Software Engineering
G22.2440-001

Session 7 – Sub-Topic 1
UML and the SDLC

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*
Introduction and Big Picture

- Requirements Analysis
  - UML: artifact production
  - Relevant
  - Coherent
  - Understandable
  - Easy to exchange
  - Complete SDLC coverage

- Analysis
- Design
- Programming
- Test
Why UML?

• Produce artifacts to communicate as a team
• Understand the « big picture »
• Keep learning about OO
Methodology War

• Booch, OMT, Coad/Yourdon, Fusion, SADT, OOSE, Schlaer/Mellor, HOOD…

• UML: a modeling meta-language to unify models used in various methodologies

• Grady Booch, Ivar Jacobson, James Rumbaugh.

• OMG standardization
Requirements Analysis

• Capture User Needs
• UML « Use-Case » Diagram

Student

Billing System

Register for Courses

Select Courses to Teach

Professor
Analysis

• Implementation independent
• Identify classes and inter-relationships
• Describe collaboration between objects that are instances of the various classes
• Class, Collaboration, Activity, and State Diagrams
Design

• Takes computer architecture into account
• Relies on class techniques to manage GUIs, distribution, persistence, concurrency
• Class, Sequence, Component, Deployment, and State diagrams
Programming

• Convert design classes into target languages: Java, SQL, C++, IDL
• Convert persistent classes to match persistence models (DBMS, ODS, persistent languages, etc.)
• etc ...
Test

• Unit testing: class diagrams
• Integration testing: component and collaboration diagrams
• System test: use-case diagram(s)
First 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
Activity Diagram

Player

Play

View High Score

menu

[start]
[highscore]
[exit]

view Highscore

Start turn=0

Roll Dice turn++

Turn<10

Update highscore

Turn<10

[true]

[false]
Activity Diagram

[Diagram showing an activity diagram with nodes and edges representing the game flow, including actions like 'view Highscore', 'Start turn=0', 'Roll Dice', 'Turn<10', 'Update highscore', and buttons like 'Play', 'View highscore', 'Exit']
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**

```
Momo : Player
game : Dice
Game
d1 : Die
d2 : Die
1: play()
2: r1=roll()
3: r2=roll()
```
Collaboration Diagram

- **menu**: 
  - [highscore]
  - [start]
  - [exit]

- **view Highscore**

- **Start**: 
  - turn=0

- **Roll Dice**: 
  - turn++

- **Turn<10**
  - [true]
  - [false]

- **Update highscore**

**Game**

- **d1 : Die**
- **d2 : Die**

**Momo : Player**

Actions:
1: play()
2: r1=roll()
3: r2=roll()
Class Diagram

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

Player
(From Use Case View)
- name : String
- score : int = 0;
- play()

Rolls

Die
- faceValue : int = 1
- roll()

1

2

Plays

Includes

DiceGame

Scoring

1

1

HighScore

1
Momo : Player

game : Dice

Game
d1 : Die
d2 : Die

2: r1=roll()
3: r2=roll()
1: play()
Sequence Diagram

• Dynamic modeling (~same as collaboration)
• Focuses on message sequencing
• Identify objects, their messages, and the message calling sequence
Sequence Diagram

: DiceGame

: Player

d1 : Die

1: play()

2: roll()

3: roll()

Activation Duration!
Sequence Diagram

1: play()
2: r1=roll()
3: r2=roll()

Reference!
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

- **Ready to play**
  - / Start game
  - Quit

- **Player ready**
  - entry: get player name

- **In progress**
  - entry: turn++
  - roll dices[ turn<10 ]
  - start

- **Cancel play**
  - cancel

- **Quit**
  - play
  - roll dices[ turn<10 ]

- **[ turn>=10 ]**
State Diagram

Ready to play
entry: get player name

In progress
entry: turn++

/ Start game
roll dices[ turn<10 ]
start

[ turn>=10 ]

Cancel play
cancel

Quit

Ready to play
Player ready
entry: get player name

Start
turn=0

Roll Dice
turn++

Turn<10

[true]

Update highscore

[false]

view Highscore

[highscore]

[start]

[exit]
Modify Schema...

```plaintext
menu

[highscore] → view Highscore
[start] → Start turn=0
[exit] → cancel
cancel

Roll Dice turn++

Turn<10

[true] → [true]
[false] → Update highscore
```

34
Modify Schemas?

- **menu**
  - [highscore]
  - [start]
  - [exit]

- **view Highscore**
- **Start turn=0**
- **Roll Dice turn++**

- **Turn<10**
  - [true]
  - [false]

- **Update highscore**
End of Analysis?

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

Player

Play

View High Score

Partially handled

Not handled
Sequence Diagram

: RealPlayer

: DiceGame
d1 : Die
d2 : Die
: HighScore

1: DiceGame()
2: Die()
3: Die()
4: Highscore()
**Sequence Diagram**

- : DiceGame
- : Player
- d1 : Die
- d2 : Die
- new : Entry
- : HighScore

1: Player(String)
2: play()
3: roll()
4: roll()
5: Entry(name:String,score:int)
6: add(Entry)
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!
Architecture Design

Presentation

Application

Persistence

File or DBMS

Play

View High Score

1 turn

score = 0

d1 : 4
d2 = 4

roll

Cancel
Layered Architecture...

- One possible architecture, others exist (seeo « A system of patterns » Bushcmann »)
- 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

Design

Analysis

- HighScore
  - $ hs: HighScore = null
  - Highscore()
  - add()
  - load()
  - save()

- Entry
  - name: String : type = initial
  - score: int : type = initial
  - Entry(name: String, score: int())

- <<Actor>>
  - Player
    - name: String
    - score: int = 0;
    - play()
    - Player()

- <<Actor>>
  - Player()
    - player

- Singleton...
  - DiceGame()
    - getInstance()
    - start()

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

- Rolls
  - Die
    - faceValue: int = 1
    - roll()
    - Die()

- Plays
  - DiceGame
    - DiceGame()
    - start()

- Includes
  - Scoring
    - HighScore
      - Highscore()
      - add()
      - Entry(name: String, score: int())()
Graphical Interface Layering: MVC
Views ?

```
<<Interface>>
Observer
(from util)

update(o : Observable, arg : Object) : void

1 turn
score = 0

d1 = 4
d2 = 4

roll

PlayerView

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

DieView

DieView(die : Die)
update(o : Observable, arg : Object) : void
```
Attention ...
MVC in action: 1 Setup

```java
: RollForm
    1: display()
    2: PlayerView(Player)
    3: addObserver(Observer)
    4: return component

: PlayerView
    5: display()

: Die
    6: DieView(Die)
    7: addObserver(Observer)
    8: return component

: DieView
```
MVC in action: 2 state change

: Die
1: getValue()
2: setValue(int)
3: notifyObservers()

: Randomizer

: DieView
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 » ...
UI « Layer »
UI Mapping Class, UI
Object Diagram: rollform

- : RollForm
- : Panel
- : Label
- : Label
- : Label
- : Label
- : Label
- theDieView
- theDieView
- thePlayerView
- momo : PlayerView
- roll : Button
- cancel : Button
- d1 : Die View
- d2 : Die View
UI/Core Separation...
Layered Architecture...

UI

Technical classes

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
(from util)
- $\text{serialVersionUID : long} = 3905348978240129619L$
- $\text{seed : long}$
- $\text{multiplier : long} = 0x5DEECE66DL$
- $\text{addend : long} = 0x8L$
- $\text{mask : long} = (1L << 48) - 1$
- $\text{BITS_PER_BYTE : int} = 8$
- $\text{BYTES_PER_INT : int} = 4$
- $\text{nextNextGaussian : double}$
- $\text{haveNextNextGaussian : boolean} = \text{false}$

Randomizer
- $\text{getInstance()}$
- $\text{getValue()}$
- $\text{Random()}$
- $\text{Random()}$
- $\text{setSeed()}$
- $\text{next()}$
- $\text{nextBytes()}$
- $\text{nextInt()}$
- $\text{nextInt()}$
- $\text{nextLong()}$
- $\text{nextBoolean()}$
- $\text{nextFloat()}$
- $\text{nextDouble()}$
- $\text{nextGaussian()}$
« 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

Abstract product →

Concrete product

Concrete factory

Abstract factory →

HighScore
(from Core)
• $ hs : \text{HighScore} = \text{null}
  • highscore()
  • add()
  • load()
  • save()

HighScoreJDBC
• highscore()
• load()
• save()

JdbcKit
• makeKit()

PersistKit
• makeKit()

HighScoreSr
• highscore()
• load()
• save()

SrKit
• makeKit()
« Persist » Dynamic Model

Attention!
DiceGame voit SrKit comme un PersistKit et HighScoreSr comme un HighScore

Seul le Realplayer sait qu'il utilise un SrKit ! DiceGame non !
Using Serialization

• Persistence propagation...
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
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();
    }
}
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...

- A component is a « non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture »
- A component conforms to and provides the physical realization of a set of interfaces.
Component diagram

- « Realize » : implement interfaces
- « Depend » : Use interfaces
- Interfaces isolate components
Component diagram...
Deployment Diagrams

- Display the physical architecture
- Associer execution units to associated handlers
- Identify connections between execution units
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);
        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((java.util.Observer)c);
        return c;
    }
    public static HighScore getInstance() {
        if (hs==null) {
            new Error("No Persist Kit declared");
        }
        return hs;
    }
    public Enumeration elements() { return entries.elements(); }
    public abstract void load();
    public abstract void save();
    public Component display() { Component c=new HighScoreView(this);
        this.addObserver((java.util.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

package Core;
public class DiceGame {
    private static int dg = null;
    private Die dies[];
    private Player thePlayer;
    DiceGame() {
        // Constructor code
    }
    public void getInstance() {
        // Method code
    }
    public void start() {
        // Method code
    }
    public void quit() {
        // Method code
    }
}

// Source file: c:/prout/Core/DiceGame.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

CORE

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

- HighScore
  - HighScore()
  - add()
  - elements()
  - load()
  - save()
  - display()
  - getInstance()

- Player
  - score : int = 0
  - turn : int = 0
  - WIN_NUMBER : int = 7
  - WIN_SCORE : int = 10
  - Player()
  - die1()
  - die2()
  - play()
  - display()
  - getName()
  - getScore()
  - getTurn()
  - setName()
  - setTurn()
  - setScore()

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

- Vector
  - from util
  - entries

- String
  - from lang
  - name

- Observable
  - from util

- DiceGame
  - DiceGame()
  - getInstance()
  - start()
  - getDie()
  - getPlayer()

- Displayable
  - display()
Problems Found

- Dynamic model to handle turns is not designed properly!
- Who really tests for the end of the game?
- Design flaw!
Here! Analysis Diagram!!

1: Player(String)
2: play()
3: roll()
4: roll()
5: Entry(name: String, score: int)
6: add(Entry)
Problem!

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

1: actionPerformed(ActionEvent)
2: rollAction()
3: getTurn()
4: [turn<10]play()
5: setValue(int)
6: setValue(int)
7: setTurn(int)
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

I forgot this one!
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

- DicePersist
- HighScore
- PersisKit
- DiceSystem
- Displayable
- Observable
- Observer
- Randomizer
- Random system

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!!
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

OK !
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!
Activity/collaboration

1 collaboration diagram partially handling Roll !!
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

Momo : Player

game : Dice

Game
d1 : Die
d2 : Die

2: r1=roll()
3: r2=roll()
1: play()

OK!
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

1: DiceGame
2: Die( )
3: Die( )
4: start( )
5: Player(String)

Player
(from Use Case View)
name : String
score : int = 0;
play()

Die
faceValue : int = 1
roll()

DiceGame

HighScore

KO!
Class/Sequence

• 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/State diagram!

Where are these methods??

Where is this attribute??

1 Object DiceGame!

Player
(from Use Case View)
- name : String
- score : int = 0;

Die
- faceValue : int = 1

Rolls
- play()

Plays

Includes

Player ready
entry: get player name

Ready to play
entry: get player name

In progress
entry: turn++

/ Start game
roll dices[ turn<10 ]

start

[ turn>=10 ]

Cancel

Quit

HighScore

Score

Play

roll dices[ turn<10 ]

[ turn>=10 ]

In progress
entry: turn++

start

cancel

play

cancel

Quit

Start game

Ready to play
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
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!

DicePersist
HighScore
PersistKit
DiceSystem
Displayable
Observable
Observer
Random
Randomizer
DiceVisualization
PersistKit
DiceSystem
Observable
Observer
Random
Randomizer
HighScore

Game Computer
SGBD computer
JBDC Connection
Play the game
File System
Save/load the highscore
Maybe a Remote a file system

OOOPPPPPSS!
General Conclusion on Dice

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

• Dice is documented: decisions occurring during analysis and design are visible and justified!
  – Dice can be taken over by somebody else! Maintenance costs are reduced! Dice is maintainable!
  – Dice may also be controlled (integrated in a software process) throughout the requirements analysis/test process …
Analysis and Design

• Dice is conforming: I said what I was going to do (before I did it) and I have done what I said I would do!
  – I could also have produced time to delivery and costs…
Analysis and Design

• Dice is evolving
  – It is possible to change the user interface
  – Additional persistence support may be provided (OO Mapping, Web server backup…)
  – And I can come up with a process to handle this evolution!
Analysis and Design

• Dice is robust:
  – Test suites campaign
  – Natural justification for MVC…

• Dice is portable:
  – Java + JDBC… (but not tested !)
Analysis and Design

• Dice is therefore maintainable, conforming, robust, and portable...

• The cost of Dice is therefore less than that of the same application developed using a «Quick and Dirty» approach
  – Maintenance costs are much higher than development costs (50% of the overall programming effort of a large company)
Analysis and Design

- Typical System Development Costs:

<table>
<thead>
<tr>
<th>System type</th>
<th>Requirements/Design</th>
<th>Coding</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command control</td>
<td>46</td>
<td>20</td>
<td>34</td>
</tr>
<tr>
<td>Spaceborn</td>
<td>34</td>
<td>20</td>
<td>46</td>
</tr>
<tr>
<td>Scientific</td>
<td>44</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Business</td>
<td>44</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>OS</td>
<td>33</td>
<td>17</td>
<td>50</td>
</tr>
</tbody>
</table>
Working with Tools...

• Drawing diagrams: must follow diagrams semantics
• Tool is used as a point of reference: all the elements created in the diagrams are stored in a single database
• Navigation: can « browser » through UML documents
Working with Tools...

• Multi-users support: Several users are able to work in parallel on the same model (configuration management)
• Code generation: Skeletons may be generated for several target languages
Working with Tools…

• Reverse engineering: Read the existing code and generate or update existing diagrams
• Integration with other tools: Editor, compiler, debugger, configuration management tool
Working with Tools...

• Model coverage at all abstraction levels: from packages to code
• Import/export: The tool must allow design import/export from/to other design tools (i.e., XMI)
A Tool...
Working with Tools

• A point of reference for:
  – Consistency control
  – Critical analysis (metrics)
  – Reporting

• Basically, DBMS scripts...
Function GetSubclasses (theSuperClass As Class, AllClasses As ClassCollection) As ClassCollection
    Dim theSuperclasses As ClassCollection
    Dim theSubclass As Class
    Dim theSubClasses As New ClassCollection

    'Print ,,"Checking subclasses for "; theSuperClass.Name
    For i = 1 To AllClasses.Count
        Set theSubclass = AllClasses.GetAt (i)
        'Print ,,"theSubclass ";theSubclass.Name
        Set theSuperclasses = theSubclass.GetSuperclasses ()
        'For j = 1 to theSuperclasses.Count
        '    Print ,,"has superclass ";theSuperclasses.GetAt (j).Name
        'Next j
        If theSuperclasses.Exists (theSuperClass) Then
            'Print ,,"Found One!"
            theSubClasses.Add theSubclass
        End If
    Next i
    'Print ,,"Found ";theSubClasses.Count; " classes"
Integration...

- Project Management Tool
- Configuration Management
- Documentation Tool
- Process Support
- Requirements and Specification tools
- Profiler and metrics
- Compilateur/Debugger
- Gui Builder
- Test Administration Tool
- Testing tools
- Model Support