Software Engineering
G22.2440-001

Session 7 – Sub-Topic 4
Micro/Macro Architectures

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Bibliographie

• UML Distilled Fowler&Scott
• UML Toolkit Eriksson&Penker
• Applying UML and Patterns Larman
• Design pattern GOF
• System of patterns Buschman&al
• Penser objet avec UML&Java Lai
• Object oriented Analysis Spadounakis
• UML Specification www.rational.com
Example

- A die
- The player throws die 10 x 2
- When total is 7, player scores 10 points
- At the end of the game, the score is collected in a score map

Requirements Analysis

- First Use Case
- Identify Actors?
- Identify possible System use cases
- External functionality!
First Use Case

- **Play:**
  - Actor: Player
  - Descr: Player rolls the dices 10 times, whenever total is 7, +10pts

- **View High Score**
  - Actor: Player
  - Descr: Player looks up highest score in read-only mode

Use Case

- Very important diagram!
- A must for requirements analysis
- A must to present an application!
- MUST BE formally commented
- Used as a reference for all remaining modeling stages
Activity Diagram

- Looks awfully close to a flow diagram
- Identify activities based on a use case
- Identify transitions between activities
Activity Diagram

- Requirements analysis or analysis phase?
- More business process than object
- Provide message calling sequence and detail
  Use-cases
- Very useful for tests...

Analysis

- Model the real world
- Implementation independent
- Determine real world classes: first class diagram
- Model system’s dynamics: collaboration diagram
Collaboration Diagram

- Identify the Objects
- Identify relationships between Objects (objects graph)
- Identify messages and message calling sequence between objects

Collaboration Diagram

- Display the objects
- Display relationships between objects
- Display message calling sequence on individual objects
Collaboration Diagram

Class Diagram

- Identify classes
- Identify **static** and **dynamic** relationships between classes
- Identify relationships’ cardinality
- Identify class attributes
- Identify methods and their **parameters**
Sequence Diagram

- Dynamic modeling (~same as collaboration)
- Focuses on message sequencing
- Identify objects, their messages, and the message calling sequence
The player is only created at the beginning of the game!
State Diagram

- Identify the states of an object
- Identify state transitions

State Diagram for a « game » object

[Diagram showing state transitions and events such as 'Ready to play', 'Player ready', 'In progress', 'Quit', 'Start game', 'Get player name', 'Turn++', 'Roll dice', 'Cancel', 'Play', 'Turn>=10']
End of Analysis?

- Verify coverage for use-case and activity diagrams…
- Use case « view highscores »?
- Use case « play » partially handled
Diagram Coverage

Sequence Diagram
End of Analysis ?

• Coverage is « pretty » good
• Consistency across schemas is correct
• 14/20
  – Dynamic model lacks detail (dynamic model for cancel?)
  – Schemas are not described in enough detail…
  – Sequence diagrams for the game are not detailed enough : a few methods are missing…

Design

• Take implementation into account
  – Handle the GUI portion
  – Handle highscores persistence
• Define a logical architecture
• Define a physical architecture
• Add technical classes allowing the implementation of such architecture !
Layered Architecture...

- One possible architecture, others exist (seeo « A system of patterns » Bushmann »)
- Layers must be as independent as possible
- « Separate » layers by relying on interfaces + abstract classes (design patterns)
Logical « packaging »

- Map the architecture on « layered » packages
- Express dependencies

« core » Layer

- Classes representing the application logic
- In fact, these are the analysis classes which are being revisited for realization purpose
Core « Layer »: First Diagram

Graphical Interface Layering: MVC
Views?

DieView(die : Die)
update(o : Observable, arg : Object) : void

PlayerView(player : Player)
update(o : Observable, arg : Object) : void

Observable
changed : boolean = false
Observable()
addObserver()
deleteObserver()
notifyObservers()
notifyObservers()
deleteObservers()
setChanged()
clearChanged()
hasChanged()
countObservers()

Player
name : String
score : int = 0;
Player()
display()

Die
faceValue : int = 1
roll()
Die()
display()
setValue()

Panel
Panel()
Panel()
constructComponentName()
addNotify()

Observer
update(o : Observable, arg : Object) : void
<<Interface>>

Attention...

Observable
changed : boolean = false
Observable()
addObserver()
deleteObserver()
notifyObservers()
notifyObservers()
deleteObservers()
setChanged()
clearChanged()
hasChanged()
countObservers()

Panel
Panel()
Panel()
constructComponentName()
addNotify()

DieView(die : Die)
update(o : Observable, arg : Object) : void

PlayerView(player : Player)
update(o : Observable, arg : Object) : void
MVC in action: 1 Setup

1: display()
2: PlayerView(Player)
3: addObserver(Observer)
4: return component
5: display()
6: DieView(Die)
7: addObserver(Observer)
8: return component

MVC in action: 2 state change

1: getValue()
2: setValue(int)
3: notifyObservers()
4: update(Observable, Object)
MVC

• Java AWT Java: Delegation Model
  – Event propagation from user interface to core application classes

• MVC:
  – State change propagation to graphical objects

Put the views in the « forms »

• Implement graphical interfaces containing views as needed…
• The UI « layer »...
Object Diagram: rollform
UI/Core Separation...
Layered Architecture...

Technical classes
UI

UI

Interface and abstract classes handling decoupling

Core
Analysis classes

Util « subsystem »

• Need for random numbers
• Use java.util « random » functionality…
• The random number generator is shared by the die…
• Another singleton…
Subsystem « util »

Random

Singleton

Randomizer

Die

Player

Value

« Util » Dynamic Model
« Persist » Layer

- Technical classes used for persistence
- Ensures Core/Persist independence
  - Ability to switch « persistent engine »
- For example:
  - Persistence by « Serialization »
  - Persistence via a relational DBMS (JDBC)

Isolation : Pattern Fabrication
Using Serialization

- Persistence propagation...

```
\text{e1} \rightarrow \text{High Score}
\text{e2} \rightarrow \text{High Score}
\text{e3} \rightarrow \text{High Score}
\text{e4} \rightarrow \text{High Score}
```
A little bit of code…that’s all folks

```java
class HighScoreSr extends HighScore implements Serializable {
  ...
  public void save() throws Exception {
    FileOutputStream ostream = new FileOutputStream(filename);
    ObjectOutputStream p = new ObjectOutputStream(ostream);
    p.writeObject(this); // Write the tree to the stream.
    p.flush();
    ostream.close();   // close the file.
  }
  public void load() throws Exception {
    FileInputStream istream = new FileInputStream(filename);
    ObjectInputStream q = new ObjectInputStream(istream);
    HighScoreSr hsr = (HighScoreSr)q.readObject();
  }
}
```

JdbPersist… Dynamic Model

- A table must be created in a relational DBMS
- Upon creation of HighScoreJDBC: Connection to DBMS via JDBC
  - **save:**
    - Perform « inserts » for each « entry »
  - **load:**
    - Select * from ..., 
    - Follow the result, 
    - create « entry » objects
A little bit of code...

```java
public class HighScoreJDBC extends HighScore {
    public static final String url="jdbc:odbc:dice";
    Connection con=null;

    public HighScoreJDBC() {
        try {
            //loads the driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection(url,
                "molli","");
        } catch (Exception e) {
            e.printStackTrace();
            new Error("Cannot access Database at"+url);
        }
        hs=this; // register unique instance !
        this.load();
    }
}
```

Jdbc Load

```java
public void load() {
    try {
        Statement select=con.createStatement();
        ResultSet result=select.executeQuery
            ("SELECT Name,Score FROM HighScore");
        while (result.next()) {
            this.add(new Entry(result.getString(1),
                result.getInt(2)));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```
public void save() {
    try {
        for (Enumeration e = this.elements();
             e.hasMoreElements()) {
            Entry entry=(Entry)e.nextElement();
            Statement s=con.createStatement();
            s.executeUpdate("INSERT INTO HighScore (Name,Score)"
                         +"VALUES('"+entry.getName()+"','"+
                         entry.getScore()+"')");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
Component diagram

• « Realize » : implement interfaces
• « Depend » : Use interfaces
• Interfaces isolate components
Deployment Diagrams

- Display the physical architecture
- Associer execution units to associated handlers
- Identify connections between execution units

Deployment Diagrams

Game Computer
- Play the game
- File System
- Maybe a Remote file system

SGBD computer
- Save/load the highscore
- JBDC Connection
Design Complete?

- Functionality coverage: compare Use-case and activity diagrams...
- Consistency between diagrams ??
  - Some inconsistencies… UI vs Core
  - Core/Persist independence partially modeled...

Generate code: code mapping

- Map into any programming language!
- OO languages: java, C++, Smalltalk
- Or: VB, C, Fortran
- As well as: SQL, Cobol...
package Core;
import Util.Randomizer;
import UI.DieView;
import java.util.*;
import java.awt.Component;
public class Die extends Observable implements Displayable {
  private int faceValue = 1;
  public int roll() {
    setValue(Randomizer.getInstance().getValue());
    return getValue();
  }
  public java.awt.Component display() {
    Component c = new DieView(this);
    this.addObserver((Observer)c);
    return c;
  }
  public void setValue(int value) {
    faceValue = value;
    this.setChanged();
    this.notifyObservers();
  }
  public int getValue() { return faceValue; }
}

package Core;
import java.util.*;
import java.awt.Component;
import UI.HighScoreView;
public abstract class HighScore extends Observable implements java.io.Serializable, Displayable {
  protected static HighScore hs = null;
  public Vector entries = new Vector();
  public void add(Entry entry) {
    entries.addElement(entry);
    this.setChanged();
    this.notifyObservers();
  }
  public Enumeration elements() {
    return entries.elements();
  }
  public abstract void load();
  public abstract void save();
  public Component display() {
    Component c = new HighScoreView(this);
    this.addObserver((Observer)c);
    return c;
  }
  public static HighScore getInstance() {
    if (hs == null) {
      new Error("No Persist Kit declared");
    }
    return hs;
  }
}

package Core;
import java.util.*;
import java.awt.Component;
import UI.HighScoreView;
public abstract class HighScore extends Observable implements java.io.Serializable, Displayable {
  protected static HighScore hs = null;
  public Vector entries = new Vector();
  public void add(Entry entry) {
    entries.addElement(entry);
    this.setChanged();
    this.notifyObservers();
  }
  public Enumeration elements() {
    return entries.elements();
  }
  public abstract void load();
  public abstract void save();
  public Component display() {
    Component c = new HighScoreView(this);
    this.addObserver((Observer)c);
    return c;
  }
  public static HighScore getInstance() {
    if (hs == null) {
      new Error("No Persist Kit declared");
    }
    return hs;
  }
}
Programming...

- Use « forward engineering » functionality provided by tools
- Then « reverse engineering »
- To achieve « round trip engineering » ;-D
- Ensure Programming/Design/Analysis consistency...

Forward engineering

```
// Source file: c:/prout/Core/DiceGame.java
package Core;
public class DiceGame {
    private static int dg = null;
    private Die dies[];
    private Player thePlayer;
    DiceGame() {
    }
    private DiceGame() {
    }
    public void getInstance() {
    }
    public void start() {
    }
    public void quit() {
    }
} // Source file: c:/prout/Core/DiceGame.java
```
Forward Engineering

```java
package Core;
import UI MainForm;
import Persist.*;
import java.awt.*;

class Main {
    public static void main(String args[]) {
        SrKit srk=new SrKit();
        JdbcKit srk=new JdbcKit();
        DiceGame dg=DiceGame.getInstance();
        Frame f=MainForm.getInstance();
        f.setSize(300,300);
        f.show();
    }
}
```

Reverse Engineering...

```java
package Core;
import UI MainForm;
import Persist.*;
import java.awt.*;

class Main {
    public static void main(String args[]) {
        SrKit srk=new SrKit();
        JdbcKit srk=new JdbcKit();
        DiceGame dg=DiceGame.getInstance();
        Frame f=MainForm.getInstance();
        f.setSize(300,300);
        f.show();
    }
```
Reverse engineering

- Does not apply to the dynamic model!
- Handle forward+modification+reverse issues
- Nothing miraculous!

Design/Implementation Comparison

```java
Die
FaceValue : int = 1
roll()
Display()
setValue()
getValue()

DieGame
DiceGame()
faceDistance()
getName()
getName()

Observable
from util

Vector
from util

HighScore
HighScore()
add()
remove()
next()
prev()

Player
Player()

score : int = 0
turn : int = 0
WIN_NUM : int = 7
WIN_SCORE : int = 10

Entry
score : int
getName()
getScore()
toString()

Displayable

CORE
```
Util

Randomizer

getInstance()
getValue()

GetRandomizer

Persist

HighScoreJDBC

load()
save()

HighScoreSR

load()
save()

JdbcKit

makeKit()

PersistKit

getInstance()
makeKit()

DiceGame

getInstance()
start()
getDie()
getPlayer()

HighScore

add()
elements()
load()
save()
display()

Serializable

Entry

score: int

Entry()
getName()
getScore()
toString()
Problems Found

• Dynamic model to handle turns is not designed properly!
• Who really tests for the end of the game?
• Design flaw!
Problem!

- Not formal enough!
- This analysis diagram was not reviewed at design time!!!
- (-4)

Redo!
Finally!

Does it work? Testing

- **Unit testing**: test each class and each method at a time
  - Class diagram
- **Integration tests**:  
  - Component diagram
- **System test**:  
  - Use Case + Activity diagram
System Test

- Ok, functionality is there ...
- And conforms to the description of the use case!
- >8->
System Test

- Test all possible paths!
- Ex:
  - 1/Start
  - 2/ roll
  - 3/ cancel
  - 4/ highscore
  - 5/ exit

Problems Found

- Scenario 1:
  - start, roll*, highscore, quit : OK
- Scenario 2:
  - highscore, : ko ! Bug
  - Design bug:
    - DiceGame creates Highscore (start)
    - If Highscore before start : bug
Solution

```java
package Core;

import UI.MainForm;
import Persist.*;
import java.awt.**;

class Main {
    public static void main(String args[]) {
        // SrKit srk=new SrKit();
        JdbcKit srk=new JdbcKit();
        DiceGame dg=DiceGame.getInstance();
        Frame f=MainForm.getInstance();
        f.setSize(300,300);
        f.show();
    }
}
```

Integration Test

![Integration Test Diagram]

MVC Test
Test Scenario

- Highscore, start, roll*
- If the MVC works properly, entry of a new highscore leads to redisplaying the list which is already opened !!
- Ok, it works…
- It is good to design well …

Summary of this Application Design

- Requirements Analysis
  - Use-case + description
  - Activity diagram
  - UI prototyping
- Analysis
  - Dynamic Model: Collaboration, Sequence, state diagrams
  - Static Model: Class Diagram
**Design**

- Architecture design (layer)
  - Package diagram, component diagram, deployment diagram
- Technical classes used to ensure isolation:
  - MVC pattern,
  - Factory pattern
- Technical classes UI and persistence
  - *Forms, Highscore*

**Programming**

- Simple conversion of design to Java
- For each UML model, it is possible to build a translation towards any target language
- Use « round-trip engineering » tools
- Programming PB : Need to update the analysis/design artifacts !!!
Feedback Problems Found at Coding!

- « auto-critique » to find the reason behind the problem.
- Improve process for next time!
Here: analysis diagrams have not been redone!
- A software process must emphasize quality!

Testing

- Functionality control: Use-case diagram
- Conformance control: Activity diagram
- Integration tests: Component diagram
- Unit tests: not done
Paying Attention to Testing!

- Current vision is too simplistic!
- Must integrate testing as part of a quality management process (change management)
- Regression testing, automated testing (test suites, test generation, tools!!)

Conclusion for this Application

- Phase:
  - Requirements analysis, analysis, design, implementation, etc.
- For each phase:
  - Put together views for the same problem
  - Static, dynamic, functional, architectural views
Consistency/Coverage

- Use-cases/Activity Diagrams
  - An activity must always be assigned to a use-case
  - All use cases must be implemented in the activity diagrams
Use-case/Activity

Activity/Collaboration

- All possible paths in the activity diagrams may be represented using collaboration diagrams!
- Beware of over-analysis!
- Only represent the most relevant scenarios!
Collaboration/Class diagram

- All the objects in a collaboration diagram have a type: the class diagram Class.
- All the relationships in a collaboration diagram must exist or may be derived from the class diagram!
- Messages exchanged are methods in the class diagram!
Collaboration/Class Diagram

Class diagram / collab / sequence

- The complete dynamic model for relations must appear in at least one sequence or activity diagram
- Any attribute change must be represented in at least one activity or sequence diagram
- Object creation or destruction must appear in at least one dynamic diagram!
Class/Sequence

KO!

A « good » solution:
- Follow the activity diagram to generate scenarios
- Follow the possible paths in the activity diagram
- If the activity diagram is not covered:
  - Granularity is not fine enough (it is the case here)
  - Class diagram over-specified
Class/State diagram...

- For each class, ask yourself if its state evolves with time?
- If so, put together a state diagram…
- Every transition in the state diagram must be verified!
Class/Package/Component diagram (Design)

- Each class must be allocated to one package, which is itself part of the overall architecture
- Every class must also be part of a component that implements a set of functionalities in that architecture !!
- Otherwise the class is not part of the architecture!
### Class/Component

![Class/Component Diagram]

### Component/Deployment

- Each component must be allocated to one execution unit in the deployment diagram!
- In general, a component cannot be part of two execution units...
- Every execution unit must have at least one component...
Component Deployment!

General Conclusion on Dice

• How different is it from a « directly coded » application ??