Software Quality

DESIGN MAINTAINABILITY

Software Quality

- defined by IEEE :
  - degree to which a system, component, or process
    - meets specified requirements
    - meets customer or user needs or expectations

- ISO defines software product quality model (ISO/IEC 25010)

Software Product Quality

Functional Suitability	Performance Efficiency	Compatibility	Usability	Reliability	Security	Maintainability	Portability
• functional completeness • functional correctness • functional appropriateness	• time behaviour • resource utilization • capacity	• co-existence • interoperability	• appropriateness recognizability • learnability • operability • user error protection • user interface aesthetics • accessibility	• maturity • availability • fault tolerance • recoverability	• confidentiality • integrity • non-repudiation • authenticity • accountability	• modularity • reusability • analysability • modifiability • testability	• adaptability • installability • replaceability

Software Quality

- software **design goal** is to achieve quality software
- quality of software design
  - fit for purpose (meet all requirements)
  - easily understandable and highly maintainable
- to improve software design quality :
  - define appropriate goal clearly and concisely
  - use appropriate design principles
  - apply design pattern
  - evaluate or measure quality attributes and redesign
  - remove bad smell
  - refactoring

Design Maintainability

- Understandability
    - how easy is it to understand
- Adaptability
    - how easy is it to change a component
- Extensibility
    - how easy is it to extend a new functionality
- Cohesion
    - how closely are the parts of a component related
- Coupling
    - how independent is a component

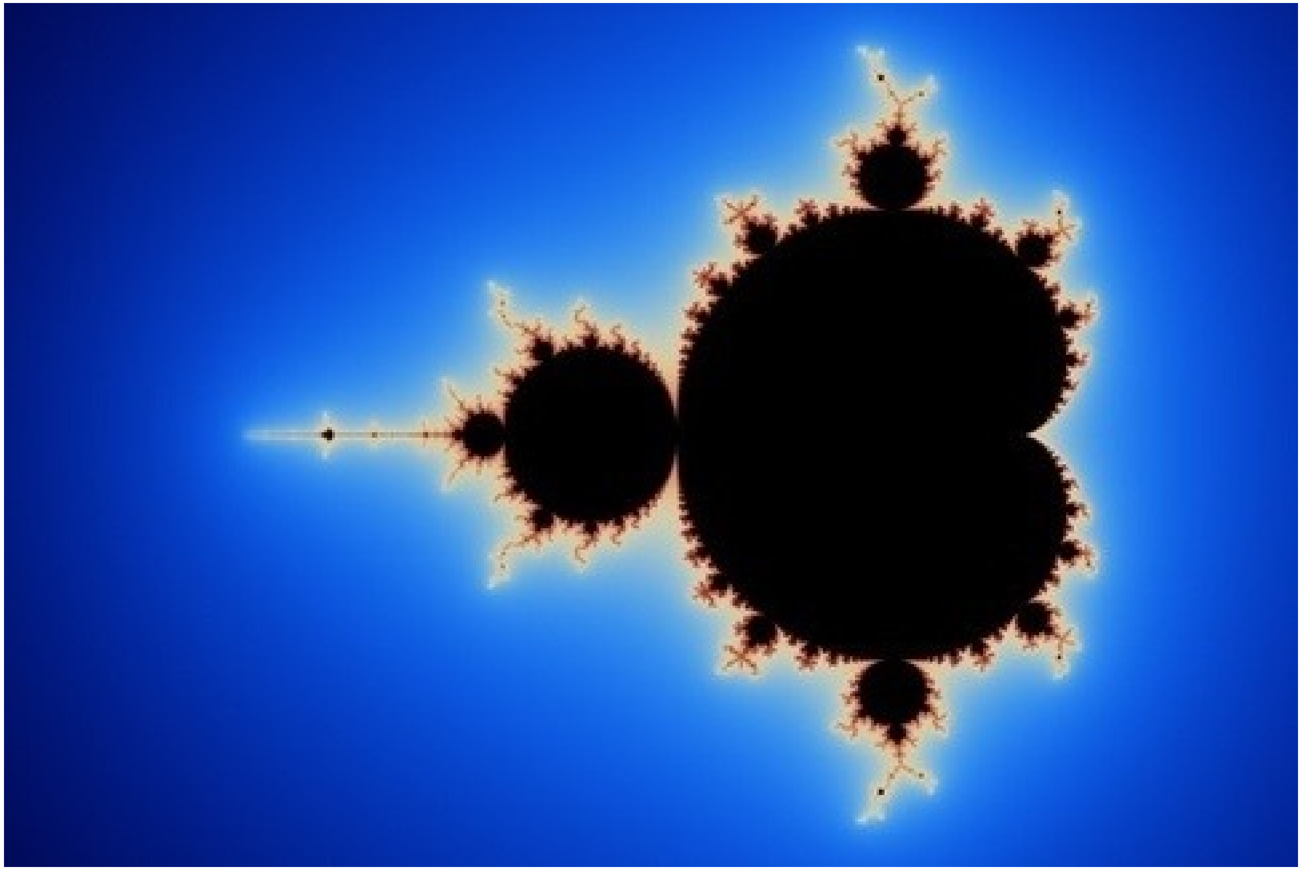
Understandability

```
_                                      =   (
                                        255,
                                      lambda
                               V       ,B,c
                             :c   and Y(V*V+B,B,  c
                               -1)if(abs(V)<6)else
               (              2+c-4*abs(V)**-0.4)/i
                 )  ;v,      x=1500,1000;C=range(v*x
                  );import  struct;P=struct.pack;M,\
            j  ='<QIIHHHH',open('M.bmp','wb').write
for X in j('BM'+P(M,v*x*3+26,26,12,v,x,1,24))or C:
            i  ,Y=_;j(P('BBB',*(lambda T:(T*80+T**9
                  *i-950*T  **99,T*70-880*T**18+701*
                 T  **9     ,T*i**(1-T**45*2)))(sum(
               [              Y(0,(A%3/3.+X%v+(X/v+
                               A/3/3.-x/2)/1j)*2.5
                             /x   -2.7,i)**2 for  \
                               A       in C
                                      [:9]])
                                        /9)
                                       )   )
```

lab(se);

HYU

Understandability



Understandability

```java
public List<int[]> getThem(){
List<int[]> l = new ArrayList<int[]>();
// this is for loop and if statement
for (int[] x : l1){if(x[0] == 4)l.add(x);}return l;}
```

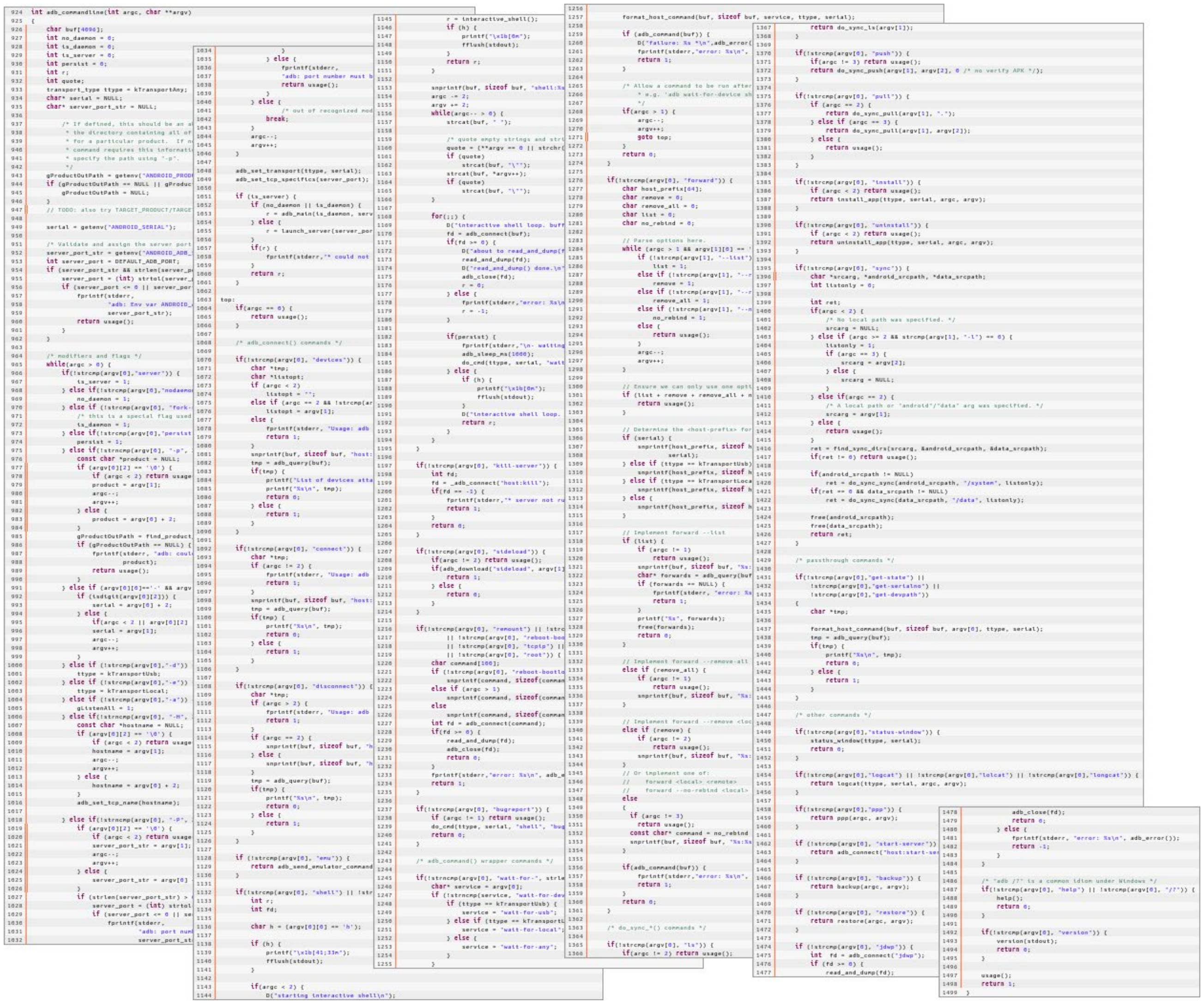
```java
// get all the flagged cells from game board
public List<int[]> getThem() {
  List<int[]> l = new ArrayList<int[]>();
  for (int[] x : l1) {
    if(x[0] == 4){
      l.add(x);
    }
  }
  return l;
}
```

Understandability

```
// get all the flagged cells from game board
public List<int[]> getThem() {
  List<int[]> l = new ArrayList<int[]>();
  for (int[] x : l1) {
    if(x[0] == 4) { l.add(x); }
  }
  return l;
}
```

```
// get all the flagged cells from game board
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard) {
    if (cell.isFlagged()) {
      flaggedCells.add(cell);
    }
  }
  return flaggedCells;
}
```

Understandability



Adaptability

```
// printing array values
public void printValues(int numbers[]) {
    for (int num : numbers)
        System.out.println(num);
}
```

```
// printing array values
public void printPositiveValue(int numbers[]) {
    for (int num : numbers)
        if (num >= 0)
            System.out.println(num);
}
```

```
// printing array values
public void printNegativeValue(int numbers[]) {
    for (int num : numbers)
        if (num < 0)
            System.out.println(num);
}
```

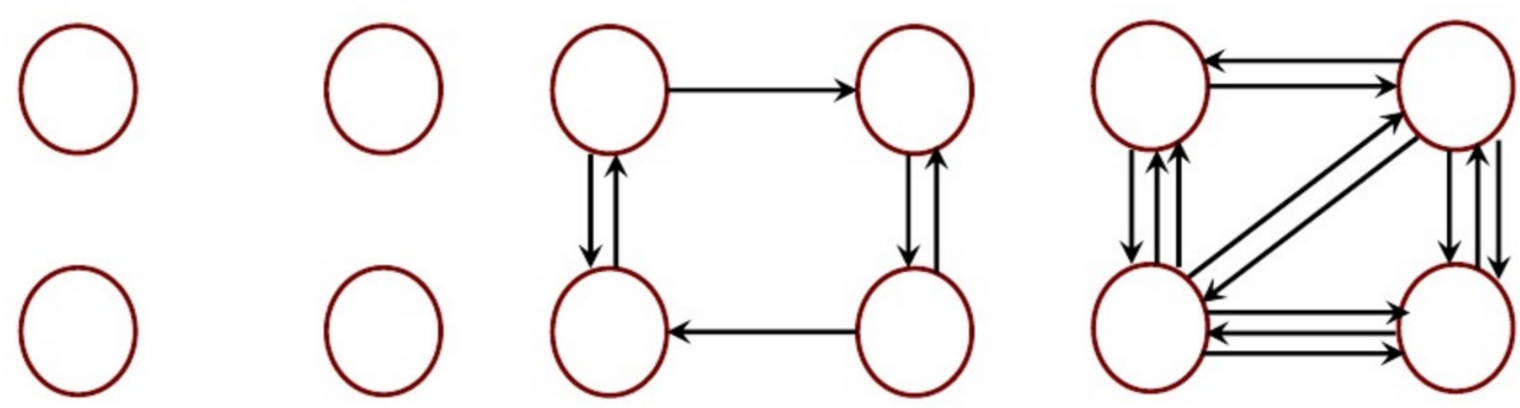
Extensibility

```java
public interface Check {
    public boolean checkCond(int num);
}

public class isPositive implements Check {
    public boolean checkCond(int num){
        return num >= 0;
    }
}

public class isEven implements Check {
    public boolean checkCond(int num){
        return (num % 2) == 0;
    }
}
```

```java
public void printValue(int numbers[], Check check) {
    for (int num : numbers)
        if (check.checkCond(num))
            System.out.println(num);
}
```

Coupling

- degree of interdependency between software modules



uncoupled
no dependency

loosely coupled
some dependencies

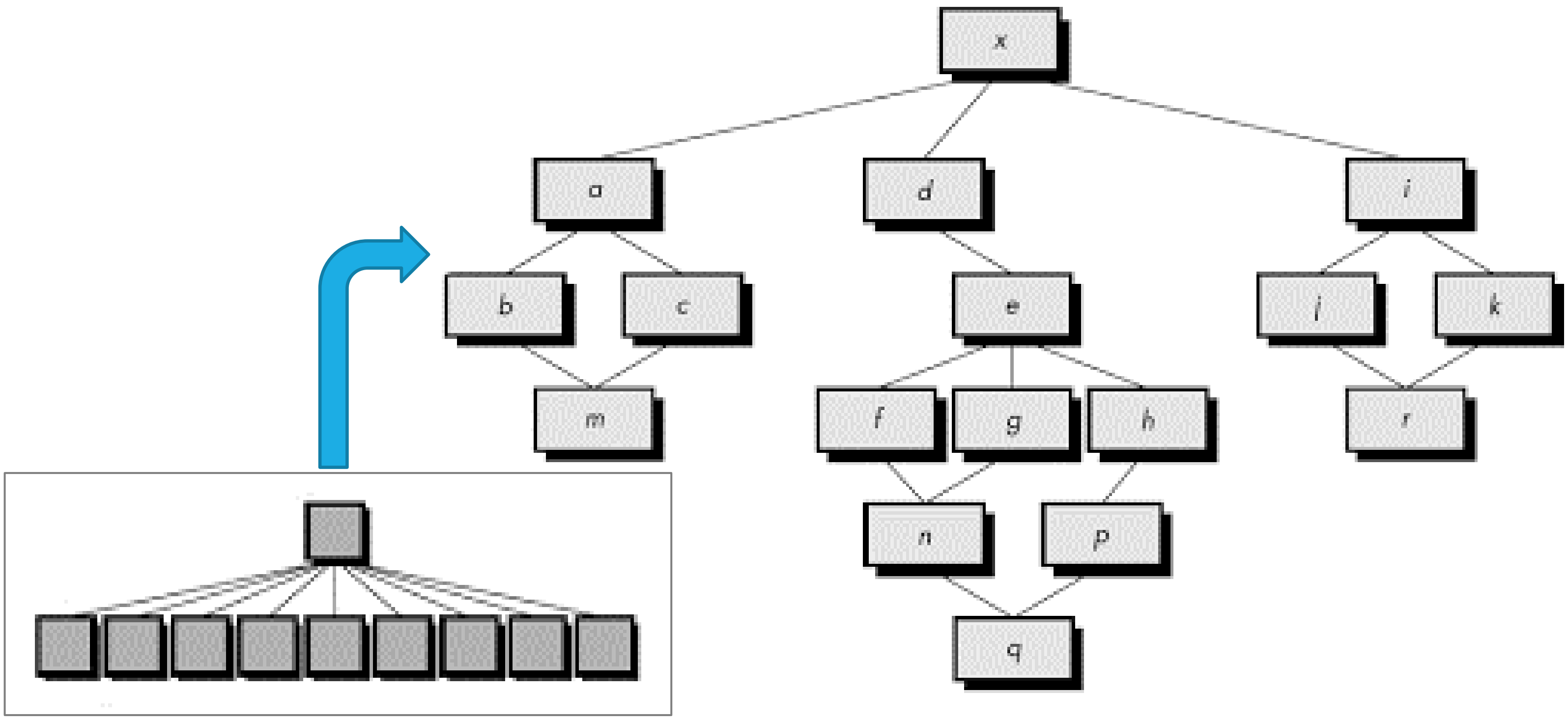
highly coupled
many dependencies

Coupling

- type of coupling
  - no coupling
  - data coupling
  - stamp coupling
  - control coupling
  - external coupling
  - common coupling
  - content coupling

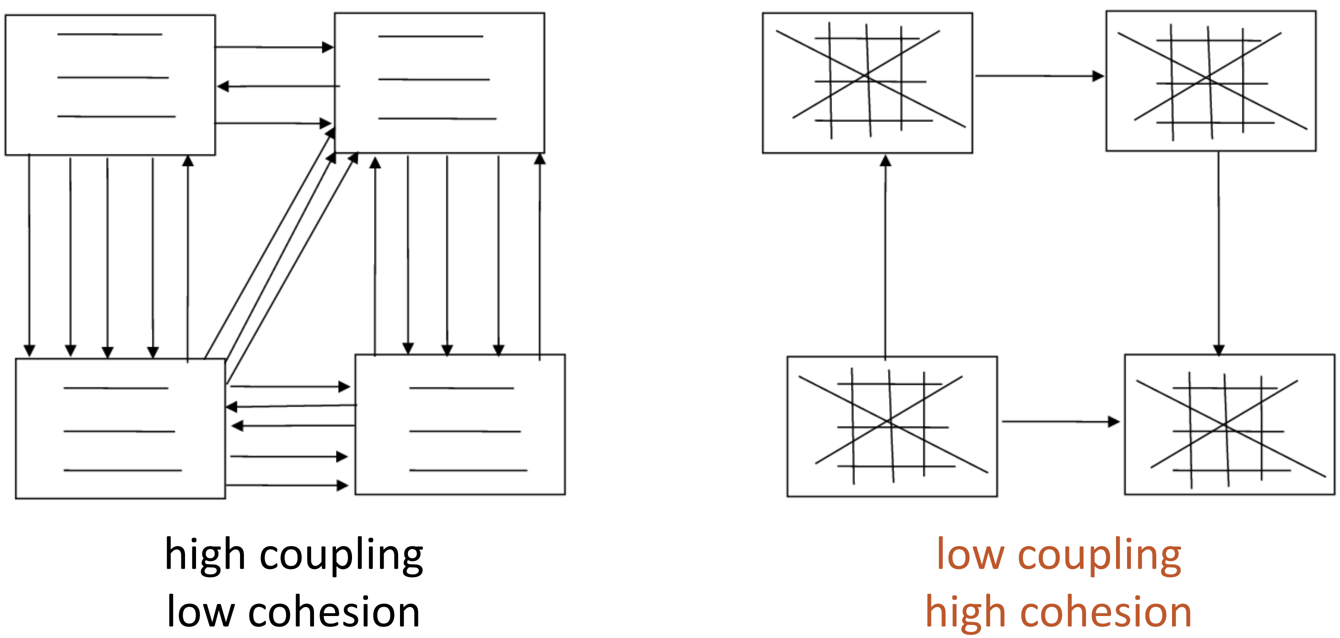
Coupling

- fan-in : number of calling modules
- fan-out : number of called modules
  - high fan-in / out suggests high coupling



Cohesion

- degree to which the elements of a module are functionally related within a module
- cohesion & coupling



Cohesion

- type of cohesion
  - functional cohesion
  - sequential cohesion
  - communicational cohesion
  - procedural cohesion
  - temporal cohesion
  - logical cohesion
  - coincidental cohesion

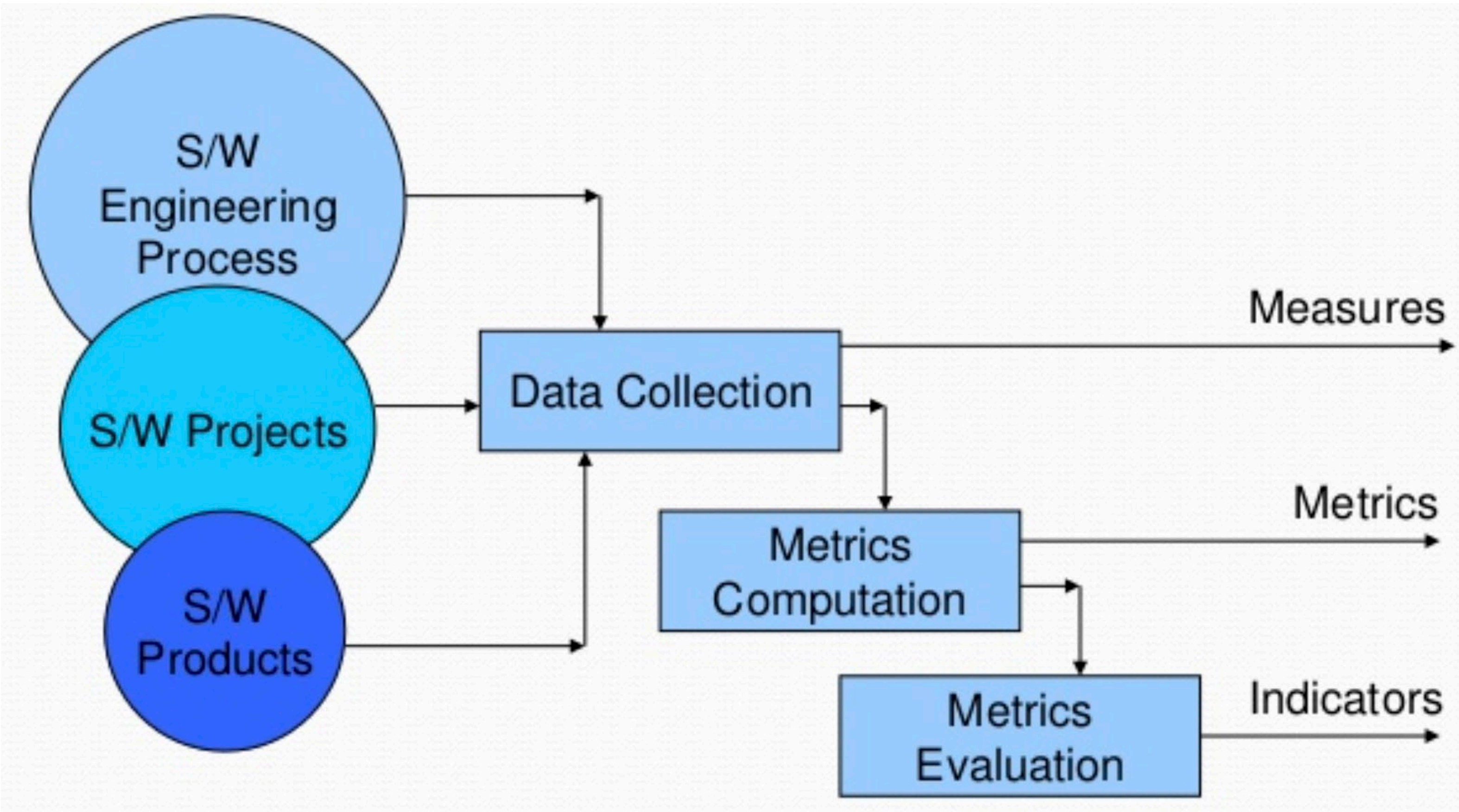
Software Metrics

Measure, Metrics, Indicators

- collects **measures** and develops **metrics** to obtain **indicators**
  - a **measures** provides a **quantitative** indication of the extent, amount, dimension, capacity, or size of some attributes of a product or process
  - IEEE defines a **metrics** as “a quantitative measure of the degree to which a system, component, or process possesses a given **attribute**.”

    - IEEE Standard Glossary of SE Terminology
  - **indicator** is a metric or combination of metrics that provide **insight** into the software process, project, or the product itself

Measure, Metrics, Indicators



Cyclomatic Complexity (CC)

- measures complexity of a module / program
  - = ***number of decisions + 1***

```
public void foo(){
  boolean jar = false;
  int tree = 9;
  if(tree == 9){
    ;
  }
  else if(jar == true){
    ;
  }
  if(tree != 9){
    ;
  }
}
```

CC =

```
public void foo2(int x){
  if(x > 10){
    ;
  }
  else{
    for(int i = 0; i < x; i++){
      ;
    }
  }
}
```

CC =

Cyclomatic Complexity (CC)

Construct	Decision	Reasoning
If..Then	+1	An If statement is a single decision.
ElseIf..Then	+1	ElseIf adds a new decision.
Else	0	Else does not cause a new decision. The decision is at the If.
#If..#ElseIf..#Else	0	Conditional compilation adds no run-time decisions.
Select Case	0	Select Case initiates the following Case branches, but does not add a decision alone.
Case	+1	Each Case branch adds a new decision.
Case Else	0	Case Else does not cause a new decision. (decisions were made at other Cases)
For [Each] .. Next	+1	There is a decision at the For statement.
Do While|Until	+1	There is a decision at the start of the Do..Loop.
Loop While|Until	+1	There is a decision at the end of the Do..Loop.
Do..Loop alone	0	There is no decision in an unconditional Do..Loop without While or Until.
While	+1	There is a decision at the start of the While..Wend or While..End While loop.
Catch	+1	Each Catch branch adds a new conditional path of execution. Even though a Catch can be either conditional or unconditional, we treat all of them the same way.
Catch..When	+2	The When condition adds a second decision.

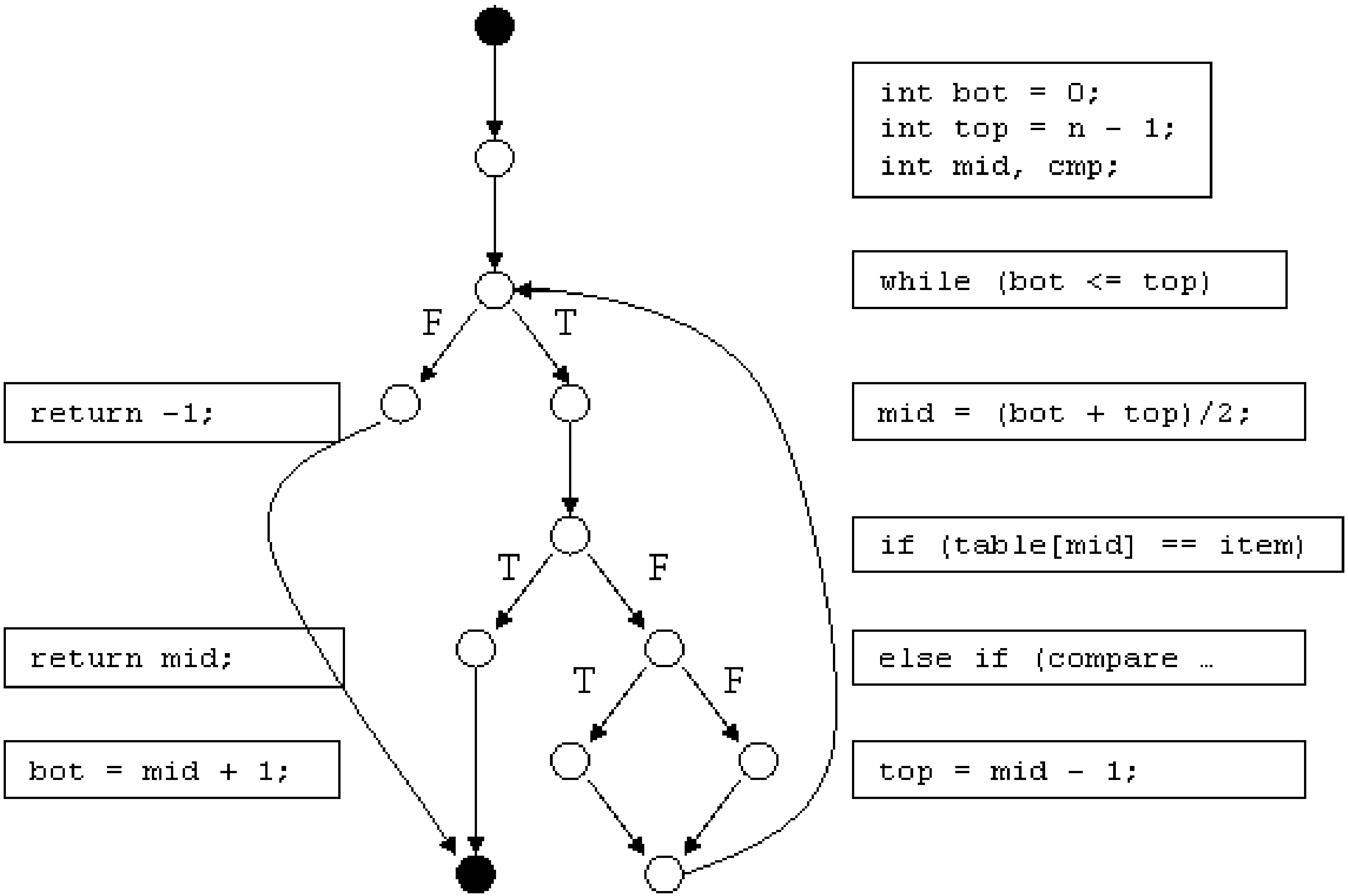
http://www.aivosto.com/project/help/pm-complexity.html

Cyclomatic Complexity (CC)

```
int BinSearch (char *item, char *table[], int n) {
   int bot = 0;
   int top = n - 1;
   int mid, cmp;
   while (bot <= top) {
      mid = (bot + top) / 2;
      if (table[mid] == item)
         return mid;
      else if (compare(table[mid], item) < 0)
         top = mid - 1;
      else
         bot = mid + 1;
   }
   return -1; // not found
}
```

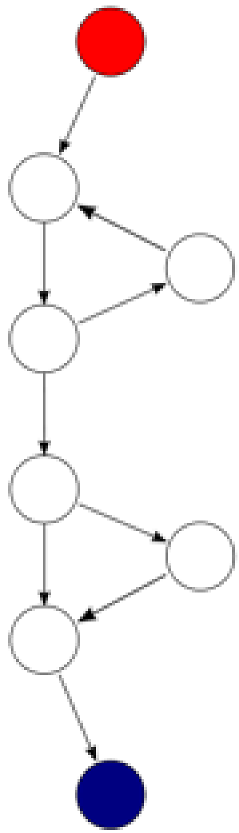
CC =

Cyclomatic Complexity (CC)



CC =

Cyclomatic Complexity (CC)



CC =

Cyclomatic Complexity (CC)

- Risk detection

CC	Type of procedure	Risk
1 -4	a simple procedure	low
5-10	a well structured and stable procedure	low
11-20	a more complex procedure procedure	moderate
21-50	a complex procedure, alarming	high
>50	An error-prone, extremely troublesome, untestable procedure	very high

Nesting Depth

- number of structuring levels

```
public void foo(int x){
   if(x > 10){
      ;
   }
}
```

Nesting Depth = 1

```
public void foo2(int x){
   int y = 0;
   while(x > 10){
      if(x > 20){
         //... y = ~
      }
      else{
         //... y = ~
      }
   }
   if(y > 10) return 1;
   else return 0;
}
```

Nesting Depth = 2

Npath

- number of (static) execution path

```
public void foo(int x){
  if(x > 10){
    ;
  }
}
```

NPath = 2

```
public void foo2(int x){
  int y = 0;
  while(x > 10){
    if(x > 20){
      //... y = ~
    }
    else{
      //... y = ~
    }
  }
  if(y > 10) return 1;
  else return 0;
}
```

NPath = 6

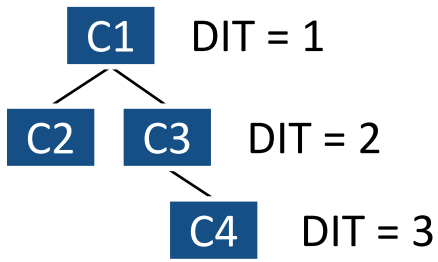
Chidamber & Kemerer Metrics

- Object-Oriented (OO) metrics designed to
  - measure **aspects** of the OO approach
  - measure **complexity** of the design
  - improve the **development** of software
    - identify outlying values; a signal of high complexity / design violations
    - taking managerial decisions
      - re-designing
      - assigning extra or higher skilled resources to develop, test, maintain

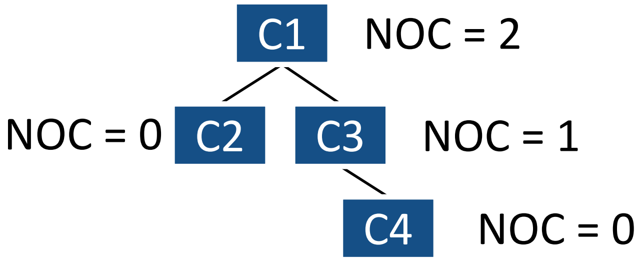
Chidamber & Kemerer Metrics

- **Weighted Methods per Class (WMC)**
  - total **complexity** of methods in a class
    - can predict required **time/effort** to develop or maintain class

- **Depth of Inheritance Tree (DIT)**
  - **depth** of **class** in inheritance tree



- **Number of Children (NOC)**
  - number of immediate **subclasses**



Chidamber & Kemerer Metrics

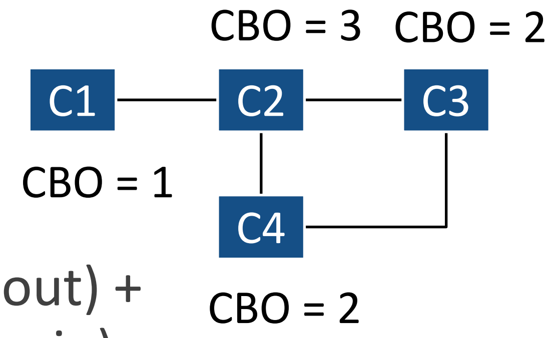
- **coupling metrics**

- **Response For Class (RFC)**
  - measures coupling in terms of method calls

  = no. of methods in class (without inherited methods) +
  no. of distinct method calls made by the methods in the class

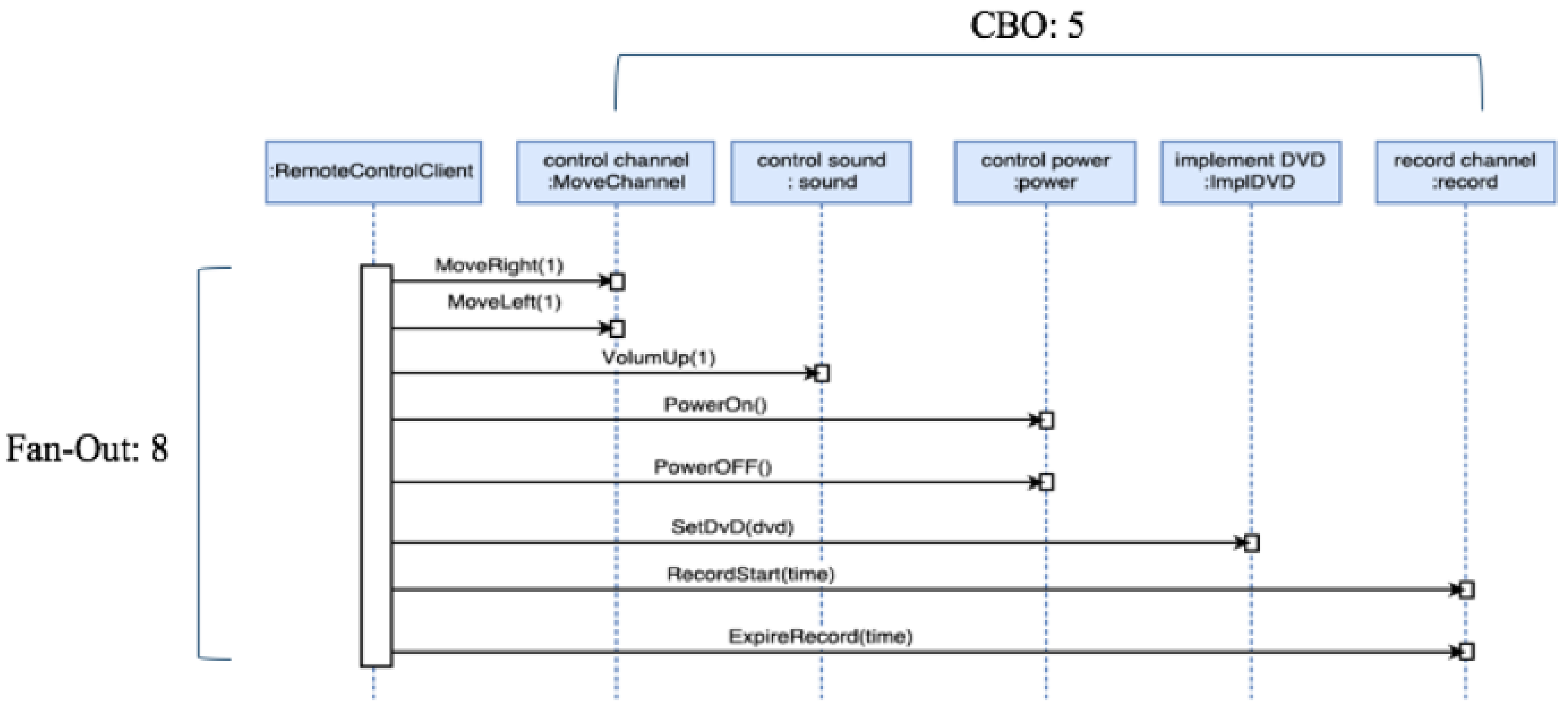
- **Coupling Between Objects (CBO)**
  - measures coupling in terms of classes

  = no. of classes that a class references (fan-out) +
  no. of classes that references the class (fan-in)



Chidamber & Kemerer Metrics

Coupling between Object (CBO)



CBO: Coupling Between Object Classes

Chidamber & Kemerer Metrics

- **Lack of Cohesion Method (LCOM)**
  - measures dissimilarity of methods in a class using instance variable or attribute

= count of no. of method pairs whose similarity is zero

given $\boldsymbol{n}$ methods $\boldsymbol{M_1, M_2, \ldots M_n}$ contained in a class $\boldsymbol{C_1}$ which also contains a set of instance variables $\boldsymbol{\{I_i\}}$

then for any method $\boldsymbol{M_i}$ we can define the partitioned set of

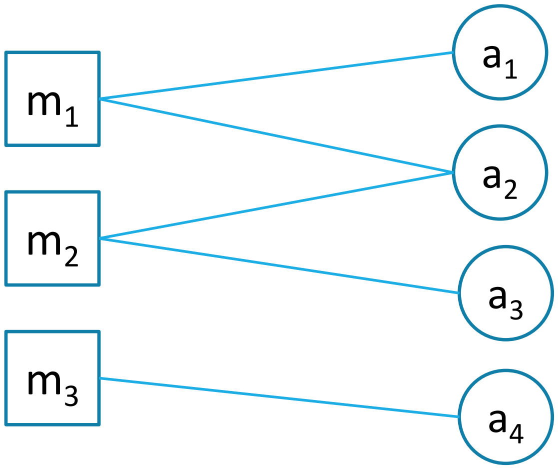
$$P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\} \quad \& \quad Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$$

then **LCOM** $= |P| - |Q|$, if $|P| > |Q|$

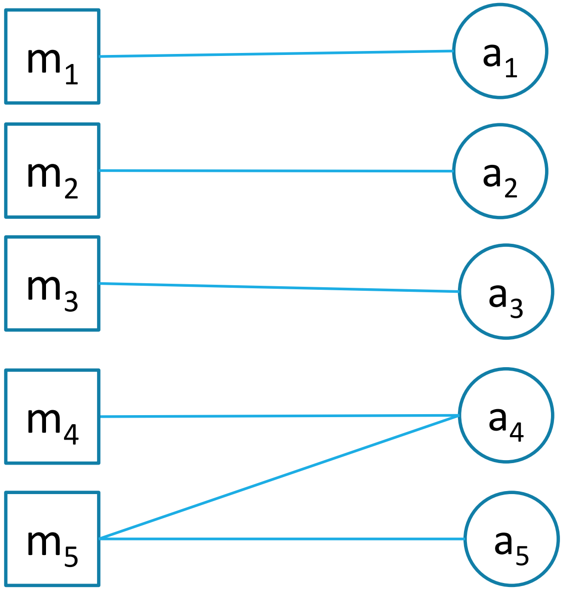
$= 0$ otherwise

Chidamber & Kemerer Metrics

Lack of Cohesion Method (LCOM)



LCOM = 2 - 1



LCOM = 9 - 1

Chidamber & Kemerer Metrics

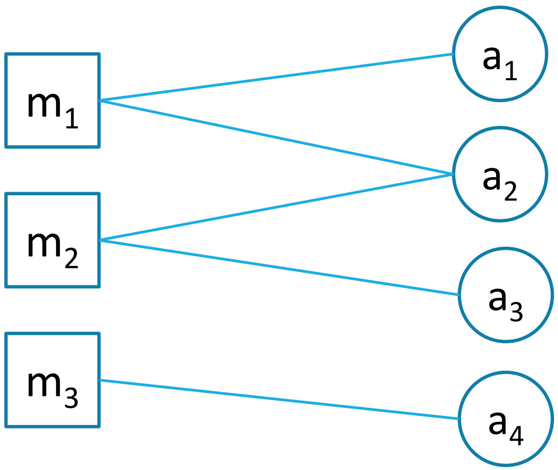
- **Lack of Cohesion of Methods (LCOM)**
  - another definition : LCOM2
  - Henderson-Seller version : LCOM-HS
    - ***M*** : number of methods in class
    - ***A*** : number of variables in class
    - ***MA*** : number of methods that access a particular variable
    - ***Sum(MA)*** : sum of ***MA*** over all attributes of class

then **LCOM2** = $\mathbf{1 - (\mathit{Sum(MA)} / (\mathit{M} \times \mathit{A}))}$

LCOM-HS = $\mathbf{(\mathit{M} - \mathit{Sum(MA)} / \mathit{A}) / (\mathit{M} - 1)}$

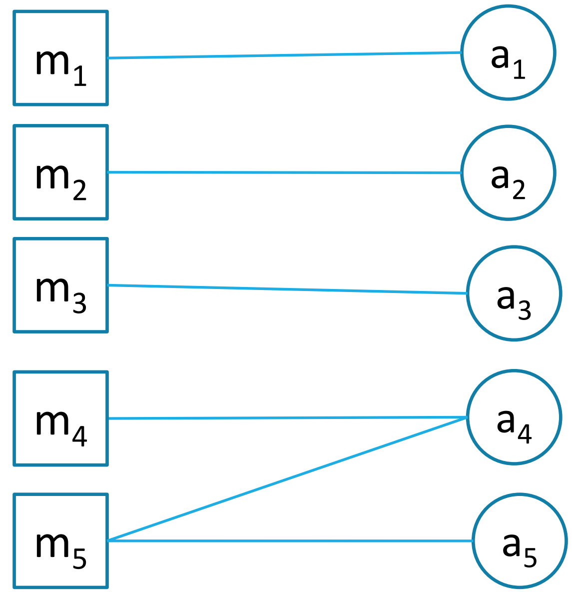
Chidamber & Kemerer Metrics

Lack of Cohesion Method (LCOM2 & LCOM-HS)



LCOM2 = 1 – (5 / 12) = 0.583

LCOM-HS = (3 - 5 / 4) / 2 = 0.875

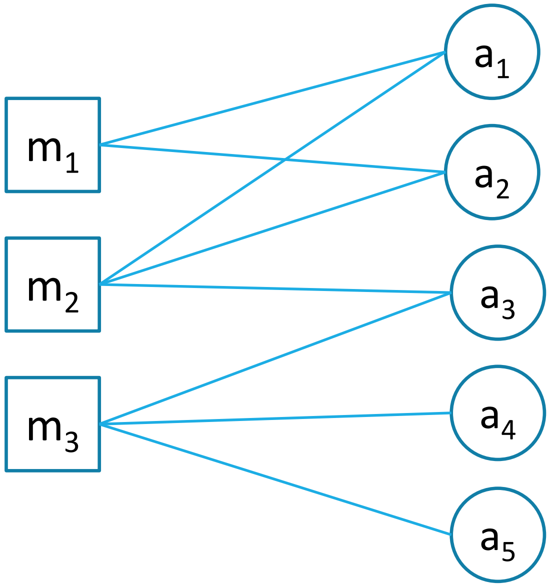


LCOM2 = 1 – (6 / 25) = 0.76

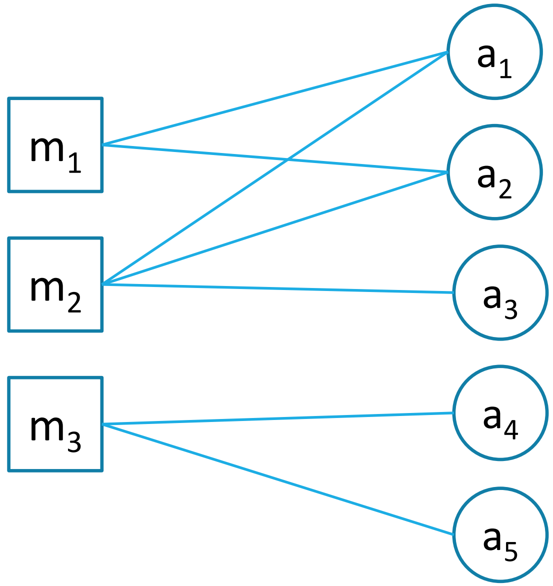
LCOM-HS = (5 - 6 / 5) / 4 = 0.95

Chidamber & Kemerer Metrics

Lack of Cohesion Method



LCOM =
LCOM2 =
LCOM-HS =



LCOM =
LCOM2 =
LCOM-HS =

Chidamber & Kemerer Metrics

- summary

Metric	Goal	Level	Complexity (To develop, to test and to maintain)	Re-usability	Encapsulation, Modularity
WMC	Low	▼	▼	▲	
DIT	Trade-off	▼	▼	▼	
		▲	▲	▲	
NOC	Trade-off	▼	▼	▼	
		▲	▲	▲	
CBO	Low	▼	▼		▲
RFC	Low	▼	▼		
LCOM	Low	▼	▼		▲

Robert C. Martin Metrics

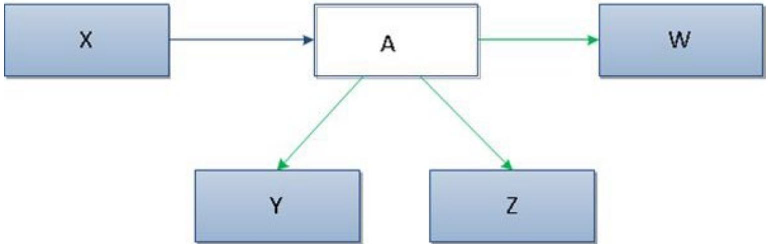
- group of OO metrics
  - proposed by “Uncle Bob” in 1994
  - do not represent attributes to access individual OO design
  - focus on **relationship between packages**
    - incoming and outgoing dependencies to determine **instability** of package

  - Efferent **C**oupling (Ce)
  - Afferent **C**oupling (Ca)
  - **I**nstability (I)
  - **A**bstractness (A)
  - Normalized **D**istance from Main Sequence (D)

Robert C. Martin Metrics

- **Efferent Coupling** (Ce)
  - number of classes in a given package which depend on classes in other packages (number of outgoing dependencies)

Ce = 2



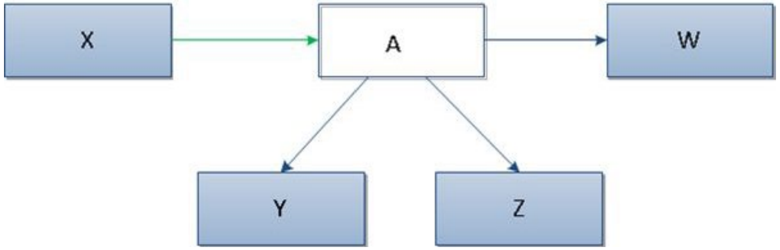
Ce = 3

  - high Ce (> 20) indicates instability of a package
    - changes in numerous external classes can cause need for changes to the package
    - preferred Ce value : 0 ~ 20

Robert C. Martin Metrics

- **Afferent Coupling (Ca)**
  - number of classes in other packages which depend on classes in a given package (number of incoming dependencies)

Ca = 3



Ca = 1

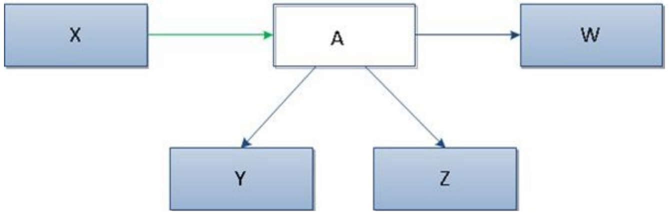
  - high Ca usually suggest high component stability
    - since many other classes depends on the class, it cannot be modified significantly as probability of spreading such changes increase
    - preferred Ca value : 0 ~ 500

Robert C. Martin Metrics

- **Instability** (I)
  - measure relative susceptibility of class to changes - potential stability

$$I = \frac{Ce}{Ce + Ca}$$

I = 2 / (2 + 3) = 0.4



I = 3 / (3 + 1) = 0.75

  - I value ranges between 0 (maximal stability) ~ 1 (maximal instability)
    - close to 1 : unstable due to possibility of easy changes to the package
    - close to 0 : more difficult to modifying due to their greater responsibility
  - preferred I value : 0 ~ 0.3 or 0.7 ~ 1

Robert C. Martin Metrics

- **Abstractness (A)**
  - measure degree of **abstraction** of package

$$A = \frac{AbstractClasses}{AbstractClasses + ContreteClasses}$$

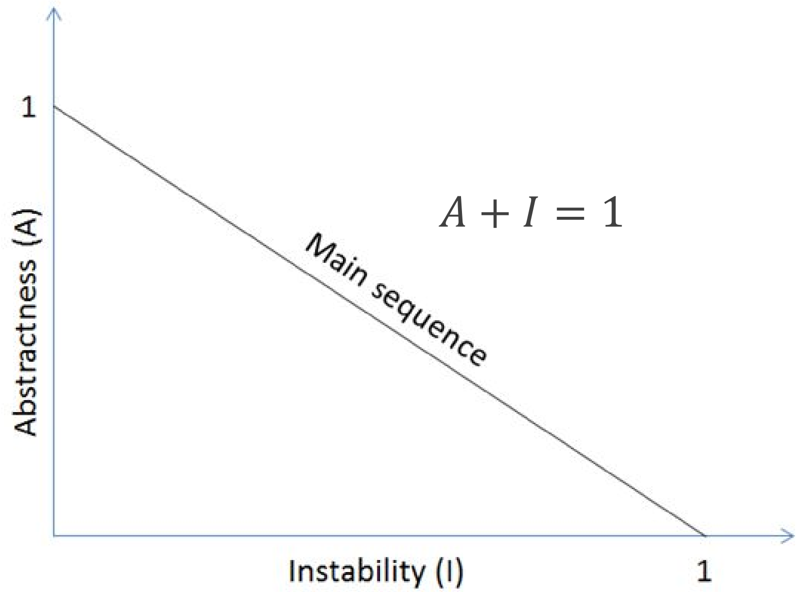
AbstractClasses : number of abstract classes in a package

ConctreteClasses : number of concrete classes in a package

  - preferred A value : extreme values close to 0 or close to 1
    - stable package (I close to 0), which are dependent on other packages, should also be abstract (A close to 1)
    - unstable package (I close to 1) should consists of concrete classes (A close to 0)

Robert C. Martin Metrics

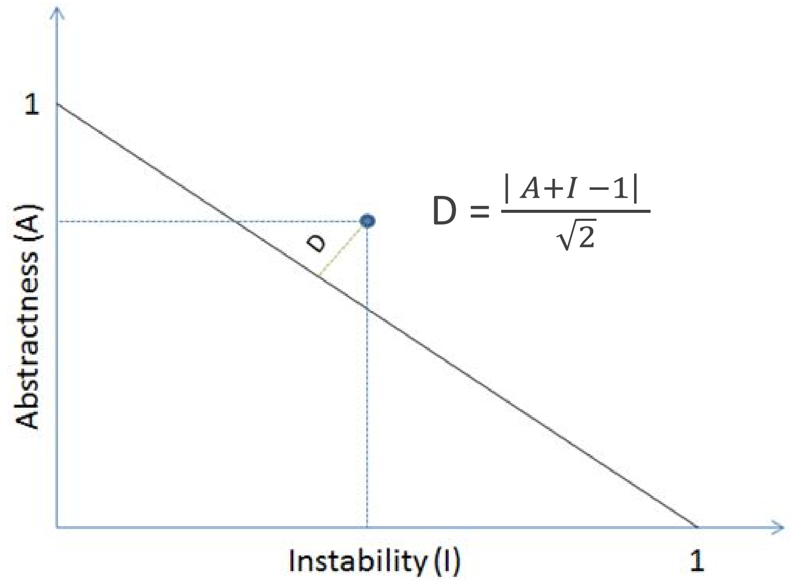
- **Main Sequence**
  - combining abstractness and stability
    - instability of the class is compensated by its abstractness



    - if Ca is small, I will be close to 1 : A should be close to 0
    - if Ca is large, I will be close to 0 : A should be close to 1

Robert C. Martin Metrics

- **Normalized Distance from Main Sequence (D)**



  - D should be as low as possible to be close to the main sequence
    - A = 0 and I = 0, package is extremely stable and concrete but undesirable as package is very rigid and cannot be extended
    - A = 1 and I = 1, impossible completely abstract package must have connection to outside for instance could be created

Design Principles

Key Design Principles

- **S**RP Single Responsibility Principle
- **O**CP Open / Closed Principle
- **L**SP Liskov Substitution Principle
- **I**SP Interface Segregation Principle
- **D**IP Dependency Inversion Principle

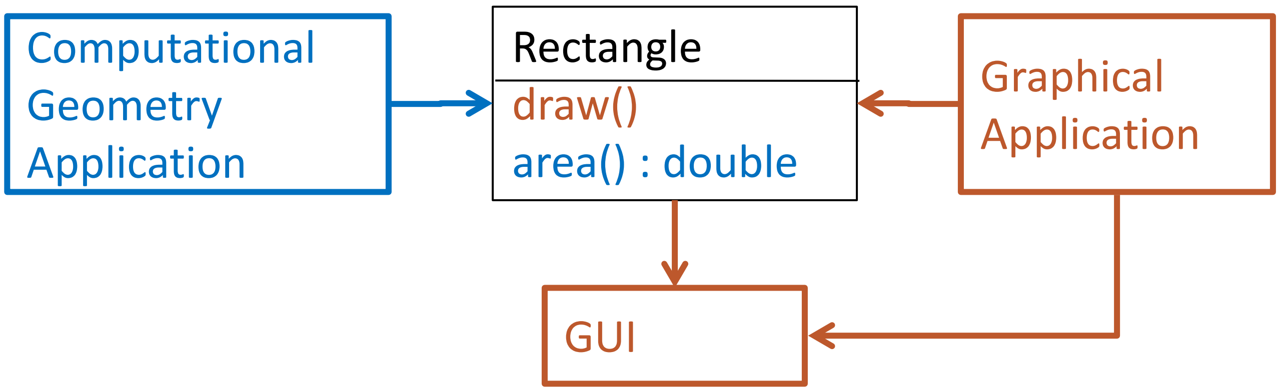
- DRY Don’t Repeat Yourself Principle
- SCP Speaking Code Principle

Single Responsibility Principle

- "A class should have one, and only one reason to change"
  - Robert C. Martin, 2003
- every object should have a single responsibility which should be entirely - Wikipedia
  - responsibility :
    - a reason to change
    - maps to requirements
    - more requirements leads to more possible change
    - more responsibilities leads to more changes in the code
    - multiple responsibilities in one class causes a possible high coupling
    - higher the coupling, higher the possibility of error on change

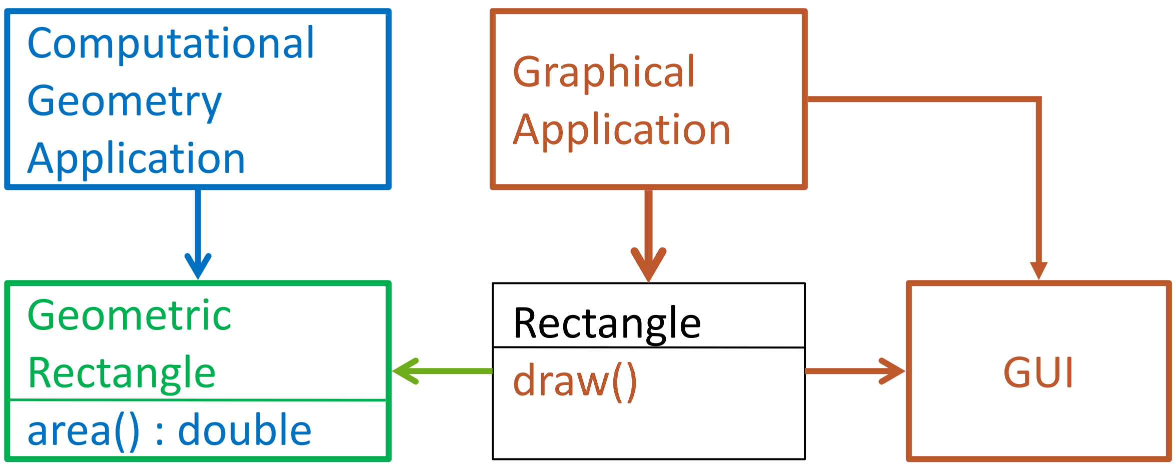
Single Responsibility Principle

- violation of the SRP causes :
  - rigidity of code
  - fragility of code



Single Responsibility Principle

- violation of the SRP causes :
  - rigidity of code
  - fragility of code



Single Responsibility Principle

- multiple responsibilities in a single class can be removed using :
  - multiple small interfaces (ISP)
  - many small classes
  - distinct responsibilities
- to yields :
  - flexible design
  - lower coupling
  - higher cohesion

Single Responsibility Principle

- SRP related bad smells
  - long methods
    - method containing too many lines of code
  - divergent changes
    - making change to a class leads to changing many unrelated methods
- coupling related bad smells
  - message chains
    - existence of series of calls

Open / Closed Principle

- "All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version"

  – Ivar Jacobson

Open / Closed Principle

- “extend the behavior of a system without having to modify the system” – Bertrand Meyer, 1988
- “software entities (classes, modules, functions, etc) should be opened for extension, but closed for modification” – Wikipedia

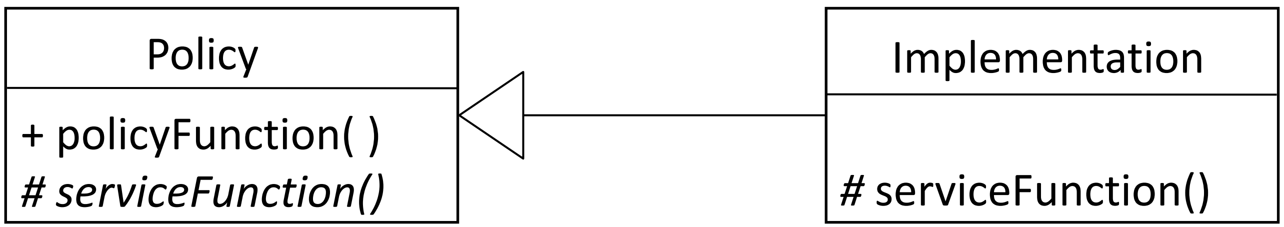
  - changing behavior without modifying the code
    - rely on abstractions not on implementations
    - do not limit the variety of implementation
    - by inheritance or abstract classes or parameterization

Open / Close Principle

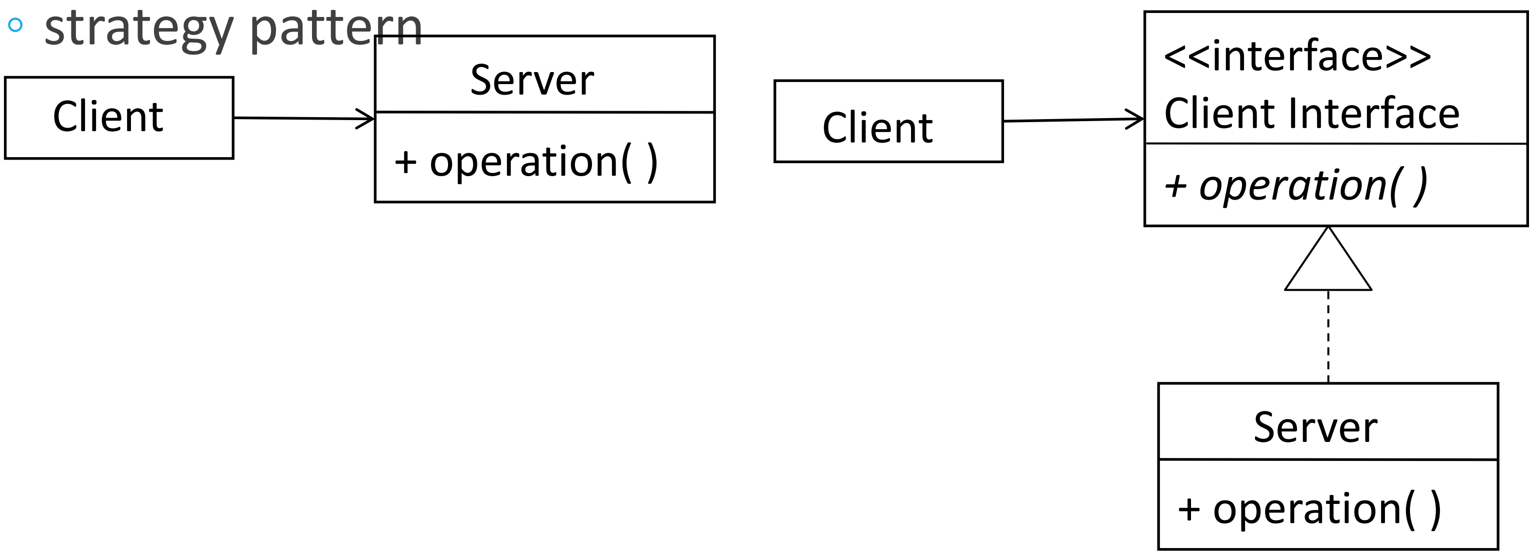
- 3 approaches to implement :
  - parameters
    - pass delegates / callbacks
  - inheritance / template methods pattern
    - child type override behavior of a base class
  - composition / strategy pattern
    - client code depends on abstraction
    - ’plug in’ model
- when to apply :
  - experience tells you – “fool me once, shame on you”
- adds complexity to design – TANSTAAFL
- no design can be closed against all changes

Open / Close Principle

- Pattern approaches to implement :
  - template methods pattern



  - strategy pattern



Liskov Substitution Principle

- "subtypes must be substitutable for their base type"

  \- Robert C. Martin

  - (strong) behavior subtyping - Barbara Liskov, 1988
  - substitutability : child class must NOT
    - remove parent class behavior
    - violate parent class intent
  - normal OOP vs. Liskov Substitution inheritance
    - IS-A relationship vs. IS-SUBSTITUTABLE-FOR relationship
    - e.g. dog is an animal vs. dog is substitutable for animal

Interface Segregation Principle

- “clients should not be forced to depend on methods they do not use” – Robert C. Martin
  - prefers small, cohesive interfaces
  - divide ’fat’ interfaces into smaller ones, as having ‘fat’ interface leads to :
    - classes having methods they do not use
    - increase in coupling
    - reduced flexibility
    - reduced maintainability

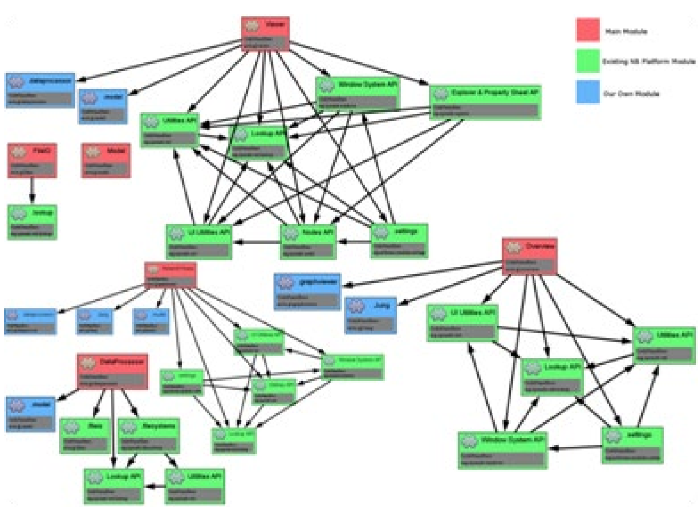
Interface Segregation Principle

- applying ISP :
  - if the ’fat’ interface is yours, separate it to smaller ones
  - if the ’fat’ interface is NOT yours, use ’Adapter’ pattern

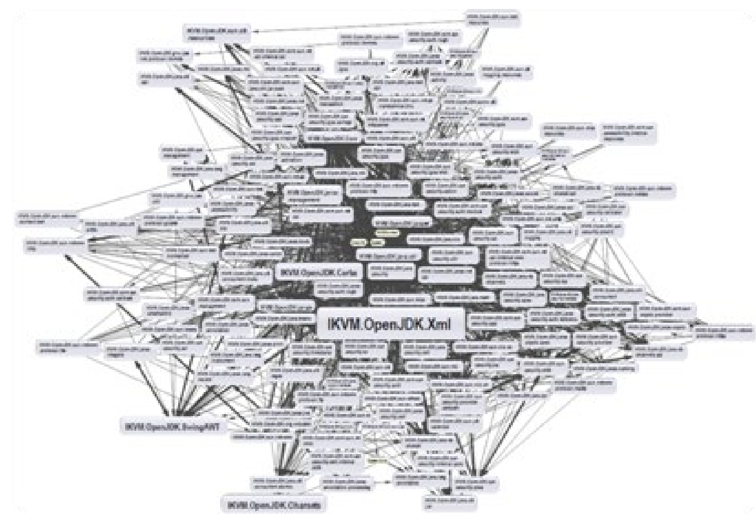
  - small interfaces
  - cohesive interfaces
  - focused interfaces
  - let the client define interfaces
  - package interfaces with their implementation

Dependency Inversion Principle

- "high-level modules should not depend on low-level modules, both should depend on abstractions"
- "abstractions should not depend on details, details should depend on abstraction" - Robert C. Martin, 2003

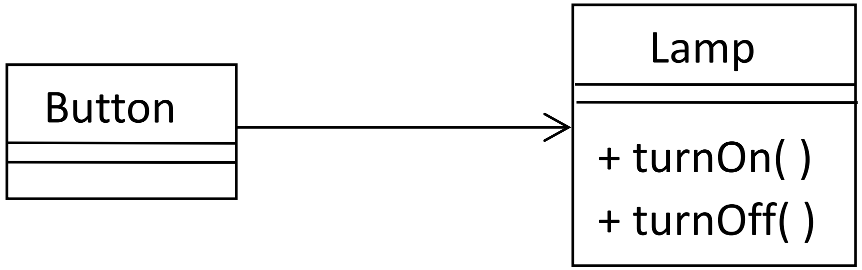


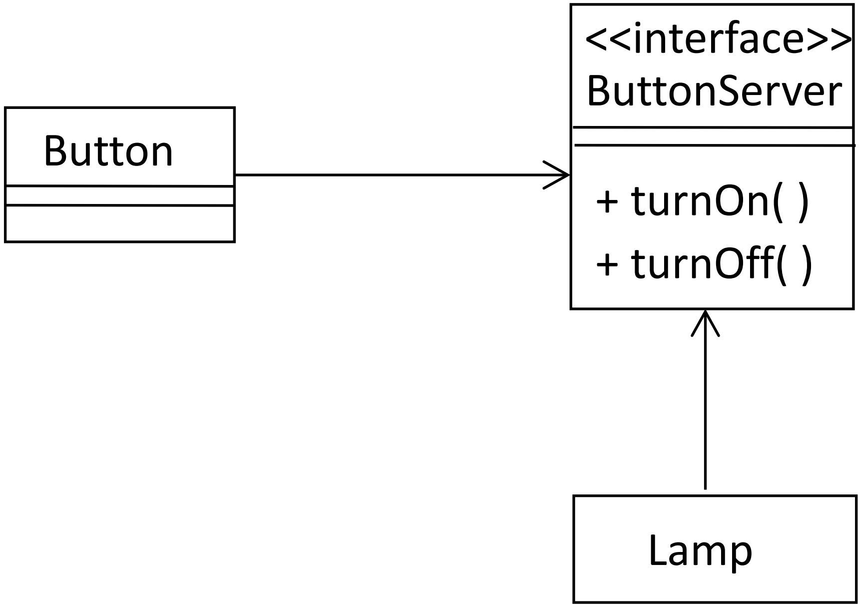
depend on abstractions



depend directly on other modules

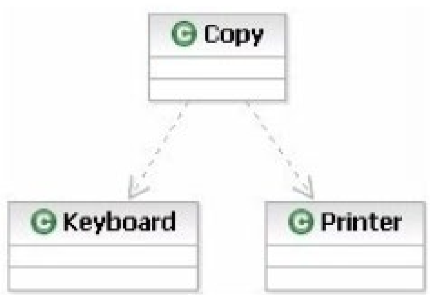
Dependency Inversion Principle

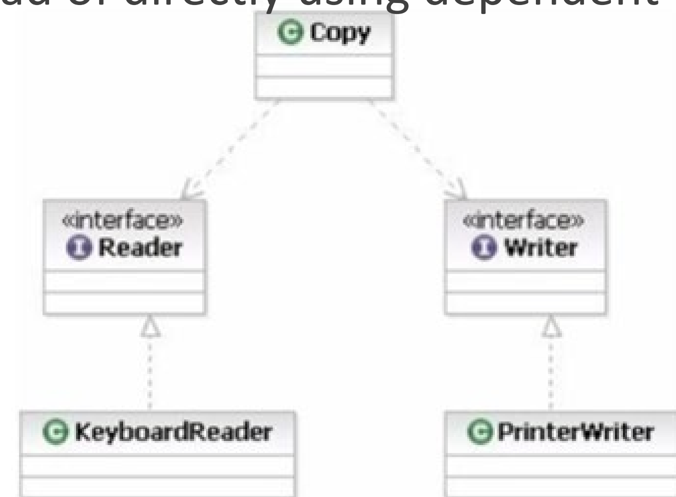




Dependency Inversion Principle

- depend on abstraction
  - to work through interfaces instead of directly using dependent classes





- desired property :
  - classes should declare what they need
  - constructors should require dependencies
  - hidden dependencies should be shown
  - dependencies should be abstractions

Dependency Inversion Principle

- Applying DIP :
  - dependency injection
    - constructor injection
    - property injection
    - parameter injection
  - Hollywood principle : Don’t call us, we’ll call you!
- DIP related smell
  - inappropriate intimacy
    - a class using the internal fields and methods of another class

Don’t Repeat Yourself

- “every piece of knowledge must have a single, unambiguous representation in the system”

  - Andrew Hunt & David Thomas, 1999

- “repetition in logic calls for abstraction, repetition in process calls for automation”
  - variation : Once and Only Once (OOO) & Duplication Is Evil (DIE)

- DRY related smells
  - shotgun surgery
    - any modification requires many small changes to many different classes

Speaking Code Principle

- “Any fool can write code that computer can understand, Good programmers write code that humans can understand” - Martin Fowler, 1999
  - code should communicate its purpose

- SCP related smell
  - comments
    - method filled with explanatory comments

Design Pattern

Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" - Christopher Alexander (Architect)
  - applicable to object-oriented design pattern
  - patterns are a solution to a problem in a context

Parts of Design Patterns

- Essential Parts :
  - pattern name
    - brief description of the design problem
    - **common vocabulary** for software designer
  - problem
    - description of the problem that the design pattern is intended to solve
  - solution
    - description of what elements are required to make up the design, their relationship and its context
  - Consequences
    - results and trade offs by applying the design pattern
    - allows comparison between different design patterns to find better fit for the actual problem

Types of Design Patterns

  - general description of the type of problem the pattern addresses
- Creational Patterns
  - everything about the creation of objects
    - initializing and configuring classes and objects
- Structural Patterns
  - how classes and objects are composed to form larger structure
    - decoupling the interface and implementation of classes and object
- Behavioral Patterns
  - Algorithms and the assignment of responsibilities between objects
    - dynamic interaction among societies of classes and objects

Types of Design Patterns

- Creational Patterns
  - class : factory method
  - object : abstract factory, builder, prototype, singleton

- Structural Patterns
  - class : adapter
  - object : adapter, bridge, composite, decorator, façade, flyweight, proxy

- Behavioral Patterns
  - class : interpreter, template method
  - object : chain of responsibility, command, iterator, mediator, memento, observer, state, strategy, visitor