subject & observer**
 - subject only knows it has some observers
 - subject doesn't know what kind they are
- **Support for Broadcast communication**
 - notification does not have a receiver
 - it is sent to all interested parties
 - observer can be added any time
- **Unexpected Updates**
 - observers don't know about each other
 - seemingly innocent action on a subject can cause a cascade of updates
 - sensitive to spurious updates
 - aggravated because update protocol does indicate what changed

Observer Pattern -Implementation Issues

- **Mapping subjects to observers**
 - simple: keep references in subject
 - wasteful if many subjects & few observers
 - alt: keep separate subj-to-obs table
- **Observing more than one subject**
 - modify update protocol to indicate which subject is notifying
- **Who triggers notification**
 - set_method of the subject
 - observer calls notify()
- **Deleting subject must not leave dangling references in observers**
 - have subject notify observers first
 - cannot just delete observer

...Observer Pattern -Implementation Issues

- **Subject state must be self-consistent before notification**
- **Avoiding the observer-specific update protocols**
 - push model: subject sends obs details of the change with the update
 - pull model: subject sends not info, observers then query the subject
- **Letting Observers specify their aspect of interest**

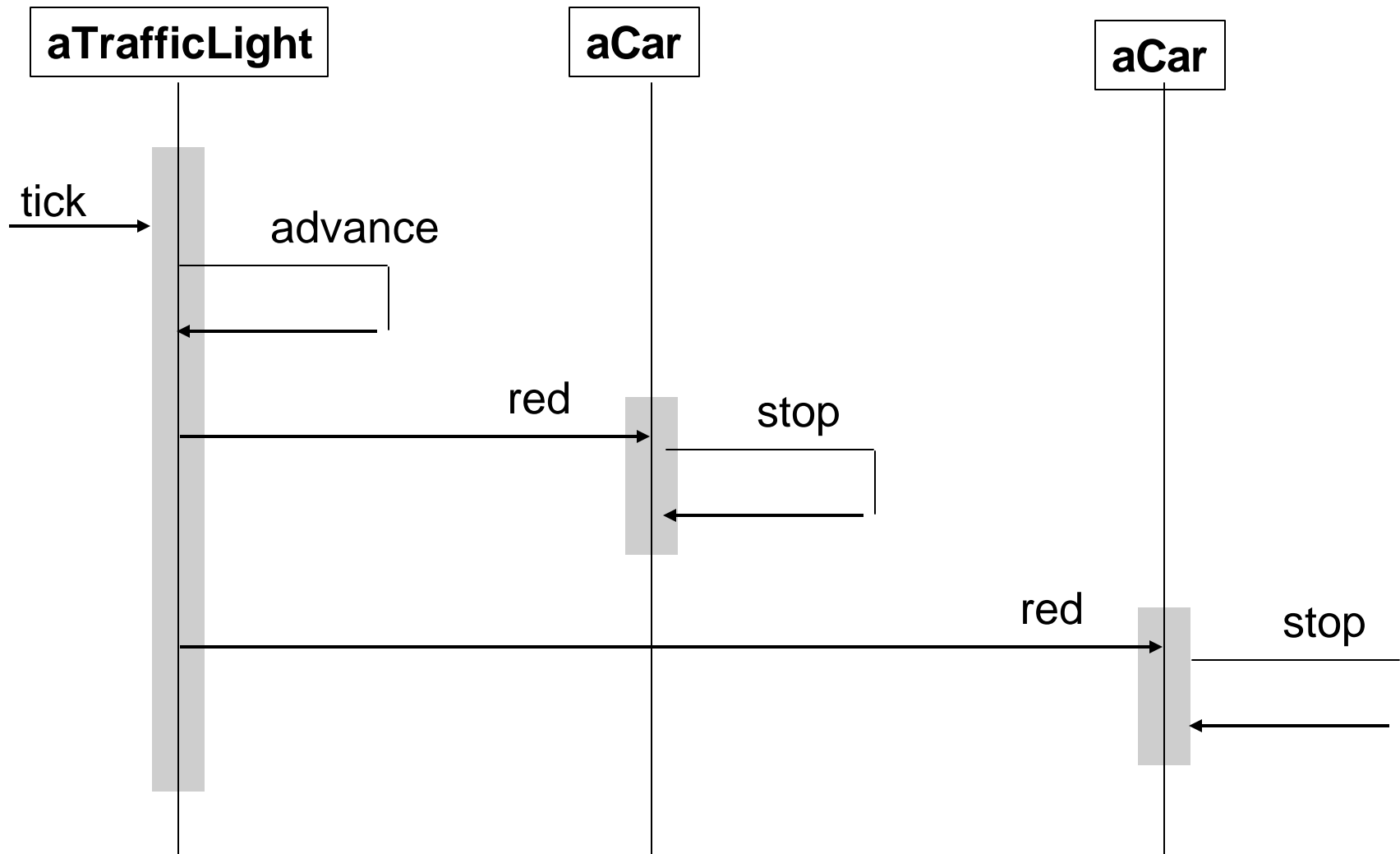
Observer Pattern -Related Patterns

Known Uses: see Gamma et al.

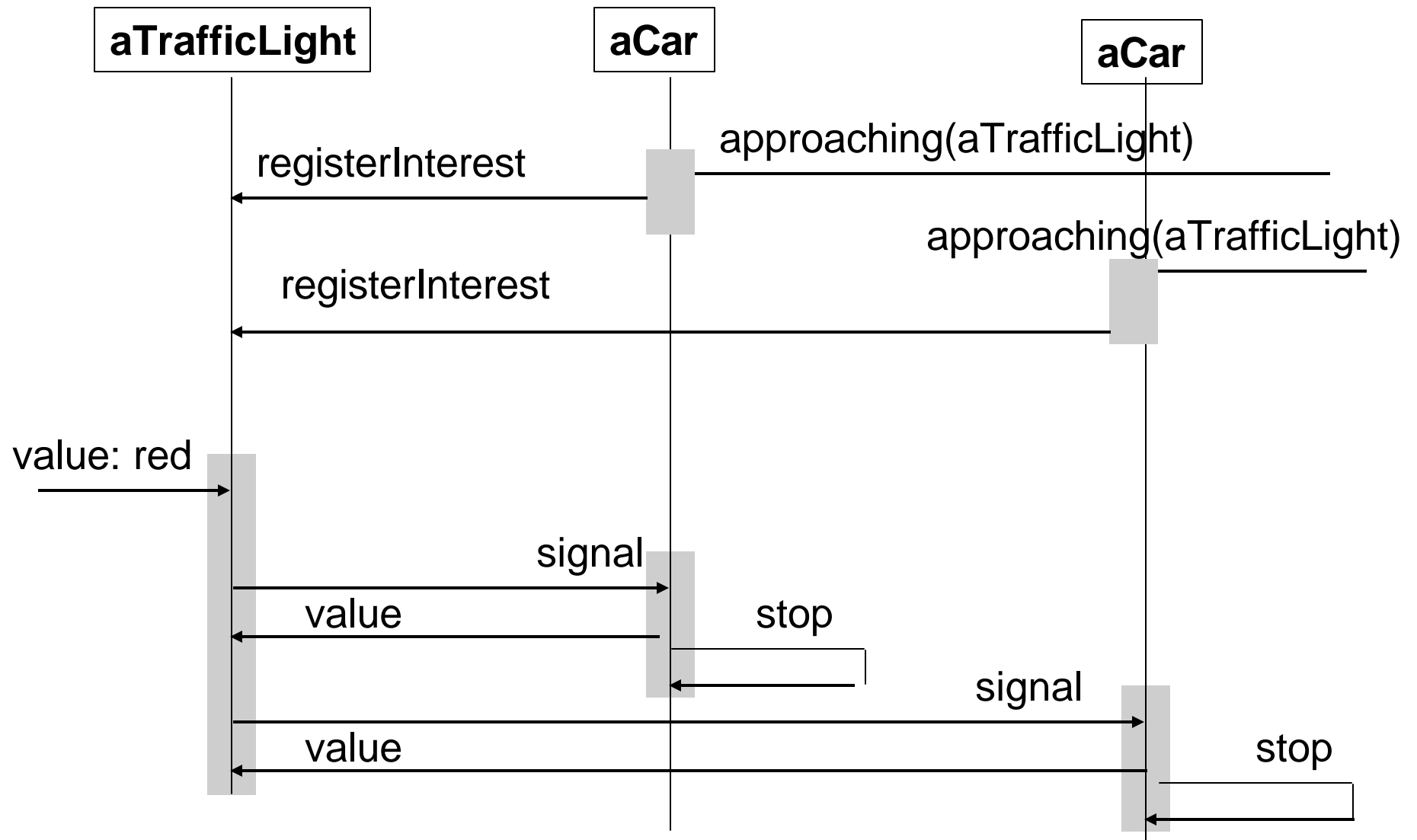
Related Patterns:

- **Mediator**
 - Can act as **Change Manager** in complex subject-observer dependencies
- **Singleton**
 - Change Manager may use a **Singleton** pattern to make it unique and global

Application Stopping for Traffic Light



Stopping for Traffic Light (as Observer Pattern)



Observer Pattern example in C++ (From Gamma et al)

```
class Subject;
class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject *theChangedSubject) = 0;
protected:
    Observer();
};
```

...Observer Pattern example in C++ (From Gamma et al)

```
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer * o);
    virtual void Detach(Observer * o);
protected:
    Subject();
private:
    List<Observer*> *_observers;
};
void Subject::Attach(Observer * o) {
    _observers->Append(o);
}
void Subject::Detach(Observer * o) {
    _observers->Remove(o);
}
void Subject::Notify(){
    ListIterator<Observer*> i(_observers);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

...Observer Pattern example in C++ (From Gamma et al)

```
class ClockTimer: public Subject { //concrete subject
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick(); //called regularly by internal timer
};

void ClockTimer::Tick() {
    //update internal time-keeping state
    // . . .
    Notify();
}
```

...Observer Pattern example in C++ (From Gamma et al)

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer *);
    virtual ~DigitalClock();
    virtual void Update(Subject *); //overrides Observer
    virtual void Draw(); //overrides Widget
private:
    ClockTimer* _subject;
};
DigitalClock::DigitalClock(ClockTimer * s){
    _subject = s;
    _subject->Attach(this);
}
DigitalClock::~~DigitalClock() {
    _subject->Detach(this);
}
```

...Observer Pattern example in C++ (From Gamma et al)

```
void DigitalClock::Update(Subject * theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw() {
    //get new values from subject
    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    int second = _subject->GetSecond();
    // . . .
    //draw the digital clock
}
```

...Observer Pattern example in C++ (From Gamma et al)

```
Class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject *);
    virtual void Draw();
    // . . .
};

//Code to create Digital and Analog Clock
ClockTimer *timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);

//Whenever the timer ticks(), the analog and digital
    clock will be updated and redraw themselves
```

Object-Oriented Patterns

Topics

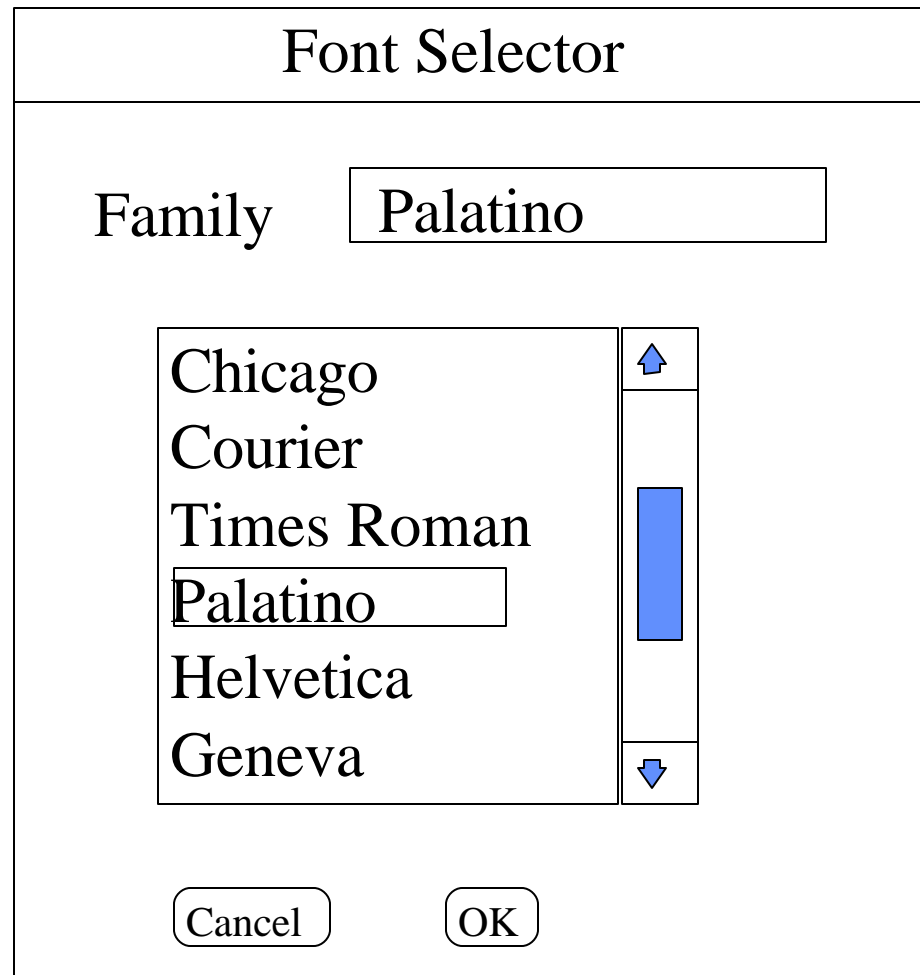
- **Brief introduction to patterns**
- **Example Patterns from Gamma et al.**

-The Observer Pattern

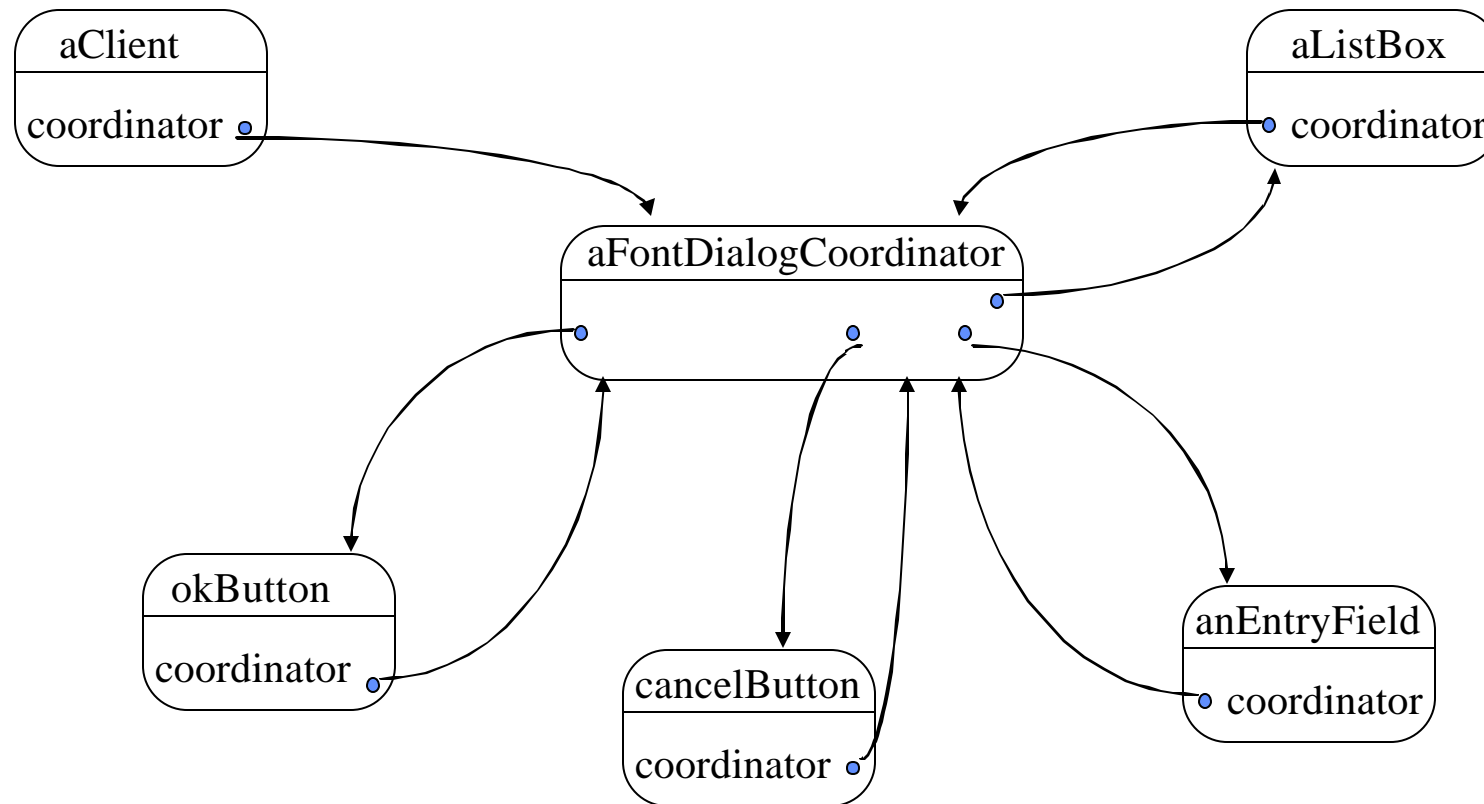
-Mediator Pattern

- **Exercise**

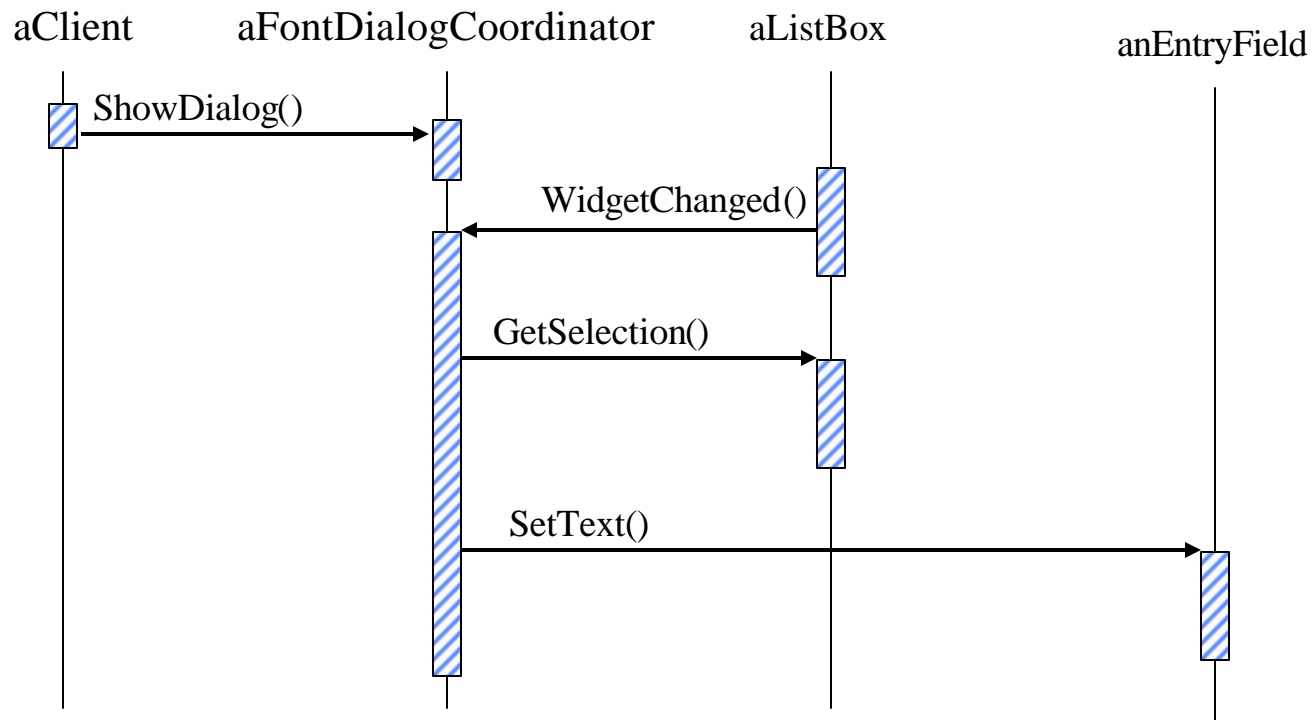
Mediator Pattern Motivation (1)



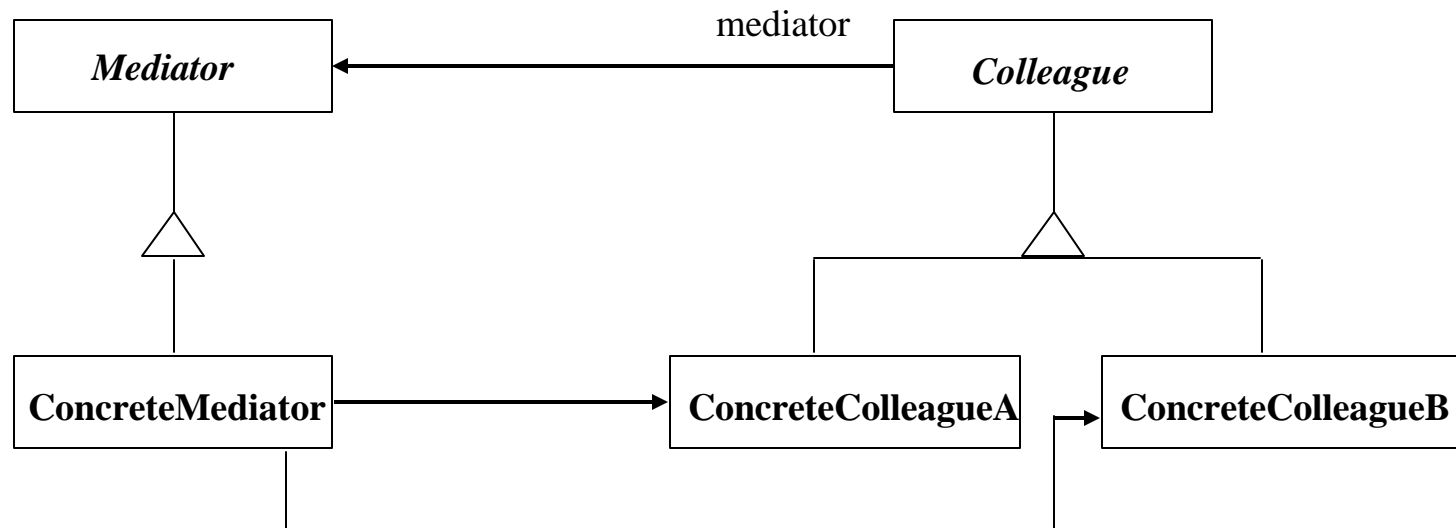
Mediator Pattern Motivation (2)



Mediator Pattern Motivation (3)



Mediator Structure



Mediator Pattern Consequences

- **limits subclassing**
- **decouples colleagues**
- **simplifies object protocols**
- **abstracts how objects collaborate**
- **centralizes control**

Mediator Sample Code (1)

```
class DialogDirector {  
public:  
    virtual ~DialogDirector();           //destructor  
    virtual void ShowDialog();  
    virtual void WidgetChanged(Widget*) = 0;  
protected:  
    DialogDirector();                   //constructor  
    virtual void CreateWidgets() = 0;  
};
```

Mediator Sample Code (2)

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();           // constructor
    virtual ~FontDialogDirector(); // destructor
    virtual void WidgetChanged(Widget*); // arg. is a pointer to Widget
protected:
    virtual void CreateWidgets();
private:
    Button* _ok;                    // attributes of this
    Button* _cancel;                // specific Dialog
    ListBox* _fontList;
    EntryField* _fontName;
};
```

Mediator Sample Code (3)

```
void FontDialogDirector::CreateWidgets () {  
  // code to create this specific Dialog  
  // this passes the current receiver to the components it builds  
  _ok = new Button(this);  
  _cancel = new Button(this);  
  _fontList = new ListBox(this);  
  _fontName = new EntryField(this);  
  
  // fill the listBox with the available font names  
  
  // assemble the widgets in the dialog  
}
```

Mediator Sample Code (4)

```
void FontDialogDirector::WidgetChanged (Widget* theChangedWidget) {  
    // this method handles a change  
    // its argument is the widget that has changed  
    if (theChangedWidget == _fontList) {  
        _fontName->SetText(_fontList->GetSelection());  
  
    } else if (theChangedWidget == _ok) {  
        // apply font change and dismiss dialog  
        // ...  
    } else if (theChangedWidget == _cancel) {  
        // dismiss dialog  
    }  
}
```

Mediator Sample Code (5)

```
class Widget {
public:
// The constructor requires being told about the DialogDirector.
// By subclass substitution, this pointer can be to any subclass
// of DialogDirector.
    Widget(DialogDirector*);
    virtual void Changed();
    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

Mediator Sample Code (6)

```
class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();    // see code in class Widget
}
```

Mediator Sample Code (7)

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);
    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);
    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
```

Object-Oriented Patterns

Topics

- **Brief introduction to patterns**
- **Example Patterns from Gamma et al.**

-The Observer Pattern

-Mediator Pattern

- **Exercise**

Exercise:

- In telephone switching there is often a call object that keeps track of the state of a call (idle, offhook, dialing, busy, ...) and services (call-forward, call-wating, busy-call-return, ...) that want to activate when certain states are reached.
- It is desirable for services to have some context to refer to so they can determine when to activate. Also services should not interfere with, or be coupled to, one another
- Sketch out a design for service objects based on the Observer and Mediator patterns -review the intent of the patterns