7

• Design patterns
  - Definition
  - Example

• Design Patterns in Distributed systems
  - Observer
  - Command
  - Memento
Design patterns

- Definition
  - A design pattern is a tried & tested solution to a common design problem

- Compare with problem frames:
  - A problem frame is a common form of a problem
  - A design pattern is a common form of a solution
    - … in the design space – there are also patterns in the implementation, e.g. standard bits of code

- As for all patterns, it's an idea, not a rule
  - Amenable to adaptation
**Design patterns**

- A design pattern is characterized by
  - A **name**
  - A description of the **problem** it aims to solve
  - A description of the **solution**
    - Elements of the design
    - Relationships among them
      - Interactions, responsibilities, collaboration
  - A discussion of the **consequences** of applying the pattern
    - Design trade-offs
An example: MVC

- One of the most famous patterns: Model-View-Controller
- Originally introduced in the Smalltalk-80 base library
- **Problem**: a good general way to handle user interface components
- **Solution**: use three different objects, with well-defined interfaces but arbitrary implementations
  - Model, View, Controller
An example: MVC

- **Model**: an object that provides a purely abstract description of the “thing” that is to be represented by the UI control
- **View**: an object that, given the data in the Model, can render it on-screen in some form
- **Controller**: an object that, given some user input (e.g., a mouse click or keypress), alters the Model (or possibly the View) according to user's intentions
An example: MVC

• The relationship between Model, View and Controller is **dynamic**
  – It can be set-up and changed at runtime
    • e.g., need to disable a GUI element to prevent issuing of invalid commands? Change its Controller to a dummy one that ignores all user input

• Each object has precise **responsibilities**
  – Described in terms of the **interfaces** it must offer to other objects
    • e.g., all Controllers must implement the same interface, regardless of their actual class
An example: MVC
An example: MVC

- The basic MVC pattern uses 1:1 relationships between Model, View, Controller
- With further massaging, these can become $n:m$ relationships
- Most often seen as multiple views for the same model
  - Hint: in a distributed system, each view can be on a different machine and use different media
Design patterns in distributed systems

- Most design patterns assume that...
  - Objects have a private state
  - Objects can communicate by invoking operations
  - Objects can exchange arbitrary data as parameters attached to such operations
  - Objects have their own control flow
    - Either their own thread, or hijacking the control flow of the caller

- All these properties can be scaled up to units of a distributed systems
  - Computation + memory + message-passing
The 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”

• This allows to keep a single copy of the data, and have multiple other objects depend on them
  - Used e.g. in multi-view MVC
  - Can be used for asymmetric replication and notification in distributed systems
The Observer pattern

- **Subject (interface)**
  - The thing to be observed
  - Maintains a set of observers

- **ConcreteSubject (object)**
  - Has the actual state
  - Provides operations to retrieve and alter the state
The Observer pattern

- **Observer (interface)**
  - The thing to be notified

- **ConcreteObserver (object)**
  - Has a local copy of the remote ConcreteSubject state
  - Goal is to keep the copy up-to-date

*(note: applicable to parts of state)*
The Observer pattern

- **Registration**
  - a.k.a., *Subscribe*
- An observer calls subject.attach(self)
- The subject adds the observer to the set of current observers

- **De-registration**
  - a.k.a., *Unsubscribe*
- An observer calls subject.detach(self)
- The subject removes the observer from the set of current observers
The Observer pattern

- The state of ConcreteSubject changes
  - Due to a call to a setState() method or due to some autonomous event

SetState() could also be called by a third party (e.g., a Controller in MVC)
The Observer pattern

- ConcreteSubject calls notify() of Subject
  - Most often, Subject is an abstract class implementing notify() — could also be an interface
The Observer pattern

- Notify() loops over all registered observers
  - Calling update() on each
  - Each observer calls getState() on the subject
Observer vs. Publish & Subscribe

- The Observer pattern is a variation of a more general protocol known as **Publish & Subscribe**
- The Subscribe part is identical to registration and de-registration via `attach()` and `detach()`
- The Publish part is more general
  - In Observer, the only cause for broadcast are changes in the state
  - In P&S, any event can be published
    - Details of the event are often sent as parameters of `update()`, not retrieved via separate `getState()`s
Implementation of Observer

```java
public class Subject {
    List<Observer> obs = new ArrayList<Observer>();

    public Observable() { super(); }
    public void attach(Observer o) { obs.add(o); }
    public void detach(Observer o) { obs.remove(o); }
    public void notify(Object data) {
        for (Observer o: obs) o.update(this, data);
    }
}

Adapted (and simplified) from java.util.Observable
```
Implementation of Observer

public interface Observer {
    public void update(Subject s, Object data);
}

Adapted (and simplified) from java.util.Observer
Implementation of Observer

```java
public class concreteSubject extends Subject {
    declarations for concrete state
    constructors etc.

    public void setState(args) {
        updates state based on arguments
        this.notify(object describing change)
    }

    public State getState(args) {
        return state based on arguments
    }
}
```
public class concreteObserver implements Observer {
    ...
    public void update(Subject s, Object data) {
        ObservedState = s.getState(args);
        Reacts to changes – for example, by updating a local copy of the Subject's state, or by redrawing a View, etc.
    }
    ...
}
Observer in a distributed system

- When applied in a distributed application
  - Subject and Observer often reside on different nodes
  - Communications among the two can be
    - Slow
    - Costly
    - Unreliable
    - Limited capacity
Observer in a distributed system

- Invoking operations across different nodes
  - Several options
    - Use CORBA, RMI, or other RPC mechanisms
    - Send a message encoding the request according to some agreed-upon protocol
    - Use ad-hoc signaling
      - e.g., on receipt of an SMS with text “update” the machine will...
Observer in a distributed system

- Invoking operations across different nodes
  - Several options
    - Use CORBA, RMI, or other RPC mechanisms
    - Send a message encoding the request according to some agreed-upon protocol
    - Use ad-hoc signaling
      - e.g., on receipt of an SMS with text “update” the machine will...

A theme for a Network Programming course

(Will discuss it later on)
Observer in a distributed system

- Establishing identity across different nodes
  - `attach()` and `detach()` are easy with local objects
    - Storing a pointer to the observer suffices
  - More complex in a distributed system
    - Need some sort of unique ID
Observer in a distributed system

- Concurrent execution of updates
  - Each node can perform whatever its own update() requires in parallel with others
  - No need for a call to update() to be blocking
    - Same holds locally, proper synchronization
    - Use broadcast for update()
Building a cost model for Observer

- **Cost for attach() and detach()**
  - One call + passing of ID for each
  - (possible hidden cost for accessing a network ID)

- **Cost for each update()**
  - One call [for update()] + passing of ID + passing of data
  - One call [for getState()] + passing of state

- **Cost for each notify()**
  - $K$ updates(), with $K =$ number of registered observers
Building a cost model for Observer

- Cost for attach() and detach()
  - One call + passing of ID for each
  - (possible hidden cost for accessing a network ID)
- Cost for each update()
  - One call [for update()] + passing of ID + passing of data
  - One call [for getState()] + passing of state
  - $K$ updates, with $K$ = number of registered observers
- Cost for each notify()
  - $K$ updates, with $K$ = number of registered observers

These are typically infrequent operations

In most systems, only performed at boot-up or shutdown

In some system, performed when a node joins/leaves the distributed system

Rarely, hugely dynamic
Building a cost model for Observer

- Cost for attach() and detach()
  - One call + passing of ID for each
  - (possible hidden cost for accessing a network ID)

- Cost for each update()
  - One call [for update()] + passing of ID + passing of data
  - One call [for getState()] + passing of state

- Cost for each notify()
  - $K$ updates(), with $K =$ number of registered observers

This part is paid at each state change
Cost proportional to (serialized) size of the state and to the number of observers
Can become HUGE!
Optimizing the distributed Observer

• We need strategies to reduce the cost of Observer in a distributed application
• Main venues:
  – Reduce the number of updates
  – Reduce the size of each update
  – Reduce the number of observers
• The particular problem will often dictate what is possible and what is not
• Strike a balance between code complexity (→ robustness) and performance (→ efficiency)
Reducing the # of updates

- Coalescing
  - At times, it is not sensible to send out many little updates: it's better to **coalesce** many setState() calls, then send out a single cumulative notify()
  - Add two operations to Subject
    - hold() - suspends all updates
    - release() - resumes sending out updates
      - Also, sends out a first notify() if there was any change w.r.t. the previous hold()
  - Risk: hold() without release()!
  - Increases code complexity (e.g., multiple calls)
Reducing the # of updates

- **Partitioning**
  - Upon registration, express interest in some subset of the state
  - Only send out updates to Observers that have expressed interest in the changed partition
  - Equivalent to having many smaller Subjects

- **Implementation**
  - Add a parameter *interest* to attach() (often, a bitmask), or
  
  Add an operation `setInterest(o,i)` to express that observer *o* is interested in facet *i* of the state
Reducing the # of updates

**Flow control**
- Stop sending further updates until the Observer has finished processing the previous set
- Also helps with the overrun concern
- Needs an additional cost to signal completion

**Implementation**
- In notify(), use an asynch invocation for update()
- Put every notified Observer in a “suspended” set
- Add an operation done() to resume an observer
- In the implementation of notify(), call done() once finished
Reducing the # of updates

- **Flow control**
  - Stop sending further updates until the Observer has finished processing the previous set.
  - Also helps with the overrun concern.
  - Needs an additional cost to signal completion.

- **Implementation**
  - In `notify()`, use an asynchronous invocation for `update()`.
  - Put every notified Observer in a “suspended” set.
  - Add an operation `done()` to resume an observer.
  - In the implementation of `notify()`, call `done()` once finished.

**Might miss intermediate states**

Applicable when the “most recent state” counts, and older states are of little interest (real-time applications).

Not applicable when all updates are significant (e.g., financial transactions).
Reducing the # of updates

• **Shifting responsibility to clients**
  - Instead of triggering an update at each `setState()`, allow clients to call `notify()` when they think that observers need to be notified
  - Only applicable if clients of the Subject have an idea about the needs of Observers
  - Reduces decoupling, makes systems more tangled
  - Increases chances of missing an update
    • i.e., client “forgets” to call `notify()`
Reducing the size of each update

- **Using small getters**
  - In our scheme, update() has a negligible payload
  - getState() is where the largest amount of data is transferred
  - Replace getState() with finer-grain getters
    - Each get...() pays the cost of 1 call + the cost for transferring the data
    - Balancing: too many getters to call, and you end up paying more than a single call to transfer the whole state
Reducing the size of each update

- Put the payload in `update()`
  - Instead of having `update()` cause a call to `getState()`, pass the state change as parameter
  - Opposite to coalescing, friendly to partition
- Implementation

```java
public void setX(T x) {
    T oldValue = x;
    this.x = x;
    notify("x", oldValue, x);  // update
}
```
Reducing the size of each update

• Push model
  - Each setX() sends full notification for that particular update
  - Observer has it all

• Pull model
  - Each setX() sends just a notify(void)
  - Observer decides if, what, when to get...()

• Intermediate models
  - Some of the information about a change is sent with update()
  - Some is retrieved by the Observer upon need
Reducing the # of observers

- Rarely we have the luxury of deciding how many observers we will have
  - e.g.: web browsers on a page from our server
- At times, it can be decided at design time
- It might be possible to keep the number of observers low by dynamic attach()/detach()
  - Balancing the cost for those with the cost for updates
- We can set a hard limit
  - the (K+1)th attach() will fail
  - QoS to already registered observers wins
Complex update strategies

When the update strategy becomes complex, it might be interesting to insulate it in a separate mediator object.
The Command pattern

- “Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations”
- Normally operations are requested by invoking a method
- With Command, operations are requested by passing an object
  - The object can carry an implementation with it
  - BUT, only few communication channels can carry code
The Command pattern

- **Command**: an interface to execute an operation
- **ConcreteCmd**: implements execution
- **Client**: creates and sends Commands
- **Invoker**: causes the execution of a Command
- **Receiver**: knows how to manage Commands
The Command pattern

1: new Command(aReceiver) → aCommand
2: StoreCommand(aCommand)
3: Execute()
4: Action()
The Command pattern

- execute() vs. action()
  - The Invoker calls execute() on the Command
  - execute() in turns calls one or more operations (action()) on the receiver to produce the desired effect

- Leeway about how much processing should be done in execute(), and how much in action()
  - The Command could be very autonomous and do all the state changing itself
  - The Command could be just a delegate and simply call an operation of the receiver
Implementing Command

```java
public interface Command {
    public abstract void execute();
}

public class Genesis implements Command {
    public void execute() { universe.start(); }
}

public class Armageddon implements Command {
    public void execute() { universe.stop(); }
}

public class MinorMiracle implements Command {
    public void execute() { universe.setState(...); }
}
```
Implementing Command

public interface Command {
    public abstract void execute();
}

public class Genesis implements Command {
    public void execute() {
        universe.start();
    }
}

public class Armageddon implements Command {
    public void execute() {
        universe.stop();
    }
}

public class MinorMiracle implements Command {
    public void execute() {
        universe.setState(...);
    }
}

Receiver, here accessed statically.
Could be a parameter set in the constructor of Command.

action() of the Receiver.
Could also be a complex set of changes, or include significant business logic
public interface Invoker {
    public void storeCommand(Command c);
}

public class PermissionInvoker {
    public void storeCommand(Command c) {
        if (requiresPermission(c)) {
            if (askUser(c)) {
                c.execute();  ← exception if “No!”
            }
        }
    }
}
public class UndoInvoker implements Invoker {
    Stack<State> undoStack = new Stack<State>();

    public void storeCommand(Command c) {
        undoStack.push(universe.getState());
        c.execute();
    }

    public void undo() {
        Universe.setState(undoStack.pop());
    }
}
Remote **Client** node requests creation of the **Command** on the local **Receiver** node. Doable, **Receiver** must provide a set of pre-defined **ConcreteCommands**. Only creation request needs to be transmitted.
Creation and dispatching of **Commands** is managed on the **Client**. Actual implementation is still on the **Receiver** (which again provides a pre-defined set).
The implementation of **Commands** is on the **Client**. Requires intimate knowledge between **ConcreteCommand** and **Receiver**. Defeats encapsulation and separation of concerns! Might require code migration.
Further separation of command management strategy from actual implementation is possible. So-called Request Queue Management Systems. Usable in high-latency, batch systems to implement logging, journaling, etc.
Goals for Command

- Implement delayed execution
  - Commands can be queued and executed later
- Implement logging/journaling/stat collection
  - A record is kept of who issued which commands to whom, execution times, etc.
- Implement undo/redo/repeat
  - Whenever a command is executed, add it to a list of undoable operations
  - Command can have undo() and redo()
  - Alternatively, can use a stack of states
Goals for Command

- Implement Command queue inspection techniques
  - Buffering and coalescing commands
    - “only last valid command counts”
  - Accumulation
    - Transform move(dx1, dy1); move(dx2, dy2) to move(dx1+dx2, dy1+dy2)

- Implement preemptible Commands
  - Allows changing your mind
    - Send! — then, you can press Cancel sending in the next 5 seconds
Goals for Command

- Allows multiple sources for the same Command
  - An icon in the tool bar
  - A menu entry
  - A keyboard shortcut
  - A scripting interface
- Allows multiple destinations for the same Command
  - “Cut” can be sent to a text, to a picture, to a sound sample...
The Memento pattern

- “Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later”
- In practice, we want an opaque container for the private state of some object
  - The owner can “lend” the state to someone else
  - Only the owner can recover the internal state
  - Still, the opaque state can be stored, transmitted etc.
The Memento pattern

- **Originator** has the state, can create **Mementos**
- **Memento** holds the state in the opaque form
- **Caretaker** can only store/retrieve/pass **Mementos**
The Memento pattern
The Memento pattern

When Caretaker requests a Memento, the Originator creates a new Memento object, fills in its state, and return the Memento to the Caretaker.
The Memento pattern

Later on, Caretaker restores the Memento; the Originator extracts its state from it, and sets the extracted state as its own state.

There is no need for the saved state to be the full state of the Originator.
Distributing Memento

Memento spans both nodes