FIT2099 Notes

1. Table of Contents

2. Good Design in Software

Some Combination of - Functionaly correct - Performs well enough - Usable - Reliable - Maintainable

these are the properties of the system, not any design artifacts

there is no algorithm for: - creating good designs - identifying good designs

Over the years, key princpiles have been identified

2.1. Dependencies

2.1.0.1. Dependency Control

- Biggest issue in design
- Controlling the extent of dependencies
- Controlling the nature of dependencies

Will have some depenencies, having fewer dependencies makes it easier to debug, modify, change the component

- form of depencies matter

2.1.0.2. Why Dependencies

- dependencies are unavoidable
- if code unit A depends on code unit B
  - Bugs in B may manifest in A
  - Changes to B may require changes to A
- **Dependencies have to:**
  - *only present when necessary*
  - *explicit*
  - *easy to understand*

2.2. Connascence

- Based on earlier ideas of *cohesion* and *coupling*

```
two components are connascent if a change in one would require the other to be
modified in order to maintain the overall correctness of the system.
            - Connascence (Wikipedia Definition)
```

2.2.0.3. Type of Connascence

Type	Description	Example
Static	obvious from code structure, auto identified by IDE	
Dynamic	Dynamically Generated	

2.2.0.3.0.1. Connascence of Name

Type: **Static** has no argument(s)

```
class Watch {
    public void testWatch(){
        ...
    }
}
```

called using

```
class Hello {
    Watch demo = new Watch();
    watch.testWatch();
}
```

2.2.0.3.0.2. Connascence of Type

Type: **Static**

has argument

```
class Watch {
    public void testWatch(int maxTick){
        ...
    }
}
```

called using:

```
class Hello {
    Watch demo = new Watch();
    watch.testWatch(1000);
}
```

2.2.0.3.0.3. Connascence of Position

Type: **Static** - where order of which things go

```
public LinkedCounter(LinkedCounter l, Counter neighbour){
    super(l);
    this.neightbour = neighbour;
}
```

watch 3:

```
public Watch3(Watch3 w){
    this.hours = new MaxCounter(w. hours);
    this.minutes = new LinkedCounter (w.minutes, this.hour);
    this.seconds = new LinkedCounter (w.seconds, this.minutes);
}
```

It has to remember the position for example `this.minutes = new LinkedCounter (w.minutes, this.hour);` has to remember the position of this.hour

2.2.0.3.0.4. Connascence of Meaning/Convention (CoM/CoC)

Type: **Static**

```
public void increment(){
    super.increment();
    if(this.getValue() == 0){
        neighbour.increment();
    }
}
```

```
public void reset(){
    value = 0;
}
```

- Documentation is **important**

2.2.0.3.0.5. Connascence of Algorithm

Type: **Static**

```
1. (message, key) ->  Encrypter
2. Encrypted Messages trasmits
3. Encrypted Message Must implement reverse of encrypter
```

must document very precisely

IPoAC - https://en.wikipedia.org/wiki/IP_over_Avian_Carriers

2.2.0.3.0.6. Connascence of Execution (CoE)

Type: **Dynamic**

Example:

```
public Watch3(){
    hours = maxCounter(24):
    minutes = new LinkedCounter(60, hours);
    seconds = new LinkedCounter(60, minutes);
}
```

Must be ran in the right order for example, hours must be run first (variable declaration)

2.2.0.3.0.7. Connascence of Timing (CoT)

Type: **Dynamic**

- Parllel Computing
- Interacting with hardware - especially real-time computing
- Distributed Computing

2.2.0.3.0.8. ## Apollo 11 Example

- Requested available memory
- Other programs
- Constant Reboot

2.2.0.3.0.9. Connascnce of Values (CoV)

Type: **Dynamic**

Where two values (variables) must be equal (the same) and if changes, it has to be changed as well

2.2.0.3.0.10. Connascnce of Identity (CoI)

Type: **Dynamic**

When two or more variables has to point the object

3. Using Abstraction in Java (Week 8 Lecture 2)

3.0.0.4. Using Abstraction at Code Level

- Abtraction is a **design principle** rather than a *programming technique*
- You do not have to write generic classes in this unit

3.0.0.5. Features of Java

3.0.0.5.0.11. Class

- is the most important mechnaism in most OO Languages (incl. Java)
  - represent single concept
  - expose a public interface that allows response in order to furfill its responsiblity
  - hide any implementation details that don't directly fullfil that responsiblity
  - ensures that its attribution are in a valid condition rather than relying on client code to maintain its state

3.0.0.5.0.12. ## Visibiltiy Modifiers

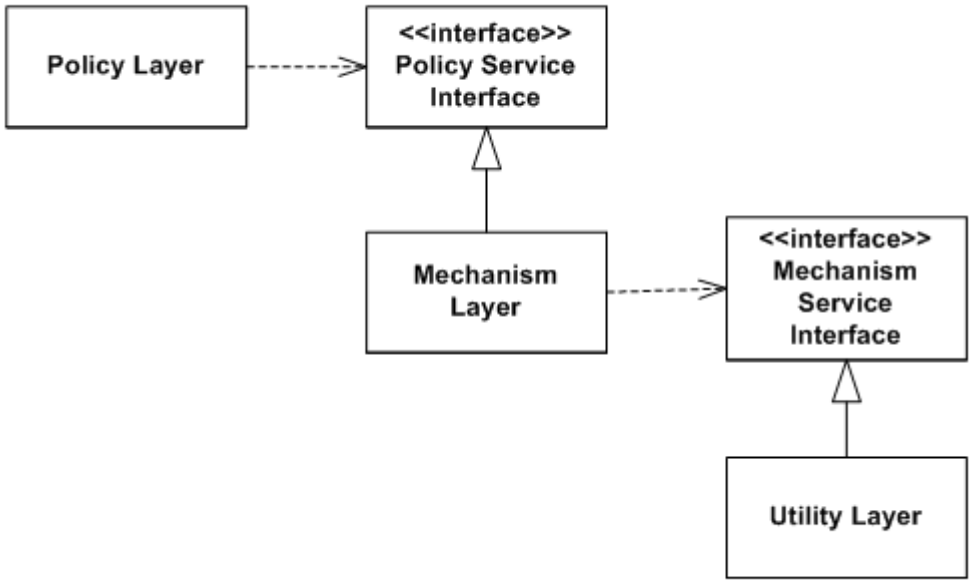
- These include `public`, `private` and `protected`
- in general when in doubt make it `private`
- only provide `getters` and `setters` if you're sure that external classes need to directly manipulate
- if you leave the visibility modifier, your class/attribute/method will be visible within the package which is declared.

3.0.0.5.0.13. ## The Abstract Class

- The `abstract` class cant be instantiated
  - may lack important components
  - such as method bodies
  - inherits the methods and attributes, this means that it can implement the public methods and the attributes specified by the **base** class.

3.0.0.5.0.14. ## Hinge Points

- Applying dependency inversion to a single relationship.
- We take a class, seperately define it's interface as an abstract entity, seperate the code. We can let the client code interact with the abstract interface. They only interact with each other through the



interface.

3.0.0.5.0.15. Packages

We want to split things up into packages. - We group a bunch of `classes` and bundle it into a subsystem. - The boundary around a package is also an encapsulation boundary.

3.0.0.5.0.16. ## Nesting Packages

- You can't put a package inside another package in Java
- `java.util.jar` is not a package within `java.util`
- If you want to use the package, you have explicitly import (e.g. `import java.util`)

3.0.0.5.0.17. Abtraction Layers

- An **abstraction layer** is the publicly accessible interface to a class, package or subsystem.
- You can create an abstraction layer by restricting visiblity as much as possible.
- One problem is to making too much public.

4. FIT2099 Week 9 Lecture A

Student data type

```
Name
StudentID
Address
```

- charactersitics behind system
- System support. Given a `studentID` return `studentName`
- Find specification of the class.

4.0.0.6. Client Supplier Relationship

- We can draw the UML

Client -> Supplier

- **Client** Watch1 "has" 2 counter attributes.
  - Client is a supplier of services to `Watch1`
  - `Watch1` is a client of Counter, and asks it to perform services such as `increment`, `reset()`
  - Inheritments making use of service to.

4.0.0.7. Software Spec: The Problem

- Hardware components
- Well-edfined public interafcaes with a hidden implementation
- Have regorous umabgiousous *specification* of behaviour

4.0.0.8. Design by Contract

- *class* desginer establishes a *software contract* between him/herselfs and the user(s) of the class he/she designs
- make this impersonal. Contract betweent he class that is the supplier and the clients of the class

4.0.0.8.0.18. Software Contract

- Documentation of the class of the technical user
- the possiblity of enforcing the contract by using exceptions and assertions

Software Contract:

```
Class Documentation

public class Documentation{

}
```

- Software designer tells the user what the class does by providing specs for the class
  - What the methods of the class need to operate correctly e.g. `assert studentID to be interger and between 00000001 to 99999999`
  - What the class will guarantee to be if is used correctly

4.0.0.9. Specification of a Class

- A specification
  - is ideally part of the implementation
    - In some languages such as *Eiffel* that is built ibn others that i can done by hand (via the use of assertions and exceptions)

    - There are also extendetions
        - Cofoja (Java)
        - Py Contracts (python)
        - Spec## and Code Contract from Microsoft Research for C## and .Net
    - Should ideally be extractable from the implementation via a tool.
        - e.g. Javadoc when using Cofoja
    - is esetnail supporting component reuse and maintenance
    - is more that just the API we havve gotten used to seeing
        - it includes **comments**, and crucially exexcutable sepcs
- The User:
    - should be able to derermine how to use the class
    - not have to look at implemenation details
- Specs forms the public interface of the class

4.0.0.10. Specs

- Preconditions ('requires')
    - things that need to be true for method to run