**Procedural vs OO Programming:** OO typically shows relationships and interactions between classes or objects. Objects may contain data in the form of fields (attributes) and code in the form of procedures (methods). An object’s procedures can access and often modify the data fields of the object they’re associated. (Java) Not ideal for languages that aren’t object based. Procedural uses procedures (routines, subroutines, functions) that contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program’s execution. (C) Better access control leads to better maintainability. Proc. = incCounter, resetCounter, incByTwo, all point to counter, OOP = MyClass counter var, setCount, getCount.

**A class is a blueprint for object(s):** Sometimes referred to as name or type, Attributes (a.k.a. data, properties), Methods(a.k.a. operations, behaviors, functions)

**An object is an instance of a class:** Instantiation: creation of an object

**Inheritance:** Sometimes referred to as extension or specialization, Superclass (a.k.a. parent class or base class), Subclass (a.k.a. child class or derived class)

**OOP Principles – Encapsulation** (information hiding, access control), **Abstraction** (focusing on [problem-specific] essentials, define objects that represent abstract "actors" that can perform work, report on and change their state, and "communicate" with other objects in the system), **Inheritance** (getting parent’s attributes and methods as well as adding new attributes and methods), **Polymorphism** (overriding methods)

**Access Modifiers: Class-level visibility** – Visible to all classes everywhere = public, visible only in its package = no mod (package-private) **Member-level visibility** – “Class” from within the class itself, “Package” from within the containing package, “Subclass” from subclasses outside the containing package, “World” everywhere.

**Class** = public, protected, no mod, private. **Package** = public, protected, no mod. **Subclass** = public, protected. **World** = public

**OOP =/= OOA/D** – you can design bad code just as efficiently with OO methodology as with any other. Putting everything in one master class is not a good idea.

**From OOP Fundamentals to Design Patterns: Abstract class** – never instantiated, serves as base class to be extended into different full implementations (inher, [adding new elements] encap, polymorph)

**Interface** – C++ it declares what should be implemented in the class (atts and method sigs), Java it allows us to put together a set of relevant methods, public methods available to the “World”

**Aggregation and composition** **– class inheritance** = is-a = subclass extending parent class. **Aggregation** = has-a = class references objects of other classes. **Composition** = special case of agg = emphasizes a containing object’s ownership of its component (semester ends, all courses must have ended)

**How should I start a software project? Req eng (FURPS+, Artifacts)**

**How can I start from requirements to codes? OOAnalysis -> Design**

**How should I manage the changes of code? Version Control**

**How can I produce high quality code? Design Patterns**

**Req Eng – 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

**Artifacts** - Use case model, supplementary specification, glossary, vision & business case, business rules

**OOA/OOD – OOA** = Finding and describing the concepts in the problem domain (use case, domain model, operation contract, SSD), **OOD** = Defining software objects and how they collaborate to fulfill the requirements (sequence diagram, communication diagram, design class diagram). Domain modeling is a conceptualization process (ID the important domain concepts, properties and relationships), Interaction diagrams (models the sequential logic of the interactions between (several) objects ordering of messages w/ respect to time), Class diagram (augment the domain model to show the system’s classes, atts, ops/methods and relationships among objects)



Project <>1-----------1..\*Consultant



**basic notation of Seq diagram**

Two options: Use the message syntax returnVar = messsage(parameter)

Use a reply message line at the end of the activation bar \*\*reply or returns\*\*

**Creation of instances Conditional messages in sequence diagrams**



**Mutually exclusive cond messages iteration over a collection**

SQD-conditional mutex

**Git/Hub** - 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

**GRASP**:

**Information Expert** – where to delegate responsibilities

**Creator** – which class is responsible for creating objects

**Low Coupling** – lower depend between classes, less impact on each class when one is changed, higher reuse potential

**High Cohesion** – responsibilities of a given element are strongly related and highly focused (break into classes)

**Controller** – assigns responsibility of dealing with system events to a non-UI class that reps a use case scenario

**Polymorphism** – responsibility of defining the variation of behaviors based on type is assigned to the type for which the variation happens

**Indirection** – supports low coupling by assigning responsibility of mediation between them to an intermediate object. (Controller between model and view)

**Pure** **Fabrication** – class that does not represent a concept in the problem domain, specially made up to achieve low coup, high cohesion and reuse potential



shows the events that external actors generate, their order, and inter-system events

Systems as black boxes

Found messages

**GoF:**

**Observer** – Motivation: 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. Subject is what Observers observe (subject is in a one-to-many relationship with observer). All obvs are notified @ any state change in subject

**Adapter** – Problem: How to resolve incompatible interfaces or provide a stable interface to similar components with diff interfaces?

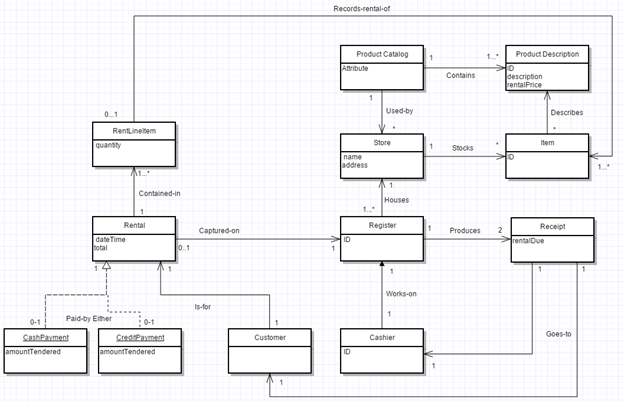
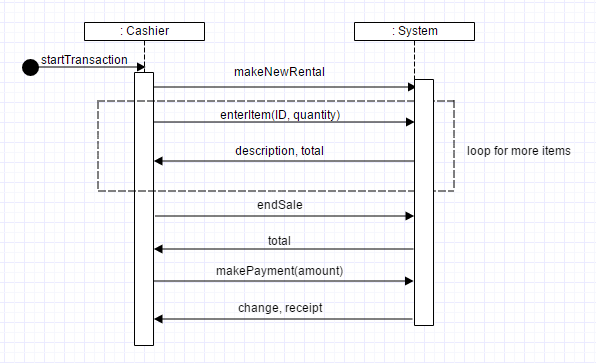
Solution: Convert original interface into another one through an intermediate adapter. Benefits: Reduces coupling to implementation specific details, clients deal with a uniform interface, polymorph and indirection reveals essential behavior provided

**Singleton** – Problem: How does one design a class that has at most one globally accessible instance? Solution: restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects.

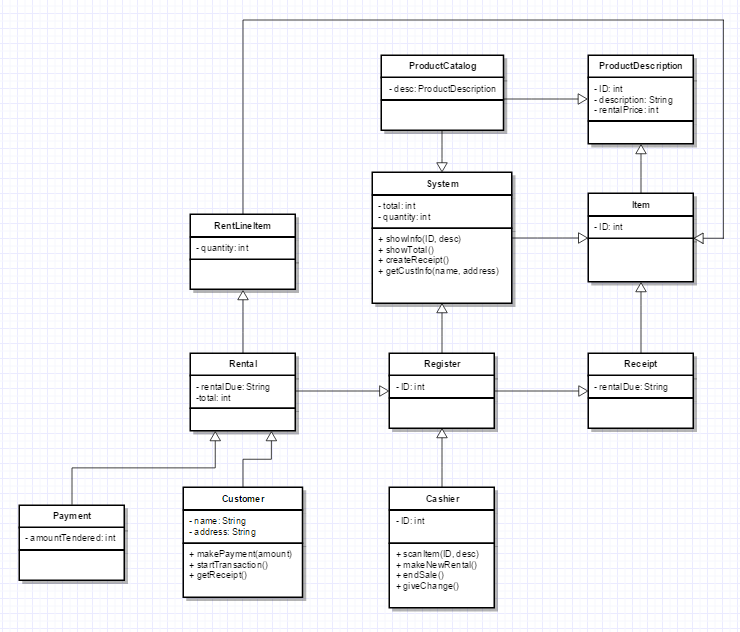
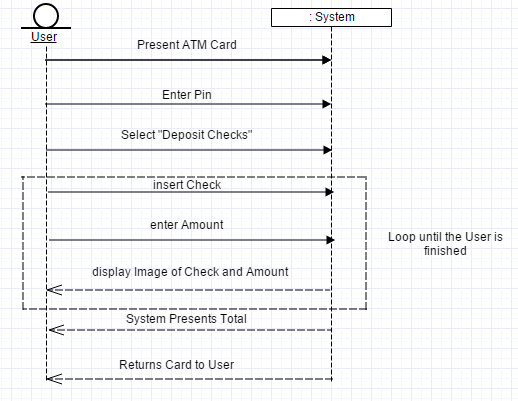
**HW2.2 - 2 leads you through the OOA/OOD (object-oriented analysis/object-oriented design) process.**

**HW2.2.1. (5 points) Write a use case for the “Process Rental” in the casual format where the use case name, primary actor, main success scenario, and alternative scenarios are described.**

**2.2.2 – Domain Model: 2.2.3 – Sequence Diagram:**

 ****

**2.2.4 - Class Diagram: 3.1 System Sequence Diagram**

**3.1 = sketch SSD for: Use Case: Deposit checks Main Scenario: 1. User presents ATM card to System. 2. User enters PIN. 3. User selects “deposit checks” function from the ATM machine. 4. User inserts a check into the ATM machine. 5. User enters amount of the check. 6. System displays a digital image of the check and also the amount. User repeats steps 4-6 until indicates done 7. System presents total 8. System returns card to User**

**4.1 MVC architecture for SimpleGUI**

***SimpleGUI.java***

class SimpleGUI implements java.util.Observer {

JFrame f; JPanel p; FlowLayout fl; JButton b1, b2; JTextField t1;

SimpleGUI() {

f = new JFrame("My GUI Test"); p = new JPanel(); fl = new FlowLayout(FlowLayout.CENTER);

b1 = new JButton("Click me"); b2 = new JButton("Exit"); t1 = new JTextField("counter value is 0");

p.setLayout(fl); p.add(b1); p.add(b2); p.add(t1); f.setSize(400, 400);

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

f.setContentPane(p);

f.setVisible(true); }

public void update(Observable subject, Object subjectChange) {

t1.setText("counter value is " + ((Integer)subjectChange).toString());}

public void addController(ActionListener controller) {

b1.addActionListener(controller);

b2.addActionListener(controller);}}

***MyGUITest.java***

public class MyGUITest {

public static void main(String[] args) {

Model model = new Model(); MyAL controller = new MyAL(); controller.addModel(model);

SimpleGUI s = new SimpleGUI(); s.addController(controller); model.addObserver(s);}}

***MyAL.java***

class MyAL implements ActionListener{

Model model;

public void addModel(Model m) {

this.model = m;}

public void actionPerformed(ActionEvent ae) {

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

System.exit(1); }

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

model.incrementValue();}}}

***Model.java***

public class Model extends java.util.Observable{

private int counter;

public void setValue(int value) {

this.counter = value; setChanged(); notifyObservers(counter); }

public void incrementValue() {

++counter; setChanged(); notifyObservers(counter);}}

**HW5.1. (5 points) Apply “Low Coupling” design pattern to the following design to generate a better alternative.**

The point of a Low Coupling design is to have lower dependency between classes. In the original, there was a big focus on the “anOrder” class and less methods/actions performed by the other classes. In my model, I think the work load is much more spread out amongst the rest of the classes, thus having a change in one class impacting the others less. There is also a higher reuse potential when using a Lower Coupling approach.

