CS 247: Software Engineering Principles

Design Patterns (Observer, MVC)

Reading: Freeman, Robson, Bates, Sierra, *Head First Design Patterns*, O'Reilly Media, Inc. 2004
- Ch 2 Observer Pattern
- Ch 12 Model-View-Controller

MVC example among gtkmm examples provided in project 2

Problem: Synchronized Views

Multiple applications:
- editor: main canvas, thumbprints, editor palette
- internet games: distributed players see all updates

Solution #1: Coupled Design

```
void Data::notifyViews() {
    percent_->update(state);
    bargraph_->update(state);
    piechart_->update(state);
}
```

Solution #2: Aggregation of Abstract Views

- Data refers to collection of abstract views.
- Notifying views means iterating through collection.
Improved Design

In addition, let's provide methods for adding and removing displays from the collection.

**Subscribe**

<table>
<thead>
<tr>
<th>Data</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>+subscribe(View)</td>
<td>+update(state)</td>
</tr>
<tr>
<td>+unsubscribe(View)</td>
<td></td>
</tr>
<tr>
<td>+change()</td>
<td>#notify()</td>
</tr>
</tbody>
</table>

**Publish**

<table>
<thead>
<tr>
<th>Percentage</th>
<th>BarGraph</th>
<th>PieChart</th>
</tr>
</thead>
<tbody>
<tr>
<td>update(state)</td>
<td>update(state)</td>
<td>update(state)</td>
</tr>
</tbody>
</table>

Observer Pattern

Problem: maintaining consistency among related objects.

Solution: Subject sends updates to collection of abstract Observers. Subject maintains collection of subscribed observers, and sends notifications to all when its state changes.

Subject

- subject
- subjectState
- getState()
- setState()
- observers
- unsubscribe(Observer)
- notify()

ConcreteSubject

- objectState
- setState()

ConcreteObserver

- update(state)
- addObserver()
- unsubscribe()

Observer

- update(state)
- notify()

Might provide methods for getting and setting state information (more later).

Observer Pattern in Action

1. edit
2. notify()
3. update(state)
4. update view

A Different Design: Push vs. Pull

- In previous design, the Subject pushes state information to Observers.
- An alternative design would have each Observer request (pull) the specific information it needs, on notification of a change.
- The decision to push or pull update information depends on the frequency of operations and the complexity of the data.

s:Subject

- state
- changes

Observer

- accessor()
- update()

Each Observer is triggered by the same abstract operation.

each Observer uses Subject’s accessor to retrieve state information.
Example

Consider the example of an automobile and its onboard status computer. The computer monitors all of the vital data about the status of the vehicle. The computer receives this information from the onboard sensors, and then relays it to the OnStar, Low Jack, Fleet Tracker, gauge cluster and dashboard systems. These systems then interpret the data and abstract it for each of their own particular uses.

For instance, the onboard status computer reports information about the status of the headlights on the vehicle (ON or OFF), this information would be irrelevant to a low jack system that is responsible for tracking the vehicle should it be stolen. All of the individual systems have unique uses for the data about the vehicle, but all of the data collectively comes from the same source.

https://www.student.cs.uwaterloo.ca/~cs247/current/patterns.shtml

Concrete Subject: Vehicle

```cpp
class Vehicle : public Subject {
public:
    Vehicle();
    Vehicle(double)
    int speed();
    void locked(bool);
    bool lightsOn();
    // here, but for demonstration sake...
    double economy;

private:
    int m_speed;
    double m_position;
    bool m_locked;
    bool m_lights;
    int m_engineTemp;
    bool m_windows;
    bool m_wipers;
    int m_fuelLevel;
    long m_vin;
    bool m_isOn;
    int m_systemTime;
    double m_economy;
};
```

```cpp
void Vehicle::driveDistance(int distance, int speed, bool atNight,
    bool inRain, bool windowsOpen) {
    m_isOn = true;
    m_engineTemp = 200;
    m_position += (double)distance; // arbitrary constant
    m_fuelLevel -= (distance * m_economy); // arbitrary constant
    m_speed = speed;
    m_lights = atNight;
    m_wipers = inRain;
    m_windows = windowsOpen;
    notify();
}
```

All manipulators include call to notify observers.

Minimal Implementation Subject / Observer

```cpp
class Subject {
public:
    void subscribe (Observer*);
    void unsubscribe (Observer*);
protected:
    void notify();
private:
    std::set<Observer*> observers_;
};
```

```cpp
class Observer {
public:
    virtual void update () = 0;
private:
    Subject *subject_;
};
```

```cpp
class Concrete Subject:
```
Concrete Observers

Each concrete observer overrides Observer `update()` method, to retrieve and display the vehicle information it cares about.

```cpp
***** Dashboard.cpp *****
void Dashboard::update ( Vehicle* s ) {
    cout << endl;
    cout << "**DASHBOARD DISPLAY**" << endl;
    // pull the properties of the vehicle
    cout << "Current speed of vehicle: " << s->speed() << endl;
    cout << "Lights are on: " << std::boolalpha << s->lightsOn() << endl;
    cout << "Engine Temperature: " << s->engineTemp() << endl;
    cout << "Odometer: " << s->odometer() << endl;
    cout << "Fuel Level: " << s->fuelLevel() << endl;
    cout << "Time: " << s->systemTime() << endl;
}
```

```cpp
***** LowJack *****
void LowJack::update ( Vehicle* s ) {
    cout << endl;
    cout << "**LOWJACK INFO UPDATE RECEIVED FROM VEHICLE**" << endl;
    // pull the properties of the vehicle
    cout << "Location of vehicle: " << s->position() << endl;
    cout << "Doors are locked: " << std::boolalpha << s->locked() << endl;
}
```

What We’ve Accomplished

The Observer Pattern minimizes coupling between Subjects that publish information and Observers that receive notifications of information.

- The Subject just knows that it has a list of Observers. It doesn’t care how many, or what type. It simply publishes notifications of changes.
- Observers subscribe to notification service (at run-time). Observers can be added and removed at run-time.

Resulting subject and observer classes are easier to reuse in other applications.

Model-View-Controller Pattern

**Idea:** Combination of design patterns (including Observer) to decouple UI code from application code (the “model”).

```
MODEL

VIEW

CONTROLLER

class Player {
    public:
        void fight();
        void flee();
        void talk();
};
```

MVC is a Compound Pattern

**Composite Pattern:** All View elements use the same uniform (abstract) base class.

**Strategy Pattern:** View delegates to Controller the strategy that maps UI events to calls to Model.

**Observer Pattern:** The Model and View implement the Observer Pattern to notify interested objects (Views) of its state changes.

VIEW

CONTROLLER

MODEL
Observer Pattern in MVC

Freeman, Freeman, Head First Design Patterns

```cpp
class Subject {
    public:
        subscribe(Observer*);
        unsubscribe(Observer*);
    protected:
        notify();
};
```

```cpp
class Player : public Subject {
    public:
        void fight();
        void flee();
        void talk();
};
```

Strategy Pattern in MVC

Freeman, Freeman, Head First Design Patterns

```cpp
class Player {
    public:
        void fight();
        void flee();
        void talk();
};
```

Composite Pattern in MVC

Freeman, Freeman, Head First Design Patterns

Will talk about Composite Pattern later in term. In essence, it provides a uniform interface for a collection of components.

The GTKmm library has implemented the Composite Pattern for us: all GTKmm elements are of type `Widget`.

MVC Example

- Click on `next` to display the next card in the deck.
- Click on `reset` to reset the deck.
### MVC Example

**VIEW**
- Query state
- Update view

**CONTROLLER**
- Call to Model
- Notify Observer of state change

**MODEL**
- SortedDeck class
  - Suits suit()
  - Faces face()
  - void nextCard()
  - void resetCards()

The Model knows nothing about the View or Controller.

### Main Program

```cpp
#include <gtkmm/application.h> // Bring in Gtk::Application
#include "model.h"
#include "controller.h"
#include "view.h"

int main( int argc, char * argv[] ) {
    // Initialize gtkmm with the command line arguments,
    // as appropriate.
    auto app = Gtk::Application::create( argc, argv, "MVC example" );

    Model model;  // Create model
    Controller controller( &model );  // Create controller
    View view( &controller, &model );  // Create view

    // Show the window and return when it is closed.
    return app->run( view );
} // main
```

### Model Implementation

The Model knows nothing about the View or Controller.

```cpp
class SortedDeck {
public:
    Suits suit();
    Faces face();
    void nextCard();
    void resetCards();
};

enum Faces { NINE, TEN, JACK, QUEEN, KING, ACE, NOFACE };
enum Suits { DIAMOND, CLUB, HEART, SPADE, NOSUIT };

class Model : public Subject {
public:
    Model();
    Suits suit();
    Faces face();
    void nextCard();
    void resetCards();
private:
    int topCard_;
};
```

```cpp
Model::Model() : topCard_(-1) {
}

void Model::resetCards() {
    topCard_ = -1;
    notify();
}

void Model::nextCard() {
    if (topCard_ == numCards-1)
        return;
    topCard_ += 1;
    notify();
}
```
### Controller

class Controller {
public:
  Controller(Model*);
  void nextButtonClicked();
  void resetButtonClicked();

private:
  Model *model_;  // has reference to Model
};

// translates commands into calls to Model
void Controller::nextButtonClicked() {
  model_->nextCard();
}
void Controller::resetButtonClicked() {
  model_->resetCards();
}

### View

class View : public Gtk::Window, public Observer {
public:
  View( Controller*, Model* );
  virtual ~View();
  void update();

private:
  Gtk::Box panels;  // components for the display
  Gtk::Box butBox;
  Gtk::Button next_button;
  Gtk::Button reset_button;
  Gtk::Image card;
  DeckGUI deck;

  Model *model_;  // Observer Pattern (to query
                  // model state)
  Controller *controller_;  // Strategy Pattern

  void nextButtonClicked();
  void resetButtonClicked();
};

### View Constructor

View::View( Controller *c, Model *m ) :
  model_(m), controller_(c), panels{},
  butBox{ Gtk::ORIENTATION_VERTICAL, 10 },
  next_button{ "next" }, reset_button{ "reset" },
  card{ deck.null() }
{
  // Sets some properties of the window.
  set_title( "CS247 MVC example" );
  set_border_width( 10 );
  set_default_size( 250, 100 );  // width, height in pixels

  // Associate GUI events with local Strategy methods
  next_button.signal_clicked().connect(
    sigc::mem_fun( *this, &View::nextButtonClicked ) );
  reset_button.signal_clicked().connect(
    sigc::mem_fun( *this, &View::resetButtonClicked ) );

  show_all();  // Display window contents
  model_->subscribe(this);  // register View as an Observer
}

### View Implementation

// Strategy Pattern -- delegate interpretation to Controller
void View::nextButtonClicked() {
  controller_->nextButtonClicked();
}
void View::resetButtonClicked() {
  controller_->resetButtonClicked();
}
// Observer Pattern -- update display upon call to update()

void View::update() {
    Suits suit = model_->suit(); // get current state
    Faces face = model_->face();
    if ( suit == NOSUIT ) // reset card image
        card.set( deck.null() );
    else
        card.set( deck.image(face, suit) );
}

Summary

The goal of design patterns is to encapsulate change.

Observer Pattern encapsulates the set of observer objects, to support dynamic addition and removal of observers.

Model-View-Controller Pattern separates UI code from application logic code.

What You Should Get From This

Comprehension

• Select appropriate design pattern to solve particular design problem.

Application

• Apply design patterns to modify given UML model.
• Implement (code fragments) of design patterns:
  - Singleton
  - Template Method
  - Strategy
  - Adapter
  - Observer
  - MVC