**GOF: Observer**

Motivation: A common side effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects

Problem: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Controller receives messages from view layer, Subject is what observers observe, The subject is in a one-to-many relationship with the observer, All observers are notified when there is any type of state change in the subject

**SAD**: Architecture factors, decisions, etc., N+1 architecture views, Use case view: Use case realization visualized in sequence diagrams, Deployment view, Deployment diagram, Data view, Activity diagram, Logic view, Diagrams: Class, Communication, Sequence, Package

**GRASP: Controller**

Problem: Who should be responsible for UI events? Solution: Assign responsibility for receiving or handling a system event to an object (i.e. a session controller) representing a use case scenario within which the system event occurs

**GOF: Adapter**

Problem: How to resolve incompatible interfaces or provide a stable interface to similar components with different interfaces? Solution: Convert original interface into another one through an intermediate adapter. Benefits: Reduces coupling to implementation specific details. Clients deal with a uniform interface. Polymorphism and Indirection reveals essential behavior provided

**GOF: Singleton**

Problem: How does one design a class that has at most one globally accessible instance? Implement the one static instance of that class in the class itself.

**GRASP Information Expert**

Problem: How to assign responsibilities to classes/objects? Solution: Assign responsibility to the class/object that has the information needed to fulfill it

**GRASP Creator**

Problem: Who creates an instance of A? Solution: Assign class B the responsibility   
if one of the followings is true (the more the better): B contains or aggregates A (in a collection), B records A, B closely uses A, B has the initializing data for A

**GRASP Low Coupling Pattern**

Problem: How to reduce the impact of change and encourage reuse? Solution: Assign a responsibility so that coupling (i.e., dependencies between classes) remains low.

High coupling means more dependencies; more things will break with a change



**GRASP High Cohesion**

Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling? Solution: Assign a responsibility so that cohesion remains high. A class with low cohesion does many related things or does too much work

**GRASP Pure Fabrication**

Problem: you must assign a responsibility to a class, but assigning it to a class that represents a domain class would ruin its low coupling and/or high cohesion. Solution: Assign a highly cohesive set of responsibilities to “made up,” fabricated class—it does not need to represent a domain concept

For NextGen POS, there is a requirement to save Sale instances in a database.

What does Information Expert tell us to do? Assign this responsibility to the Sale class, since Sale has the data that needs to be saved. Invent a new class that is solely responsible for saving objects (Persistent Storage Class)

Benefits: High Cohesion: fabricated class focuses on a very specific responsibility Reuse: fine-grained pure fabrication classes with specific responsibilities are relatively easy to understand and reuse in other applications. Pure fabrication principle leads to many other reusable design patterns, including most of the GoF patterns

**GRASP Polymorphism**

Problem: How to handle alternatives/behaviors based on type? How to create pluggable software components? Solution: When related alternatives/behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies. EX for tax calculator figure out and configure for common interface between POS and tax calculator.

**GRASP Indirection**

Problem: Where to assign a responsibility, to avoid direct coupling between two (or more) components? Solution: Assign the responsibility to an intermediate component to mediate between other components so that they are not directly coupled. (Persistent Storage). Benefits: High Cohesion: fabricated class focuses on a very specific responsibility, Reuse: fine-grained pure fabrication classes with specific responsibilities are relatively easy to understand and reuse in other applications

**Errors:** An error is a human activity resulting in software containing a fault, a fault is the manifestation of an error, a fault may result in a failure, a failure is the runtime, observable manifestation of a fault

@Test

public void testComputeFactorial() { System.out.println(“Testing computeFactorial"); assertEquals("24", Fac.computeFactorial(4)); assertEquals("120", Fac.computeFactorial(5));}

ASSERT: True, False, NotNull, Null, equals(uses equals), same(uses ==), notSame @Test(expected=IllegalArgumentException.class)

**public class PersistentStorage {**

private static PersistentStorage ps = null;

private Connection con = null;

private PersistentStorage() {

String url = "jdbc:ucanaccess://Parameters.accdb.mdb";

try{ con = DriverManager.getConnection(url);} catch(SQLException ex){while (ex!=null){ System.out.println ("SQL Exception: " + ex.getMessage ()); ex = ex.getNextException();}

**public static PersistentStorage getInstance() {**

if(ps==null) ps = new PersistentStorage(); return ps;}

**public void save(String clickTime, int counterValue) {**String insert = "insert into counter values (" + clickTime + ", " + counterValue + ")";stmt.executeUpdate(insert); stmt.close(); }

**public int getLastValue**

int lastValue = 0;

String query = "select counterValue from counter";

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) lastValue++;

stmt.close(); return lastValue;

**public class Model extends java.util.Observable {**

PersistentStorage.getInstance().save(dateFormat.format(date), counter);

setChanged();

notifyObservers(counter);

**public class SimpleGUI implements java.util.Observer {**

JFrame f; JPanel p; FlowLayout fl; JButton b1,b2; JTextField tf; MyAL al;

SimpleGUI () {

f = new JFrame("My MVC Test");

p = new JPanel();

fl= new FlowLayout(FlowLayout.CENTER);

b1 = new JButton("Click me");

b2 = new JButton("Exit");

tf = new JTextField("counter value is ... ...");

p.setLayout(fl); p.add(b1); p.add(b2); p.add(tf);

f.setSize(400,400);

f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE;

f.setContentPane(p);

f.setVisible(true);}

**public void update(Observable subject, Object subjectChange) {**

tf.setText("counter value is " + ((Integer) subjectChange).toString());}

public void addController(ActionListener controller) {

b1.addActionListener(controller);

b2.addActionListener(controller);}}

**public class MyAL implements ActionListener {**

Model model;

public void addModel(Model m) {

this.model = m;}

public void actionPerformed(ActionEvent ae){

if (ae.getActionCommand().equals("Exit")){

PersistentStorage.getInstance().close();

System.exit(1);}

else if (ae.getActionCommand().equals("Click me")){

model.incrementValue();}} }

**updateObservers**

for(int i=0; i<observers.size();i++)

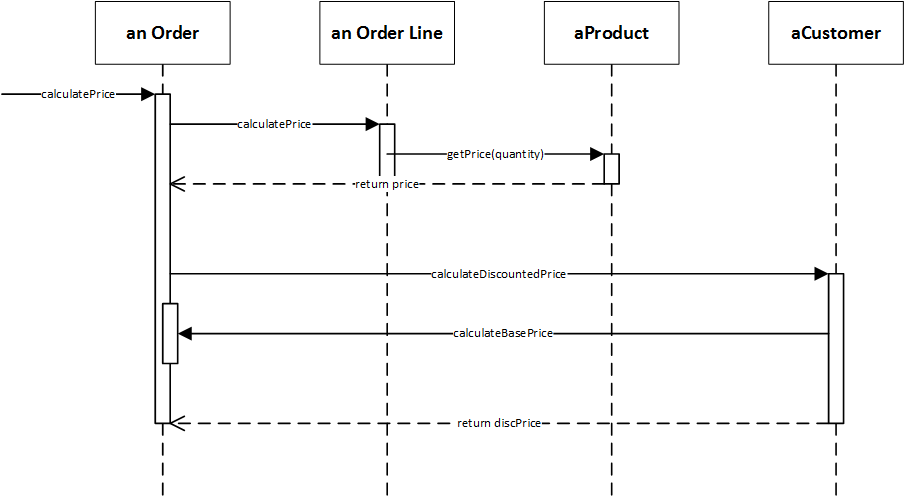
observers.get(i).update(this);}

**OOA**: Finding and describing the concepts in the problem domain| Use case, Domain model

**OOD:** Defining software objects and how they collaborate to fulfill the requirements**|** Sequence diagram, Class diagram

Open diamond: class belongs to a collection, can belong to more than one collection

Arrow: “is a”, inheritance, more specific version



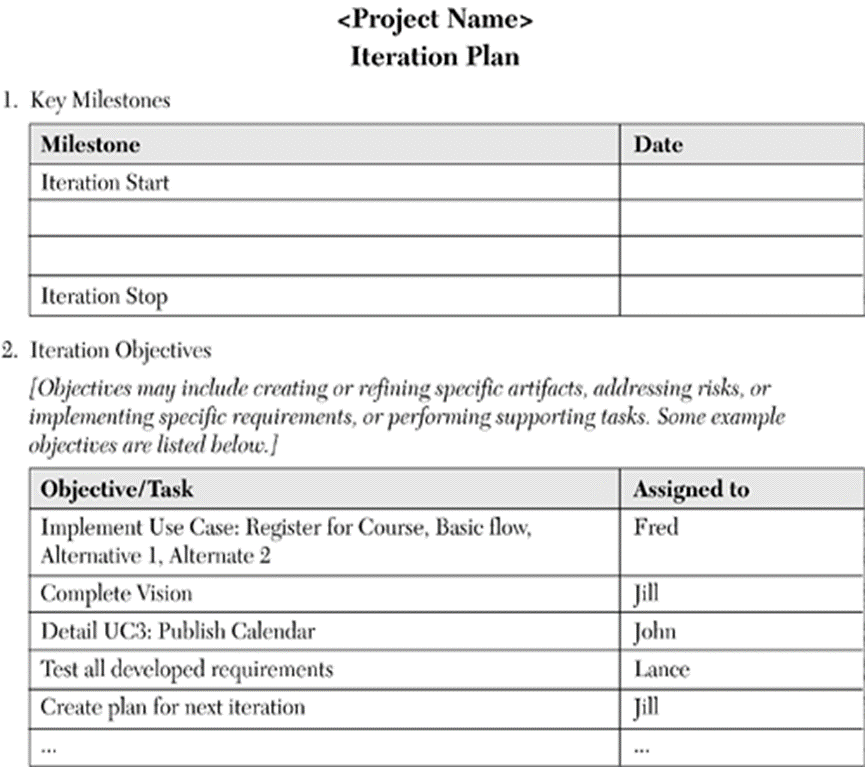
Under the “Information Expert” design pattern, the class that has the information needed is the class that receives the responsibility. For saving instances of Rental to the database, this would mean that the Rental class, which contains the data for each instance would be responsible for saving to the database. Because the use of this design pattern would mean that every class is responsible for saving itself to the database, the system would have low cohesion. This low cohesion is because the Rental class no longer just contains information about rentals, and also has to contain code that connects to the database and manages that connection instead of being coupled to a class elsewhere in the code that manages these things.

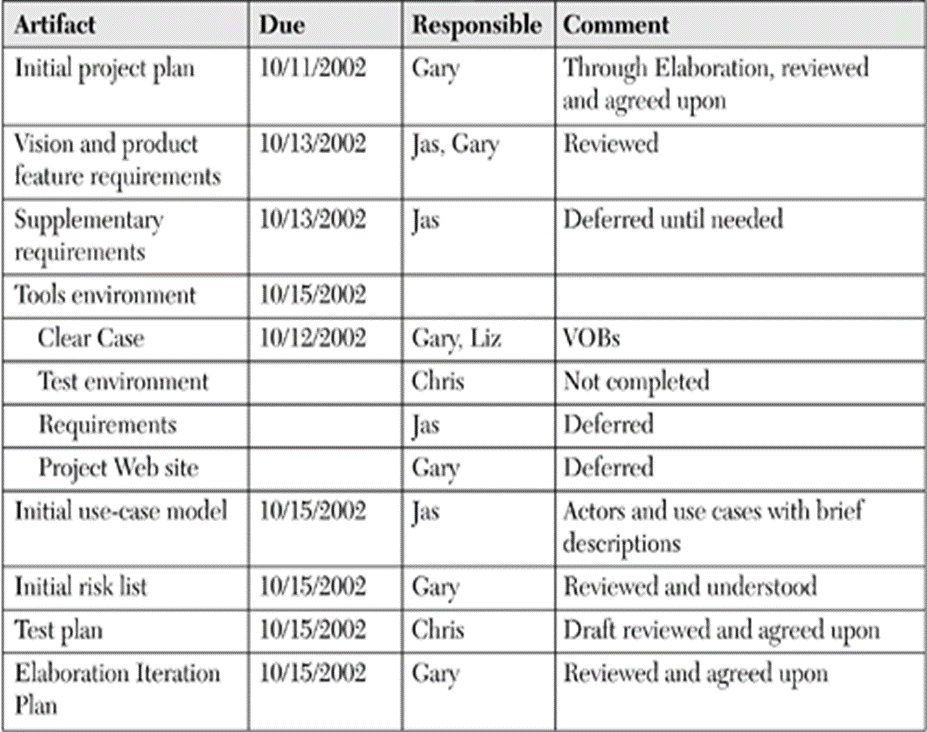
**FURPS+**

**Functional**: features, capabilities, security **Usability**: human factor, help, documentation **Reliability**: frequency of failure, recoverability **Performance**: response time, accuracy, resource usage **Supportability**: maintainability, internationalization **+**: implementation, interface, operation, packaging, legal

**Sequence:** bars+vertical Boxes, contain loops+cond. **Class Diagram:** Actual methods and variables and relationships. **Domain:** visual repr. of conceptual classes + info

**Communication diagram**: Illustrates object interactions in a graph format where objects can be placed anywhere on the diagram and the messages are numbered by the communication order





Repository: Central location storing a copy of all files.

**check in**: adding a new file to the repository

**check out**: downloading a file from the repo to edit it

you don't edit files directly in the repo; you edit a local working copy

once finished, the user checks in a new version of the file

**commit:** checking in a new version of a file(s) that were checked out

**revert:** undoing any changes to a file(s) that were checked out

**update:** downloading the latest versions of all files that have been recently committed by other users