An Introduction to Design Patterns

(Patterns of Gamma et al.)

Object-Oriented Patterns

Topics

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

  -The Observer Pattern
  -Mediator Pattern
  -Memento Pattern
- Applications of Patterns

Object-Oriented Patterns

Topics

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

  -The Observer Pattern
  -Mediator Pattern
  -Memento Pattern

Design Patterns -Origins

- **Climate [Early 1990's]**
  - OO programming was taking a strong foothold
  - Many companies running into trouble being unable to test, debug, and bring out new product versions
  - Code size was exhibiting $O(n^2)$ code growth (n=number of types, or classes)
  - Testing was exhibiting $O(n^2)$ test case growth (n=number of types or classes)
  - OO seen as "silver bullet" was a set-back to software engineering
  - OO "Software Engineering" proposed, but did not want to follow "rules of engineering"
  - OO programmers were solving many of the same problems repeatedly, often poorly
  - Problem was worse in OO than procedure oriented programming because problem architectures became the code architectures.

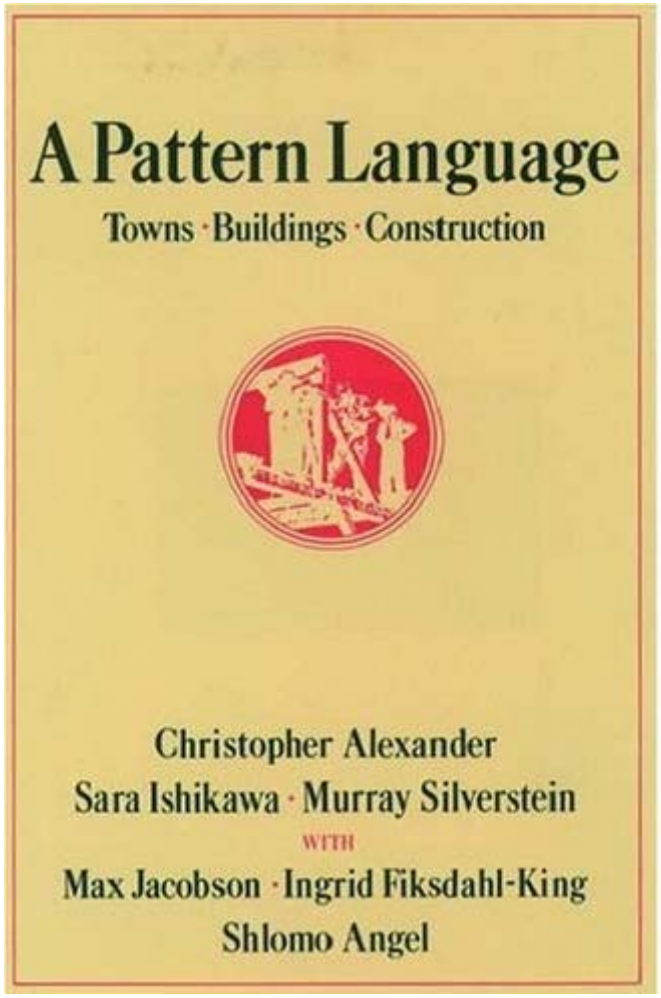
Design Patterns -Origins

- **Climate [Early 1990's]**
  - Proposal was that software should be designed and constructed the way other engineering systems were built.
  - But, Engineering, as a profession, adopts codes of conduct that outlaws practices that lead to repeated errors. (A technique that attempts to keep the same errors from re-occurring.)
  - Practitioners get licensed, or certified, and must prove that they followed established professional practices.
  - Computer Science, or programming, wanted the benefit of engineering approaches, but did not want to adopt a code of conduct (i.e. a building code)
  - It turns out Computer Science wanted to act more like architects (of buildings) than the civil engineers (that must approve the compliance to code).
  - But they still wanted to call it "Software Engineering"

Design Patterns -Origins

- **Climate [Early 1990's]**
  - What guidance exists regarding how architects deal with re-occurring problems.
  - In 1977 Christopher Alexander published an influential book: "A Pattern Language".
  - Suggested patterns that could serve a re-usable solutions of re-occurring problems in Architecture.
  - It was a collection of suggested, optional, patterns –it was not a building code.

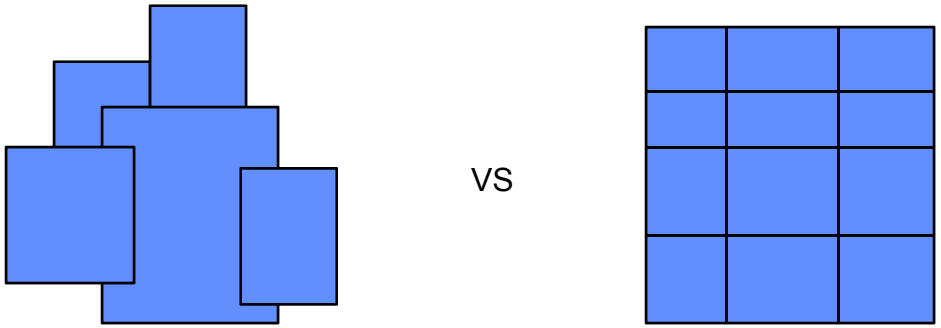
Modelled after: Christopher Alexander's Design Patterns.



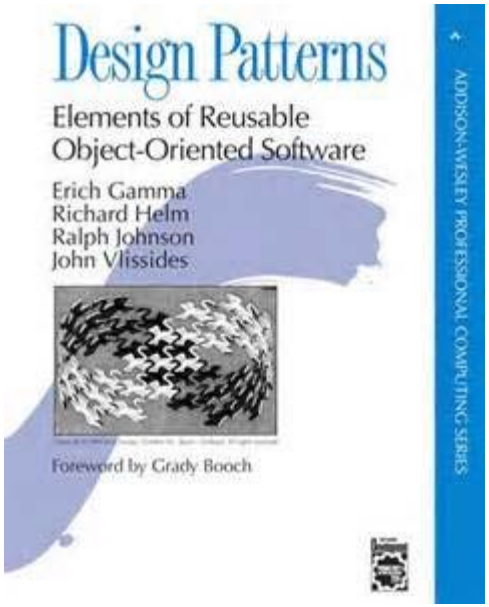
- ***A Pattern Language: Towns, Buildings, Construction*** **is a 1977 book on architecture, urban design, and community livability.**
- **253 patterns**
- **Patterns solve problems that occur in architecture from parts of a room to entire communities**
- **Re-usable with discussion of trade-offs**
- **Have anti-patterns**

Alexander Pattern #253

- **Pattern Name:** “Light On Two Sides of Every Room"
- **Problem:** Observation that people strongly prefer rooms with windows (light) on more than one wall). They will “gravitate” to rooms that are lit on more than one side when there is a choice.
- **Solution:** Create structure so many rooms can have exterior walls on more than one side
- **Consequences:** Discussion of how it co-ordinates and interferes with other patterns
- **Sample :** Expressed in form of blue-print or sketch

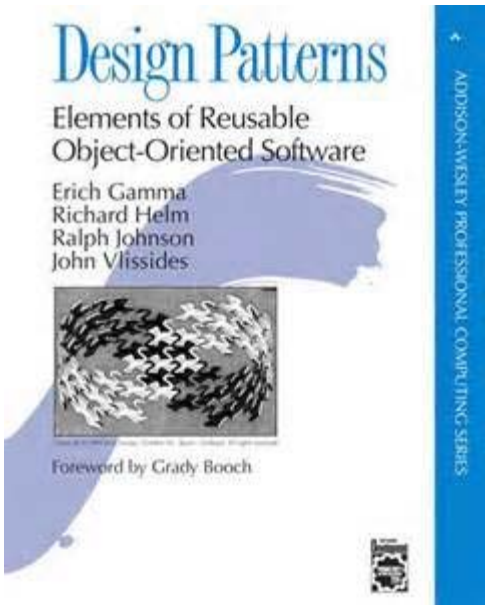


Design Patterns -Origins



- **Climate [Early 1990's]**
  - OO programming researchers Gamma et al adopted Alexander's work as a template and in 1994 published "Design Patterns"
  - A collection of 23 patterns that solve typical and re-occurring problems in object-orient programming
  - It went on to be come the best selling book ever written about object-oriented programming and has been extremely influential.

The Patterns of Gamma et al. [1994]



- **23 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**
- **Have anti-patterns**

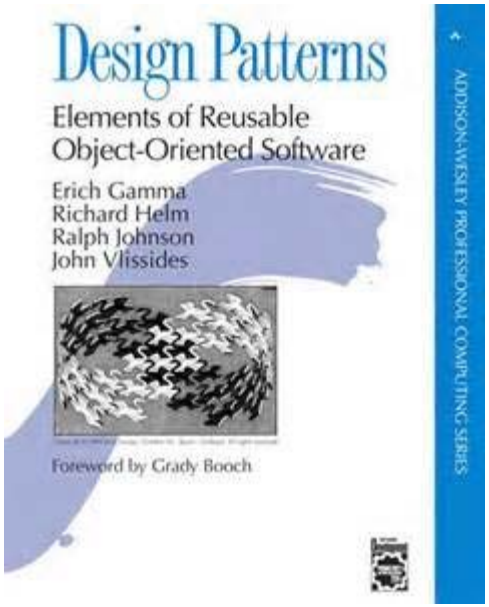
Gamma's Patterns (Essential Elements)

- **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.

Design Patterns

- **Aim**
  - record experience in designing object-oriented software in a form that people can use effectively re-use to solve their own problems
  - To serve as a alternative to “building codes” found in engineering practices.
  - To provide a design vocabulary above the programming language level and suitable for OO programming
- **Are**
  - problem description and illustrated solution with examination of trade-offs
  - solution expressed in terms of co-operating classes and objects
  - a design vocabulary that is not as fine-grained as programming language vocabulary
  - -domain independent

The Patterns of Gamma et al. [1994]



- **The gamma patterns were discovered, not invented.**
- **The rule was that it must be found in existing code and more than once.**
- **They have anti-patterns but the importance of having anti-patterns was not emphasized, or understood, at the time.**
- **(anti-patterns are patterns for poor solutions.)**
- **Emphasis is on decoupling and encapsulation which address the n^2 code explosion problem, often at the cost of performance.**

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 an architecture.)**

- **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**

Why Can't This Be a Pattern?

- **Pattern Name:** -"Door Is Not Ajar"
- **Problem:** The class is over and you want to leave but the door is shut.
- **Solution:** The first one to arrive at the door opens it allowing themselves and others to leave.
- **Consequences:** Nobody can leave until someone has opened the door. Door might get left open.
- **Sample Code**: none shown.

Why Can't This Be a Pattern

- **Pattern Name:** -"Door Is Not Ajar"
- **Problem:** The class is over and you want to leave but the door is shut.
- **Solution:** The first one to arrive at the door opens it allowing themselves and others to leave.
- **Consequences:** Nobody can leave until someone has opened the door. Door might get left open.
- **Sample Code**: none shown.

- A pattern must have an anti-pattern (a bad way of solving the problem) that is sufficiently likely.

Why Can't This Be a Pattern

- **Pattern Name:** -"Save The World "
- **Problem:** There is a Q137 thermo-nuclear bomb in the classroom lecture cabinet and the timer is ticking with 3 min to explosion that will destroy the campus.
- **Solution:** Rotate the detonator hatch cover 30 degrees counter clockwise, open it then snip the wires in the order: green, yellow, then red.
- **Consequences:** Getting the wire order wrong will cause bomb to explode.
- **Sample Code**: none shown.

Why Can't This Be a Pattern

- **Pattern Name:** -"Save The World "
- **Problem:** There is a Q137 thermo-nuclear bomb in the classroom lecture cabinet and the timer is ticking with 3 min to explosion that will destroy the campus.
- **Solution:** Rotate the detonator hatch cover 30 degrees counter clockwise, open it then snip the wires in the order: green, yellow, then red.
- **Consequences:** Getting the wire order wrong will cause bomb to explode.
- **Sample Code**: none shown.

- A pattern must be useful for solving other problems, not just the one used to illustrate it.

Object-Oriented Patterns

Topics

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

-The Observer Pattern

-Mediator Pattern

-Memento Pattern

- Exercise

Example: Observer Pattern

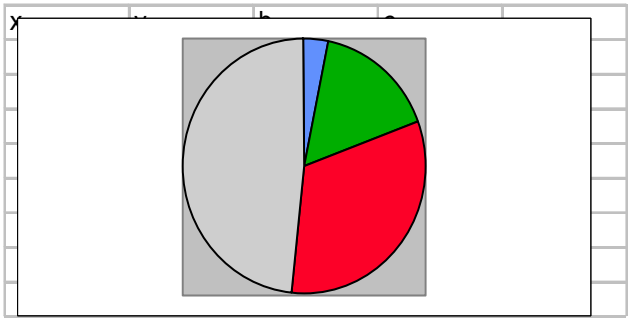
Observer Pattern:

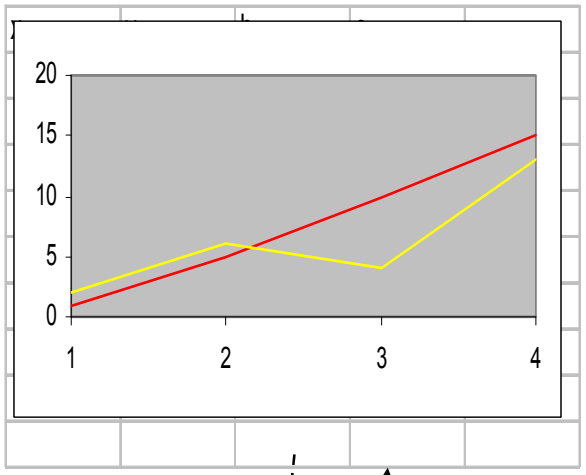
Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

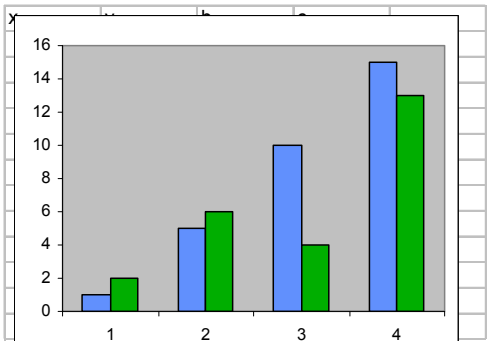
(Notice the obvious usefulness to GUI applications)

Also known as: Dependents, Publish-Subscribe

Observers







Observers

Requests,
Modifications

Change
Notification

x	y
1	2
5	6
10	4
15	13

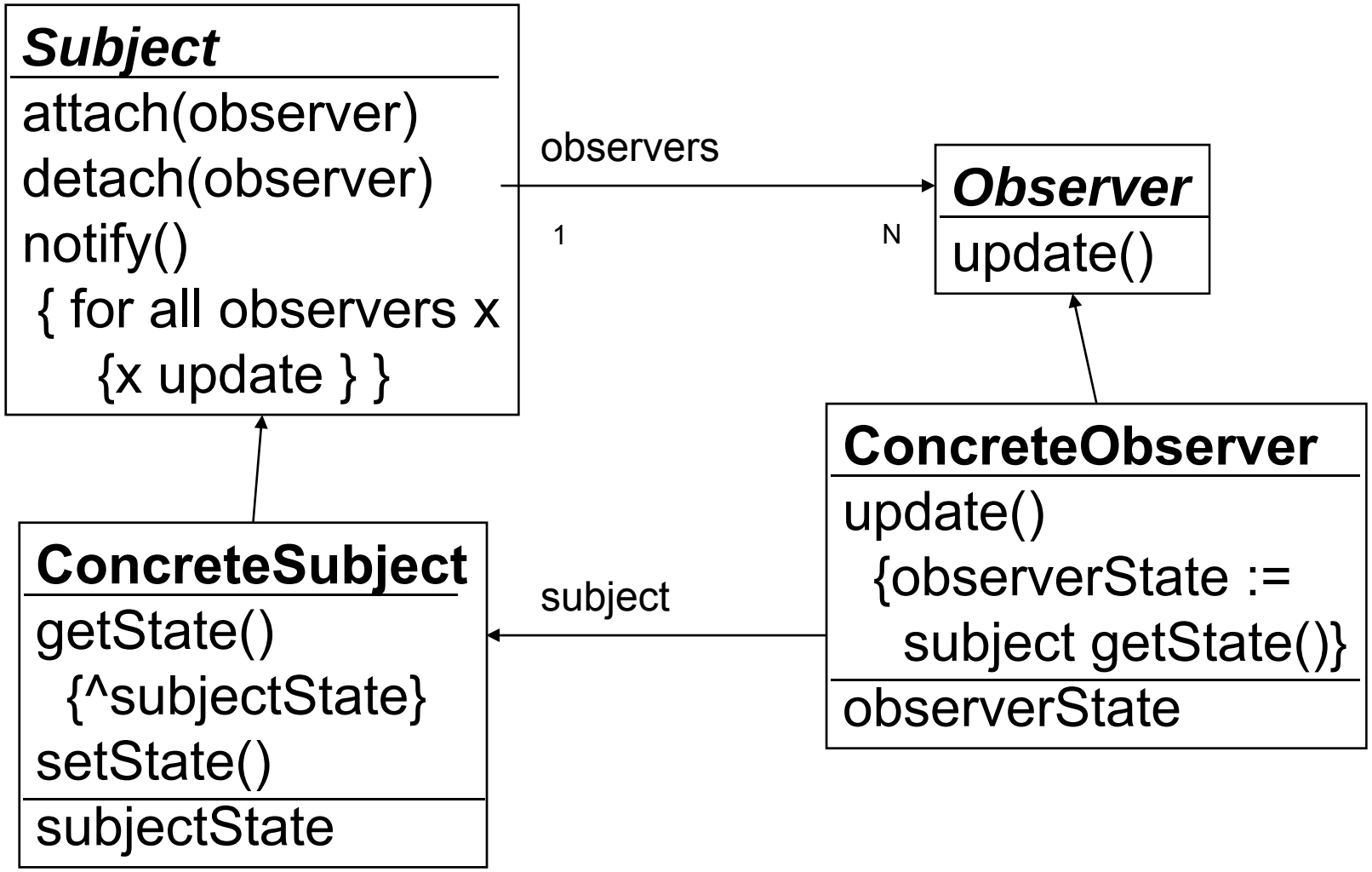
Subject

Subjects and Observers

- **Subjects will notify their observers (dependents) whenever the subject changes state; subjects don't know, or care, who the observers are and what they are watching for.**

- **Observers will register interest in the subject, and query the subject for current state information when notified of a state change**

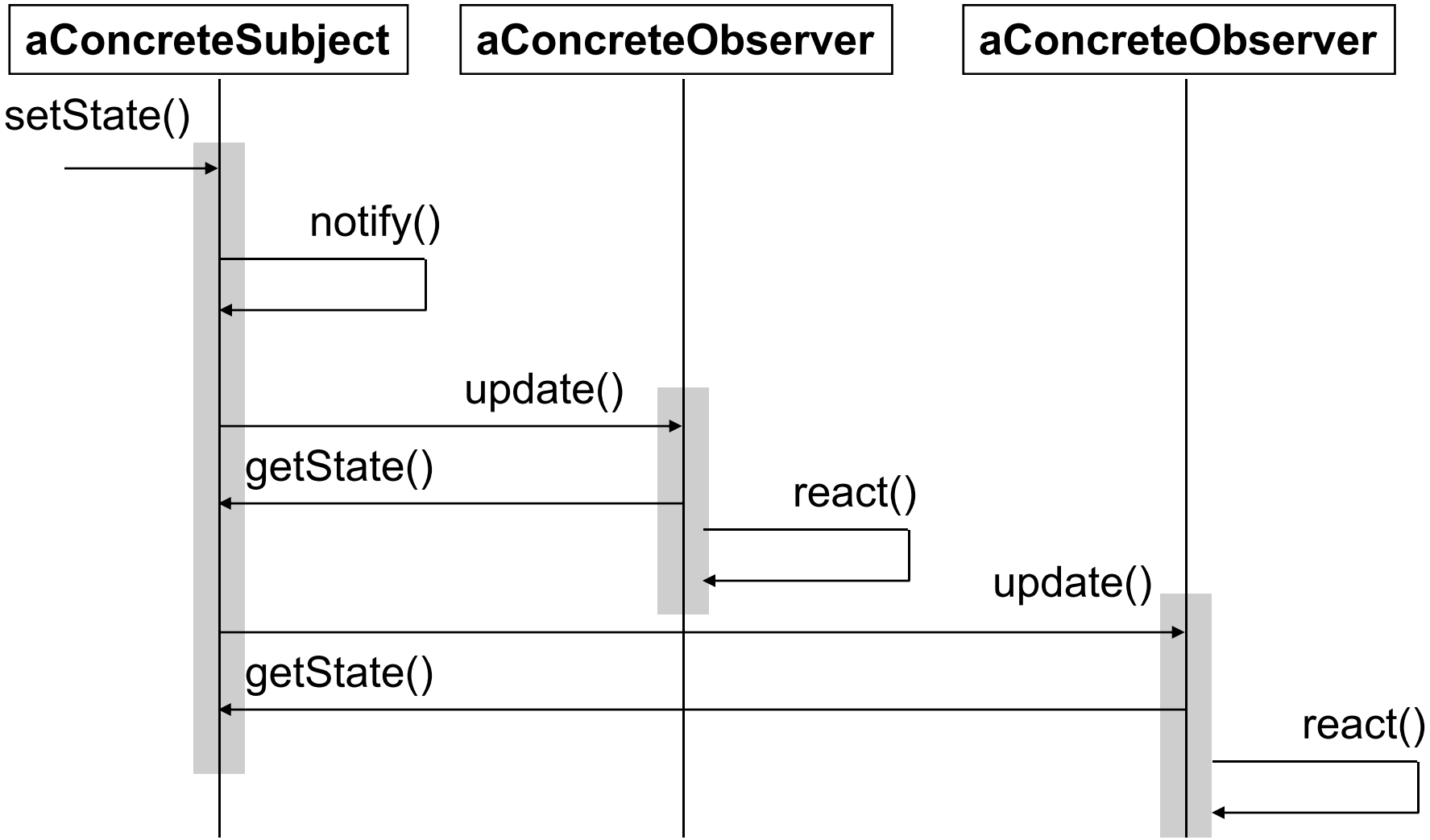
Observer Pattern (OMT Structure)



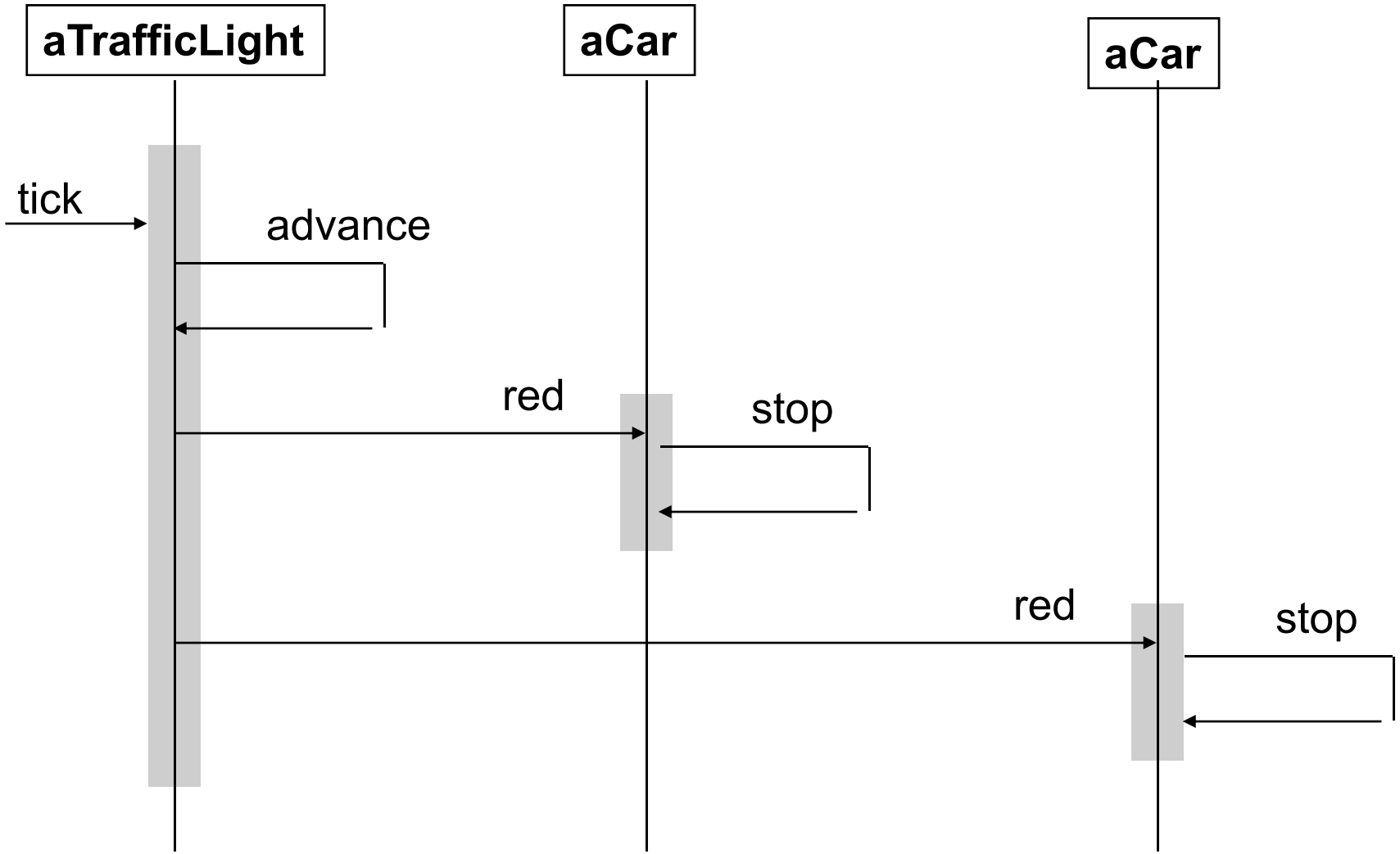
Observer Pattern -Participants

- **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**

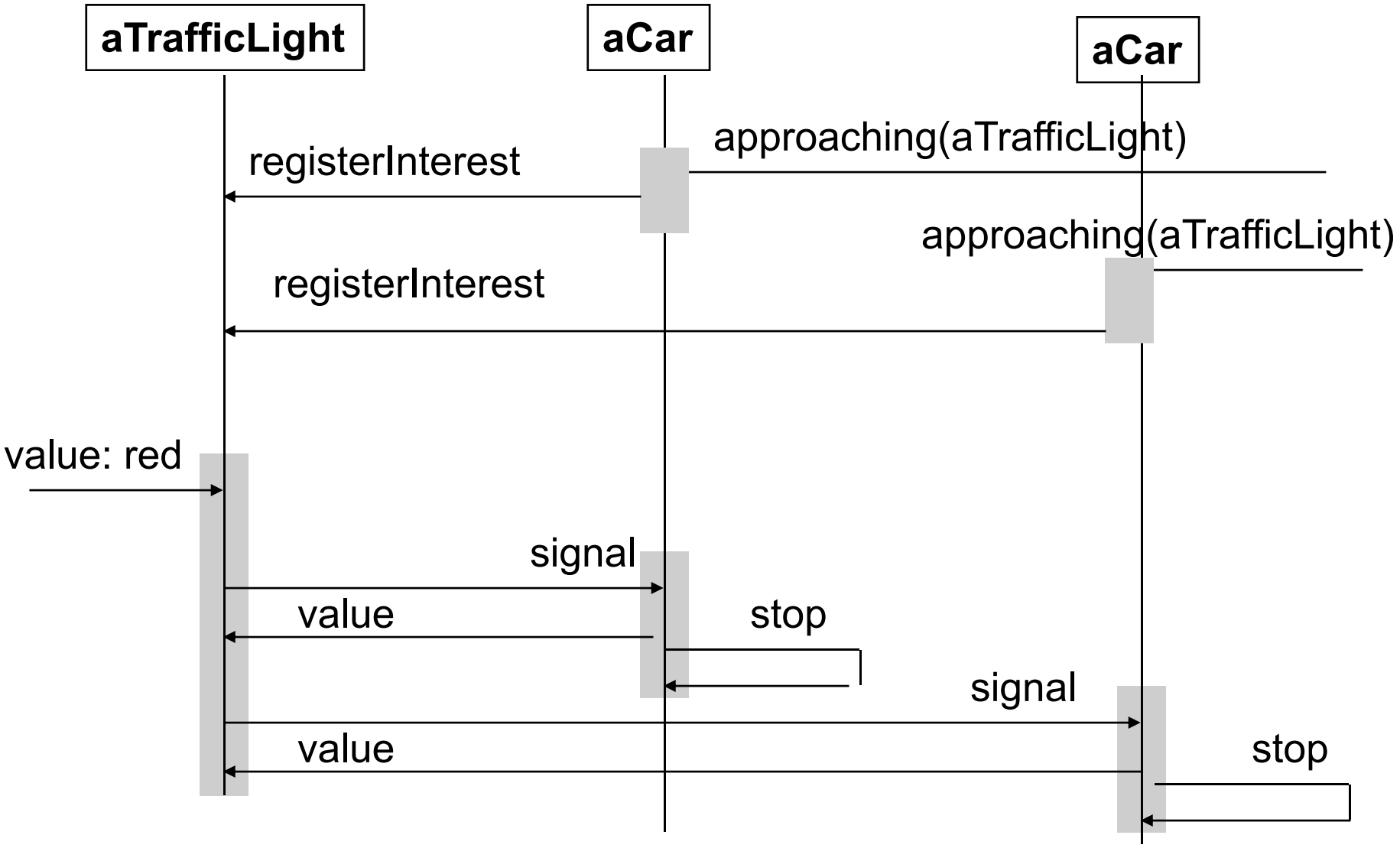
Observer Pattern Collaborations



Application Stopping for Traffic Light



Stopping for Traffic Light (as Observer Pattern)



Gamma Example

Observers

Update()

getState()

Tick()

Time and Data Object
Current time:
Current date:

Subject

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 *);
  virtual void Detach(Observer *);
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
```

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 not 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 observer details of the change with the update

  -pull model: subject sends no info, observers then query the subject

- Letting Observers specify their aspect of interest

- (What if observer detaches as a result of being notified?)

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

Object-Oriented Patterns

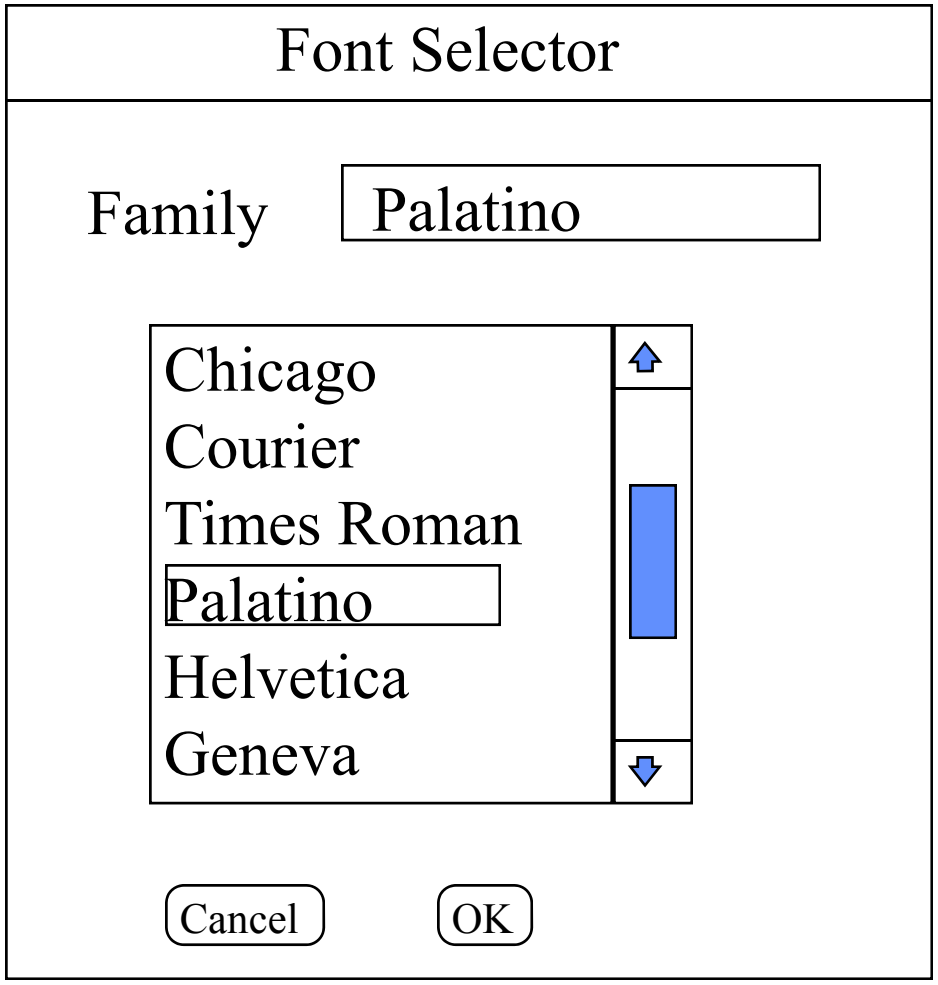
Topics

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

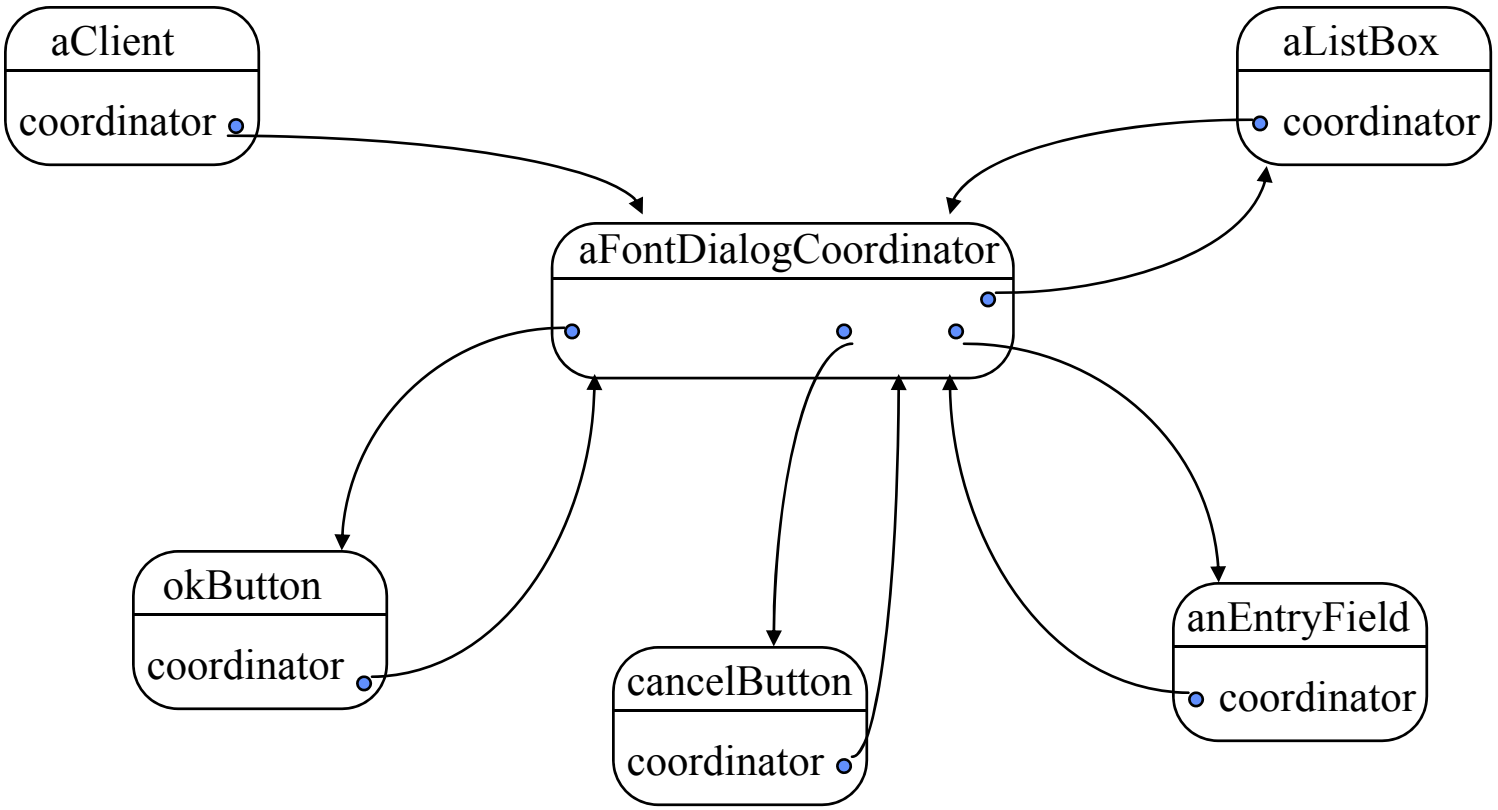
  -The Observer Pattern

  -Mediator Pattern
- Exercise

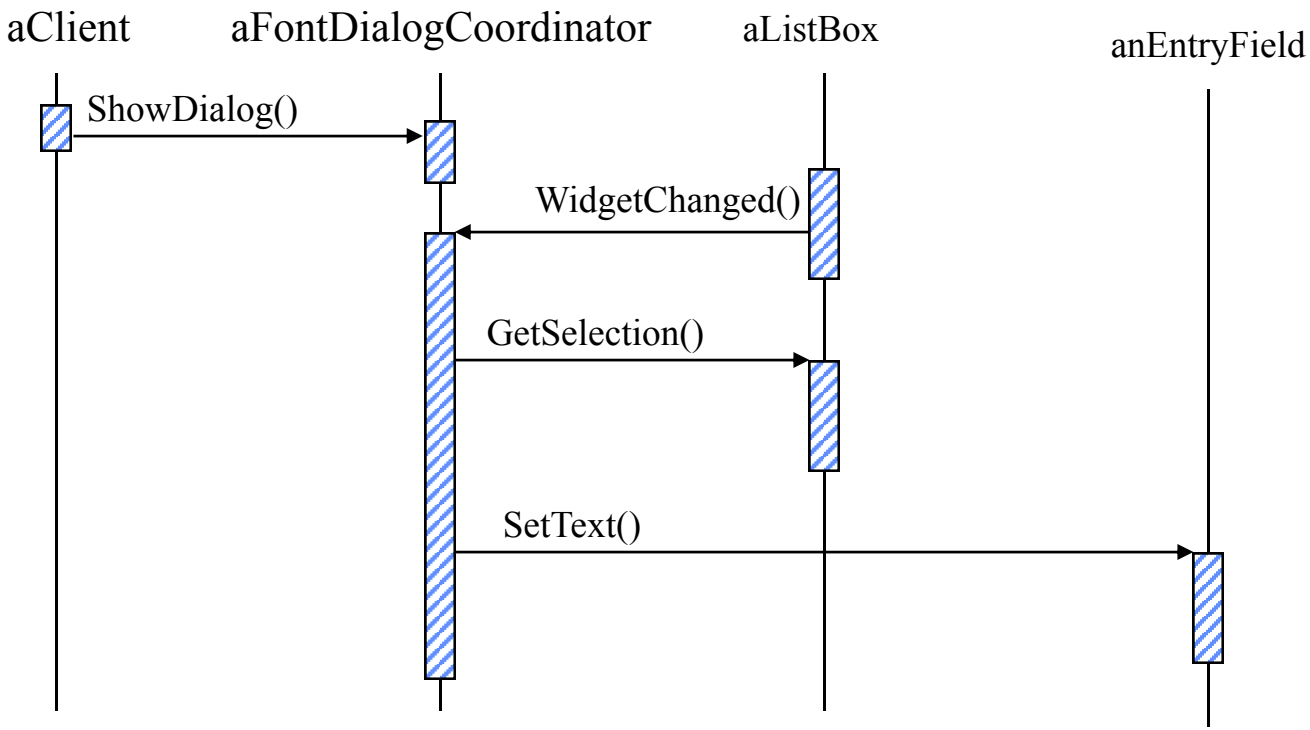
Mediator Pattern Motivation (1)



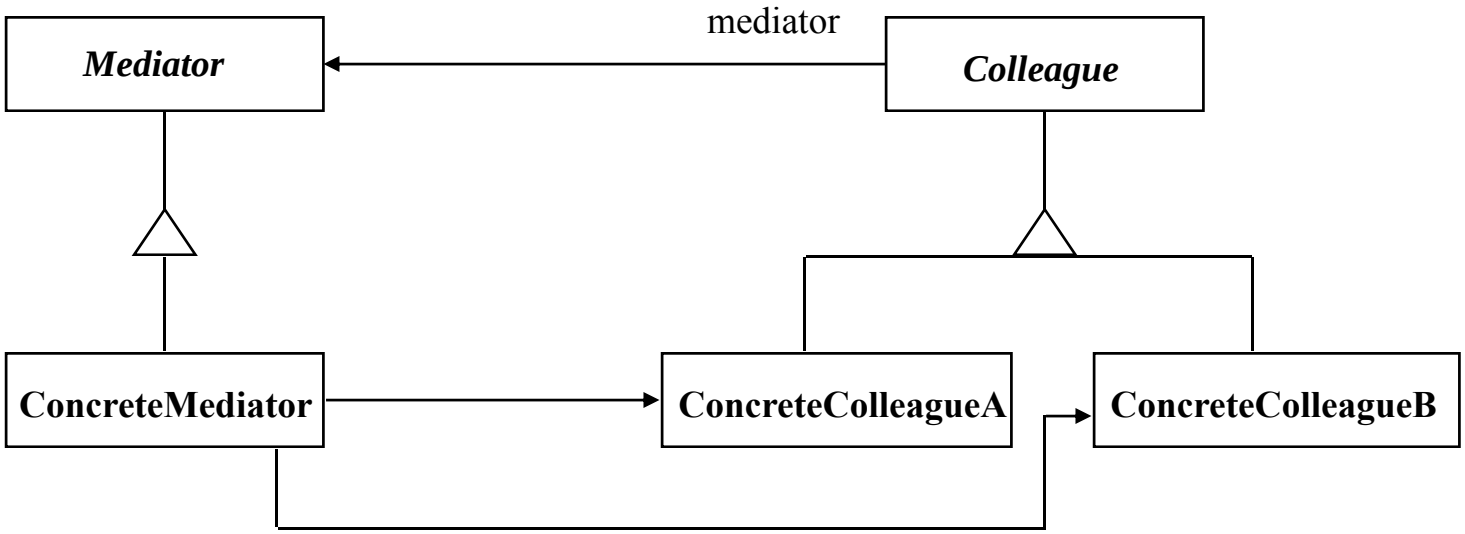
Mediator Pattern Motivation (2)



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:                // attributes of this
    Button* _ok;        // specific Dialog
    Button* _cancel;
    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 must passed the DialogDirector.
// but 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 –Applying these two patterns

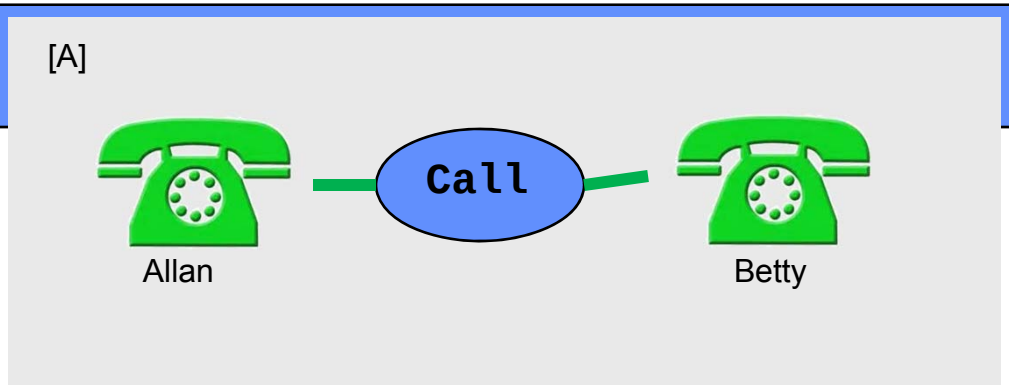
Exercise (Industry Example)

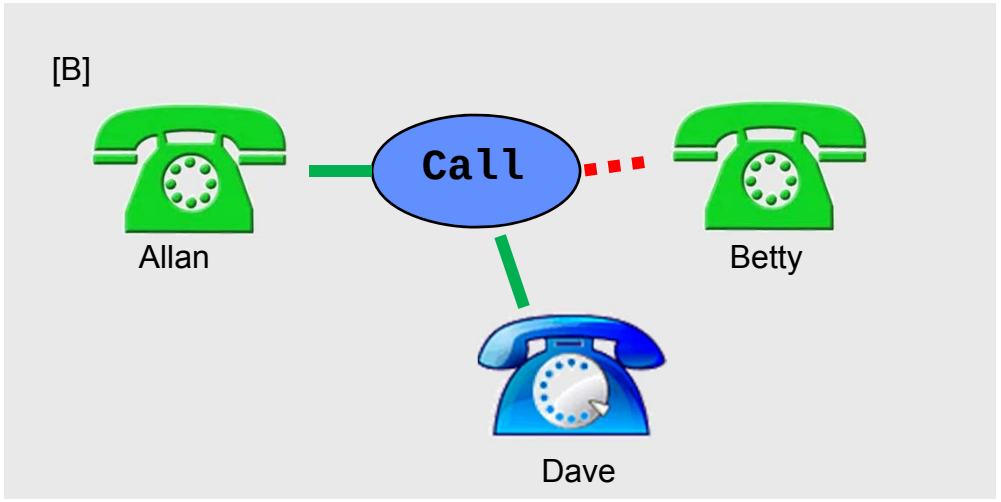
- **In telephone switching there is often a call object that keeps track of the state of a call(idle, offhook, dialing, busy, …) and services (call-forward, call-wating, busy-call-return, …) that want to activate when certain states are reached or certain events (off-hook, digits, flash) occur.**

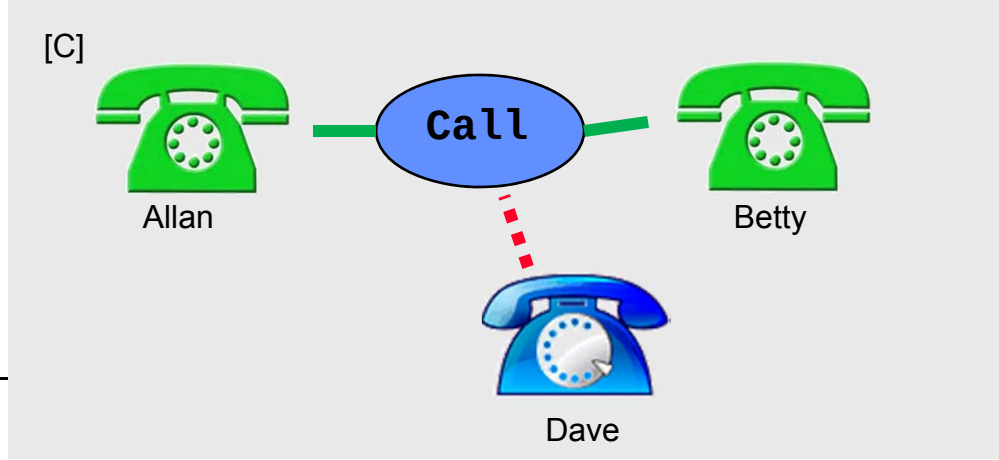
- **It is desirable for services to have some context to refer to so they can determine when to activate. Also services should not interfere with, or be coupled to, one another**

- **Sketch out a design for service objects based on the Observer and Mediator patterns -review the intent of the patterns**

Service Interaction (Good)



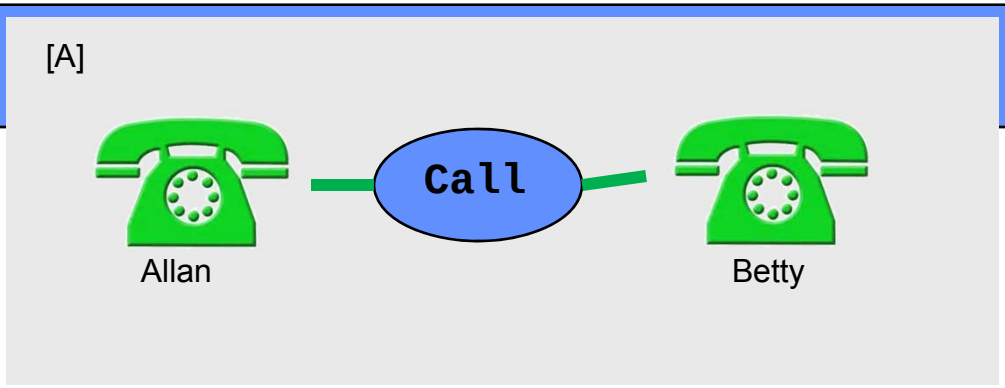


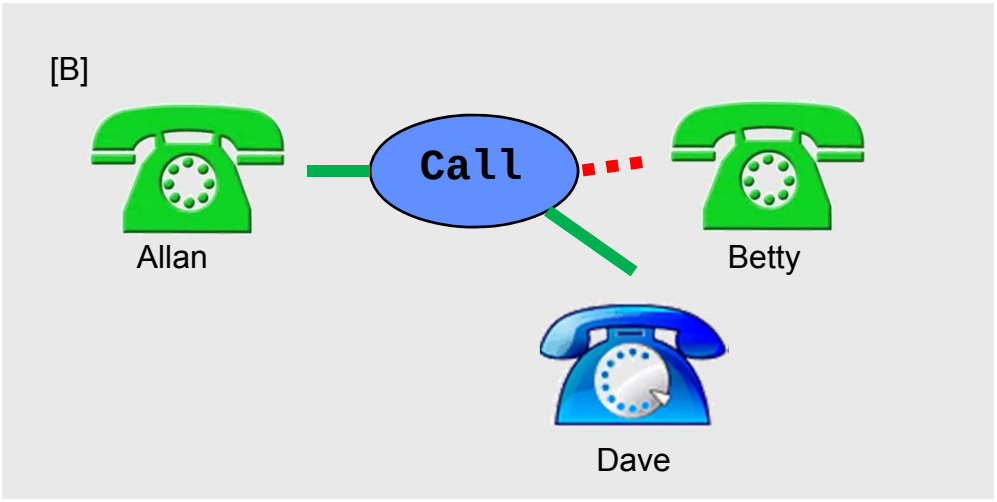


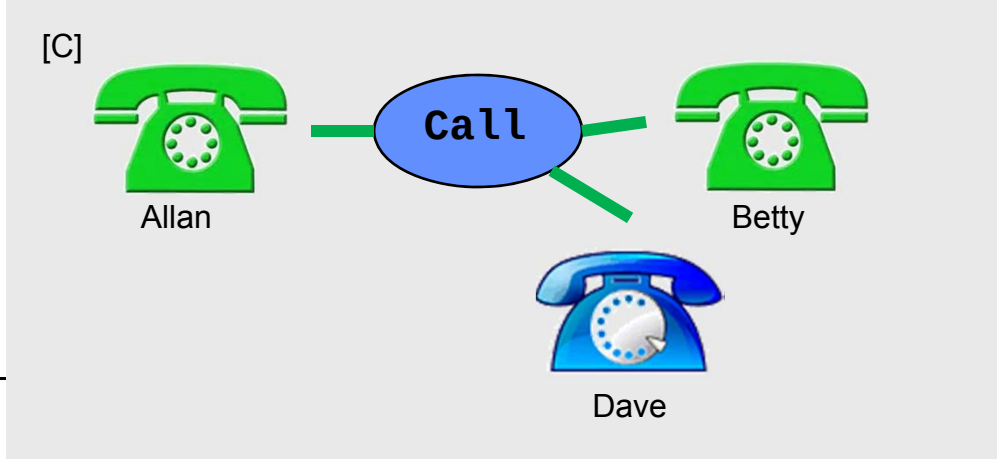
Scenario (Call Waiting):

- **Allan calls Betty [A]**
- **While they are talking Dave calls Allan**
- **Allen hears a "beep" because he subscribes to call-waiting service**
- **Allen flashes (links)**
- **The Allan-Betty call is put on hold and Allan can talk to Dave [B]**
- **Allen flashes (links)**
- **Allan-Betty call is re-connected and Dave is on put hold [C]**
- **Dave hangs up**

Service Interaction (Good)



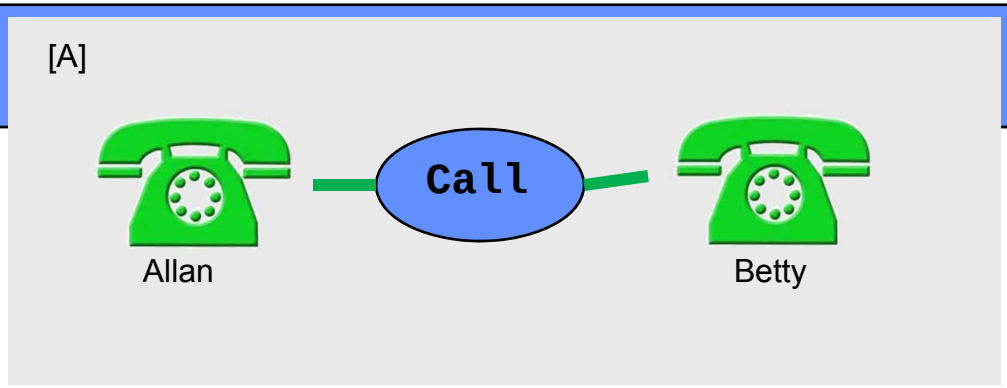


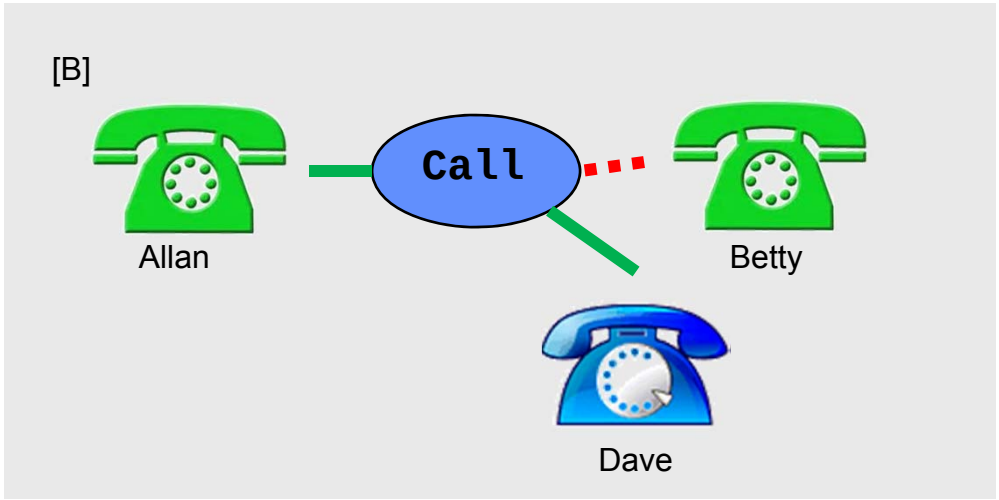


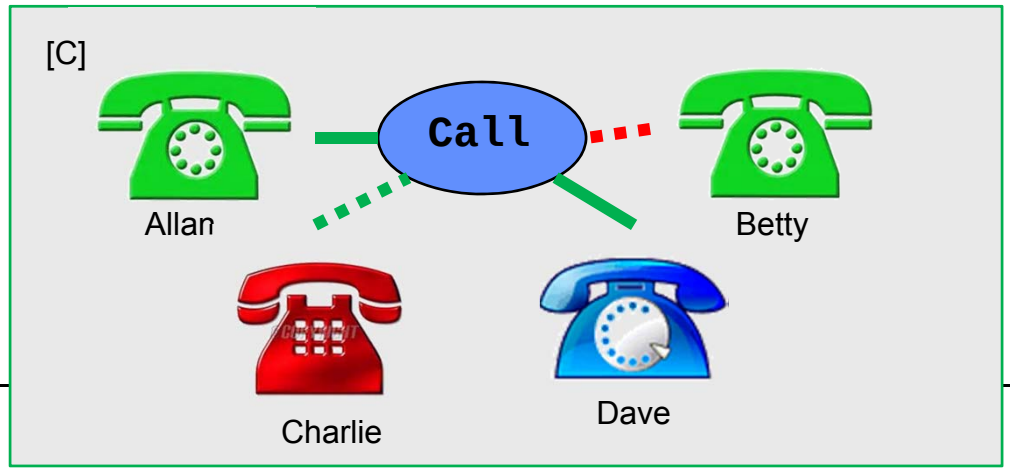
Scenario (3-Way Call, Call Waiting):

- **Allan calls Betty [A]**
- **Allan flashes**
- **Betty is put on hold, Allan gets second dialtone and calls Dave. [B]**
- **Asks Dave if he wants to join a 3-way call with Betty**
- **Allan flashes (links)**
- **Allan, Betty, and Dave are conferenced together in a 3-way call [C].**

Service Interaction (Bad)



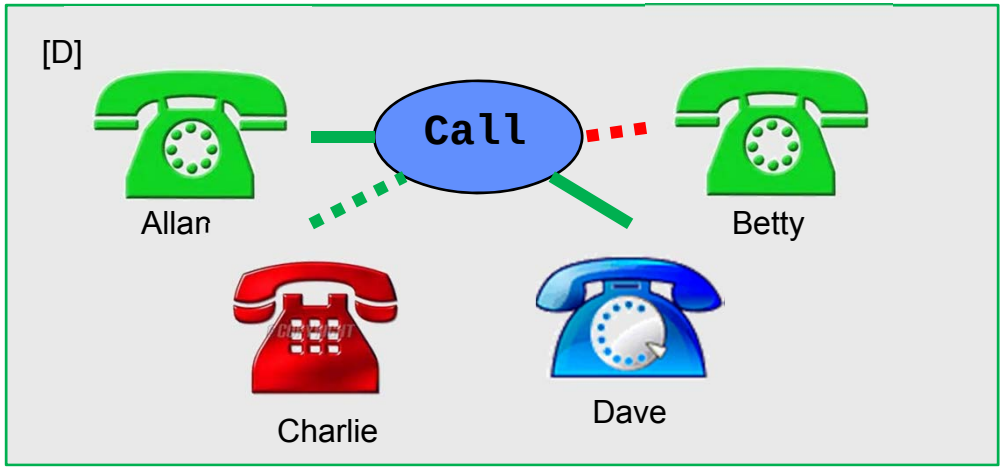


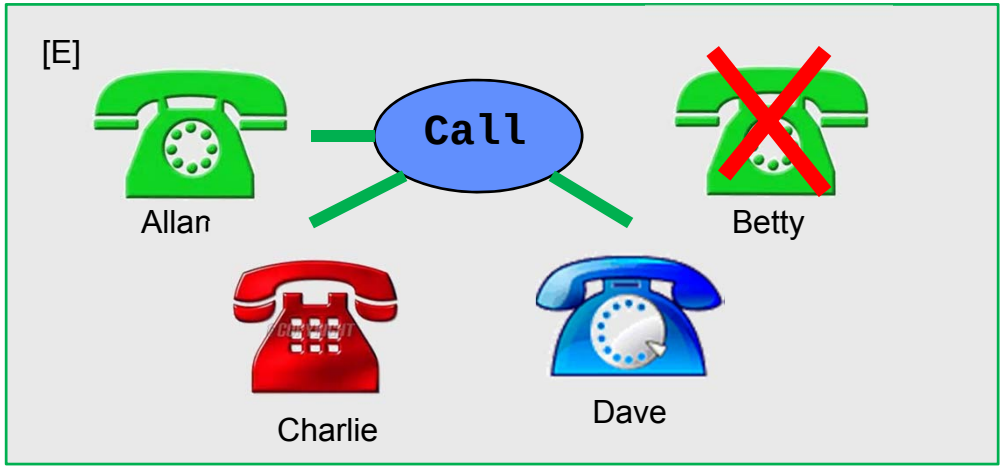


Scenario (3-Way Call, Call Waiting):

- **Allan calls Betty [A]**
- **Allan flashes (links)**
- **Betty is put on hold, Allan gets second dialtone and calls Dave. [B]**
- **Allan asks Dave if he wants to join a 3-way call with Betty**
- **While Allan is talking to Dave, Charlie calls Allan [C]**
- **Allan hears a "beep" because he subscribes to call-waiting service.**

…Service Interaction (Bad)

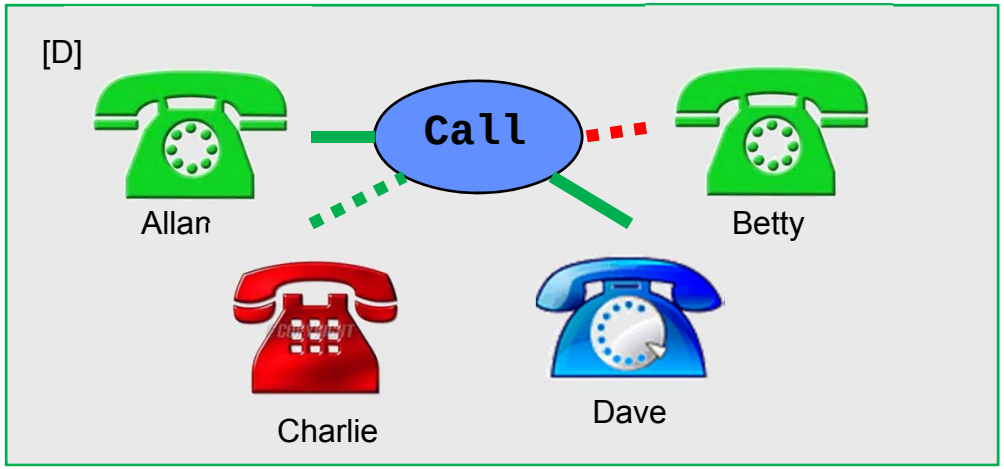


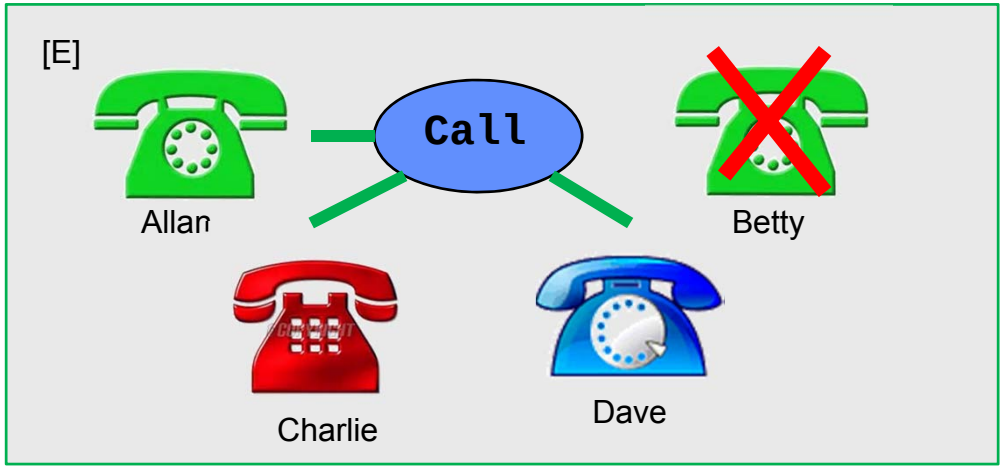


Scenario (3-Way Call, Call Waiting):

- **Allan flashes (links)**
- **Allan is connected to Charlie, Dave and Betty are on hold [D]**
- **Allan is confused because his 3-way call did not link and is talking to Dave.**
- **Allan flashes**
- **Allan, Chalie, and Dave are conferenced together in a 3-way call, and Betty is dropped from the call.**

…Service Interaction (Bad)





Scenario (3-Way Call, Call Waiting):

- **This bad interactions was not found by the programmers or testers.**
- **It was found, and reported, by the customers.**
- **The code had "evolved" to the point where these problems could not be caught during development and testing.**

OLD Code Structure (Anti-Pattern)

Call

```
Event event;
State state;
handle_event(Event e){
…
if(three_way_call == running)
    three_way_call_handle(e);
if(call_waiting == running)
   call_waiting_handle(e);
if(message_answer == running)
  message_answer_handle(e);
…
}
```

Three Way Call Service

```
Event event;
State state;
three_way_call_handle(Event e){
…
if(call_waiting == running)
    deny_service();
if(call_forward == running){
   call_forward(e);
   make_3waycall(e);
}
if(message_answer == running)
  deny_service(e);
…
}
```

Given N Services:
Call object is O(N) code complexity
Each Service Object is O(N) code complexity
Overall: $O(N^2)$ Code complexity or worse
$O(N^2)$ Testing complexity or worse

OLD Code Structure (Anti-Pattern)

Call

```
Event event;
State state;
handle_event(Event e){
…
if(three_way_call == running)
    three_way_call_handle(e);
if(call_waiting == running)
   call_waiting_handle(e);
if(message_answer == running)
  message_answer_handle(e);
…
}
```

Three Way Call Service

```
Event event;
State state;
three_way_call_handle(Event e){
…
if(call_waiting == running)
    deny_service();
if(call_forward == running){
   call_forward(e);
   make_3waycall(e);
}
if(message_answer == running)
  deny_service(e);
…
}
```

That is:
The Call object's code had to worry about all the possible services that could run, and each service had to worry about what all the other services might be doing. Code complexity was at least $O(n^2)$ for n possible services.

OLD Code Structure (Anti-Pattern)

Call

```
Event event;
State state;
handle_event(Event e){
…
if(three_way_call == running)
    three_way_call_handle(e);
if(call_waiting == running)
   call_waiting_handle(e);
if(message_answer == running)
  message_answer_handle(e);
…
}
```

Three Way Call Service

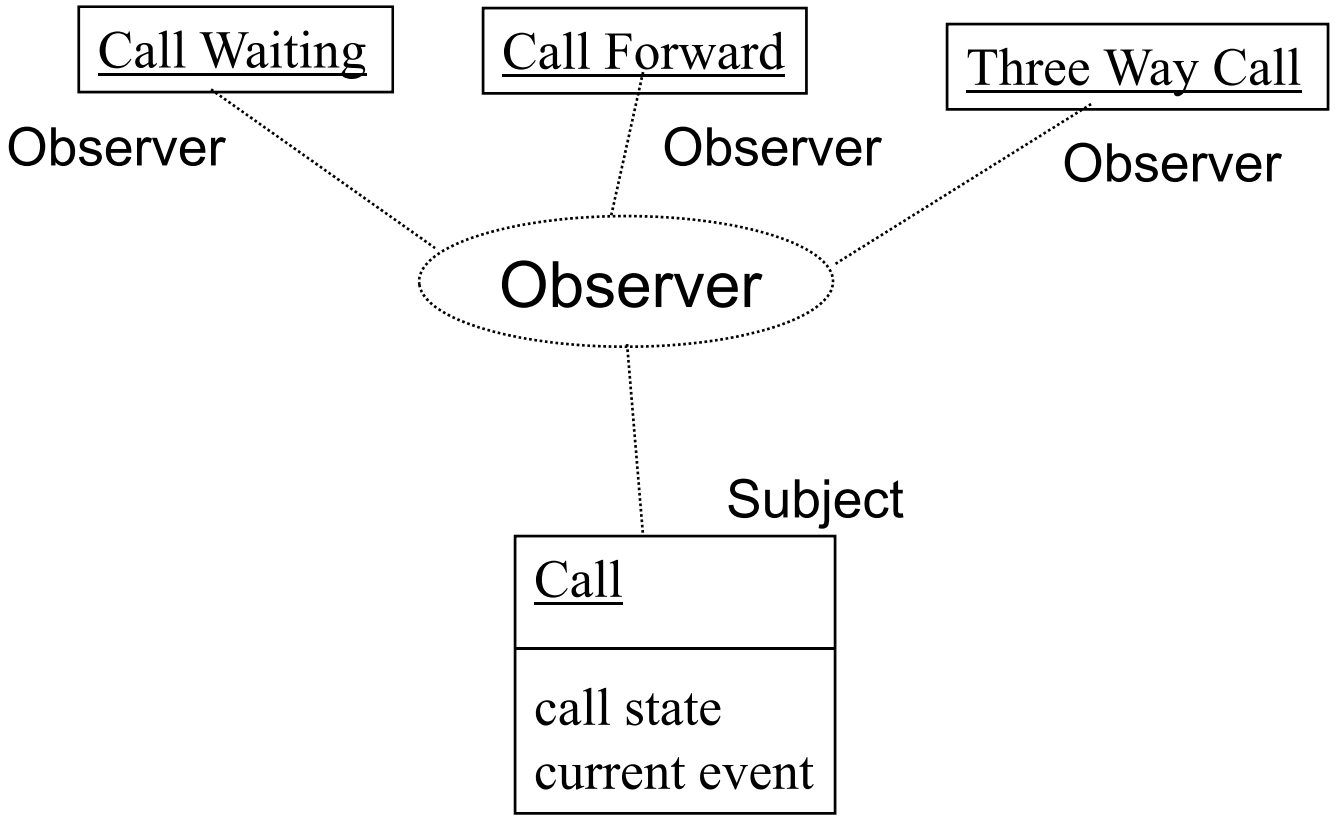
```
Event event;
State state;
three_way_call_handle(Event e){
…
if(call_waiting == running)
    deny_service();
if(call_forward == running){
   call_forward(e);
   make_3waycall(e);
}
if(message_answer == running)
  deny_service(e);
…
}
```

At the time this code architecture was developed there were only about a dozen services, but that number grew to well over 100.

Fixing The Problem

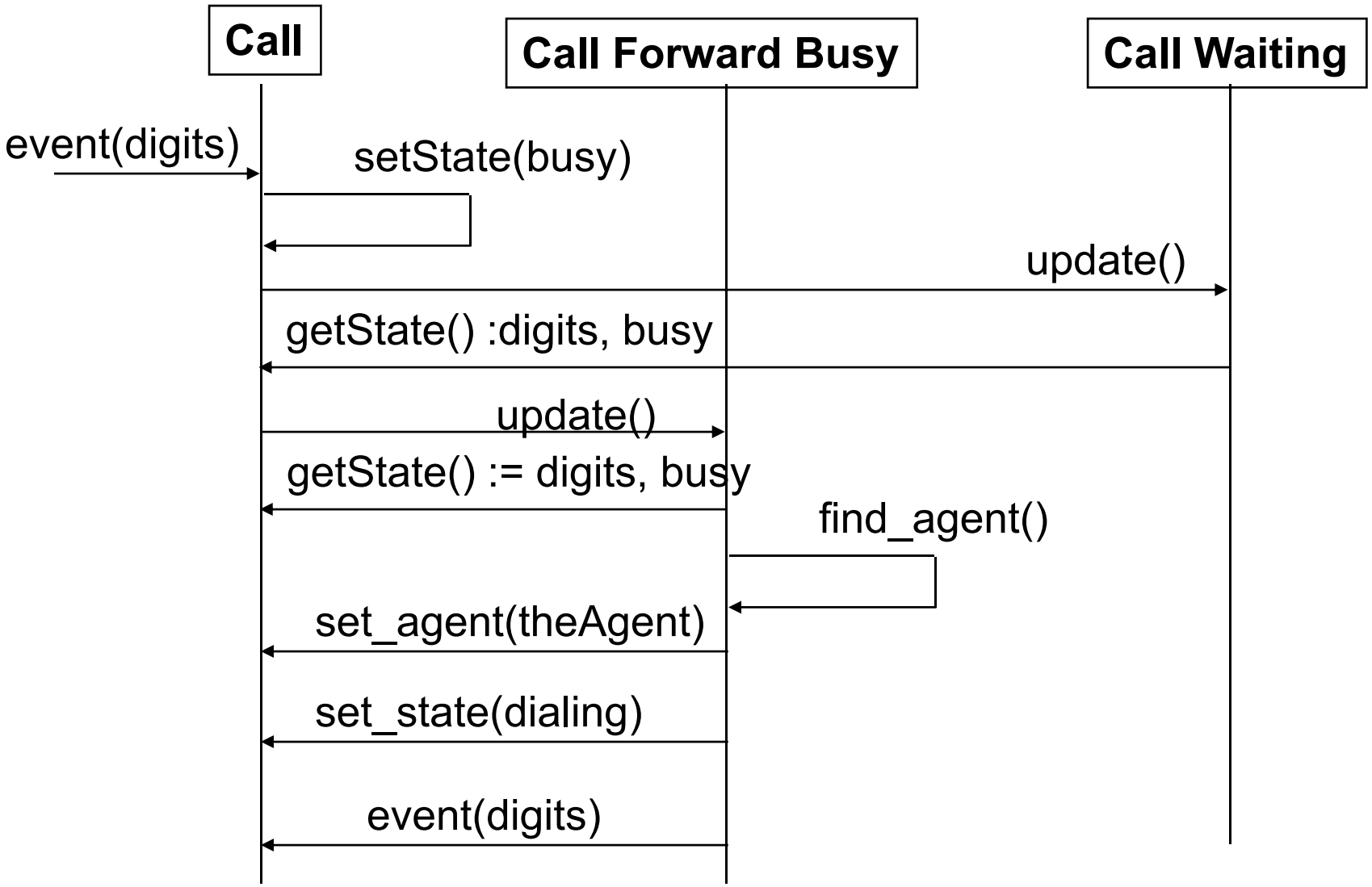
- A solution was sought to overcome this N^2 problem
- The solution incorporated both object-oriented programming and Gamma patterns.
- To what extend do you think the overcame the problems?

Solution part 1- services as observers

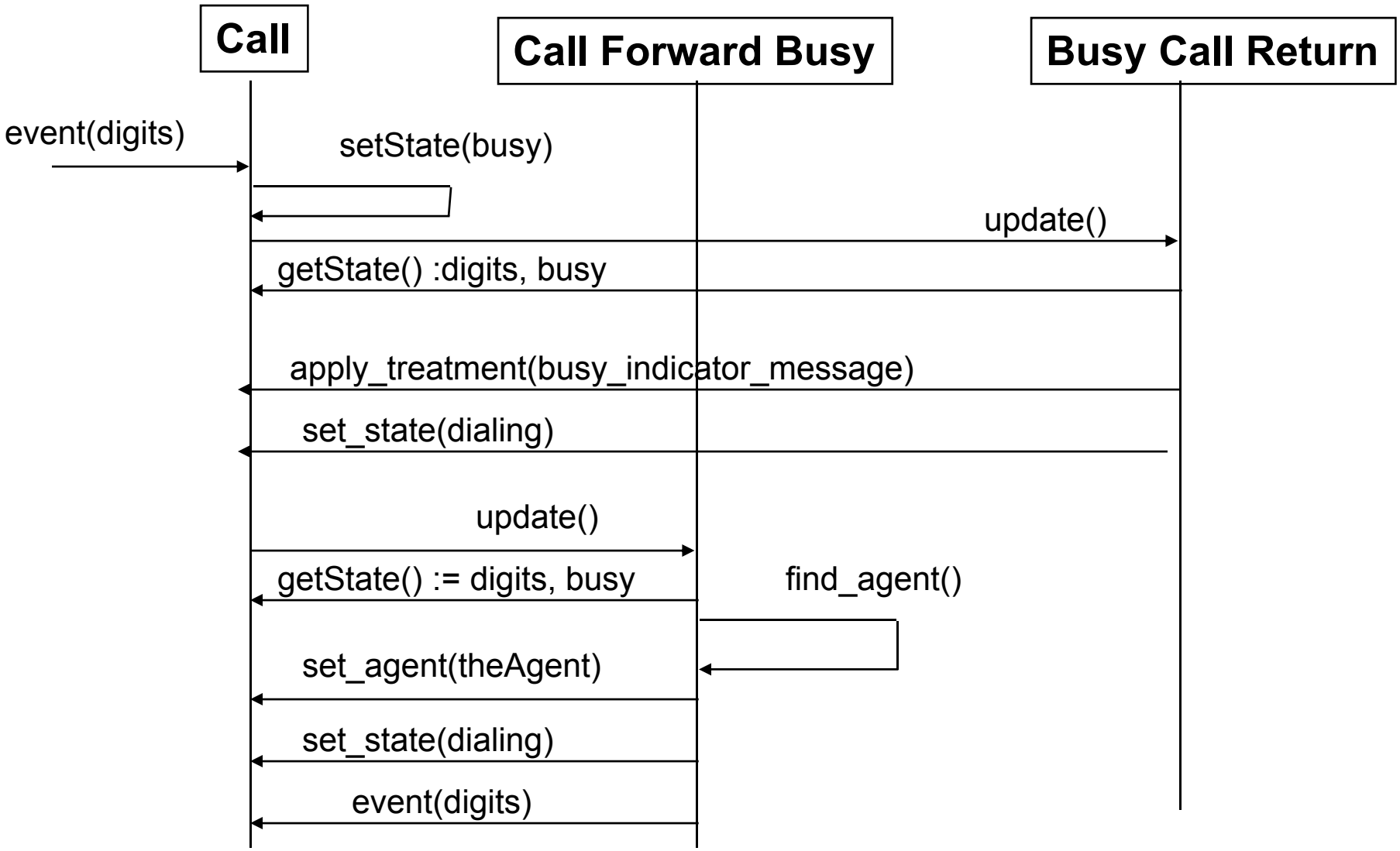


Call object: O(1) complexity, Each Service O(1) complexity.
Overall O(N) code and Test complexity

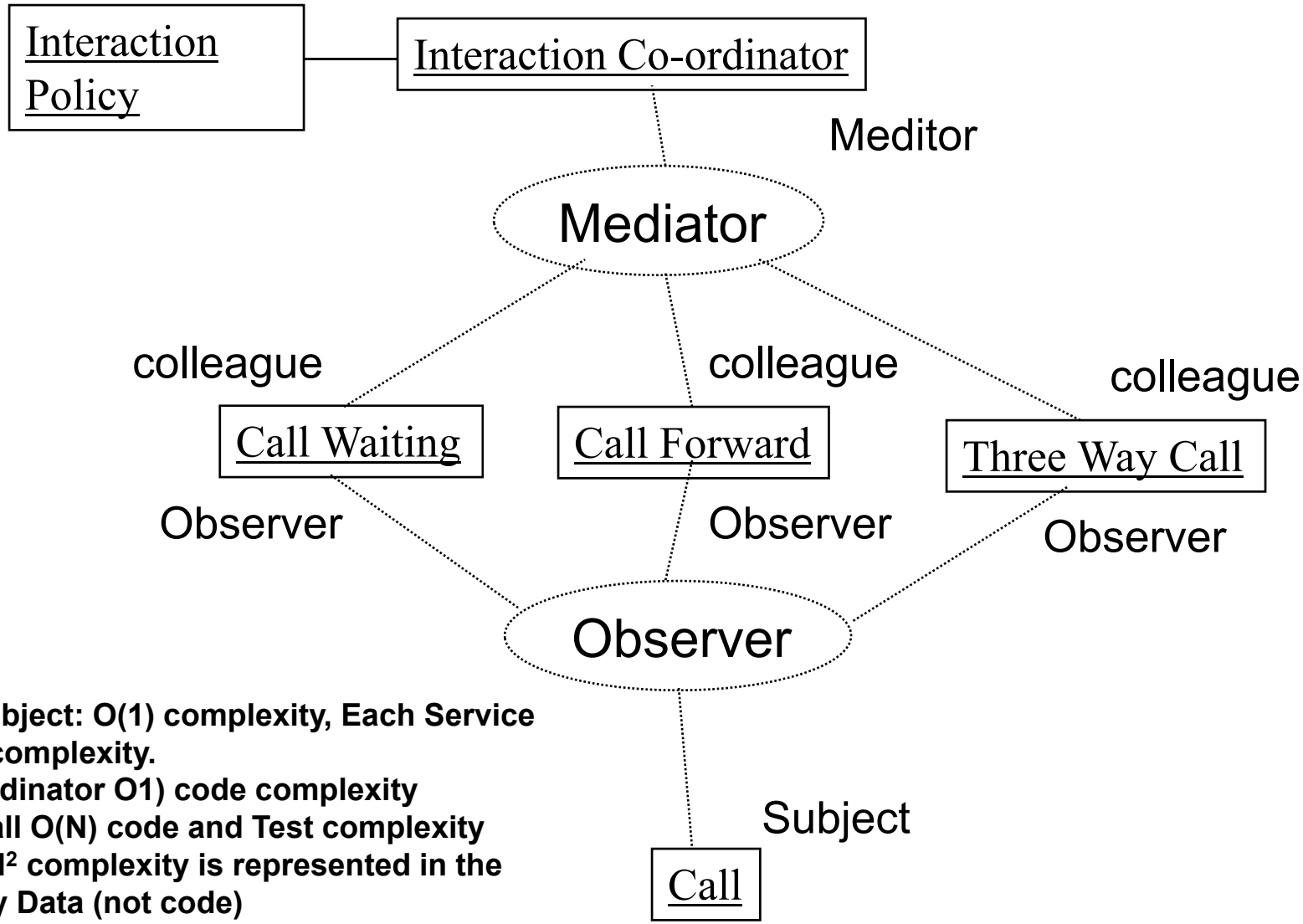
Services as Observers



Services as Observers -Interaction Problem

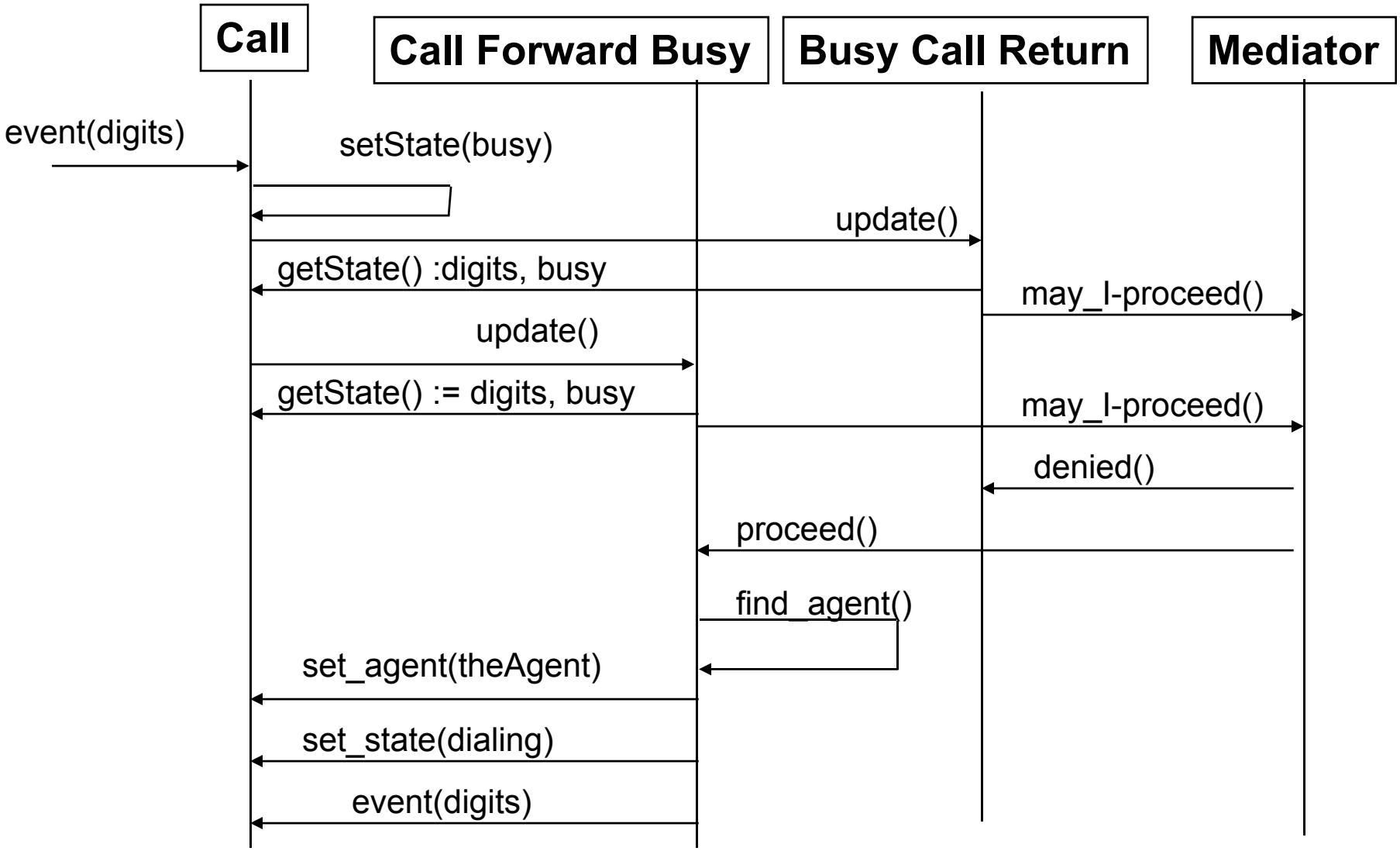


Solution part 2- mediating service interactions



Call object: O(1) complexity, Each Service O(1) complexity.
Co-ordinator O1) code complexity
Overall O(N) code and Test complexity
Any N^2 complexity is represented in the Policy Data (not code)

Mediating service interactions



Fixing The Problem

- **New Problem:**
- **How does the call object know that it has heard from all the services that might want to "chime in".**
- **This problem was solved with yet another pattern (Command Pattern) –but that is another story**

Memento Pattern

- **Memento Pattern (Gamma et al.)**

Memento Pattern

- We examine the Memento pattern as presented by Gamma et al.
- Application: this is the basis of many applications implement the "undo" operation.
- Application: this is the basis of how Cookies and Sessions implement state in the stateless world of HTTP-based web applications
- Also known as Token pattern

Memento Pattern (Gamma et al.)

Intent

- **Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.**

Memento Pattern (Gamma et al.)

Motivation

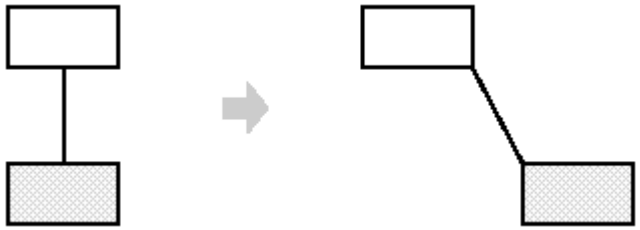
- Sometimes necessary to record the internal state of an object.
- This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
- You must save state information somewhere so that you can restore objects to their previous states.
- But objects normally encapsulate some, or all, of their state, making it inaccessible to other objects and impossible to save externally.
- Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.

Memento Pattern (Gamma et al.)

Consider for example a graphical editor that supports objects joined by connector.

A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them.

The editor ensures that the line stretches to maintain the connection and maintains connectivity to some point on the objects.



Memento Pattern (Gamma et al.)

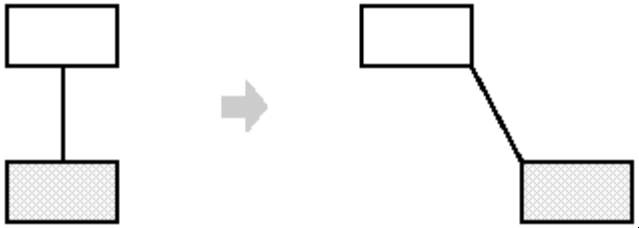
A well-known way to maintain connectivity relationships between objects is with a constraint-solving system.

We can encapsulate this functionality in a ConstraintSolver object.

ConstraintSolver records connections as they are made and generates mathematical equations that describe them.

It solves these equations whenever the user makes a connection or otherwise modifies the diagram.

ConstraintSolver uses the results of its calculations to rearrange the graphics so that they maintain the proper connections.



Memento Pattern (Gamma et al.)

Supporting undo in this application isn't as easy as it may seem.

An obvious way to undo a move operation is to store the original distance moved and move the object back an equivalent distance.

However, this does not guarantee all objects will appear where they did before.

Suppose there is some slack in the connection. In that case, simply moving the rectangle back to its original location won't necessarily achieve the desired effect.

Memento Pattern (Gamma et al.)

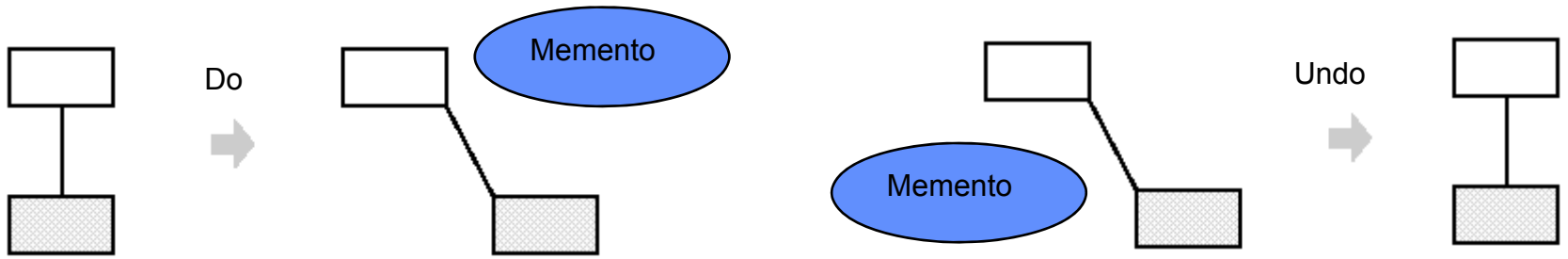
- We can solve this problem with the Memento pattern.
- A memento is an object that stores a snapshot of the internal state of another object—the memento's originator.
- The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state.
- The originator initializes the memento with information that characterizes its current state.
- Only the originator can store and retrieve information from the memento—the memento is "opaque" to other objects (caretakers).

Memento Pattern (Gamma et al.)

- The editor requests a memento from the ConstraintSolver as a side-effect of the move operation.
- The ConstraintSolver creates and returns a memento, an instance of a class SolverState in this case. A SolverState memento contains data structures that describe the current state of the ConstraintSolver's internal equations and variables.

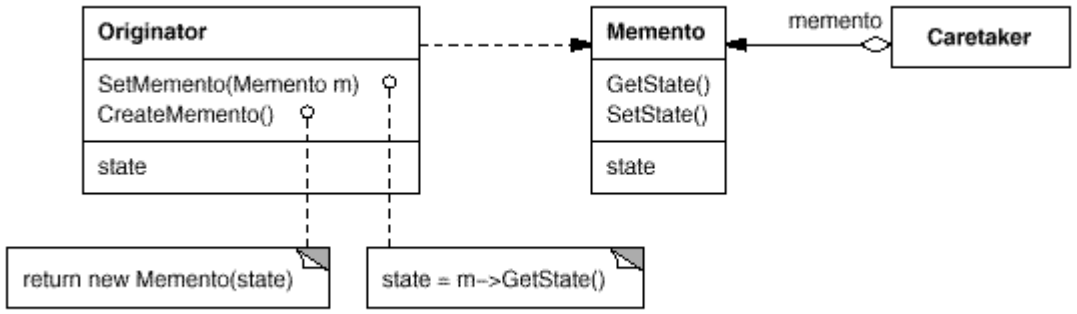
- Later when the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver.

- Based on the information in the SolverState, the ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state.



Memento Pattern (Gamma et al.)

Structure

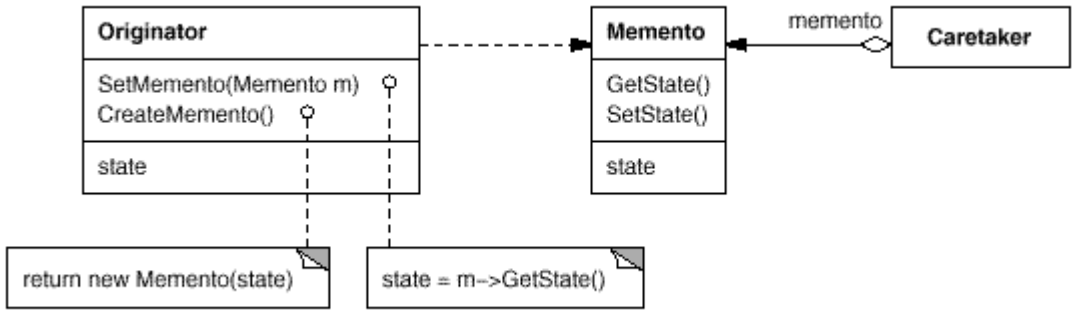


Roles

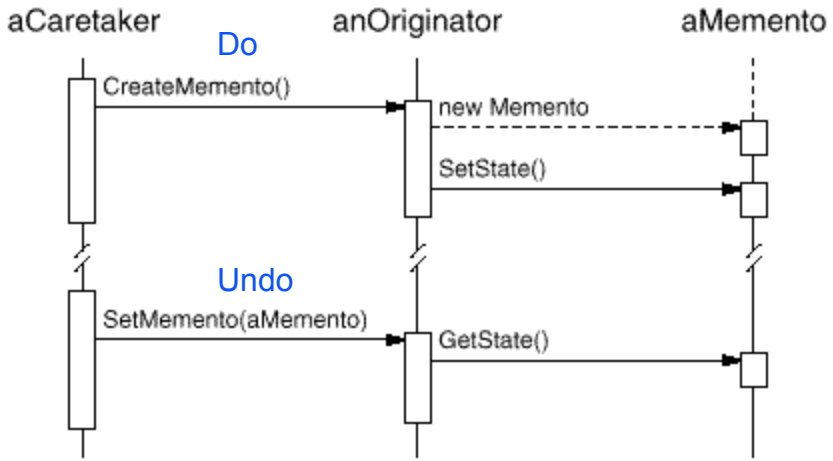
- **Memento**: store internal state, hides state for caretaker
- **Originator** (e.g. Constraint Solver): creates memento, uses memento to restore state of the app
- **Caretaker** (e.g. Undo Mechanism): keeps memento until needed. Does not examine or modify the memento's contents

Memento Pattern (Gamma et al.)

- **Structure**



- **Collaborations**



Memento Pattern (Gamma et al.)

Consequences:

- **Memento avoids exposing information that only originator should manage. Shields objects from having to know details of state.**
- **Simplifies the Originator: clients keep (take care of the memento object)**
- **Might be expensive: anticipating that client might want to undo.**
- **How to hide details so caretaker cannot see, or manipulate them?**
- **Hidden cost of caring for mementos**

Application to Web Programming:

- **Servers and HTTP are stateless.**
- **During an request from client browser, server places things that must be remembered into a memento (cookie) and returns it to client as part of HTTP response.**
- **On subsequent request client sends memento (cookie) along with the request.**
- **Server uses memento (cookie) to re-establish what the client was doing.**
- **One of the first applications of this was implementing the "Shopping Cart" metaphor on web pages.**

Memento Pattern (Gamma et al.)

Language Support (How to Implement Memento)

- Mementos have two interfaces: a wide one for originators and a narrow one for caretakers.
- Ideally the implementation language will support two levels of static protection.
- C++ lets you do this by making the Originator a `friend` of Memento and making Memento's wide interface `private`.
- Only the narrow interface should be declared `public`.

Memento Pattern (Gamma et al.)

```
class State;

    class Originator {
    public:
        Memento* CreateMemento();
        void SetMemento(const Memento*);
        // ...
    private:
        State* _state;      // internal data structures
        // ...
    };

    class Memento {
    public:
        // narrow public interface
        virtual ~Memento();
    private:
        // private members accessible only to Originator
        friend class Originator;
        Memento();

        void SetState(State*);
        State* GetState();
        // ...
    private:
        State* _state;
        // ...
    };
```

Memento Pattern (Gamma et al.)

Sample Code (For Graphics/Constraint Solver Example) from Gamma et al.

Memento Pattern (Gamma et al.)

```
 class Graphic;
        // base class for graphical objects in the graphical
editor

    class MoveCommand {
    public:
        MoveCommand(Graphic* target, const Point& delta);
        void Execute();
        void Unexecute();
    private:
        ConstraintSolverMemento* _state;
        Point _delta;
        Graphic* _target;
    };    };
```

Memento Pattern (Gamma et al.)

```
class ConstraintSolver {
   public:
       static ConstraintSolver* Instance();
       void Solve();
       void AddConstraint(
           Graphic* startConnection, Graphic* endConnection
       );
       void RemoveConstraint(
           Graphic* startConnection, Graphic* endConnection
       );

       ConstraintSolverMemento* CreateMemento();
       void SetMemento(ConstraintSolverMemento*);
   private:
       // nontrivial state and operations for enforcing
       // connectivity semantics
   };

   class ConstraintSolverMemento {
   public:
       virtual ~ConstraintSolverMemento();
   private:
       friend class ConstraintSolver;
       ConstraintSolverMemento();

       // private constraint solver state
   };
```

Memento Pattern (Gamma et al.)

```
 void MoveCommand::Execute () {
        ConstraintSolver* solver =
ConstraintSolver::Instance();
        _state = solver->CreateMemento(); // create a
memento
        _target->Move(_delta);
        solver->Solve();
    }

    void MoveCommand::Unexecute () {
        ConstraintSolver* solver =
ConstraintSolver::Instance();
        _target->Move(-_delta);
        solver->SetMemento(_state); // restore solver state
        solver->Solve();
    }
```

Memento Pattern Application

Example Application: Memento Based Iteration in C++

We review several iteration ideas ending with one inspired by the Memento pattern.

External Iteration

Bad because it provides no encapsulation or a general scheme that works for all containers. Relies on mapping. We are iterating over integers not elements directly.

```
for (int i = 0; i<v.size(); i++)
    v[i] = data[i];
```

Memento Pattern Application

Iteration Protocol (Internal Iteration)

Container provides a way for clients to ask for elements without exposing its internal storage management

```
class Collection {
     public:
     void reset() {index = 0;}
     bool hasMore() {return index < size;}
     void advance() {index++;}
     T & element() const {return *buffer[index];
     private:
       T ** buffer;
       int size;
       int index;

};

//client code
for (v.reset(); v.hasMore(); v.advance() )
       cout << v.element();
```

Note with this approach we cannot have nested iteration. That is, cannot have two loops iterating over the same container at the same time.

Memento Pattern Application

External Iterators

External object keeps track of iterator location. This allows multiple iterators to work over the same collection

```
class Collection {
     friend class CollectionIterator;
     public:

     private:
       T ** buffer;
       int size;
};

class CollectionIterator{
  public:
     CollectionIterator(Collection & c) :col(c) {index = 0;}
     void reset() {index = 0;}
     bool hasMore() {return index < size;}
     void advance() {index++;}
     T & element() {return *(col.buffer[index]);
  private:
     Collection & col;
     int index;

};
```

Memento Pattern Application

```
//client code
Collection v;
CollectionIterator iter(v);

while (iter.hasMore() ) {
    cout << iter.element();
    iter.advance();
    }


//client code  --nested iterator
Collection v;
CollectionIterator iter1(v), iter2(v);

for(iter1.reset(); iter1.hasMore(); iter1.next())
  for(iter2.reset(); iter2.hasMore(); iter2.next() )
       cout << iter1.element() <<  iter2.element();
```

Memento Pattern Application

Memento Based Iteration

Container provides an internal iteration protocol but allows multiple clients at the same time. Each client returns their old element, which the container uses to select the next element to give the client. (Based on Mememto Programming Pattern)

```
class Collection {

     public:
     T * first() {return buffer[0]; }
     bool hasMoreAfter(T * element) {
         for(int i = 0; i<size-1; i++)
             if(buffer[i] == element) return true;
         return false;

     }
     T * nextAfter(T & element) {
         for(int i = 0; i<size; i++)
             if(buffer[i] == element) return buffer[++i];
         return element;

     private:
       T ** buffer;
       int size; bufferSize;
};
```

Memento Pattern Application

```
//client code
Collection v;


for(T* item = v.first(); v.hasMoreAfter(item); v.nextAfter(item))
   cout << item;

//client code --nested iteration

for(T* item = v.first(); v.hasMoreAfter(item); item = v.nextAfter(item))
   for(T* item2 = v.first(); v.hasMoreAfter(item2); item = v.nextAfter(item2))
       cout << *item << *item2;
```