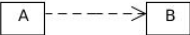







Amalgamation of knowledge

Question 1

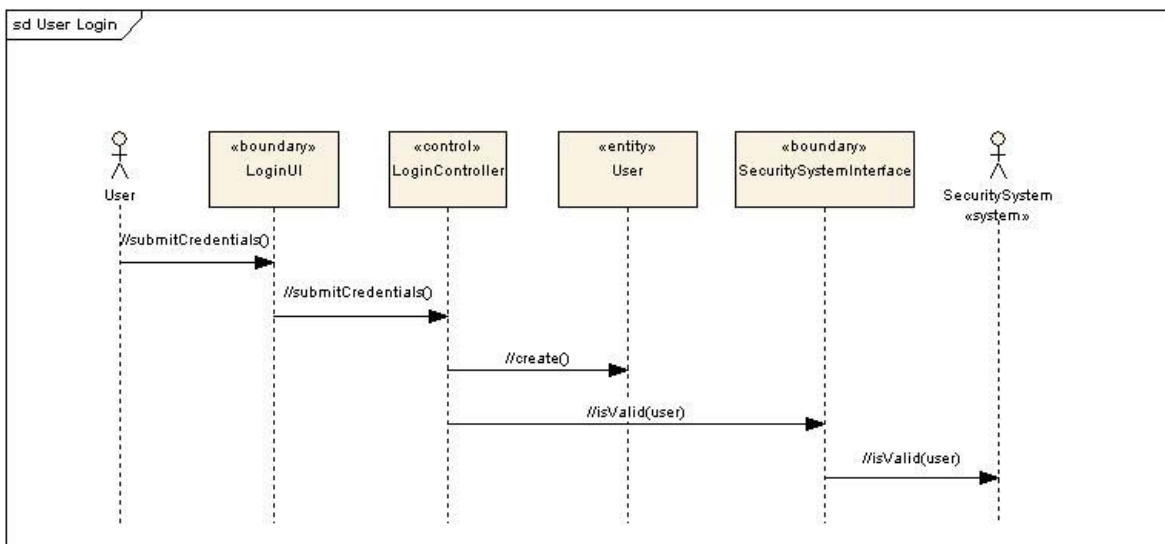
CRC, UML

UML

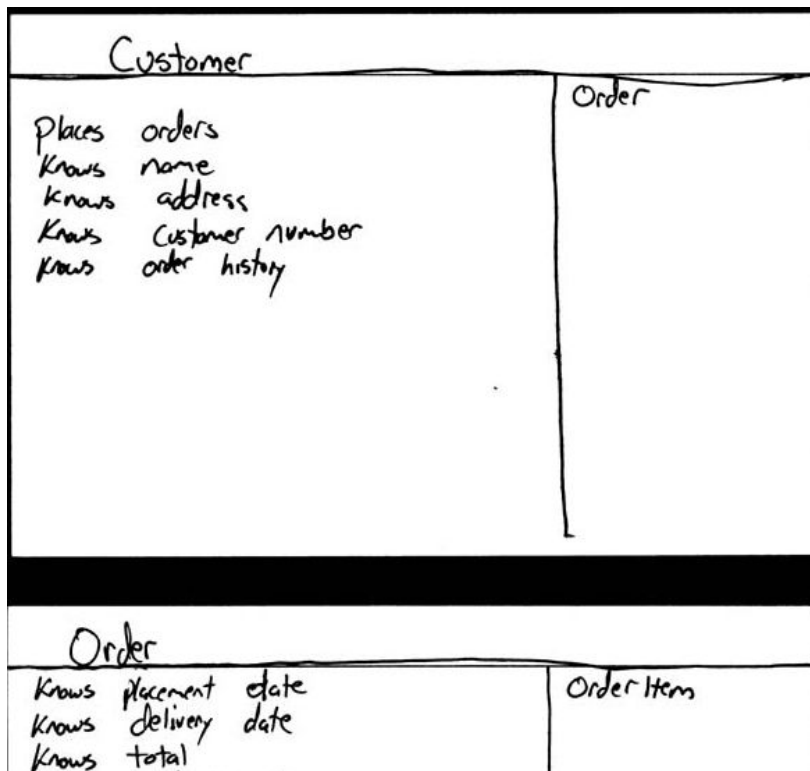
Relationship	Depiction	Interpretation
Dependency		A depends on B This is a very loose relationship and so I rarely use it, but it's good to recognize and be able to read it.
Association		An A sends messages to a B Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A.
Aggregation		An A is made up of B This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.
Composition		An A is made up of B with lifetime dependency That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.

Relationship	Depiction	Interpretation
Generalization		A generalizes B Equivalently, B is a subclass of A. In Java, this is extends.
Realization		B realizes (the interface defined in) A As the parenthetical name implies, this is used to show that a class realizes an interface. In Java, this is implements, and so it would be common for A to have the «interface» stereotype.

Sequence Diagram



CRC (Class Responsibility Card/Collaborator Card)



in UML,

|class containing variable | <>-----|Class of that variable |

closed arrow is inheritance.

|subclass|-----|> |super class|

or

|class|- - - - - |> |interface|

Question 2

Programming by Contract & Inheritance

- Liskov principle: The sub-classes have to be able to be swapped out for a super class and everything should behave as normal

It states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.).

- Demeter principle: No transitivity. If Google has a deal with Apple and knows it's net earnings, and Apple has another deal with Microsoft and knows microsoft's net earnings, Google should not be able to find out anything about Microsoft through it's deal with Apple.

In this case, an object A can request a service (call a method) of an object instance B, but object A should not "reach through" object B to access yet another object, C, to request its services.

- Equals function: public boolean equals(Object obj)
 - if (this == obj) return true;

- if (obj == null) return false;
- if (getClass() != obj.getClass()) return false;
- ClassNameOfThing other = (ClassNameOfThing) obj;
- Then check yo member variables

Question 3

Generic Types and Polymorphism

- Similar to the lab with Set<E> or Graph<E>.
- Covariance: Birds lay eggs. More specifically, a duck lays a duck egg.
We can be more specific on the left hand side and say a Duck lays an egg, but we can't be more specific on the right hand side and say a bird lays a duck egg. LHS can get more specific and still equal RHS. (Require more)
- Contravariance: Uni students can write in C, Java, HTML, Python, etc. Web designers can write in HTML.
We can be less specific on the RHS and say Uni students can write in HTML, but Web designers can't write in C, Java, HTML, Python, etc. RHS can get less specific and still equal LHS. (Require less)

Covariance:

```
class Super { Object getSomething(){} }
class Sub extends Super { String getSomething() {} }
```

Sub#getSomething is covariant because it returns a subclass of the return type of Super#getSomething (but fullfills the contract of Super.getSomething())

Contravariance

```
class Super{ void doSomething(String parameter) }
class Sub extends Super{ void doSomething(Object parameter) }
```

Sub#doSomething is contravariant because it takes a parameter of a superclass of the parameter of Super#doSomething (but, again, fullfills the contract of Super#doSomething)

Question 4

Design Patterns

- **Composite pattern:** Instead of saying if obj class is rectangle, do this, if circle do this, blah blah blah, make the rectangle and circle classes implementations of a Shape class, and then just need to call Shape.doThing(), where here Shape is an interface.
 - Need to use INTERFACE
- **Decorator pattern:** Use an abstract class, but instead of it being a problem with working down, this time it is so you can do different things with them. Like the strategy pattern. So you can make a new shape that is coloured in red, or a shape that is coloured in

green, or the original plain shape. The upside is that you still get all the functionality of the plain shape, but you have the added functionality of the new decorated classes.

- How to:

Abstract class (decorator) implements base interface (shape) and is called by concrete types below (rectangle, circle) and concrete types above (red shape, blue shape).

INTERFACE -> ABSTRACT CLASS

- **Iterator pattern:** implement an Iterator class inside the iterable object class with functions first, next, is_done and current_item. The object also has a create_iterator function. This is so that you can call `Object.Iterator i = object.create_iterator()` (which returns the iterable version of the object) and then `i.first()` will start/initialise, `i.next()` is like `i++`, `i.current_item()` returns the value, Etc, etc.

NOTE: member variable in iterator class is

```
private java.util.Iterator iteratorMemberVariable;
```

and this has function

```
iteratorMemberVariable.next()
```

which should be used like

```
try {
    current = iteratorMemberVariable.next();
}

catch (NoSuchElementException ex){
    current = -999;
}
```

- **Observer pattern:** Have an interface (AlarmListener) and then in your main system, have an array of listeners who need to be updated if something happens. There will need to be a register function to add them to the alert system. A soundAlarm function runs through each listener in the array and calls `((AlarmListener)e.next()).alarm()`. Each listener implements the AlarmListener interface.
 - Need to use INTERFACE
- **Strategy pattern:**

Surprise Question

Locks and synchronisation

- **Reentrant locks vs Synchronized functions:** Reentrant locks have extended capabilities
 - The ability to have more than one condition variable per monitor. Monitors that use the synchronized keyword can only have one. This means reentrant locks support more than one wait()/notify() queue.
 - The ability to make the lock "fair". "[fair] locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order." Synchronized blocks are unfair.
 - The ability to check if the lock is being held.
 - The ability to get the list of threads waiting on the lock.

- Bad things about reentrant locks:
 - Need to add import statement.
 - Need to wrap lock acquisitions in a try/finally block. This makes it more ugly than the synchronized keyword.
 - The synchronized keyword can be put in method definitions which avoids the need for a block which reduces nesting.
- Lock code:

```
final Lock lock = new ReentrantLock(); //Create the lock for the BoundedQueue
final Condition full = lock.newCondition(); //Used for producer waiting when queue is full
```

```
// How to use condition
function (Lock lock, Condition full)
if (full){ //If the buffer is full we need to sleep
    full.await(); //Release the lock and wait until it is returned to us
}
```

- If a **FUNCTION** is declared synchronised, it is available to only one object at a time
 - The data can still be accessed though, just not the function
- While a thread is waiting on a lock, it does not do anything else (unless reentrance lock). The other threads still continue to work though
- Deadlock occurs if a thread is waiting on a lock, and the other thread is also waiting on a lock. Ex, if main calls thread1Function, there will be a deadlock.

```
synchronized thread1Function(){
    thread2Function();
}

synchronized thread2Function(){
    thread1Function();
}
```