in real life

# Design Patterns

PA14[13]5

Mikael Svahnberg[1]

2016-04-06

[1] Mikael.Svahnberg@bth.se

in real life

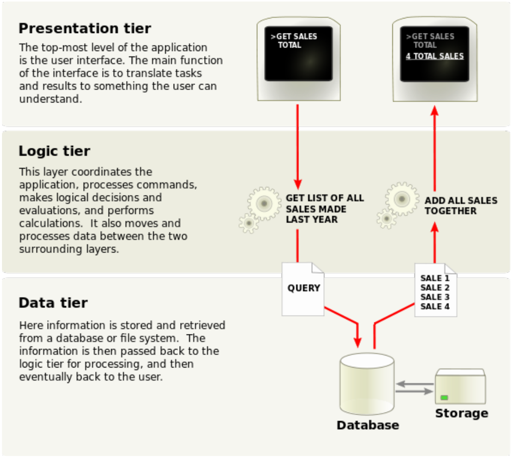
## Gang of Four Design Patterns and Architecture Patterns

- Architecture
  - Layered
  - MVC
- Design Patterns
  - Observer
  - Singleton
  - Strategy
  - State
  - Abstract Factory

## Layered

- Problem: *You have groups of subtasks that depend on other subtasks at different levels of abstractions*
- Solution: *Put the subtasks into layers, each representing a specific level of abstraction*
  - Minimise connections between layers (low coupling)
  - Assign a clear responsibility to each layer (high cohesion)
- Examples: Thee Tier Architecture, Windows 2000 Architecture

## Example: Thee Tier Architecture



**Presentation tier**

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

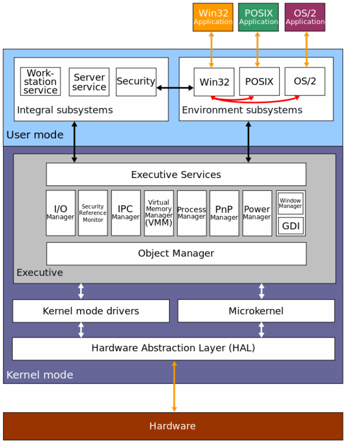
**Logic tier**

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

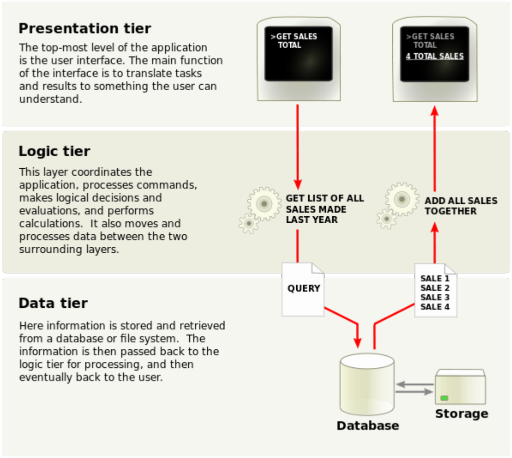
# Example: Windows 2000 Architecture



## Model View Controller: MVC

- Problem: *You have an interactive application. How should you divide responsibilities for presenting, managing, and storing data?*
- Solution: *Divide your system into three parts:*
  - Model: Maintain persistency and consistency of the data
  - View: Presentation of the data (may be more than one view)
  - Controller: Handle user input and manage business rules
- Example: Thee Tier Architecture

# Example: Thee Tier Architecture



**Presentation tier**

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

**Logic tier**

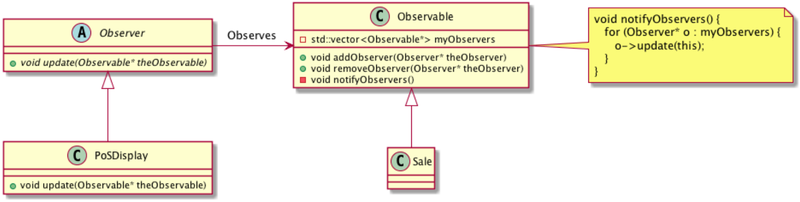
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

## Observer

- Problem: *How should one object (A) keep track of the state of another object (B)?*
- Solution: *Give B a pointer to A and ask it to notify when there are changes.*
- Illustration:


A Observer
void update(Observable* theObservable)
Observes
C Observable
std::vector<Observable*> myObservers
void addObserver(Observer* theObserver)
void removeObserver(Observer* theObserver)
void notifyObservers()
C PoSDisplay
void update(Observable* theObservable)
C Sale

```
void notifyObservers() {
  for (Observer* o : myObservers) {
    o->update(this);
  }
}
```

## Java Problem 1: Multiple Inheritance

- Problem: What if you are already extending something? Multiple inheritance is not possible in Java.
- Solution:
  - re-implement all methods of Observable :-(

## Java Problem 2: Observe multiple observables

- Problem: What if you want to observe many things
- Solution:
  - One giant switch/case statement
  - Inner Classes
  - Anonymous Inner Classes
  - Lambda function

# Java Problem 2, Alternative 1

```
// Alternative 1: Inner Classes
// ---

class DictionaryView {
    public MyFancyView(DictionaryObservable theDictObs, BannerAdObservable theAdObs) {
theDictObs.addObserver(new DictObserver());
theAdsObs.addObserver(new AdObserver());
    }

    private class DictObserver implements DictionaryObserver {
       public void update(DictionaryObservable dict) {
// Logic for updates on Dictionary in update method
       }
    }

    private class AdObserver implements BannerAdObserver {
       public void update(BannerAdObservable banner) {
// Logic for updates on Banner Ads in update method
       }
    }
}
```

## Java Problem 2, Alternative 2

```
// Alternative 2: Anonymous Inner Classes
// ---

class DictionaryView {
    public MyFancyView(DictionaryObservable theDictObs, BannerAdObservable theAdObs
theDictObs.addObserver(new DictionaryObserver() {
  @override
  public update(DictionaryObservable dict) {
    // Logic for updates on Dictionary in update method
  }
});
theAdsObs.addObserver(new AdObserver()); // Modify this in the same way
    }
}
```

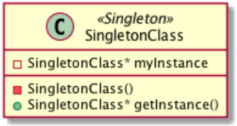
## Java Problem 2, Alternative 3

```
// Alternative 3: Lambda Function
// ---

class DictionaryView {
    public MyFancyView(DictionaryObservable theDictObs, BannerAdObservable theAdObs
theDictObs.addObserver(
  (dict) -> System.out.println("Do stuff on " +dict.toString())); // Magic and much

theAdsObs.addObserver(new AdObserver()); // Modify this in the same way
    }
}
```

# Singleton

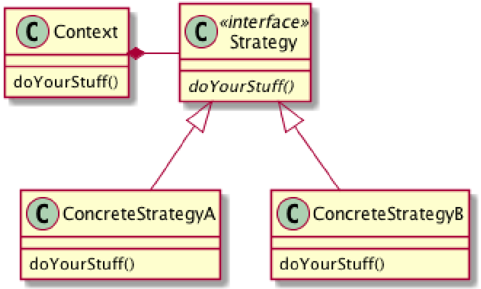
- Problem: *How do I ensure that a class has only one instance in the system, with a global point of access?*
- Solution: *Delegate the creation of the instance to a* `static` *method in the class.*
- Example:

```
class SingletonClass {
public:
  static SingletonClass* getInstance() {
    if (!myInstance) {
      myInstance = new SingletonClass();
    };
    return instance;
  }
private:
  SingletonClass() {};
  static SingletonClass* myInstance ;
};

SingletonClass* SingletonClass::myInstance=NULL;
```



# Strategy

- Problem: *There are different ways of doing the same thing; I want an extensible way of selecting between them.*
- Solution: *Use polymorphism to implement each different way.*
- Example:

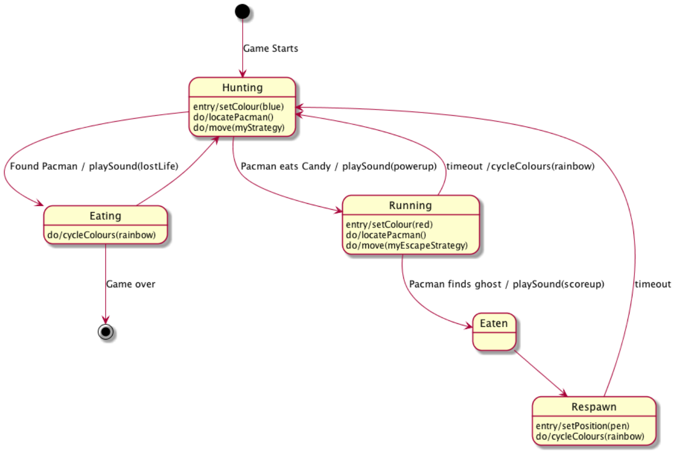


(A more concrete example: Spellcheckers)

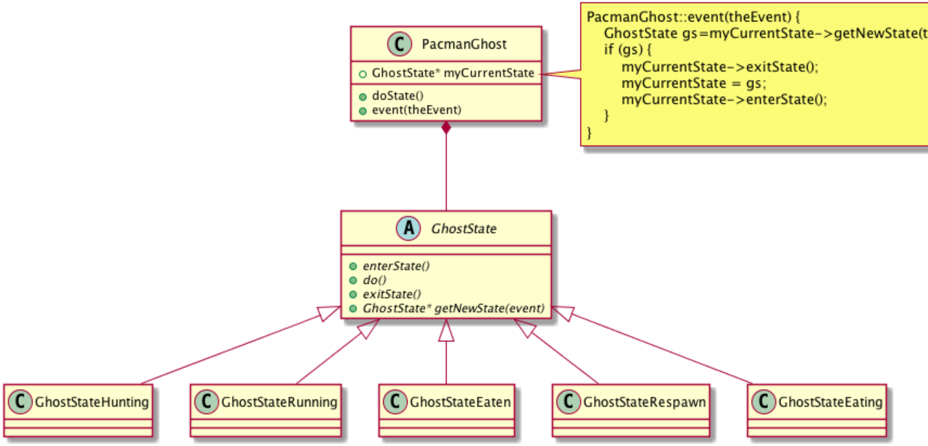
## State

- Problem: *You have a stateful system and want this to be mimicked by your class structure*
- Solution: *Implement it as a strategy pattern*
- Example:

# Example: State Diagram



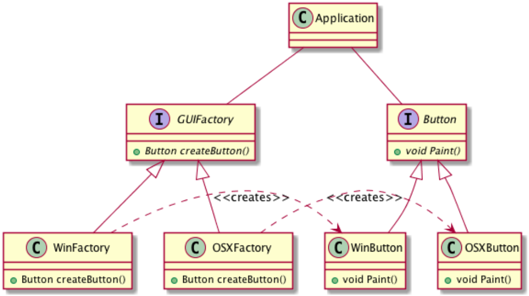
# Example: Class Diagram



```
PacmanGhost::event(theEvent) {
  GhostState gs=myCurrentState->getNewState(t
  if (gs) {
    myCurrentState->exitState();
    myCurrentState = gs;
    myCurrentState->enterState();
  }
}
```

# Abstract Factory

- Problem: *There are different ways to initiate the system, depending on the context*
- Solution: *Use a strategy-like solution to create the right objects*
- Example:


Application
GUIFactory
Button createButton()
Button
void Paint()
<<creates>>
<<creates>>
WinFactory
Button createButton()
OSXFactory
Button createButton()
WinButton
void Paint()
OSXButton
void Paint()