Lecture 13

- UML diagrams
  - state-chart diagrams
- Design Pattern

Hat Problem

- Three persons each one wearing a hat that is either red or blue with 50% probability.
- A person can see the colours of the other two hats but not its own colour.
- Each player can either make no prediction or give a guess on the colour of its own hat.
- The team wins if at least one player guessed correctly and no player guessed a wrong answer.
- There is no form of communication between the players, in particular they do not know which decision any of the other players made.
- The players can agree beforehand on a common fixed strategy.

Hat Problem

- Simple strategy: Player A always guesses colour red, the other players make no guess. Probability of winning is 50%.
- Common strategy: Players agree that when a player sees two hats with identical colours, then he guesses the opposite colour, if he sees two different colours he makes no guess. Chance of winning as a team is 75%
- 25% RRR, BBB: all three players guess wrong
- 75% RRB, RBR, RBB, BBR, BRR: one player guesses correctly, the other two players make no guess

UML Statechart Diagrams

- Statechart diagrams lie within the behavioral view of a system and render the states and responses of a class.
- Statechart diagrams describe the behaviour of a class in response to external stimuli.
- Statechart diagrams describe the life cycle of an object.
- Statechart diagrams contain
  - States
  - Transitions

Finite State Machines (FSM)

- States
  - An FSM is exactly in one current state.
  - An FSM has an initial state.
  - An FSM may change state due to a transition.
- Transitions
  - Transitions connect states with each other.
  - Transitions carry a label or a guard condition that triggers the transition.
  - Transitions can be defined by a transition matrix.

FSM Example

- States: A, B, C, A is initial state
- Labels: 0, 1
- Transitions:
  - (A, 0) -> B, (A, 1) -> C, (B, 0) -> B
  - (B, 1) -> C, (C, 0) -> C, (C, 1) -> B

\[ \begin{array}{ccc}
0 & 1 & 1 \\
1 & 1 & 0 \\
2 & 0 & 0 \\
\end{array} \]
**FSM Example**

- Input sequence: 011010
- State sequence: A-B-C-B-B-C-C
- State B: even number of 1’s in sequence
- State C: odd number of 1’s in sequence

**Statechart Diagrams**

- States are classes that define status conditions that an object may satisfy during its existence.
- States are used to model the situations during the life of an entity in which it satisfies some condition, performs some activity, or waits for some occurrence.
- An entity remains in a state for a finite and non-instantaneous time.

**Statechart Diagrams**

- Transitions are associations between states. They are used to model relationships between the different states of an entity.
- An entity in one state will perform actions and possibly enter another state when an event occurs and a condition is satisfied (if specified), this occurrence is known as the firing of a transition.

**Statechart Diagram Diagram**

- Transitions are denoted as solid arrows between states and are labeled with a transition symbol.
- Transitions relate source state and target state vertices.
- Transitions may have square brackets containing a boolean guard condition that must be satisfied in order to enable the transition to fire.
- Transitions may have a forward slash followed by one or more actions that result when the transition fires.

**Example Statechart Diagram**

- Initial pseudo state
- State creation
- Transition
- Guard
- Action
- Not Selected
- Selected
- Click event (inside object) / add to selected graphics
- Click event (outside object and not shift key) / delete from selected graphics
- Click event (shift key pressed)
The Mediator pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interaction independently.

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects, in the worst case every object ends up knowing about every other. Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tends to reduce it again. Lots of interconnections make it less likely that an object can work without the support of others-the system acts as if it were monolithic.

The dialog box uses a collection of widgets such as, menus, list boxes, buttons, entry fields. Often there are dependencies between the widgets in the dialog, for example the marital status buttons get disabled when the entry field is empty, or selecting an entry in the list box changes the content of the entry field, and typing in the entry field changes the highlighted entry in the list box.

Different dialog boxes will have different dependencies between widgets, which makes it impossible to simply reuse a standard set of widget classes. Instead widget classes have to be customized to reflect dialog-specific dependencies, which would require a large number of separate subclasses for different types of dialogs.
**Mediator Pattern**

- Encapsulating the collective behavior in a separate Mediator object avoids these problems.
- A Mediator object is responsible for controlling and coordinating the interactions of a group of objects.
- The Mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly.
- The objects only know the Mediator thereby reducing the number of interactions.

**FormDialogDirector**

- The FormDialogDirector is the mediator between the widgets in the dialog box.
- The FormDialogDirector knows the widgets in a dialog and coordinates their interaction.
- The FormDialogDirector acts as a hub of communications for widgets.

**DialogDirector**

- DialogDirector is an abstract class that defines the overall behavior of a dialog.
- Clients call the ShowDialog operation to display the dialog on the screen.
- CreateWidgets is an abstract operation for creating the widgets of a dialog.
- WidgetChanged is another abstract operation, widgets call it to inform their director that they have changed.
- DialogDirector subclasses override CreateWidgets to create the proper widgets, and they override WidgetChanged to handle the changes.
class DialogDirector {
public:
    virtual void ShowDialog();
    virtual void WidgetChanged(Widget *) = 0;
protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
class Widget {
public:
    Widget(DialogDirector *);
    virtual void Changed();
    virtual void HandleMouseEvent(MouseEvent& event);
private:
    DialogDirector* director_;
};
void Widget::Changed() {
    director_->WidgetChanged(this);
}
class ListBox : public Widget {
public:
    ListBox(DialogDirector *);
    virtual const string GetSelection();
    virtual void HighLight(string selection);
    virtual void SetList(list<string> newlistItems);
    virtual void HandleMouseEvent(MouseEvent& event);
private:
    list<string> listItems;
};
class EntryField : public Widget {
public:
    EntryField(DialogDirector *);
    virtual const string GetText();
    virtual void SetText(const string newtext);
    virtual void HandleMouseEvent(MouseEvent& event);
private:
    string text;
};
class Button : public Widget {
public:
    Button(DialogDirector *);
    virtual void SetText(const string newtext);
    virtual void HandleMouseEvent(MouseEvent& event);
    virtual void Activate();
    virtual void DeActivate();
private:
    bool active;
};
class FormDialogDirector : public DialogDirector {
public:
    FormDialogDirector();
    virtual void WidgetChanged(Widget *);
protected:
    virtual void CreateWidgets();
private:
    ListBox* list;
    EntryField* field;
    Button* ok_button;
    Button* cancel_button;
};
**Mediator Sample Code**

```cpp
class FormDialogDirector::CreateWidgets()
{
    list = new ListBox(this);
    field = new EntryField(this);
    ok_button = new Button(this);
    cancel_button = new Button(this);
    ok_button->DeActivate();
    ok_button->SetText("OK");
    cancel_button->Activate();
    cancel_button->SetText("Cancel");
    // fill the ListBox with the available names
    list->SetList(...);
}
```

```cpp
class FormDialogDirector::WidgetChanged
(Widget* ChangedWidget) {
    if (ChangedWidget==list)
        field->SetText(list->GetSelection());
    if (ChangedWidget==field) {
        list->Highlight(field->GetText());
        if (field->GetText() !="")
            ok_button->Activate();
        else
            ok_button->DeActivate();
}
    if (ChangedWidget==ok_button)
        ...
}
```

**Mediator Applicability**

Use the Mediator pattern when

- A set of objects communicate in well-defined complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that is distributed between several classes should be customizable without a lot of subclassing.

**Mediator Pattern Structure**

![Mediator Pattern Structure Diagram]

**Mediator Pattern Participants**

- Mediator
  - defines an interface for communicating with Colleague objects.
- ConcreteMediator
  - Implements cooperative behavior by coordinating Colleague objects.
- Colleague classes
  - Each colleague knows its mediator
  - Each colleague communicates with its Mediator whenever it would have otherwise communicated with another colleague

**Mediator Pattern Collaborations**

- Colleagues send and receive requests from a Mediator object.
- The Mediator implements the cooperative behavior by routing requests between the appropriate colleagues
Mediator Pattern Consequences

- The Mediator pattern limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only, Colleague classes can be reused.
- The Mediator pattern decouples colleagues. A mediator promotes loose coupling between colleagues. You can vary and reuse Colleague and Mediator classes independently.

Mediator Pattern Consequences

- The Mediator pattern simplifies object protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain and extend.
- The Mediator pattern abstracts how objects cooperate. Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

Mediator Pattern Implement.

- Omitting the abstract Mediator class. There is no need to define an abstract Mediator class when colleagues work with only one mediator. The abstract coupling that the Mediator class provides lets colleagues work with different subclasses and vice versa.

Mediator Pattern Implement.

- Colleague-Mediator communication. Colleagues have to communicate with their mediator when an event of interest occurs. One approach is to implement the Mediator as an Observer. Colleague classes act as Subjects, sending notifications to the mediator whenever they change state. The mediator responds by propagating the effects of the change to the other colleagues.
**Singleton Pattern**

- The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.
- Examples:
  - There can be many printers in a system but there should only be one printer spooler.
  - There should be only one instance of a WindowManager (GrainWindowingSystem).
  - There should be only one instance of a filesystem.

**Singleton Pattern**

- How do we ensure that a class has only one instance and that the instance is easily accessible?
- A global variable makes an object accessible, but does not keep you from instantiating multiple objects.
- A better solution is to make the class itself responsible for keeping track of its sole instance. The class ensures that no other instance can be created (by intercepting requests to create new objects) and it provides a way to access the instance.

**Singleton Pattern**

Use the Singleton pattern when

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

**Singleton Structure**

```
Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueinstance
singletonData
return uniqueinstance
```

**Singleton Participants**

- Singleton
  - Defines an Instance operation that lets clients access its unique instance. Instance is a class operation (static member function in C++)
  - May be responsible for creating its own unique instance
  - Client
    - Accesses a Singleton instance solely through Singleton’s Instance operation.

**Singleton Consequences**

- Controlled access to sole instance
  - Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
- Reduced name space
  - The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
Singleton Consequences

- Permits refinement of operations and representations
  The Singleton class may be subclassed and it is easy to configure an application with an instance of this extended class at run-time.
- More flexible than class operations
  An alternative is to use static member functions in C++. However it is difficult to change the design to allow more than one instance of a class and static member functions in C++ are never virtual, so subclasses can not override polymorphically.

Singleton Implementation

- Ensuring a unique instance
  The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a static class operation that guarantees that only one instance is created.

Singleton Sample Code

class Singleton {
  public:
    static Singleton* Instance();
    // clients access the Singleton exclusively through
    // the Instance() member function
  protected:
    Singleton();
    // the constructor is protected, such that a client
    // which tries to instantiate a Singleton object gets
    // a compiler error
  private:
    static Singleton* instance_;
};

Singleton Sample Code

Singleton* Singleton::instance_ = 0;
// initialize static member data of class Singleton

Singleton* Singleton::Instance()
{
  if (instance_ == 0) // if not created yet
    instance_ = new Singleton; // create once
  return instance_;
}