**Error Handling:**

Failure: Incorrect behavior; Fault (Bug, Defect); Error: Mistake made by human

Method contract: Operation Signature; Precondition; Postconditions; invariant; Frame condition

**R.E.P.A.I.R. R** – Runtime : (do nothing/let it die) **E** – Exception (catch and handle an exception at throw point) **P** – Propagation (allow exceptions to move up system to point where you will handle them) **A** – Atomic (transactions – ensure something either happens or it doesn’t) **I** – Isolation (keep other processes from seeing changes. i.e. until transaction commits, nothing has happened) **R** – Resumption (retry, code tries to “heal” itself)

**Testing**: Assert.assertEquals(message, expect, actual)

**Whitebox**: all lines of code; each branch direction; find ”plausible faults”

**Blackbox**: don’t look at the code, review specification for the class;

“Plausible” faults (specific values)； Equivalence partitions(one per case);

Limits: Boundary conditions (boundary one off of the spec limit) Edge case (extreme limit)Corner case (situation caused by multiple issues) :

**EX1** If (x <0)..if(y<0).. if(x>30 && y<7).. statement Coverage: (x= -1, y=-1), (x = 3, y = -1), (x=34, y = 3), (x = 34, y = 9)

Branch Coverage: +(x=30,y=7)  **EX2** TEST the range of 0 - 1000 inclusive: Equivalence partitions -45,48,3432 Boundary Condition:-1,0-1000,100 **EX3** if(<0){}; if(>80){}elseif(>50&&<60){}else{}Statement:-2,90,55,45 Branch:-2,90,55,45,75

**Design Reviews**

Design Criteria(FURPS)**F**unctionality /traceability: can trace the code backward; **U**sability|**R**eliability|**P**erformance|**S**ecurity/safety/sustainability( ability to extend in the future)

Quality Factors: Maintainability(how long system has to up)Testability()Reliability/Availability(mean time to failure)Precision and Accuracy(ex. how many decimal places to keep)Security()Performance()Portability(code compile in different operating systems) Extensibility(plugins)Interoperability(communicate with other apps)Usability

**Refactoring**

Bad Smells: Duplicate Code, Long Method, Large Class (violate Low Cohesion, Single Responsibility), Long Parameter List, Switch statements, Message Chains

**Enterprise architectures**

Microservices: Keep each service small & separated

Union (Clean) Architecture: Can only call from outer layer to inner layer

**Law of Demeter** (model.getGame().getPlayer().get); Tell, Don’t Ask (for e: E..e.getxxx); **Open/Closed** (Switch): **Liskov Substitution** (instance of) **Dependency Inversion**: Program to interfaces not implementation: List<int> list = new ArrayList<int>();

**Design Patterns**

**Singleton**: static accessor method; private constructor; problem: Anyone can get reference to class

ADV: global access DISADV: coupling, no reuse

**Monostate:** Make a class where all attributes are class attributes

**Factory:**（Switch） …getM; if(M.equals(..)){…}else if(M.equals(…)){…}

-Replace the switch statement with polymorphism to support open-closed principle;

-Create subclasses of X (one for each type) that each know their special behavior.

**Command**: Encapsulate commands in objects, so we can queue them, undo them or make macros.

**Null Object:** using null to indicate the absence of an object to delegate an operation to

**Observer**： Listeners;

**Visitor**: apply different operations to a collection of objects. -(Double Dispatch): calling the right method based on type or receiver AND arguments.

**Flyweight**: extract attributes shared amongst all instances into a class (SHIP-SHIPTYPE)

**Façade**: Provide an interface to the application and hide the complexity of the actual model

**strategy pattern**: put the unique behavior in its own class; Allow selection of algorithm to vary by object and time

**Packaging:**

**package by layer**: executives, managers, employees

**package by feature**: front office, back office, accounting, personnel, mail room

+High Modularity – reduces coupling between packages

+Easier Code Navigation –related code in same directory

+Minimizes Scope, can use package private to hide most details

**Deployment Principles**: Release—Reuse;

Versioning: x.y.z: x = major revision – incompatible API changes

y = minor revision - add new functionality but still backwards compatible

z = patch revision – backwards compatible bug fixes

**Acyclic Dependency**: Do not have cyclic dependencies in your packages

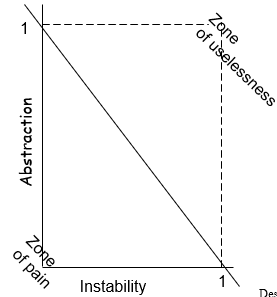
**Dependency Injection** is very popular because: It reduces hard coded dependencies in your code. It allows you to substitute mock classes during testing. It makes the code easier to maintain and change

**Design Metrics** : Distance = A + I – 1;Closer to zero, better the design

– Stable Dependencies Principle: Ci packages depend on me);Co (packages I depend on):

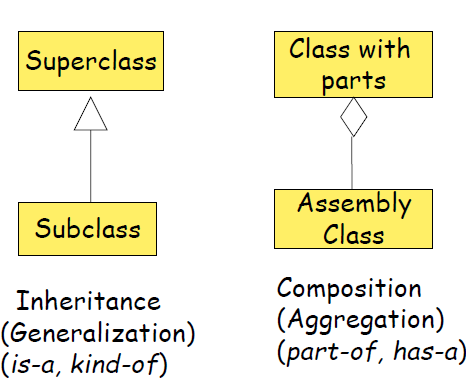
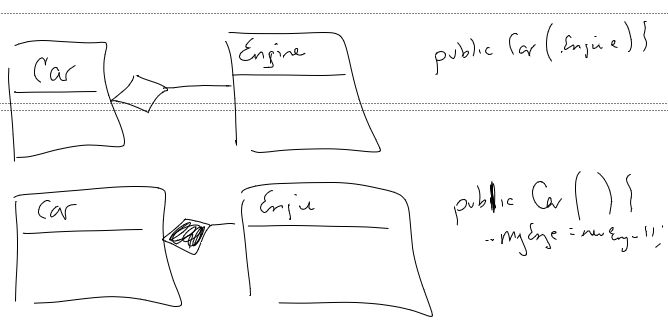
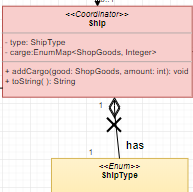
Instability = Co / (Co + Ci)

– Stable Abstractions Principle: A = Na/ (Na + Nc) ; 0 = Totally Concrete ; 1 = Totally Abstract



**UML:**

**Class**: + public # protected – private ~ package {abstract} static final: {readonly}

**Sequence**:

