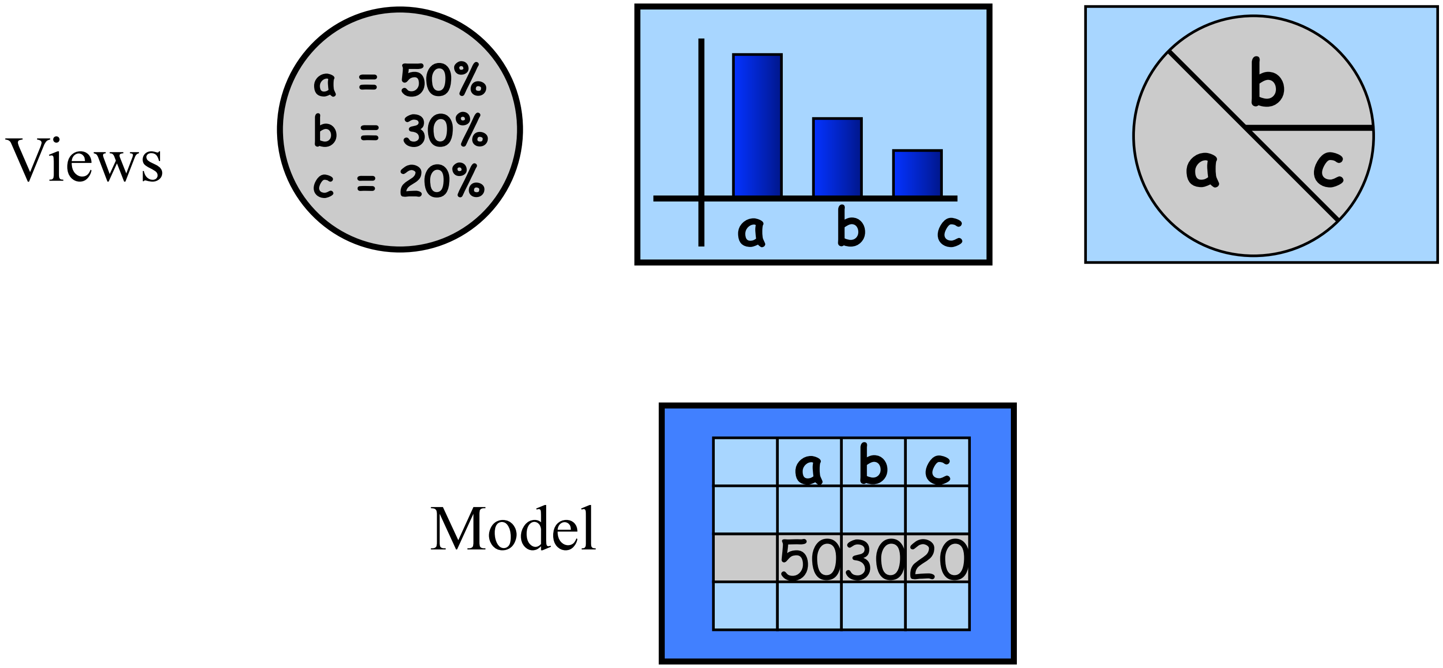
CS 247: Software Engineering Principles

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

Synchronized Views

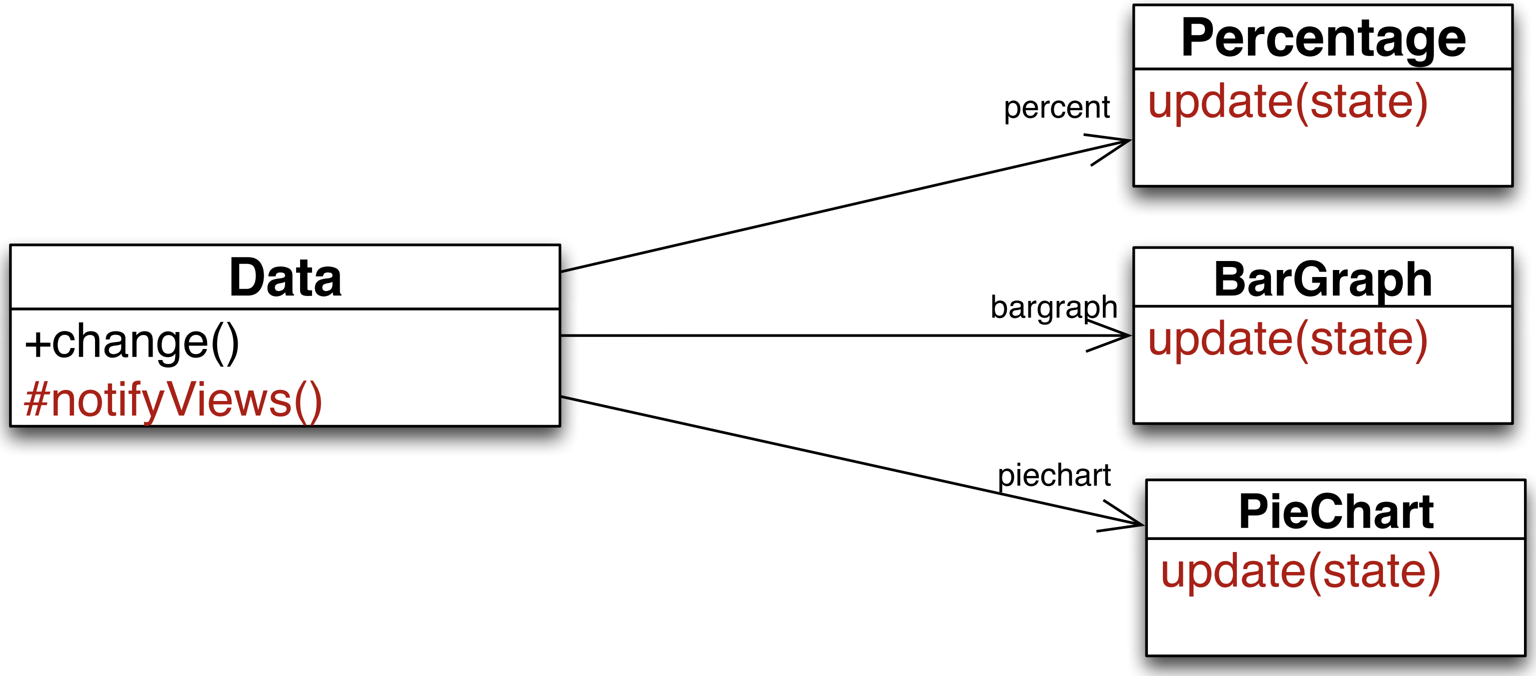


	a	b	c
	50	30	20

Multiple applications

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

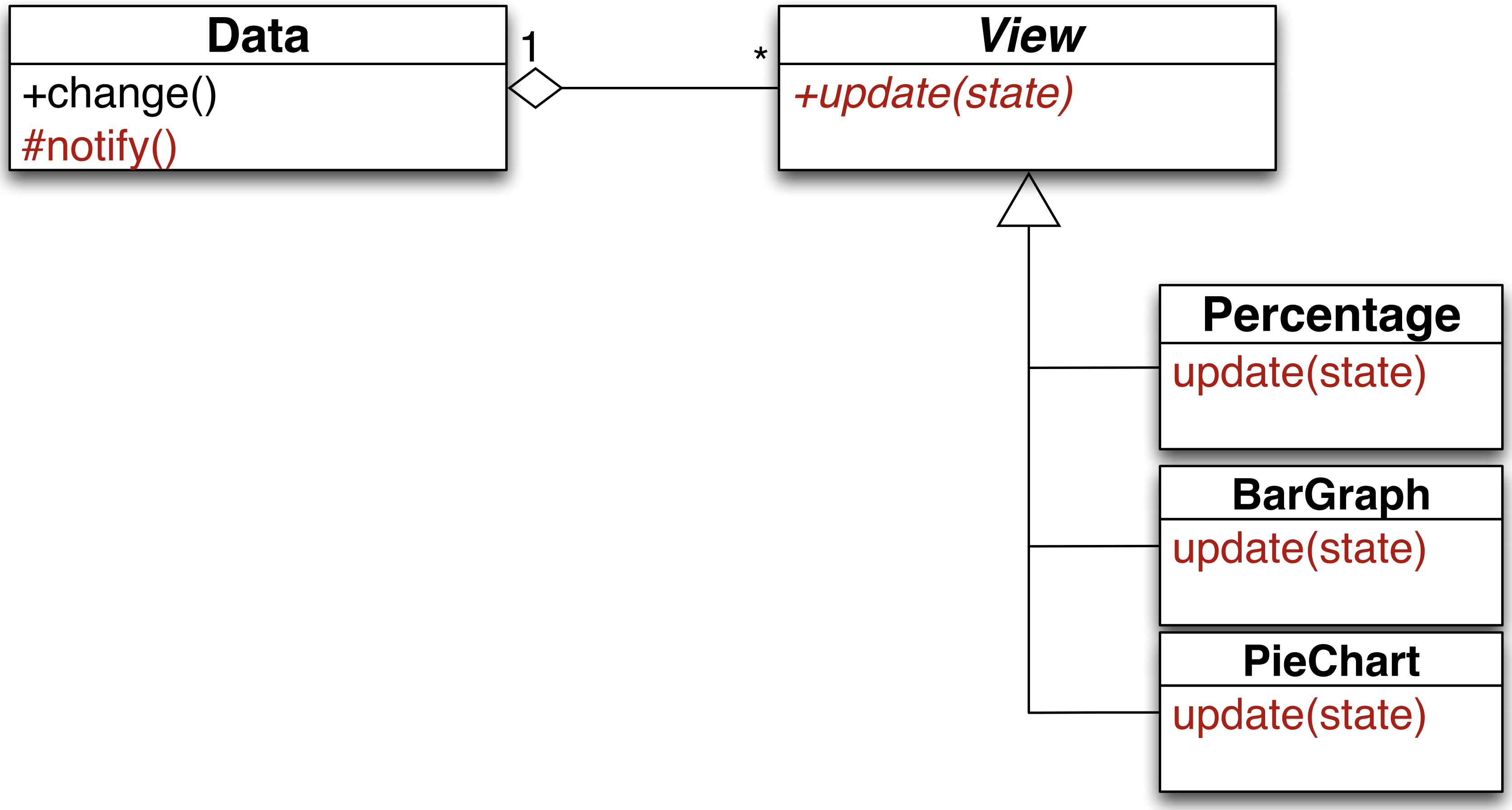
Solution #1: Coupled Design



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

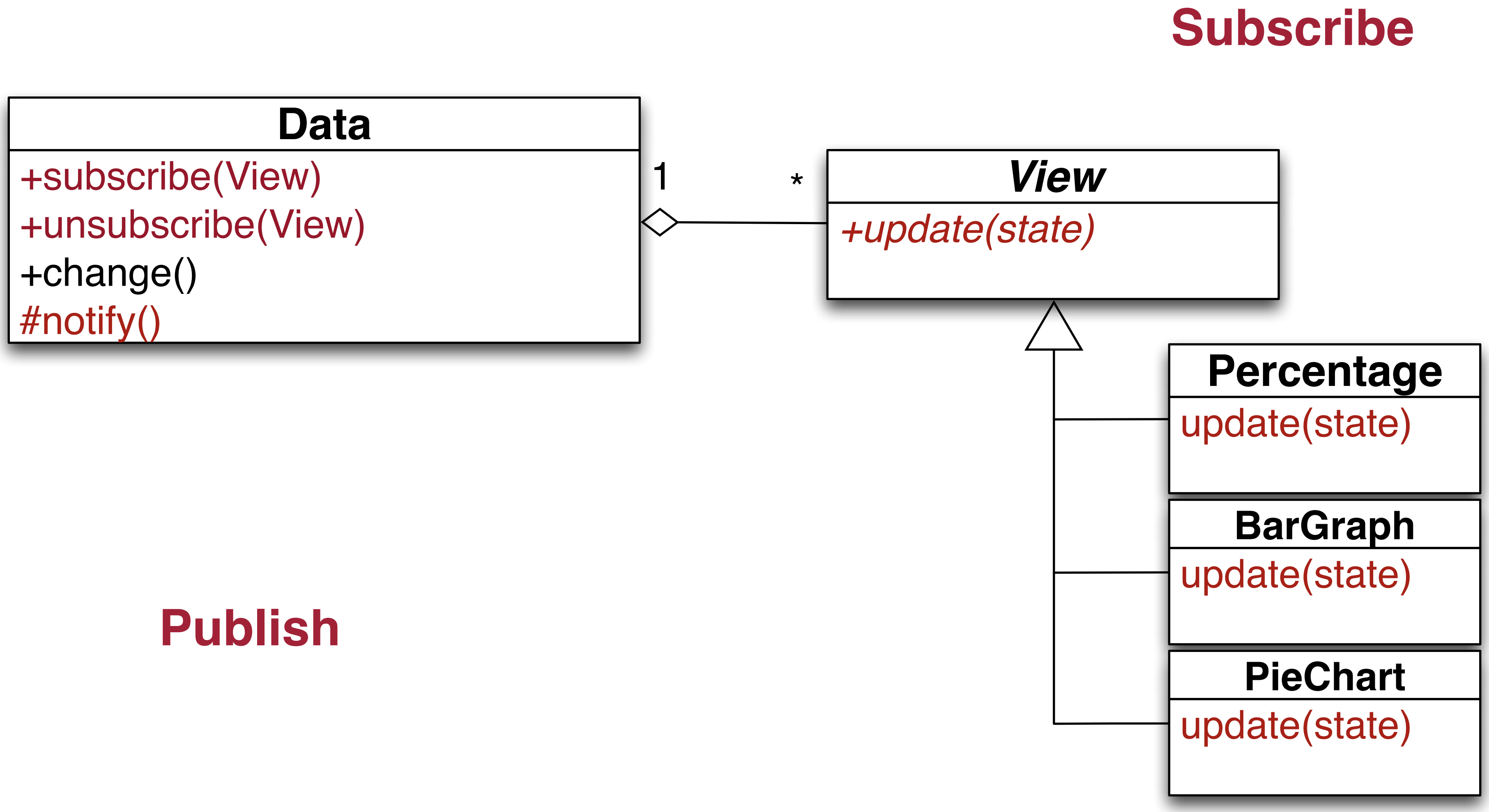
Solution #2: Aggregation of Abstract Views

- 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.

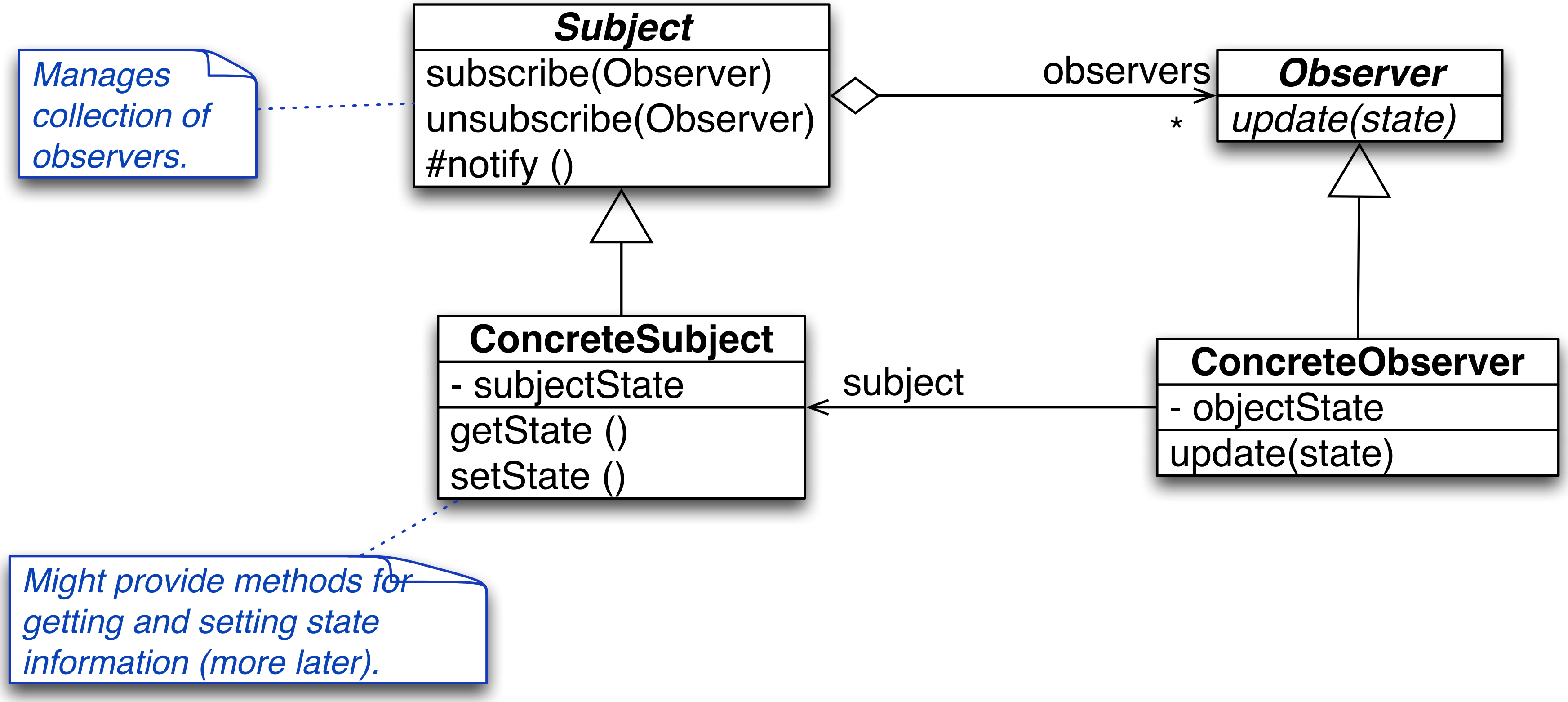


Observer Pattern

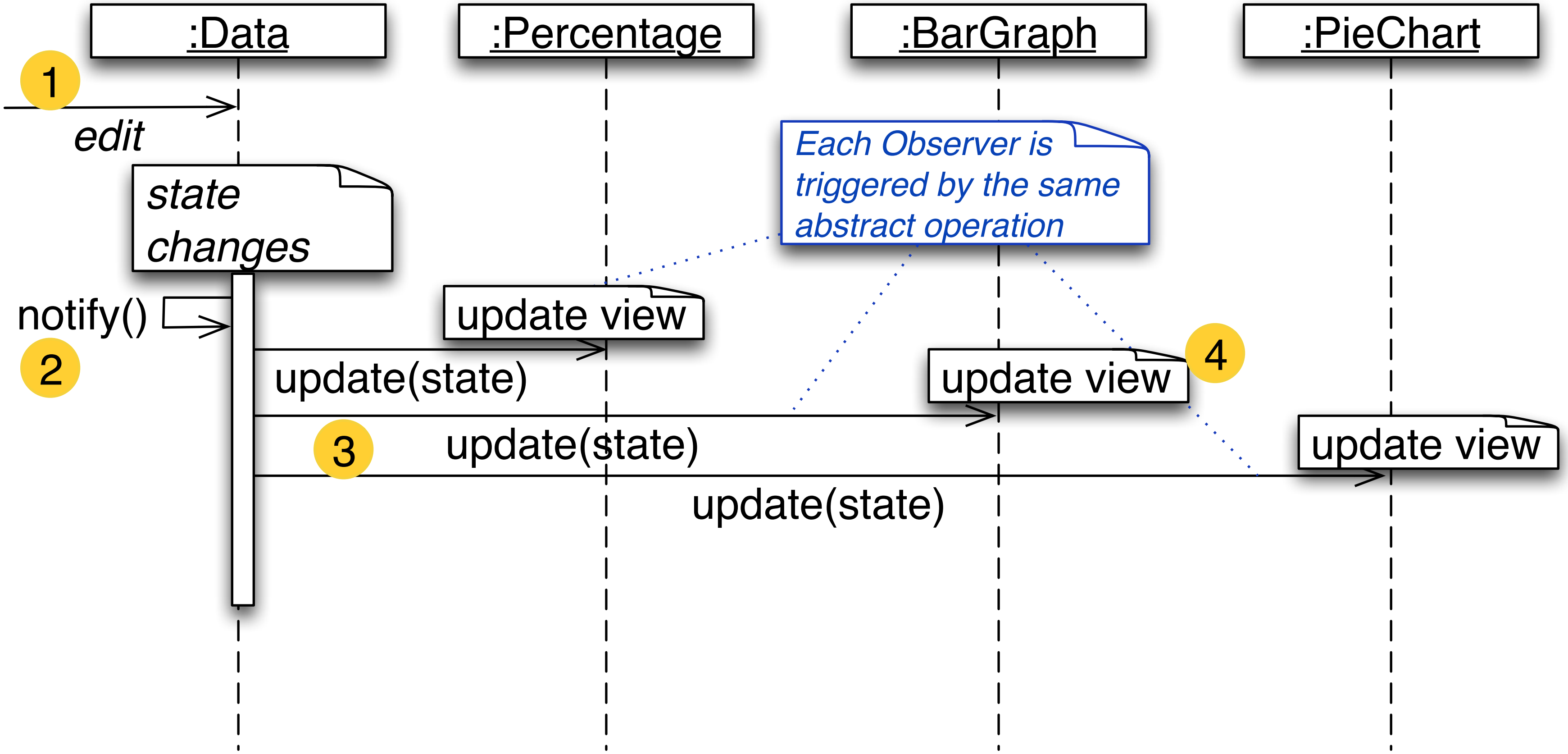
Problem: maintaining consistency among related objects

Solution: Subject sends updates to collection of abstract Observers

Subject maintains collection of subscribed observers, and sends notifications to all when its state changes.

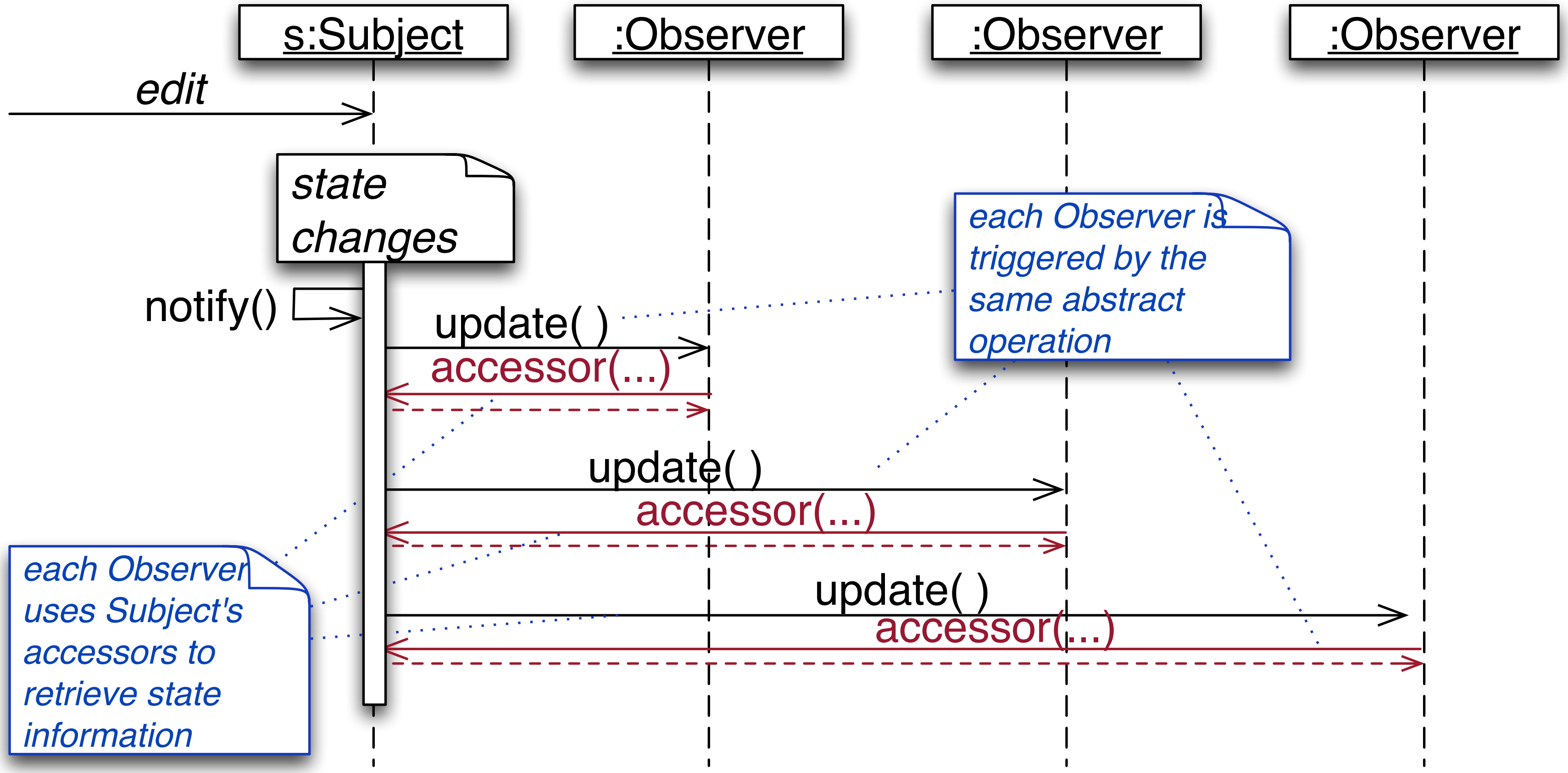


Observer Pattern in Action



A Different Design: Push vs. Pull

- In previous design, the Subject pushes state information to Observers.
- An alternative design would have each Observer request 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.



each Observer is triggered by the same abstract operation

each Observer uses Subject's accessors to retrieve state information

Example

Consider the example of an automobile and its onboard status computer. The computer monitors the 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, and gauge cluster and dash board 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

Minimal Implementation Subject / Observer

```
class Subject {
public:
   void subscribe (Observer*);
   void unsubscribe (Observer*);

protected:
   void notify();

private:
   typedef std::set<Observer*> Observers;
   Observers observers_;
};
```

```
class Observer {
public:
   virtual void update () = 0;
};
```

Concrete Methods of Abstract Subject

```
void Subject::subscribe (Observer* o)
{
   observers_.insert (o);
}


void Subject::unsubscribe (Observer* o)
{
   observers_.erase (o);
}


void Subject::notify ()
{
   Observers::iterator i;
   for (i = observers_.begin (); i != observers_.end (); i++)
      (*i)-> update ();
}
```

What We've Accomplished

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

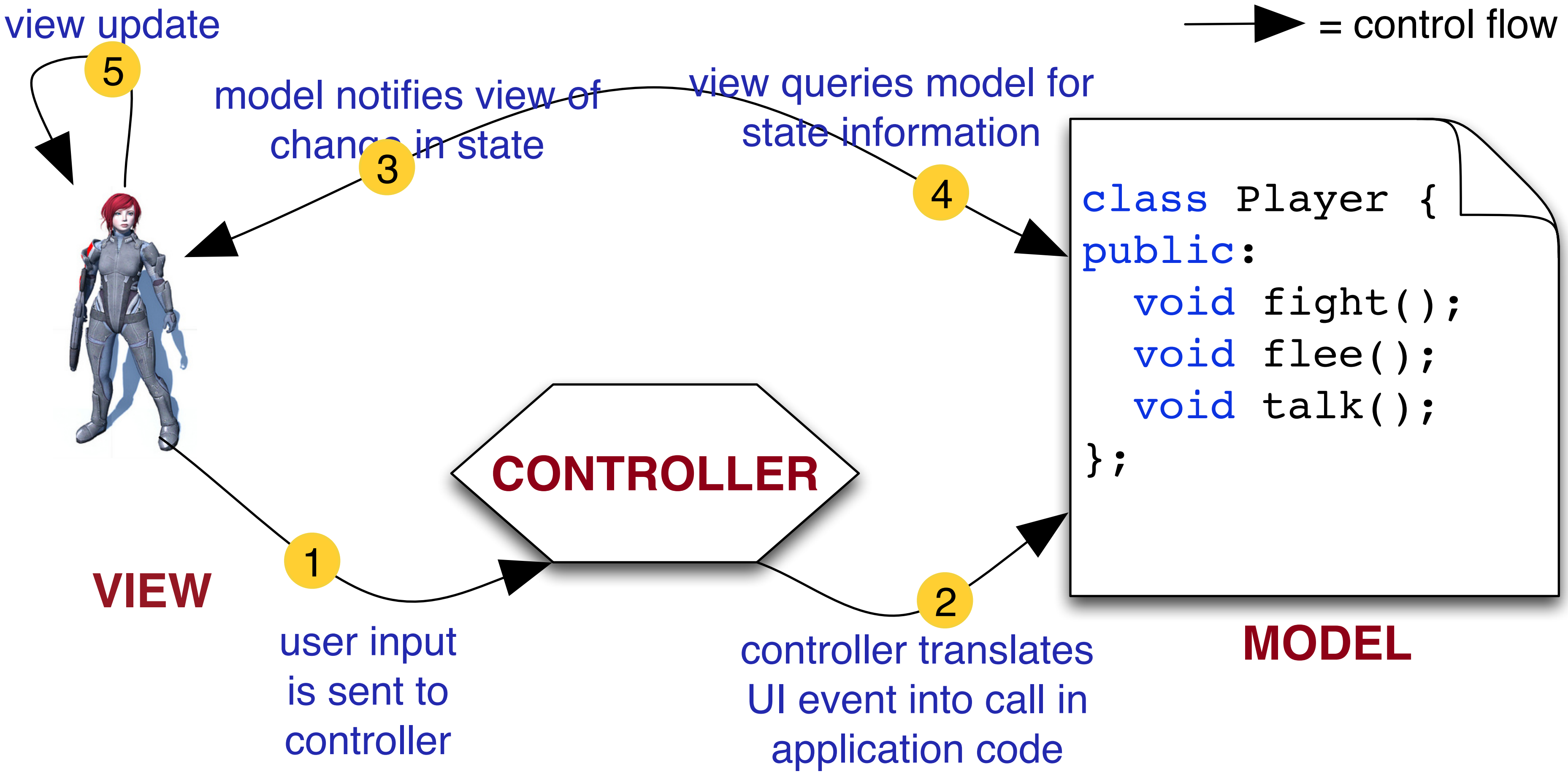
- The Subject just knows that it has a list of Observers
  It doesn't care how many, or what type
  It simply publishes notifications of changes
- Observers subscribe to notification service (at run-time)
  Observers can be added and removed at run-time

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

Model-View-Controller Pattern

Freeman, Freeman, *Head First Design Patterns*

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



view update

5

model notifies view of change in state

3

view queries model for state information

4

= control flow

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

CONTROLLER

VIEW

1

user input is sent to controller

2

controller translates UI event into call in application code

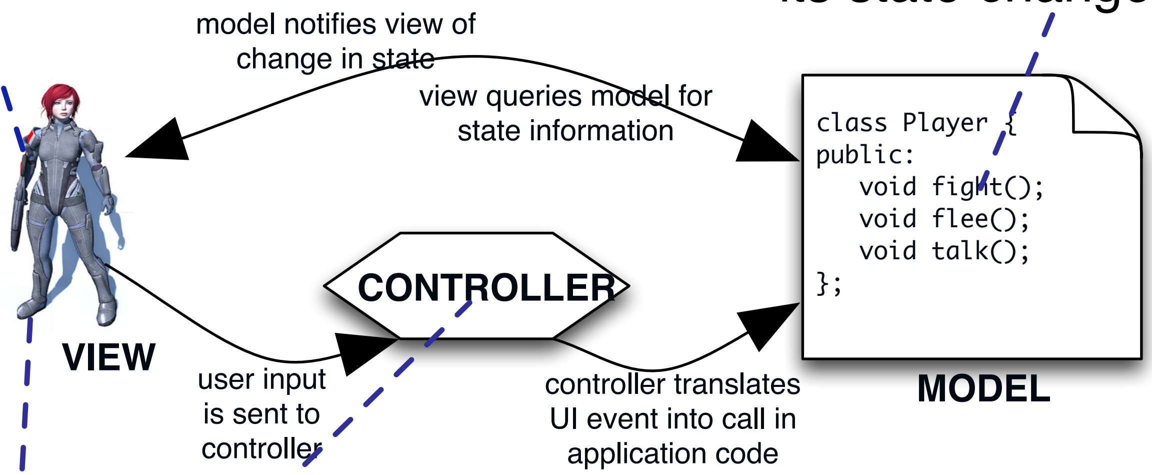
MODEL

MVC is a Compound Pattern

Freeman, Freeman, *Head First Design Patterns*

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

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


model notifies view of change in state
view queries model for state information
VIEW
CONTROLLER
user input is sent to controller
controller translates UI event into call in application code
MODEL

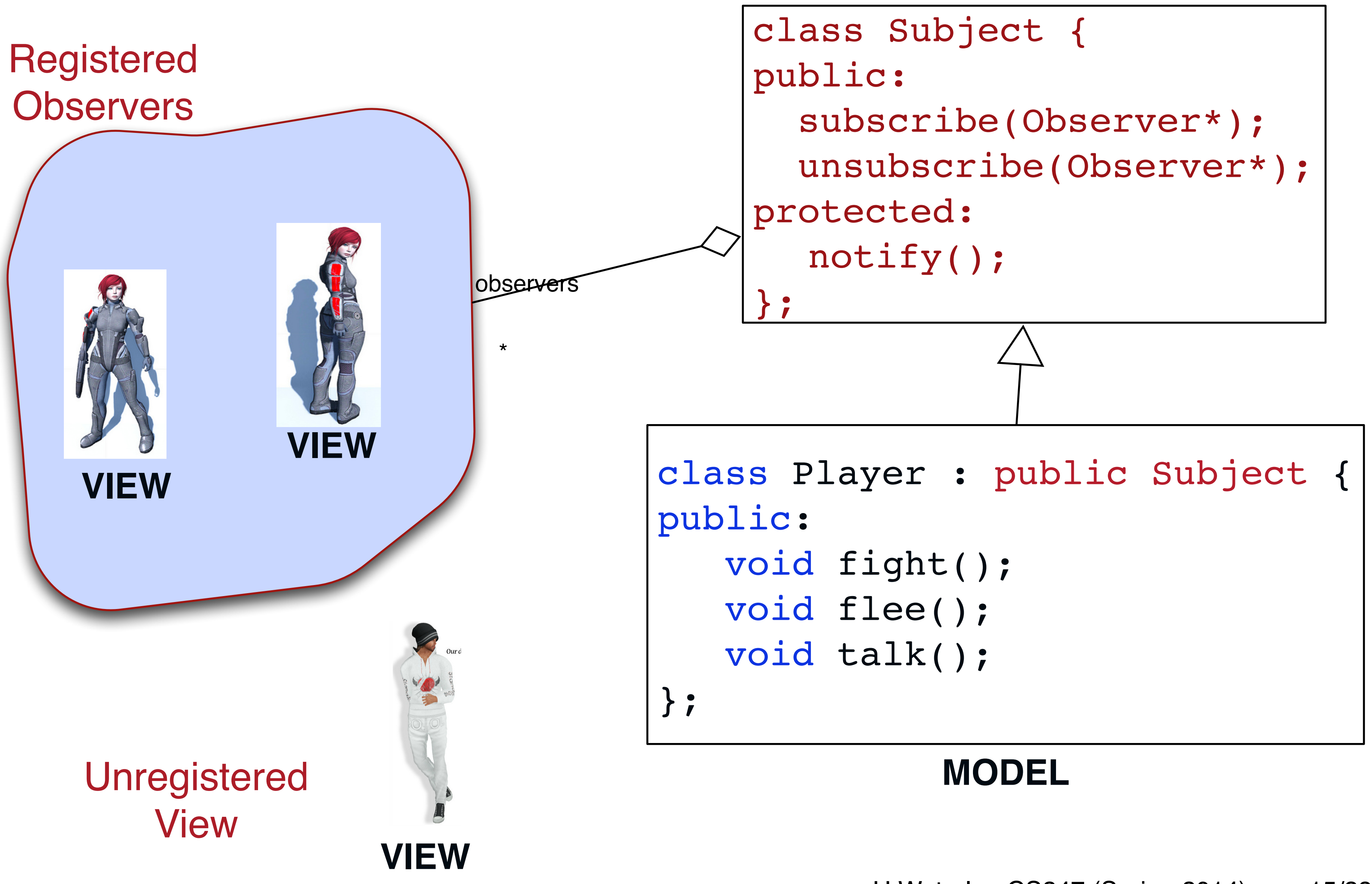
```
class Player {
public:
   void fight();
   void flee();
   void talk();
};
```

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

→ = control flow

Observer Pattern in MVC

Freeman, Freeman, *Head First Design Patterns*



Registered Observers

VIEW

VIEW

observers

*

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

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

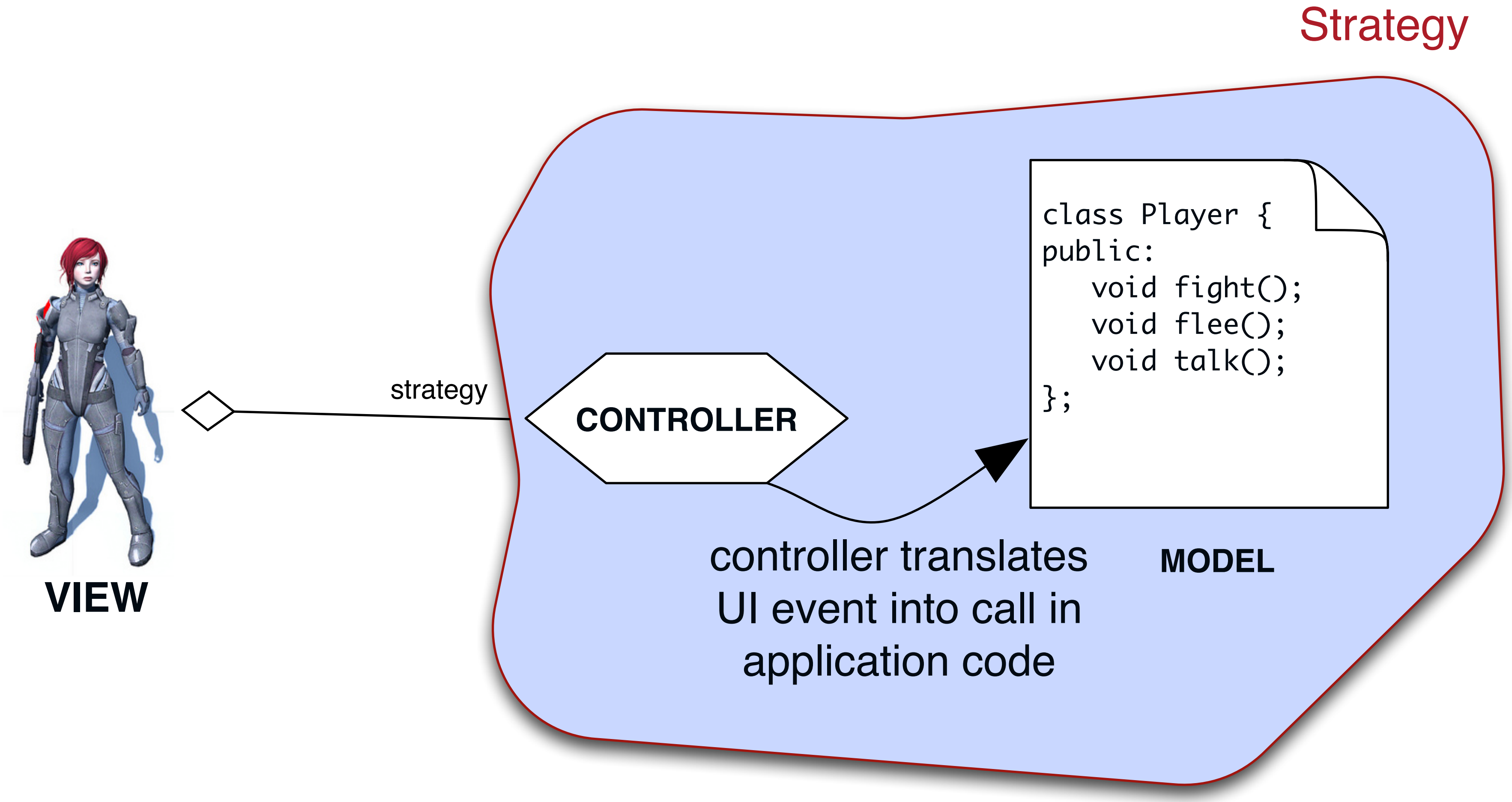
MODEL

Unregistered View

VIEW

Strategy Pattern in MVC

Freeman, Freeman, *Head First Design Patterns*



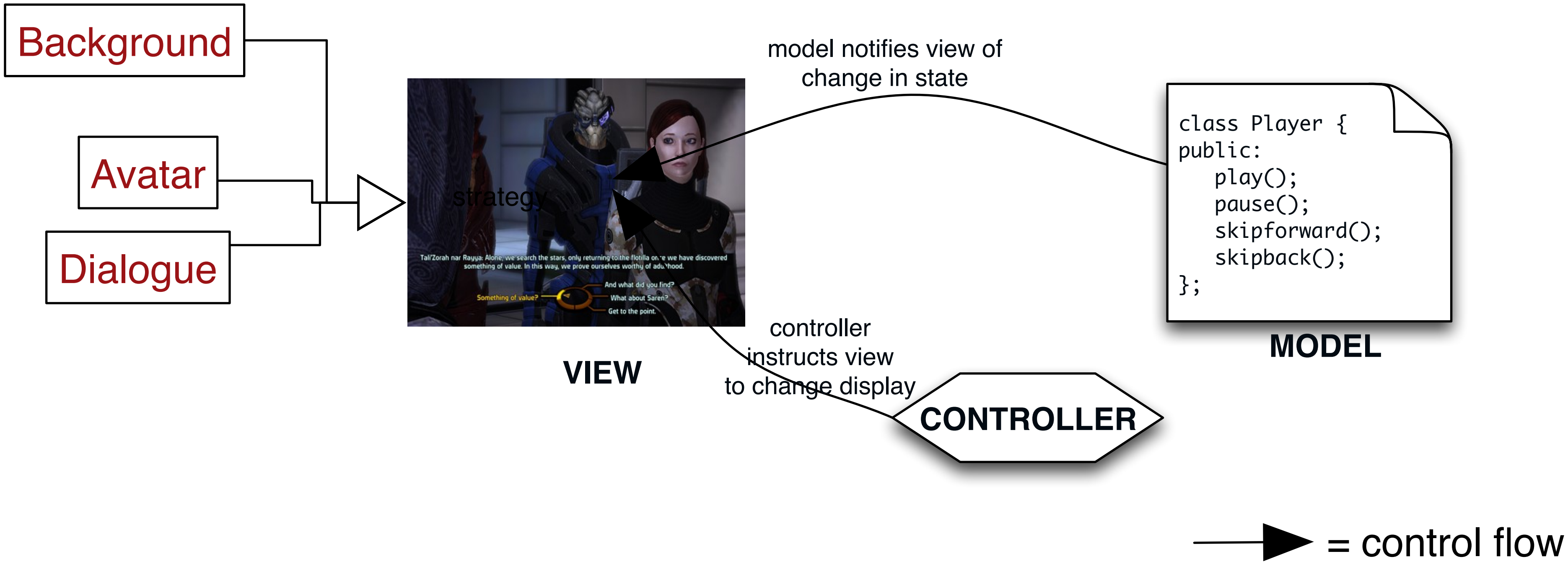
```
class Player {
public:
   void fight();
   void flee();
   void talk();
};
```

controller translates UI event into call in application code

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.


Background
Avatar
Dialogue
strategy
model notifies view of change in state
VIEW
controller instructs view to change display
CONTROLLER
MODEL
= control flow

Tali'Zorah nar Rayya: Alone, we search the stars, only returning to the flotilla once we have discovered something of value. In this way, we prove ourselves worthy of adulthood.

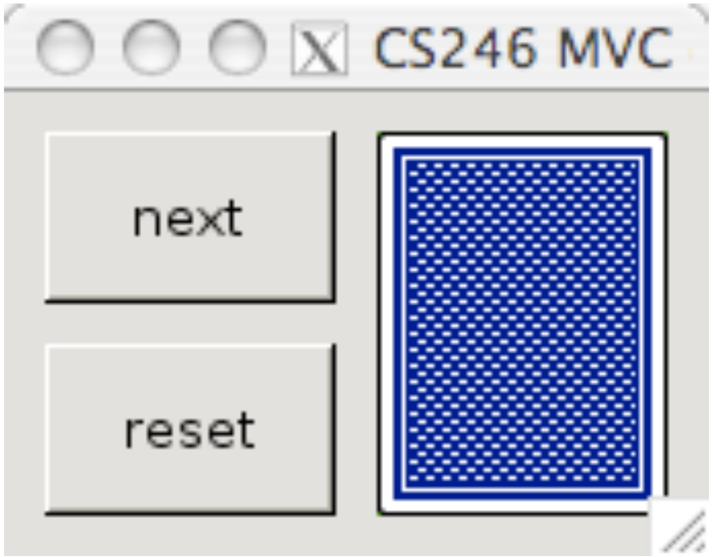
Something of value?
And what did you find?
What about Saren?
Get to the point.

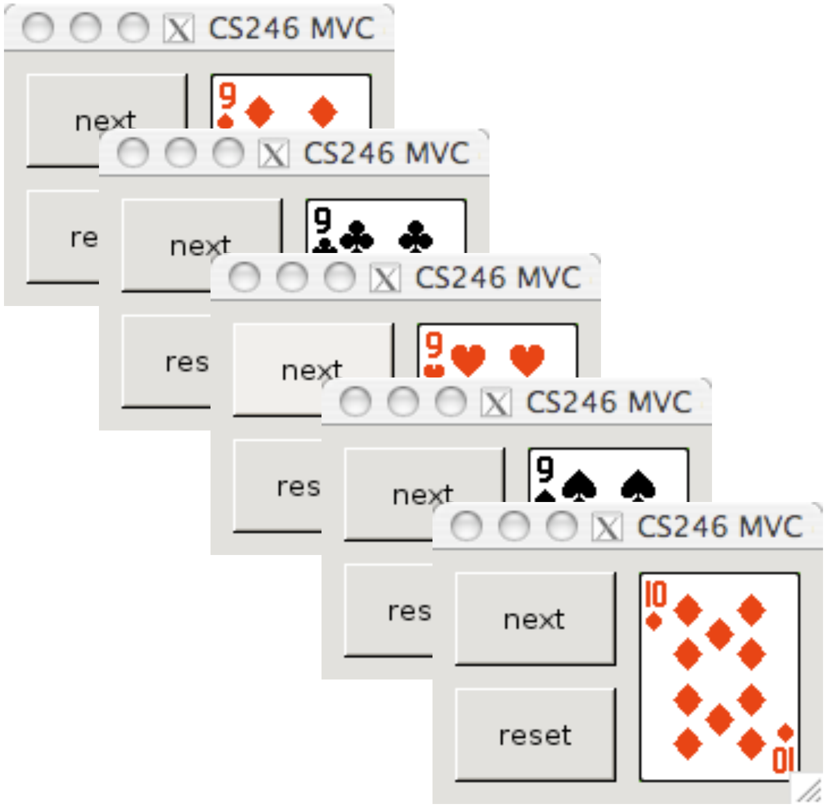
```
class Player {
public:
   play();
   pause();
   skipforward();
   skipback();
};
```

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

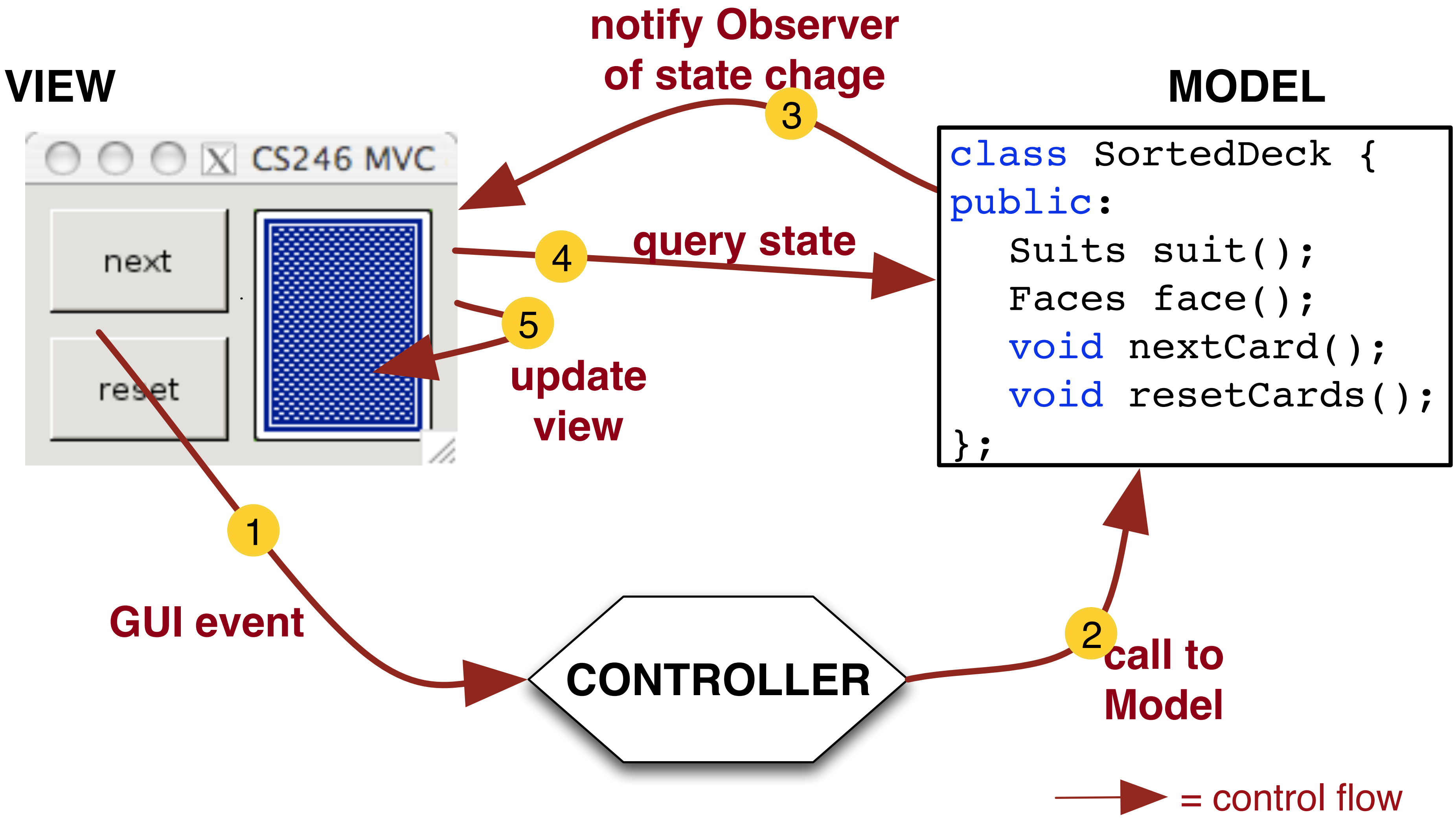
MVC Example

- click on next to display the next card in the deck
- click on reset to reset the deck





MVC Example



VIEW

MODEL

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

CONTROLLER

1 GUI event

2 call to Model

3 notify Observer of state chage

4 query state

5 update view

= control flow

Main Program

```
int main( int argc, char * argv[] ) {
  Gtk::Main  kit( argc, argv );         // Initialize gtkmm

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

  Gtk::Main::run( view );               // Show the window

  return 0;
}
```

Model

```
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_;
};
```

Model implements the Observer Pattern

- *set of observers*
- *subscribe/unsubscribe*
- *notify*

Model Implementation

The Model knows nothing about the View or Controller.

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

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

void Model::nextCard() {
  if (topCard_ == numCards-1)
    return;

  topCard_ += 1;
  notify();
}
```

```
Faces Model::face() {
  if (topCard_ == -1)
      return NOFACE;

  return (Faces)
    (topCard_ / numSuits);
}

Suits Model::suit() {
  if (topCard_ == -1)
      return NOSUIT;

  return (Suits)
    (topCard_ % numSuits);
}
```

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::HBox panels;         // components for the display
  Gtk::VBox 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), ...
{
  // Sets some properties of the window.
  ...

  // 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();

  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();
}
```

View Implementation 2

```
// 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.