

Lecture 12

- Observer Pattern
- UML
 - Class Diagrams (repetition)
 - Use Case Diagrams
 - Sequence Diagrams
 - Statechart Diagrams

Binder with Course Material

- A binder with extra material for course OOPK01 is now available at student-expedition for copying
 - Composite and Observer pattern from the book "Design Patterns"
 - UML tutorial chapter 4 from the book "UML in a nutshell"
 - UML quick reference
 - Lecture slides
 - Lab assignments
 - STL tutorial and quick reference

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Example: Graphical editor with multiple views of the same graphic objects. If a graphical object is manipulated in one view, it notifies the other views to display the changes made to it.

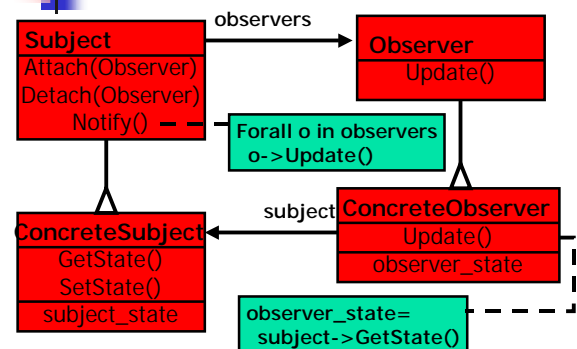
Observer Pattern

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- The key objects in the Observer pattern are Subject and Observer.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.
- In response, each observer will query the subject to synchronize its state with the subject's state.

Observer Applicability

- Use the Observer pattern in any of the following situations
 - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - When a change to one object requires changing others, and you do not know how many objects need to be changed.
 - When an object should be able to notify other objects without making assumptions about who these objects are.

Observer Structure



Observer Participants

- Subject
 - Knows its observers. Any number of Observer objects may observe a subject.
 - Provides an interface for attaching and detaching Observer Objects.
- Observer
 - Defines an updating interface for objects that should be notified of changes in a subject

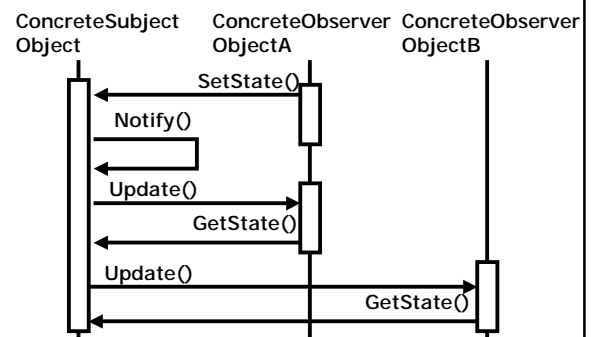
Observer Participants

- ConcreteSubject
 - Stores a state of interest to ConcreteObserver objects.
 - Sends a notification to its observers when its state changes.
- ConcreteObserver
 - Maintains a reference to a ConcreteSubject object
 - Stores state that should stay consistent with the subject state.
 - Implements the Observer updating interface to keep its state consistent with the subject state.

Observer Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observer's state inconsistent with its own.
- After being informed of a change in the ConcreteSubject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the object.

Observer Sequence Diagram



Observer Sequence Diagram

- Note that the Observer object that initiates the change request with SetState() postpones its update until it gets a notification from the subject.
- In this scenario Notify() is called by the subject, but it can be called by an observer or by another kind of object (see implementation issues)

Observer.h

```

#include <list>
#include <algorithm>
#include <string>

class Subject;

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

Observer.h

```
class Subject
{
public:
    virtual ~Subject() {};
    void attach(Observer* obs);
    void detach(Observer* obs);
    void notify();

private:
    list<Observer*> observers_;
};
```

Observer.h

```
class ConcreteSubject : public Subject
{
public:
    ConcreteSubject(string newname);
    void setstate(int newstate);
    int getstate();
    string getname();
private:
    int state_;
    string name_;
};
```

Observer.h

```
class ConcreteObserver : public Observer
{
public:
    ConcreteObserver(string newname);
    void attachsubject(ConcreteSubject*
subject);
    void detachsubject(ConcreteSubject*
subject);
    virtual void update(Subject* subject);

private:
    list<ConcreteSubject*> subjects_;
    string name_;
};
```

Observer.cc

```
#include "observer.h"
Observer::Observer() {};
void Subject::attach(Observer* obs)
{
    observers_.push_back(obs);
};

void Subject::detach(Observer* obs)
{
    observers_.erase(find(observers_.begin(
), observers_.end(), obs));
};
```

Observer.cc

```
void Subject::notify()
{
    for (list<Observer*>::iterator
iter=observers_.begin();
iter!=observers_.end(); ++iter) {
        (*iter)->update(this);
    };
};

ConcreteSubject::ConcreteSubject(string
newname) : Subject(),
name_(newname), state_(0)
{
};
```

Observer.cc

```
void ConcreteSubject::setstate(int newstate)
{
    state_=newstate;
    notify(); // notify all observers that state
              // has changed
}

int ConcreteSubject::getstate()
{
    return state_;
}

string ConcreteSubject::getname()
{
    return name_;
}
```

Observer.cc

```
ConcreteObserver::ConcreteObserver(string
newname) : Observer(), subjects_(0),
name_(newname)
{
};

void ConcreteObserver::AttachSubject
(ConcreteSubject* subject)
{
    subjects_.push_back(subject);
    subject->attach(this);
};
```

Observer.cc

```
void ConcreteObserver::detachsubject
(ConcreteSubject* subject)
{
    subjects_.erase(find(subjects_.begin(),
subjects_.end(),subject));
    subject->detach(this);
    cout << "Subject detached!" << endl;
};
```

Observer.cc

```
void ConcreteObserver::update(Subject*
subject)
{
    list<ConcreteSubject*>::iterator iter;

    iter=find(subjects_.begin(),subjects_.e
nd(),subject);
    if (iter != subjects_.end())
        cout << "Observer " << name_ << "
queries Subject " << (*iter)->getname()
<< " for new state " << (*iter)-
>getstate() << endl;
}
```

Main.cc

```
#include "observer.h"
int main(){
    ConcreteSubject s1("s1");
    ConcreteSubject s2("s2");
    ConcreteObserver o1("o1");
    ConcreteObserver o2("o2");
    ConcreteObserver o3("o3");
    o1.AttachSubject(&s1);
    o1.AttachSubject(&s2);
    o2.AttachSubject(&s1);
    o3.AttachSubject(&s2);
    s1.SetState(5);
    s2.SetState(2);
}
```

Program Output

In s1.setState(5) notify observers o1 and o2
Observer o1 queries Subject s1 for new state 5
Observer o2 queries Subject s1 for new state 5

In s2.setState(2) notify observers o1 and o3
Observer o1 queries Subject s2 for new state 2
Observer o3 queries Subject s2 for new state 2

Observer Consequences

- The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Observer Consequences

- Abstract coupling between Subject and Object
 - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject does not know the concrete class of any observer.
- Support for broadcast communication
 - Unlike an ordinary request, the notification that a subject sends need not specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it.

Observer Consequences

- Unexpected Updates
 - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation to the subject may cause a cascade of updates to observers and their dependent objects.
 - This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. Without additional protocol observers have to query the entire state of the subject to deduce the changes.

Observer Implementation

- Mapping subjects to their observers
 - The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. An alternative is to use an associative look-up (multimap) to maintain the subject-observer mapping.

Observer Implementation

- Observing more than one subject
 - It might make sense in some situations for an observer to depend on more than one subject. It is necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.

Observer Implementation

- Who triggers the update
 - The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify() to trigger the update? There are two options.
 - Have state-setting operations on the Subject call Notify after they change the subject's state
 - Make clients responsible for calling Notify at the right time.

Observer Implementation

- Subject calls Notify
 - The advantage of this approach is that clients do not have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient
- Clients call Notify
 - The advantage here is that the client can wait to trigger the update until after a series of state changes has been made. The disadvantage is that clients have an added responsibility to trigger the update, causing possible errors.

Observer Implementation

- Subject calls Notify

```
void ConcreteSubject::setstate(int newstate)
{
    state_=newstate;
    notify(); // notify all observers that //
             // state has changed
}
```

```
Subject s1;
subject.setstate(6);
// automatic call to notify
Subject.setstate(5);
```

Observer Implementation

- Observer or client calls Notify

```
void ConcreteSubject::setstate(int newstate)
{
    state_=newstate;
}
```

```
Subject s1;
subject.setstate(6);
subject.setstate(5);
Subject.notify(); // explicit call to notify
```

Observer Implementation

- Dangling references to deleted subjects
 - Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference. In general simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

Observer Implementation

- Making sure subject is self-consistent before notification
 - It is important to make sure Subject is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state. This self-consistency rule is easy to violate when Subject subclass operations call inherited operations.

Observer Implementation

- Avoiding observer-specific update protocols : the push and pull model
 - Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update.
 - Push model: the subject sends observers detailed information about the change
 - Pull model: the subject sends only a minimal notification and observers ask for details explicitly

Observer Implementation

- Pull model:
 - The pull model emphasizes the subject's ignorance of its observers. The pull model may be inefficient, because Observer classes must infer what changed without help from Subject.
- Push model:
 - The push model assumes that the subject knows something about its observers' needs.
 - The push model might make observers less reusable because Subject classes make assumptions about Observer classes that might not always be true.

Observer Implementation

- Specifying modifications of interest explicitly.
 - You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this is to use the notion of *Aspects* for Subject objects.

```
void Subject::Attach(Observer*, Aspect& interest);
void Observer::Update(Subject*, Aspect& interest);
```

Unified Modeling Language

- Use case diagrams describe the functionality of a system and users of the system. They contain the following elements
 - Actors which represent users of the system, including human users and other systems
 - Use cases which represent functionality or services provided by a system to users.

Unified Modeling Language

- UML object model describes the static structure of the problem domain, it contains
 - Class diagram describes the class attributes (name and type), operations and the relationship among classes. A class diagram contains classes and associations
 - Object diagram A class model describes all possible situation, whereas an object model describes a particular situation. An object diagram contains objects and links which represent the relationship between objects

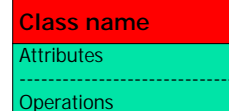
Unified Modeling Language

- The UML dynamic model describes all temporal and dynamic aspects of the system, for example interaction among objects it contains
 - Sequence diagrams describe interactions among classes which are modeled as exchanges of messages. Sequence diagrams contain class roles, lifelines, activations and messages
 - Scenarios describe interactions among objects
 - Statechart diagrams describe the states and responses of a class. They contain states and transitions.

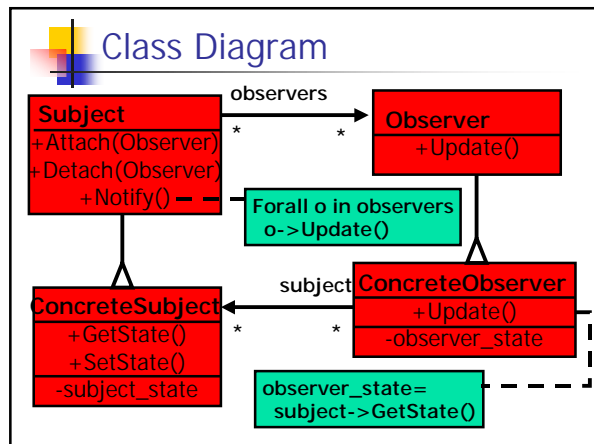
Class Diagrams

- Class diagrams describe the static structure of a system rather than its behaviour. Class diagrams contain the following elements:
 - Classes Which represent entities with common characteristics or features. These features include attributes (data members), operations (member functions) and associations
 - Associations which represent relationship that relate two or more other classes where the relationships have common characteristics or features.

Class Diagram



<div style="background-color: red; color: white; padding: 2px;">Account</div> <div style="background-color: #00ffcc; padding: 2px;">-number : int -balance : float</div> <hr style="border-top: 1px dashed black;"/> <div style="background-color: #00ffcc; padding: 2px;">+withdraw(amount : float) +deposit(amount : float)</div>	<pre>class Account { private: int number; double balance; public: void withdraw(double amount); void deposit(double amount); };</pre>
--	---

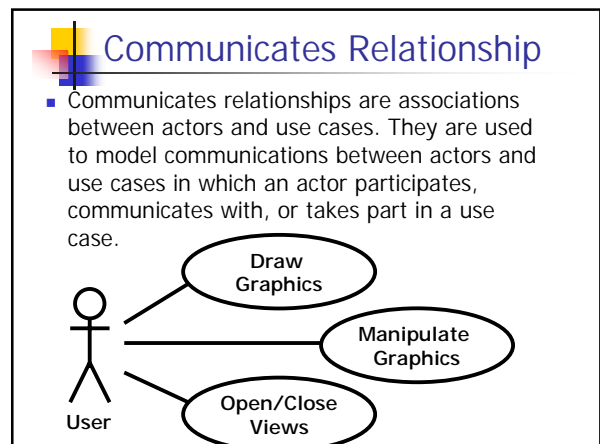


- ### DIA
- DIA is a diagram creation program from Linköping University
<http://www.lysator.liu.se/~alla/dia/dia.html>
 - It can be used to draw many different kinds of diagrams
 - Entity relationship diagrams
 - UML diagrams
 - Flowcharts
 - Network diagrams
 - It is available within the OOPK01 course module with the command : **dia**

- ### Use Case Diagrams
- Use case diagrams render the user view of a system.
 - Use case diagrams describe the functionality provided by the system or class to external actors.
 - Use case diagrams contain
 - Actors
 - Use cases
 - Their relationships

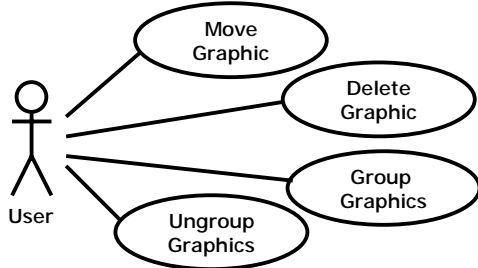
- ### Actors
- Actors are classes that define roles that objects external to a system may play. They are used to model users outside of a system that interact directly with the system as part of coherent work units. This includes human users and other systems
 - Actors are characterized by their external view rather than their internal structure.
 - Actors participate in interactions involving message exchanges and actions with systems.
 - Actors have goals to be achieved by interacting with the system.

- ### Use Cases
- Use cases are classes that define units of functionality or behavior provided by the system.
 - Use cases specify the external requirements of the system and the functionality offered by the system.
 - Use cases are *specified* by *sequence diagrams* representing the external interaction sequences among the systems and its actors.
 - Use cases are *realized*, or implemented, by *collaboration diagrams* representing the internal refinement of the services provided by the system.



Manipulate Graphics Use Case

- Detailing the functionality of the Manipulate Graphics Use Case

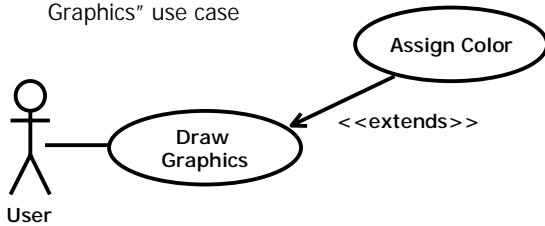


Extends Relationships

- Extends relationships are generalizations between use cases. They are used to model relationships between use cases in which a base use case instance may include the behavior specified by an extending use case, subject to conditions specified in the extension.
- Extends relationships are used to capture exceptional behavior or variations of normal behavior.

Extends Relationships

- The arrow from the "Assign Color" use case to the "Draw Graphics" use case is labeled with an <<extends>> stereotype to indicate that this use case is an option of the "Draw Graphics" use case

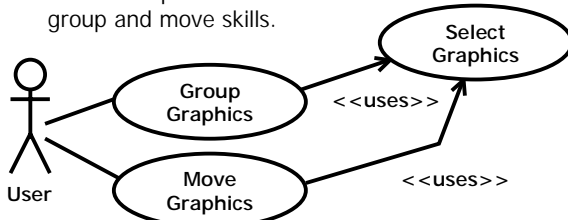


Uses Relationships

- Uses relationships are generalizations between use cases. They are used to model relationships between use cases in which a base use case instance will also include the behavior specified by a common use case.
- Use relationships are used to share common behavior among use cases.

Uses Relationships

- The arrow from the "Group Graphics" and "Move Graphics" use case to the "Select Graphics" use case is labeled with an <<uses>> stereotype to indicate that the "Select Graphics" use case is included in the group and move skills.



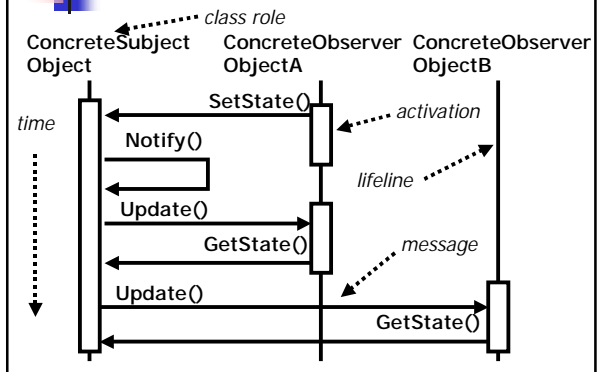
Sequence Diagrams

- Sequence diagrams describe interactions among classes. These interactions are modeled as exchanges of messages.
- Sequence diagrams focus on classes and the messages they exchange to accomplish some desired behavior.

Sequence Diagrams

- Sequence diagrams are a type of interaction diagrams that contain
 - Class roles** represent roles that objects may play within the interaction.
 - Lifelines** represent the existence of an object over a period of time.
 - Activations** represent the time during which an object is performing an operation.
 - Messages** represent communications between objects.

Observer Sequence Diagram



Observer Sequence Diagram

