投票 最多可选1项

设置

Are you ready ?

A Yes

B No

提交

Review – Ch13 Architectural Design

- Architectural Styles
  - Data-centered architectures, Data flow architectures, call and return architectures, Object-oriented architectures, Layered architectures
  - Cloud, MVC (django), Microservice

- How to do architecture design
  - architectural considerations: Economy , Visibility , Spacing , Symmetry , Emergence
  - context and archetypes
  - review / analysis

Software Engineering

Part 2 Modeling

Chapter 14 Component-Level Design

Contents

Contents

14.1 What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as

  "A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

- *OO view:*

  A component contains a set of collaborating classes.

- *Conventional view:*

  A component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

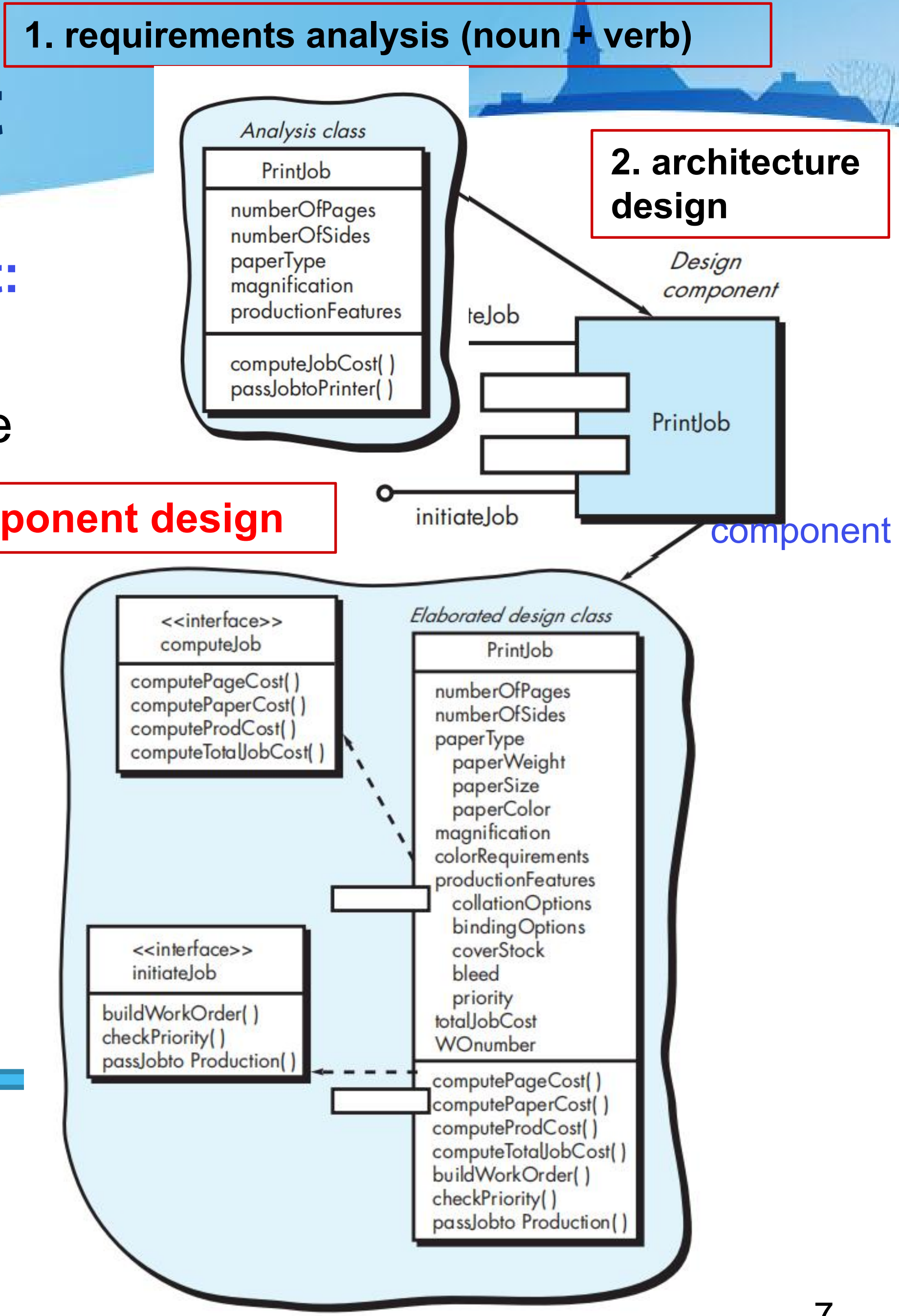
14.1.1 OO Component

Elaboration of a design component:

- Analysis class: problem domain
- Infrastructure class: support service

Each class has been fully elaborated to include all attributes and operations that are relevant to its implementation.

Example: a sophisticated print shop.
1. collect the customer's requirements at the front counter.
2. cost a print job.
3. pass the job on to an automated production facility.

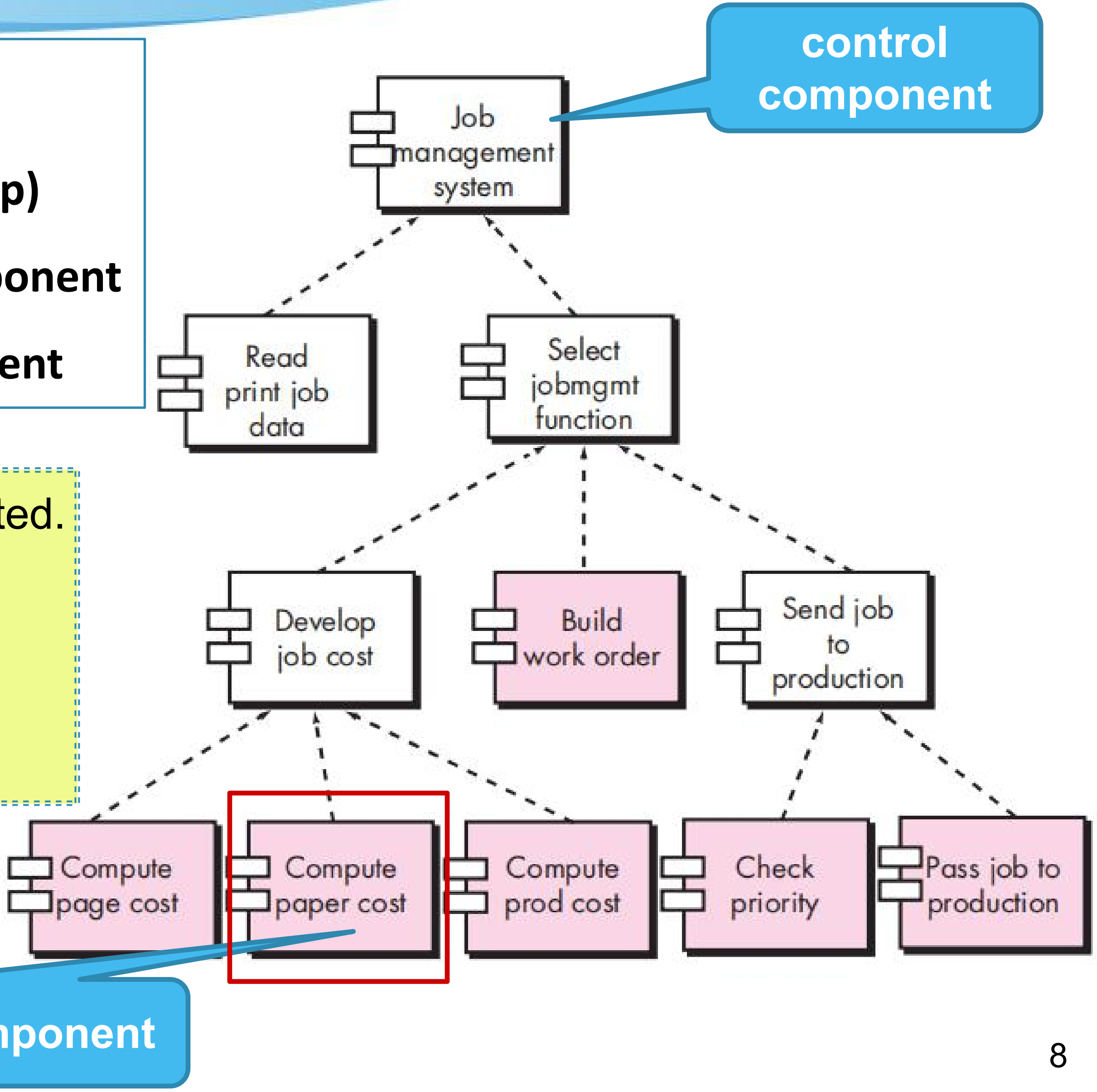


14.1.2 Traditional Component

Hierarchy

- **control component (top)**
- **problem domain component**
- **infrastructure component**

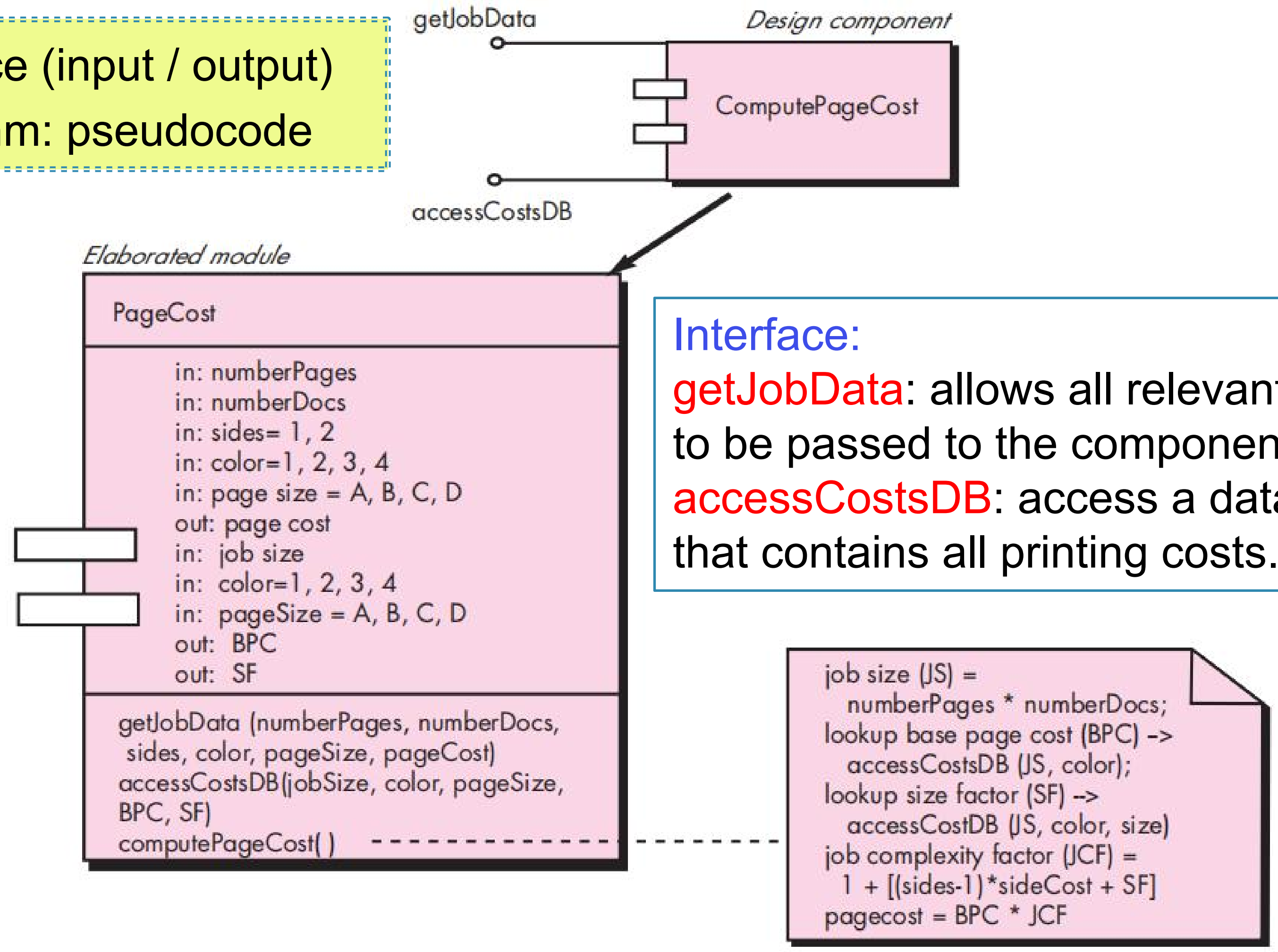
Each module is elaborated.
- data structures
- data or control object
- interface
- algorithm



14.1.2 Traditional Component

ComputePageCost： compute the printing cost per page based on specifications provided by the customer.

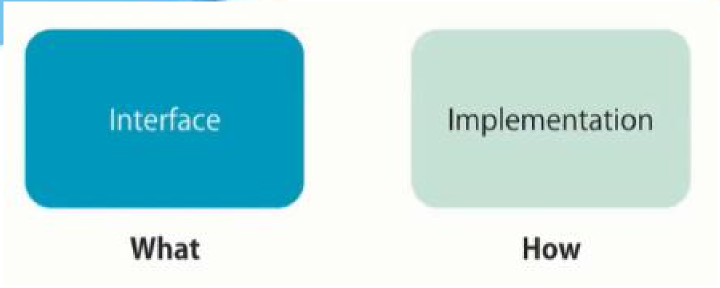
- interface (input / output)
- algorithm: pseudocode



```
job size (JS) =
   numberPages * numberDocs;
lookup base page cost (BPC) -->
   accessCostsDB (JS, color);
lookup size factor (SF) -->
   accessCostDB (JS, color, size)
job complexity factor (JCF) =
  1 + [(sides-1)*sideCost + SF]
pagecost = BPC * JCF
```

Interface:
getJobData: allows all relevant data to be passed to the component.
accessCostsDB: access a database that contains all printing costs.

14.2.1 Class-based component Design



Basic Design Principles for OO

SOLID

1. Single responsibility principle (SPR)
2. The Open-Closed Principle (OCP)
3. The Liskov Substitution Principle (LSP)
4. The Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

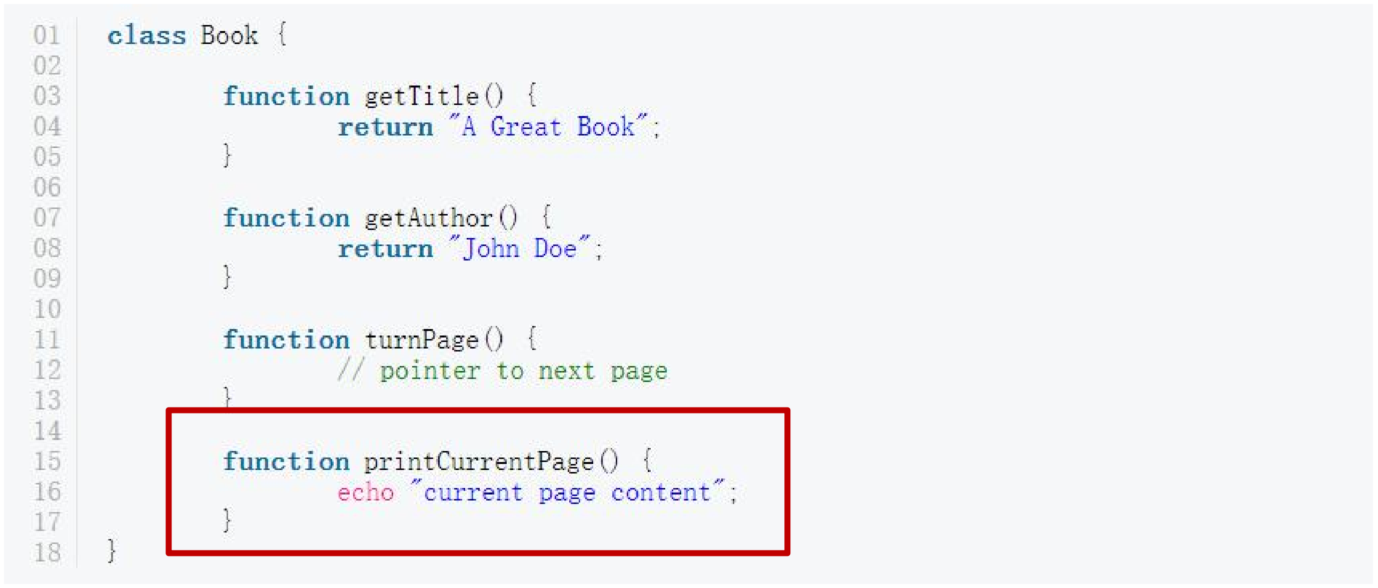
1. The Release Reuse Equivalency Principle (REP)
2. The Common Closure Principle (CCP)
3. The Common Reuse Principle (CRP)

14.2.1 Class-based component: Basic Design Principles

- Single responsibility principle (SPR)

Classes should have a single responsibility and thus only a single reason to change.

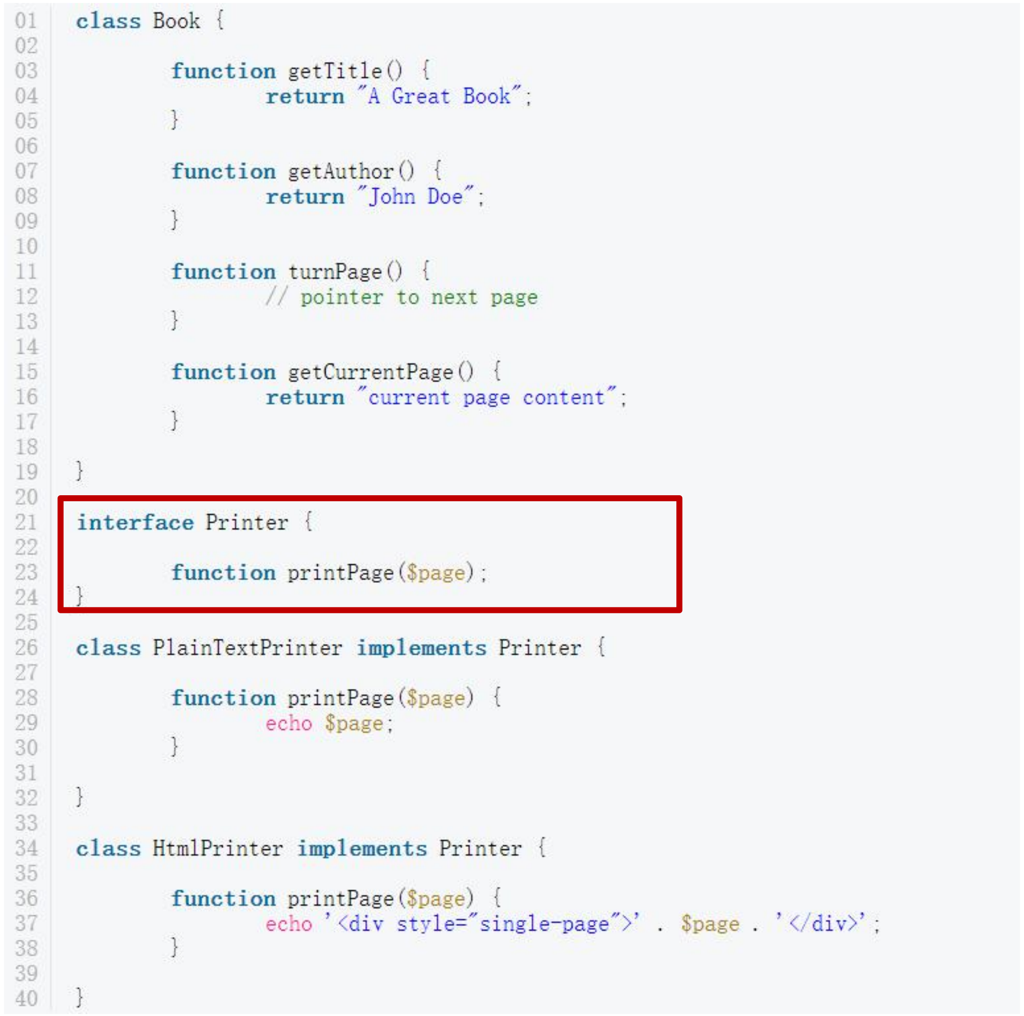
(High cohesion)



```
class Book {

        function getTitle() {
                return "A Great Book";
        }

        function getAuthor() {
                return "John Doe";
        }

        function turnPage() {
                // pointer to next page
        }

        function printCurrentPage() {
                echo "current page content";
        }
}
```

14.2.1 Class-based component: Basic Design Principles

- Single responsibility principle (SPR)

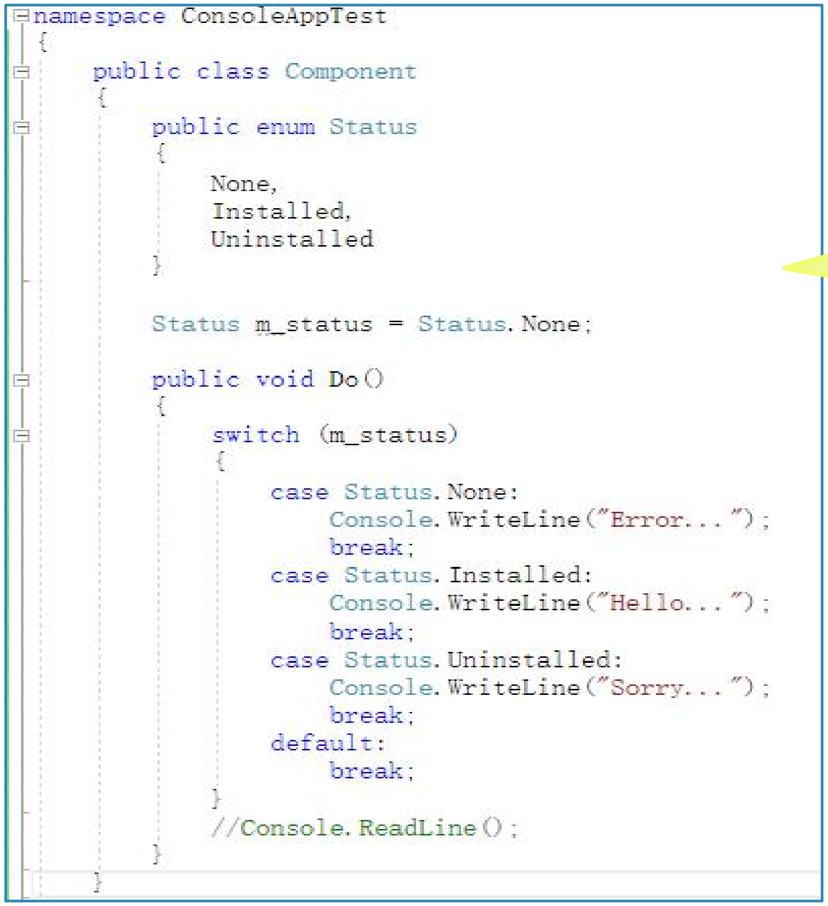


```
class Book {

        function getTitle() {
                return "A Great Book";
        }

        function getAuthor() {
                return "John Doe";
        }

        function turnPage() {
                // pointer to next page
        }

        function getCurrentPage() {
                return "current page content";
        }

}

interface Printer {

        function printPage($page);
}

class PlainTextPrinter implements Printer {

        function printPage($page) {
                echo $page;
        }

}

class HtmlPrinter implements Printer {

        function printPage($page) {
                echo '<div style="single-page">' . $page . '</div>';
        }

}
```

Interface: what
Implement: how

14.2.1 Class-based component: Basic Design Principles

- The Open-Closed Principle (OCP). *“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*



```
namespace ConsoleAppTest
{
    public class Component
    {
        public enum Status
        {
            None,
            Installed,
            Uninstalled
        }

        Status m_status = Status.None;

        public void Do()
        {
            switch (m_status)
            {
                case Status.None:
                    Console.WriteLine("Error...");
                    break;
                case Status.Installed:
                    Console.WriteLine("Hello...");
                    break;
                case Status.Uninstalled:
                    Console.WriteLine("Sorry...");
                    break;
                default:
                    break;
            }
            //Console.ReadLine();
        }
    }
```

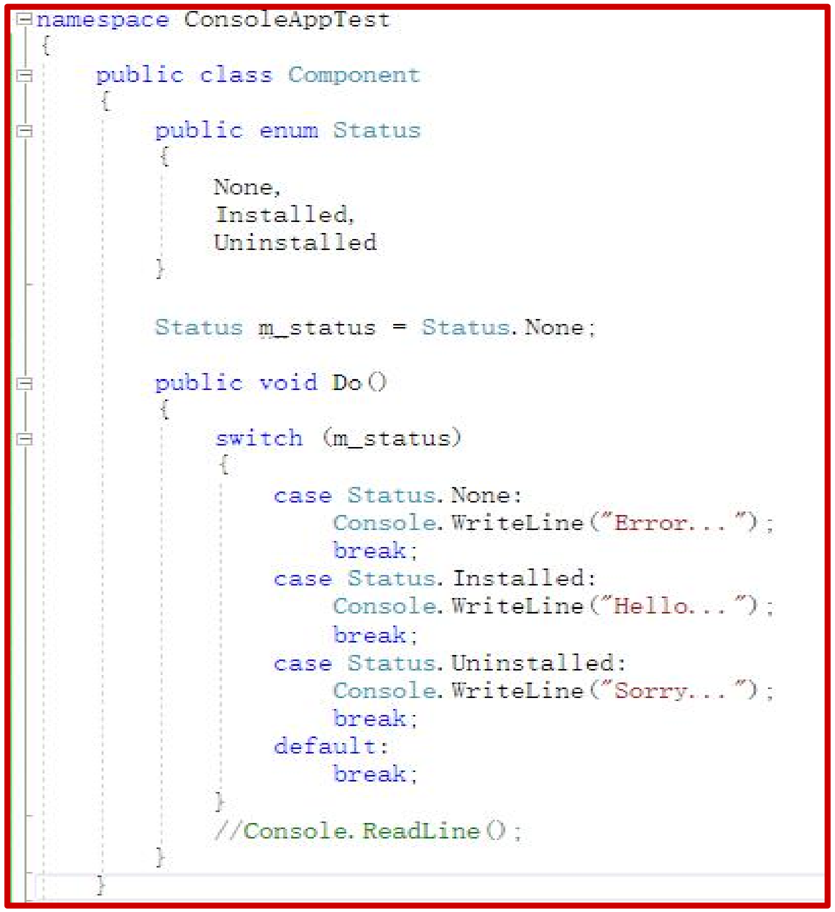
New status: Configured, Blocked?

How to change the code?

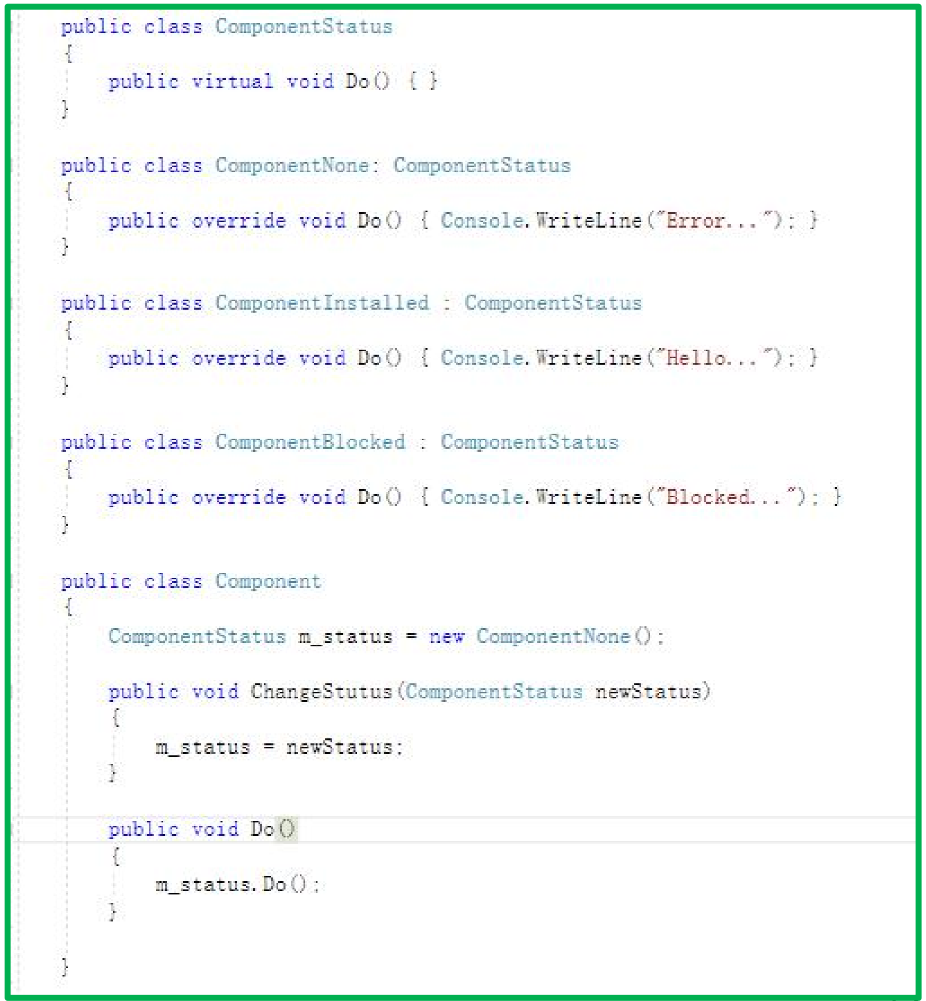
https://code-maze.com/open-closed-principle/

14.2.1 Class-based component: Basic Design Principles

- OCP: *be open for extension，but closed for modification.*



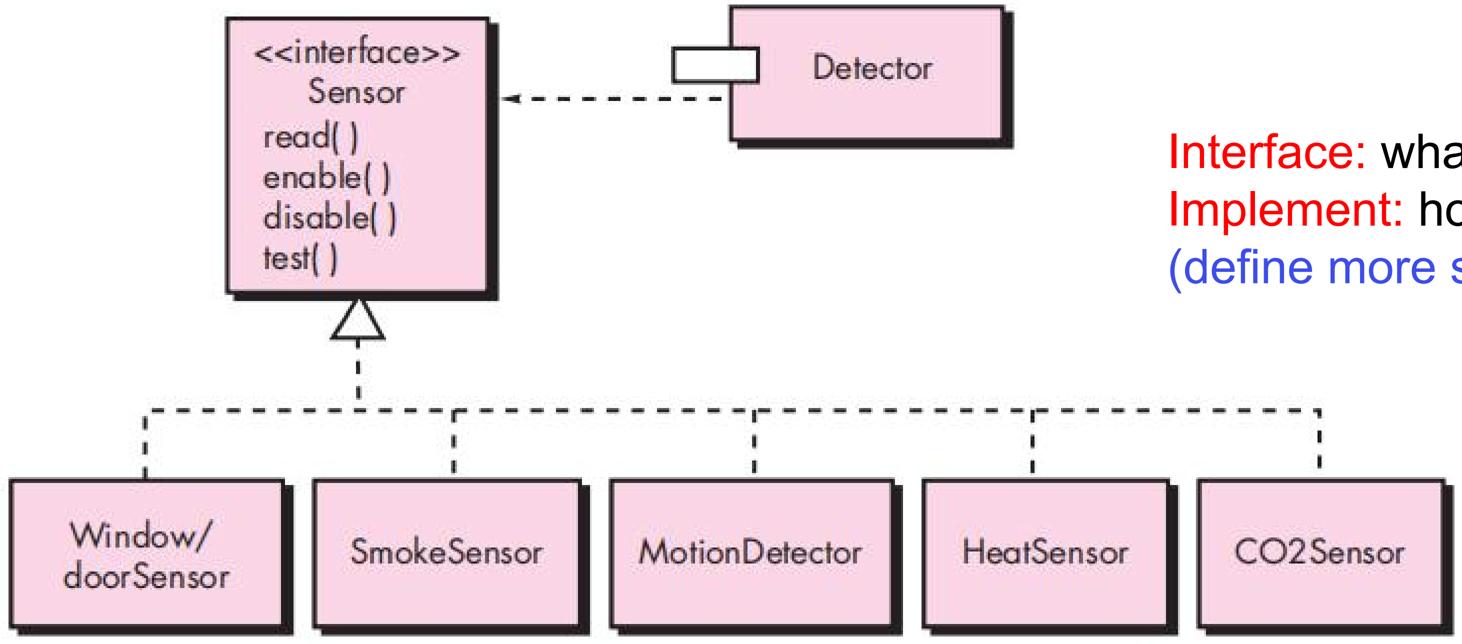
```
namespace ConsoleAppTest
{
    public class Component
    {
        public enum Status
        {
            None,
            Installed,
            Uninstalled
        }

        Status m_status = Status.None;

        public void Do()
        {
            switch (m_status)
            {
                case Status.None:
                    Console.WriteLine("Error...");
                    break;
                case Status.Installed:
                    Console.WriteLine("Hello...");
                    break;
                case Status.Uninstalled:
                    Console.WriteLine("Sorry...");
                    break;
                default:
                    break;
            }
            //Console.ReadLine();
        }
    }
```



```
public class ComponentStatus
{
    public virtual void Do() { }
}

public class ComponentNone: ComponentStatus
{
    public override void Do() { Console.WriteLine("Error..."); }
}

public class ComponentInstalled : ComponentStatus
{
    public override void Do() { Console.WriteLine("Hello..."); }
}

public class ComponentBlocked : ComponentStatus
{
    public override void Do() { Console.WriteLine("Blocked..."); }
}

public class Component
{
    ComponentStatus m_status = new ComponentNone();

    public void ChangeStutus(ComponentStatus newStatus)
    {
        m_status = newStatus;
    }

    public void Do()
    {
        m_status.Do();
    }

}
```

14.2.1 Class-based component: Basic Design Principles

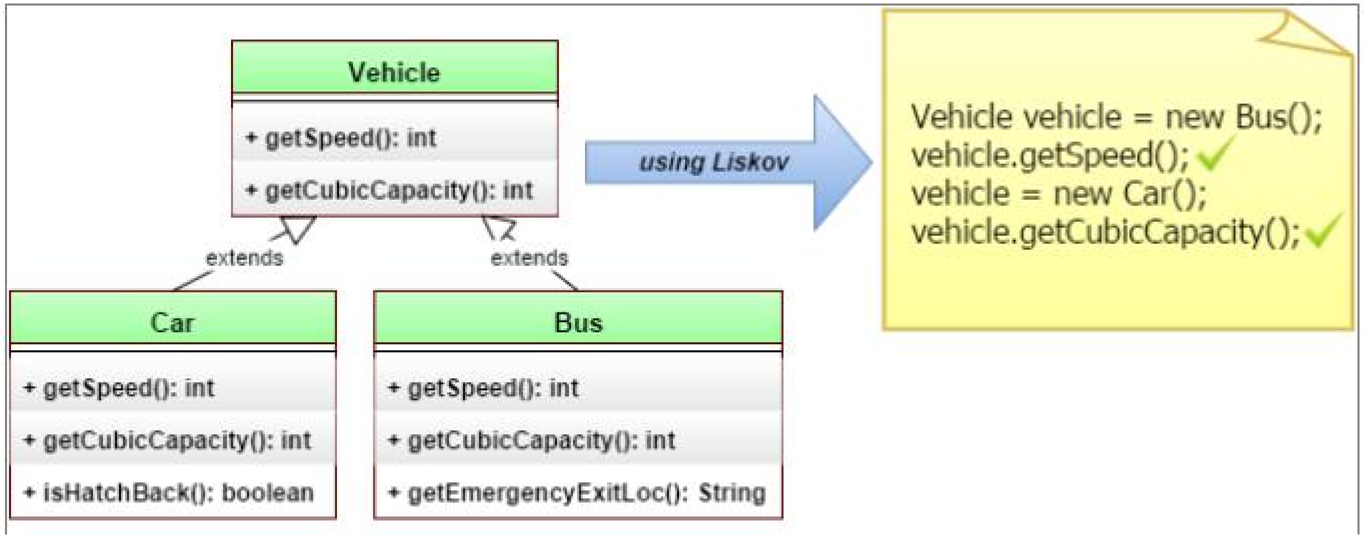
- *be open for extension，but closed for modification.*



Interface: what
Implement: how
(define more sensors)

14.2.1 Class-based component: Basic Design Principles

- The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes.*

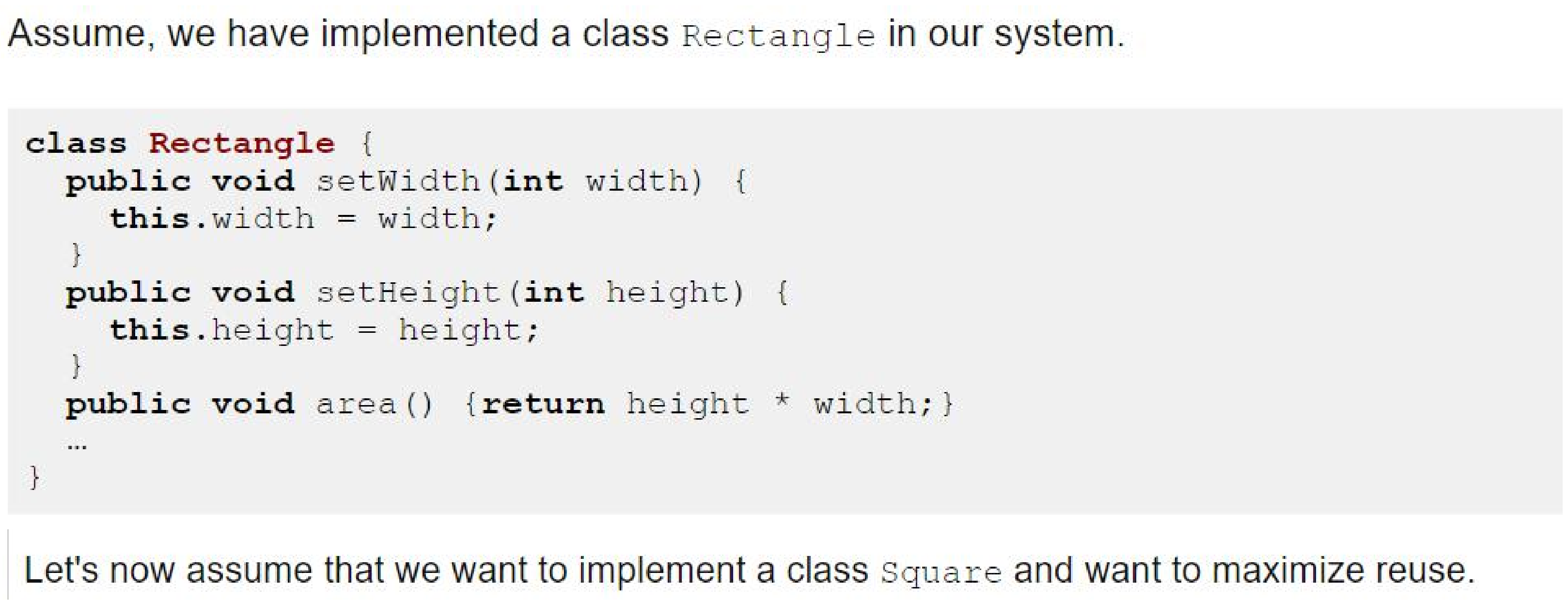


```
Vehicle vehicle = new Bus();
vehicle.getSpeed(); ✓
vehicle = new Car();
vehicle.getCubicCapacity(); ✓
```

subtype objects can replace super type objects without affecting the functionality inherent in the super type.

14.2.1 Class-based component: Basic Design Principles

- The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes.*



Assume, we have implemented a class `Rectangle` in our system.

```
class Rectangle {
  public void setWidth(int width) {
    this.width = width;
  }
  public void setHeight(int height) {
    this.height = height;
  }
  public void area() {return height * width;}
  …
}
```

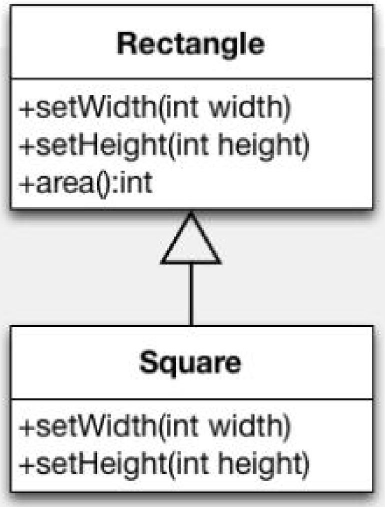
Let's now assume that we want to implement a class `Square` and want to maximize reuse.

14.2.1 Class-based component: Basic Design Principles

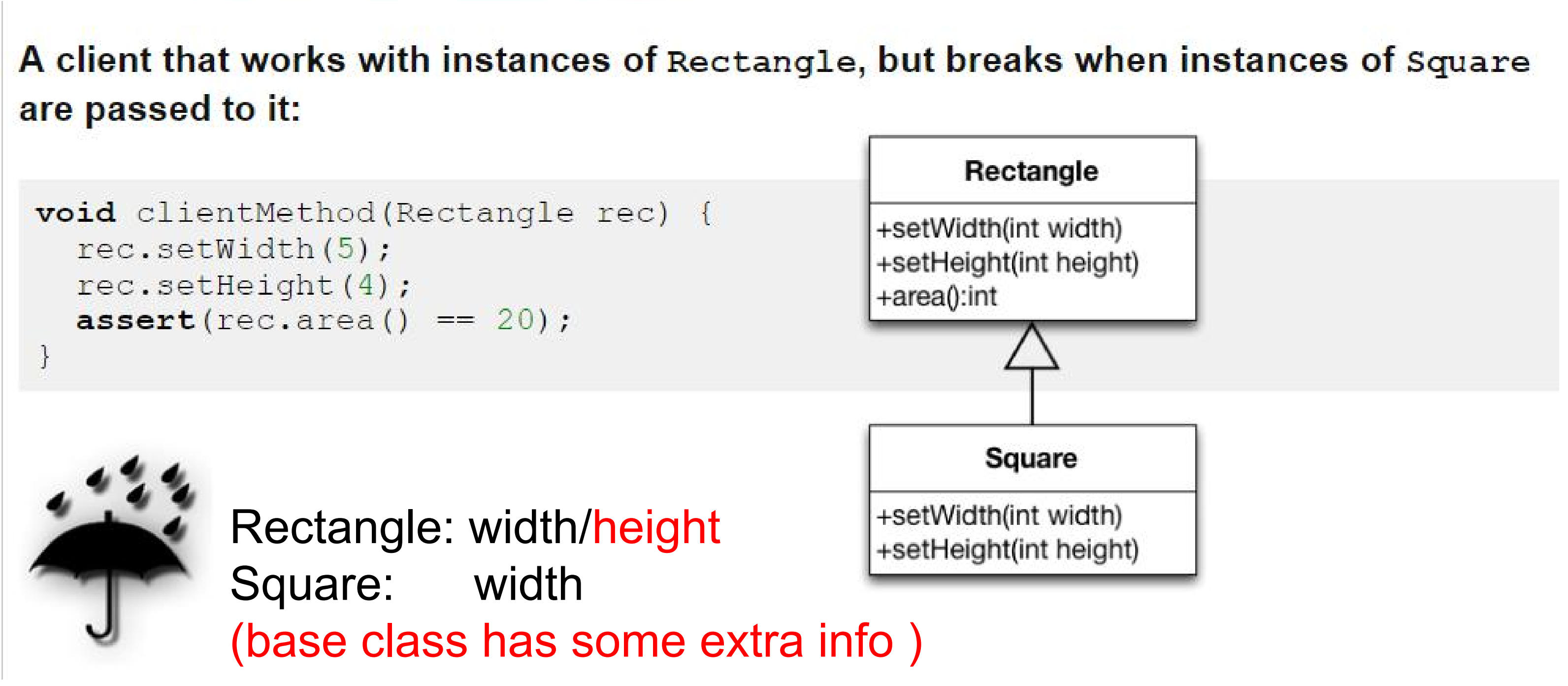
- The Liskov Substitution Principle (LSP). *“Subclasses should be substitutable for their base classes.*

Implementing `Square` as a subclass of `Rectangle`:

```
class Square extends Rectangle {
  public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
  }
  public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
  }
  ...
}
```



14.2.1 Class-based component: Basic Design Principles



A client that works with instances of `Rectangle`, but breaks when instances of `Square` are passed to it:

```
void clientMethod(Rectangle rec) {
  rec.setWidth(5);
  rec.setHeight(4);
  assert(rec.area() == 20);
}
```

Rectangle: width/height
Square: width
(base class has some extra info)

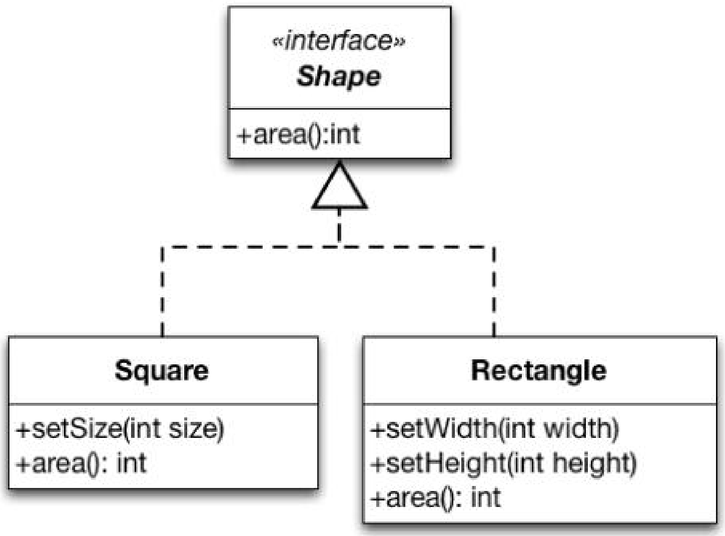
The clientMethod method makes an assumption that is true for Rectangle: setting the width respectively height has no effect on the other attribute. This assumption does not hold for Square.

The Rectangle/Square hierarchy violates the Liskov Substitution Principle (LSP)! Square is behaviorally not a correct substitution for Rectangle.

14.2.1 Class-based component: Basic Design Principles

- The Liskov Substitution Principle (LSP). *“Subclasses should be substitutable for their base classes.*

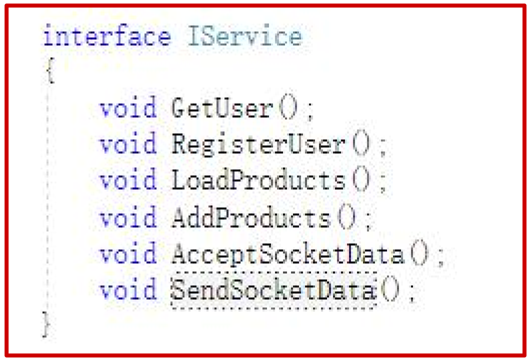
Rectangles and Square - LSP Compliant Solution



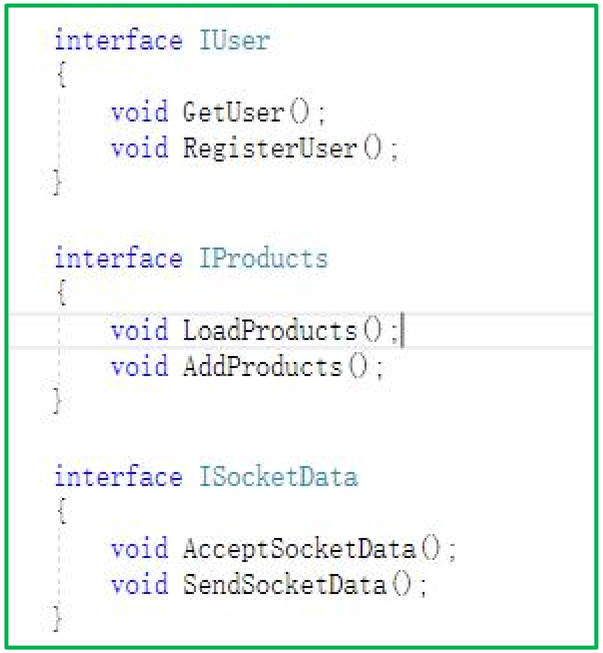
subtype objects can replace super type objects without affecting the functionality inherent in the super type.

14.2.1 Class-based component: Basic Design Principles

- The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface.*



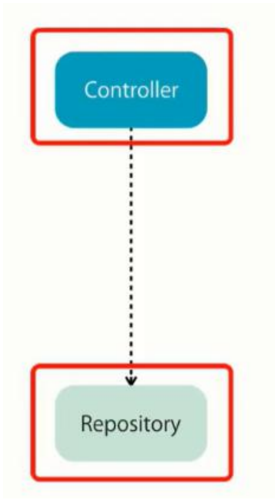
```
interface IService
{
    void GetUser();
    void RegisterUser();
    void LoadProducts();
    void AddProducts();
    void AcceptSocketData();
    void SendSocketData();
}
```

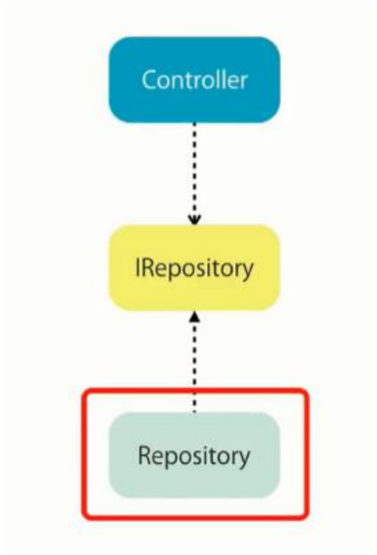


```
interface IUser
{
    void GetUser();
    void RegisterUser();
}

interface IProducts
{
    void LoadProducts();
    void AddProducts();
}

interface ISocketData
{
    void AcceptSocketData();
    void SendSocketData();
}
```

14.2.1 Class-based component: Basic Design Principles

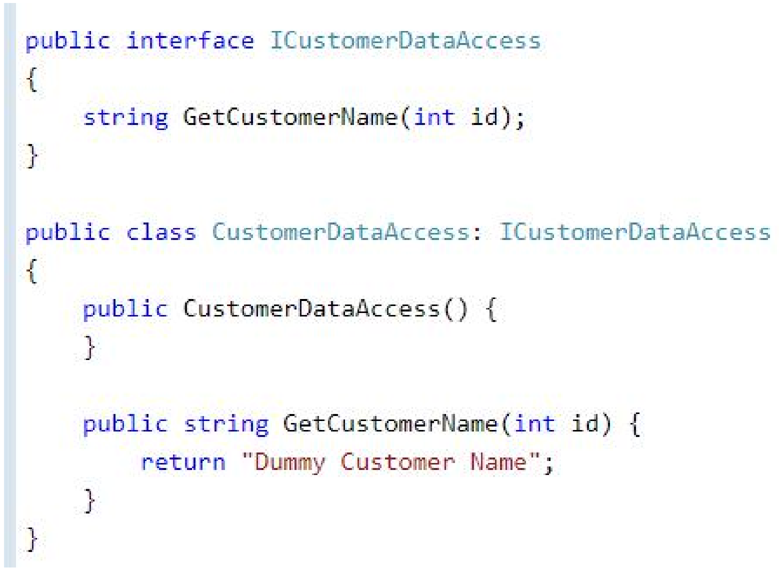
- Dependency Inversion Principle (DIP).
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.



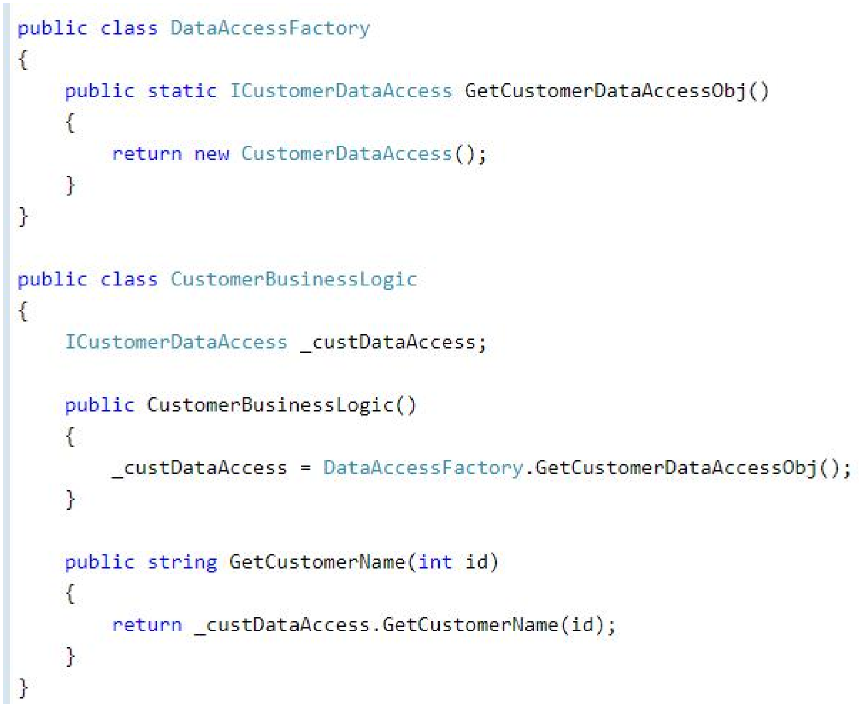


14.2.1 Class-based component: Basic Design Principles

- Dependency Inversion Principle (DIP).
  - High-level modules (CustomerBusinessLogic) should not depend on low-level modules (CustomerDataAccess). Both should depend on abstractions.
  - Abstractions (ICustomerDataAccess) should not depend on details (CustomerDataAccess) . Details should depend on abstractions.



```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess() {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}
```



```
public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}

public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

https://www.tutorialsteacher.com/ioc/dependency-inversion-principle

14.2.1 Class-based component: Basic Design Principles

- The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release."*

  (release control module)

- The Common Closure Principle (CCP). *"Classes that change together belong together."*

- The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together."*

14.2.1 Class-based component: Basic Design Principles

Please give an example for each OO Design principle:

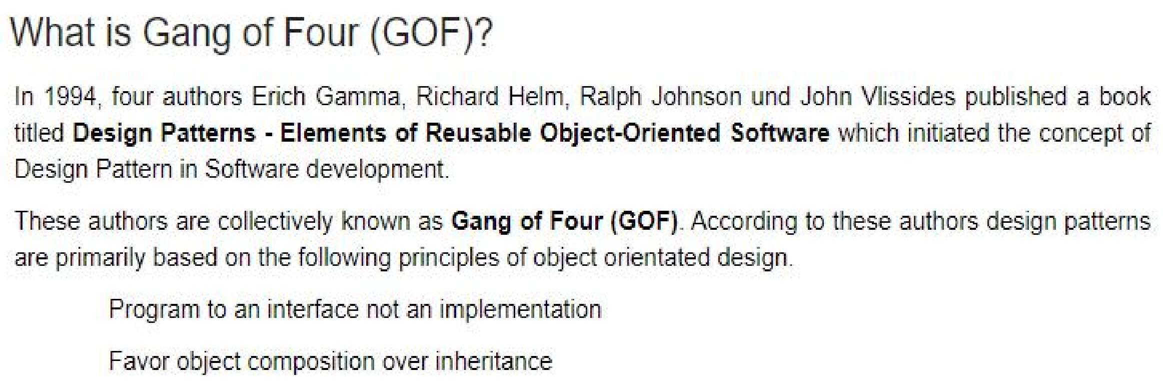
- TheSingle responsibility principle (SPR)
- The Open-Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Dependency Inversion Principle (DIP)
- The Interface Segregation Principle (ISP)

- The Release Reuse Equivalency Principle (REP).
- The Common Closure Principle (CCP).
- The Common Reuse Principle (CRP).

14.2.2 OO: Design Guidelines

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces
  - Interfaces provide important information about communication and collaboration
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

14.2.1 Design Pattern for Java (23 patterns)

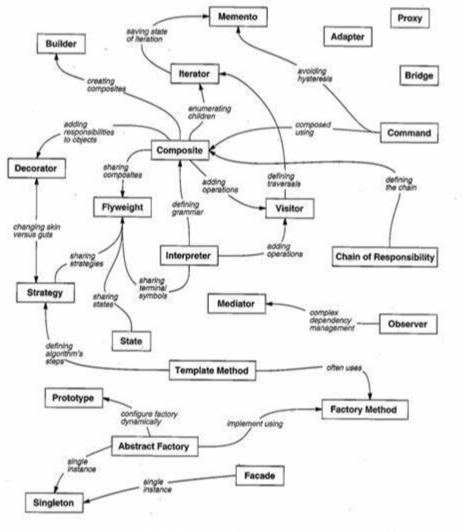


What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance



Design Pattern Relationships

See the interesting discussion in Wikipedia (2019):

"Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code."

https://www.tutorialspoint.com/design_pattern/design_pattern_quick_guide.htm

Review

- How to do component design?
  - OO Component **Basic Design Principles for OO**

Please give an example for each OO principle(SOLID):

- TheSingle responsibility principle (SPR)
- The Open-Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Dependency Inversion Principle (DIP)
- The Interface Segregation Principle (ISP)

- The Release Reuse Equivalency Principle (REP).
- The Common Closure Principle (CCP).
- The Common Reuse Principle (CRP).

Take a break

Ten minutes

投票　最多可选1项　　设置

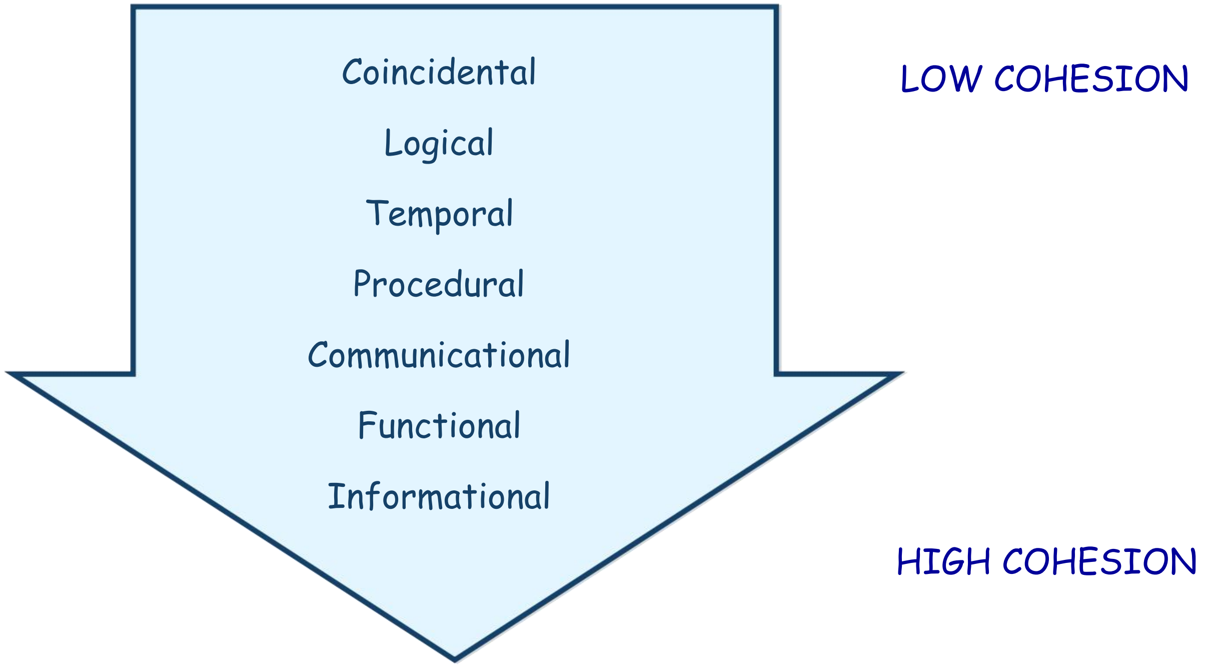
How many lines of code have you written so far?

- A　<= 1,000L
- B　1,000L – 10,000L
- C　10,000L – 50,000 L
- D　>= 50,000 L

提交

14.2.3 Cohesion

- Conventional view:
  - the “single-mindedness” of a module
- OO view:
  - *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself



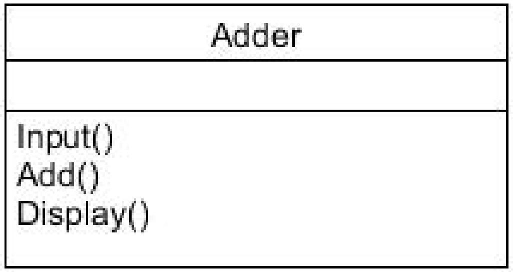
14.2.3 Cohesion

- Coincidental (worst degree)
  - Parts are unrelated to one another
- Logical
  - Parts are related only by the logic structure of code
- Temporal
  - Module’s data and functions related because they are used at the same time in an execution
- Procedural
  - Similar to temporal, and functions pertain to some related action or purpose

14.2.3 Cohesion

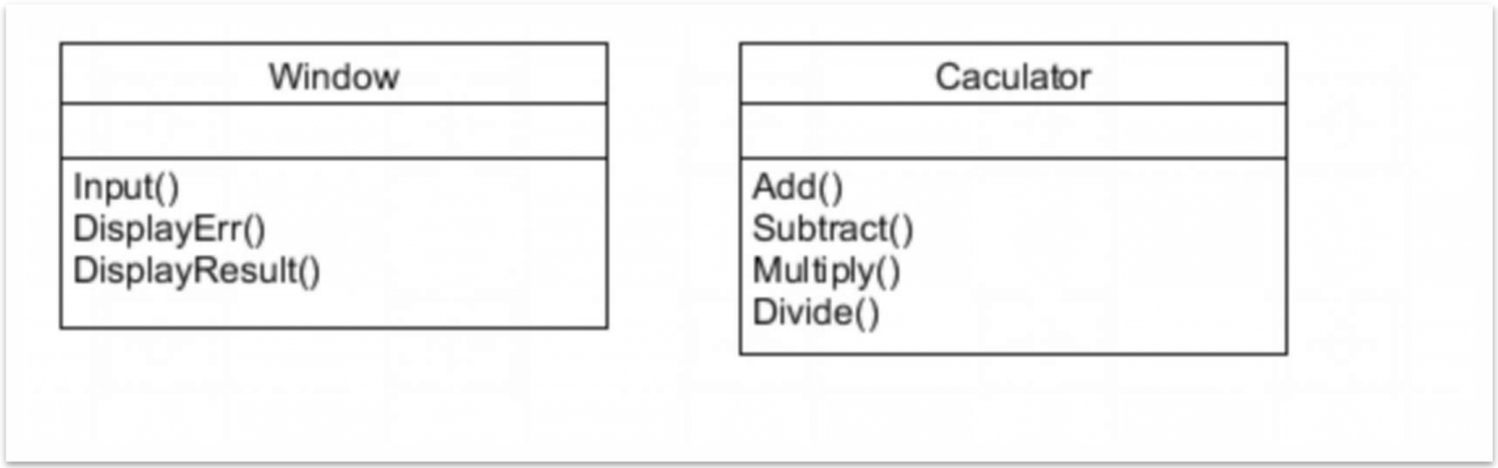
- Communication
  - Operates on the same data set
- Functional (ideal degree)
  - All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function
- Informational
  - Adaption of functional cohesion to data abstraction and object-based design

Example



Temporal

Low cohesion



Functional

High cohesion

Question

The following statements describe modules in a program. For each, decide whether the module is likely to have a high or low degree of cohesion. If cohesion is low, explain why.

a. Module “InventorySearchByID” searches the records in inventory to see if any match the specified range of ID numbers. A data structure is returned containing any matching records.

b. Module “ProcessPurchase” removes the purchased product from inventory, prints a receipt for the customer and updates the log.

c. Module “FindSet” processes the user's request, determines the set of items from inventory that match the request, and formats the items into a list that can be shown to the customer.

Answer

a. Module “InventorySearchByID” can be expected to have high cohesion; it performs only one type of functionality (a search).

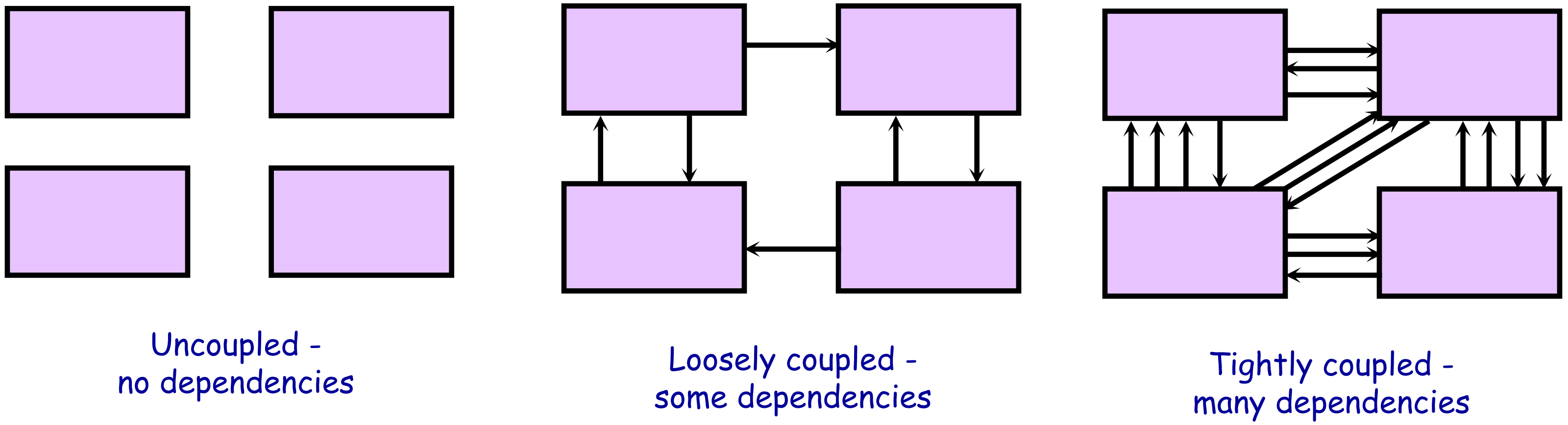
b. Module “ProcessPurchase” can be expected to have relatively low cohesion, since it involves very different functionalities: printing a receipt for use by the user is logically quite different from updating a data store.

c. Module “FindSet”can be expected to have relatively low cohesion, since it invokes very different functionalities: parsing input, a search through data, and output formatting.

14.2.4 Coupling

Low coupling

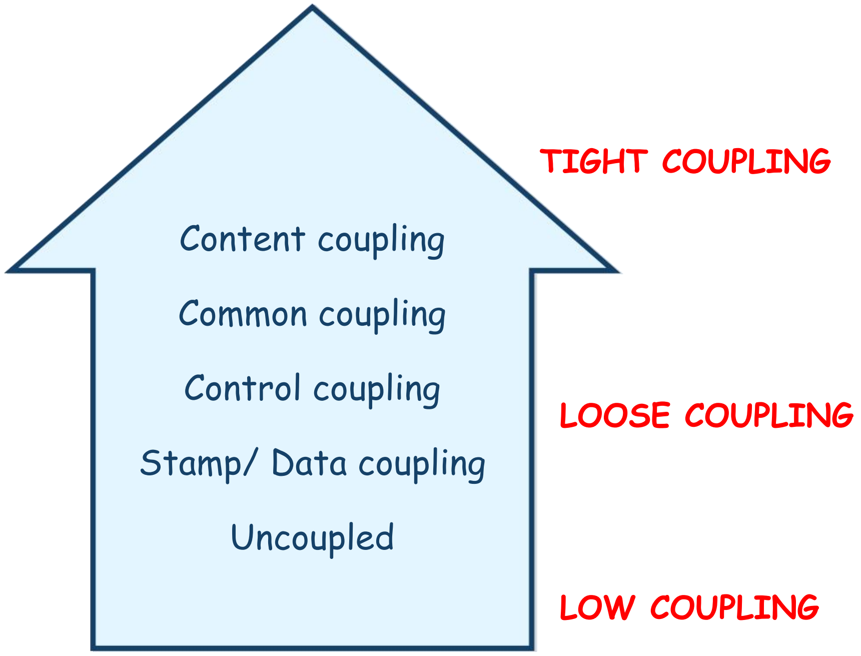
- Conventional view:
  - The degree to which a component is connected to other components and to the external world
- OO view:
  - a qualitative measure of the degree to which classes are connected to one another



14.2.4 Coupling

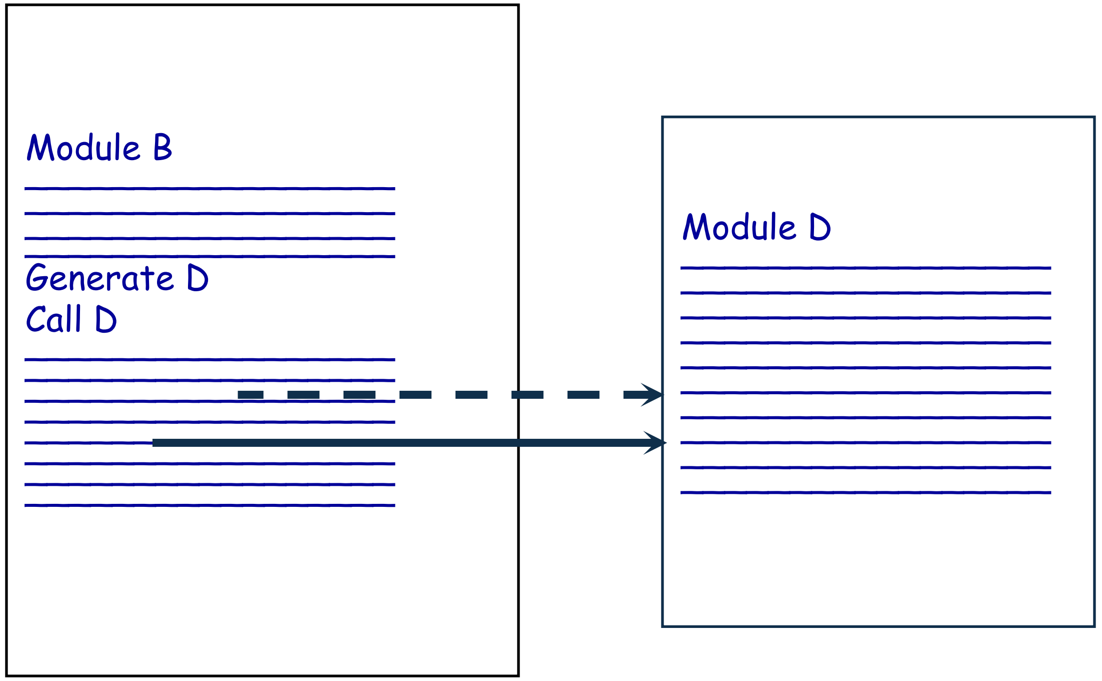
- Content coupling
- Common coupling
- Control coupling
- Stamp/Data coupling

Low coupling



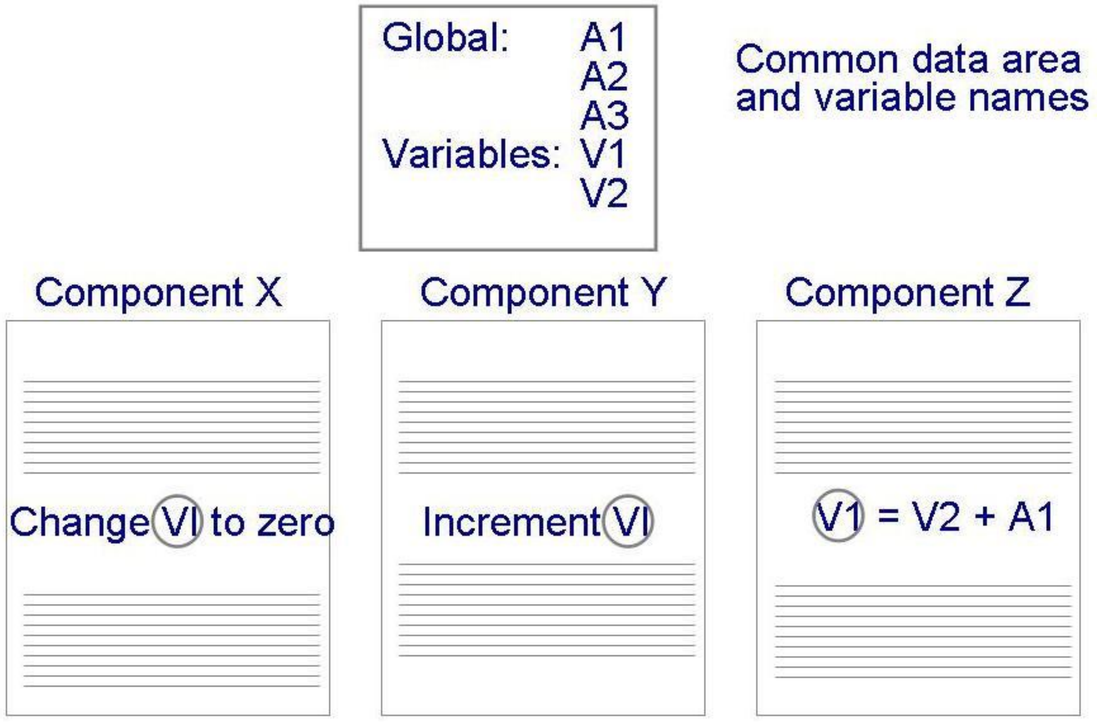
14.2.4 Coupling： Content Coupling

- The least desirable case： One module actually modifies another (one component modifies an internal data item in another component, or when one component branches into the middle of another component)



14.2.4 Coupling: Common Coupling

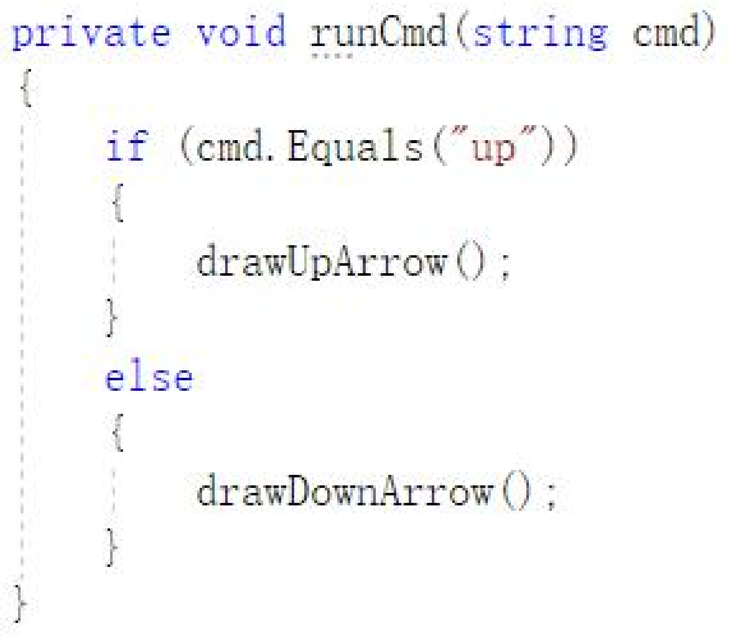
- Making a change to the common data means tracing back to all components that access those data to evaluate the effect of the change



14.2.4 Coupling: Control Coupling

- When one module passes parameters or a return code to control the behavior of another module

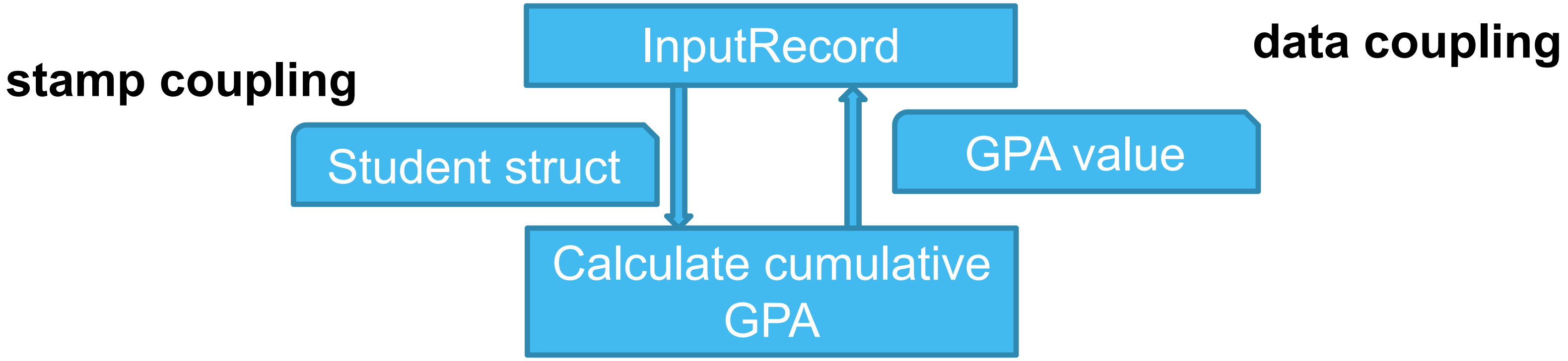
Flag or command



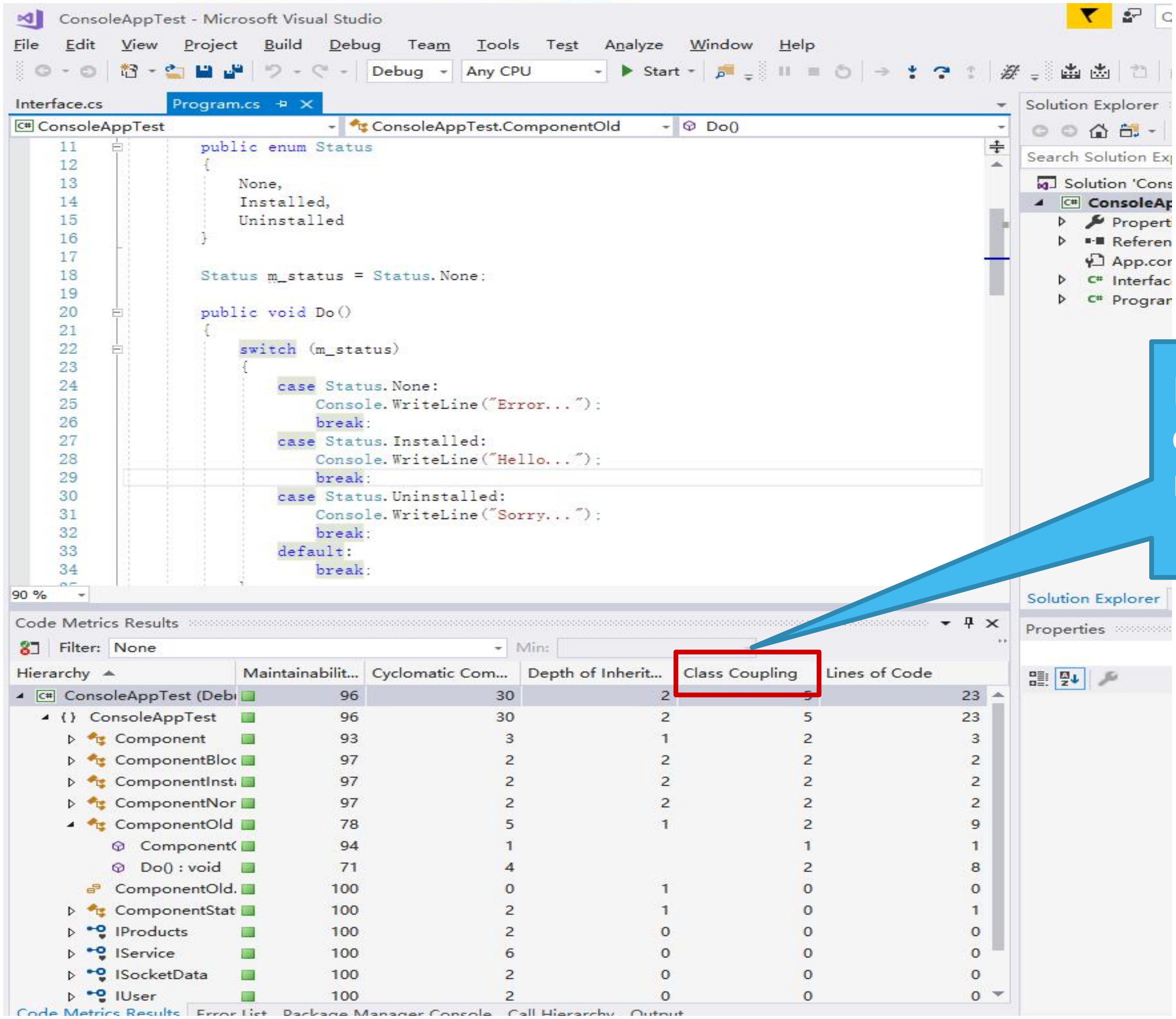
```
private void runCmd(string cmd)
{
    if (cmd.Equals("up"))
    {
        drawUpArrow();
    }
    else
    {
        drawDownArrow();
    }
}
```

14.2.4 Coupling: Stamp and Data Coupling

- **Stamp coupling** occurs when complex data structures are passed between modules
  - Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization
- If only data values, and not structured data, are passed, then the modules are connected by **data coupling**
  - Data coupling is simpler and less likely to be affected by changes in data representation



14.2.4 Coupling for class


ConsoleAppTest - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug Any CPU Start
Interface.cs Program.cs
ConsoleAppTest ConsoleAppTest.ComponentOld Do()
Solution Explorer
Search Solution Exp
Solution 'Cons
ConsoleAp
Propert
Referen
App.cor
Interfac
Prograr
90 %
Code Metrics Results
Filter: None Min:
Solution Explorer
Properties
Code Metrics Results Error List Package Manager Console Call Hierarchy Output

```
        public enum Status
        {
            None,
            Installed,
            Uninstalled
        }

        Status m_status = Status.None;

        public void Do()
        {
            switch (m_status)
            {
                case Status.None:
                    Console.WriteLine("Error...");
                    break;
                case Status.Installed:
                    Console.WriteLine("Hello...");
                    break;
                case Status.Uninstalled:
                    Console.WriteLine("Sorry...");
                    break;
                default:
                    break;
```

Hierarchy	Maintainabilit...	Cyclomatic Com...	Depth of Inherit...	Class Coupling	Lines of Code
ConsoleAppTest (Debu	🟩 96	30	2	5	23
{ } ConsoleAppTest	🟩 96	30	2	5	23
Component	🟩 93	3	1	2	3
ComponentBloc	🟩 97	2	2	2	2
ComponentInst	🟩 97	2	2	2	2
ComponentNor	🟩 97	2	2	2	2
ComponentOld	🟩 78	5	1	2	9
Component(	🟩 94	1		1	1
Do() : void	🟩 71	4		2	8
ComponentOld.	🟩 100	0	1	0	0
ComponentStat	🟩 100	2	1	0	1
IProducts	🟩 100	2	0	0	0
IService	🟩 100	6	0	0	0
ISocketData	🟩 100	2	0	0	0
IUser	🟩 100	2	0	0	0

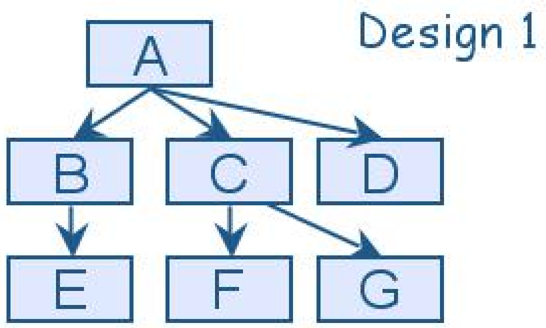
Measures number of classes that are referenced. Lower value are better.

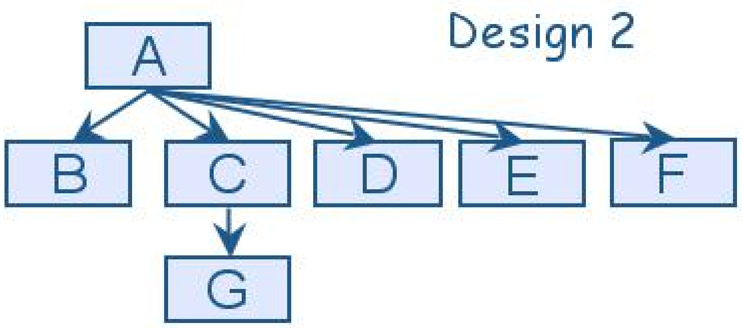
14.2.4 Complexity

Complexity(Henry) :   C = (FanIn × FanOut)* 2

FanIn: The number of its immediately superordinate (parent) modules. Higher value indicates high reusability.

FanOut: The number of its immediately subordinate modules. Higher value indicates more complex. But too small (such as alwasy 1) is also not good.





Design1: A: fan-in: 0 (A is called by nothing)
fan-out: 3 (A call B,C,D)

Design2: A/G:
fan-in: ?   fan-out: ?

Aim: high fan-in at the lower levels of the hierarchy.

optimum fan-out: 7±2

https://it.toolbox.com/blogs/craigborysowich/design-principles-fan-in-vs-fan-out-050407

14.2.4 Complexity

McCabe's cyclomatic complexity （McCabe 1976）

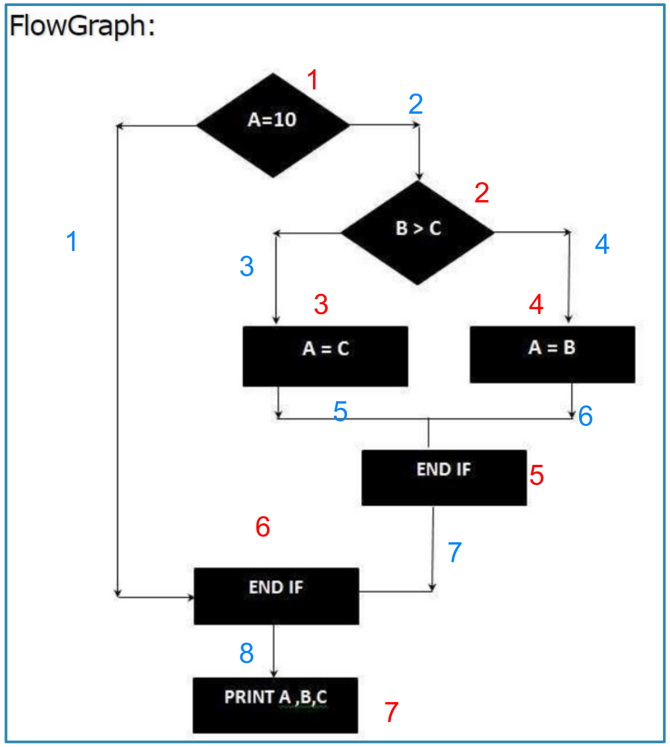
Guide: V(G) <= 10

M = e – n + 2p

e = number of edges of the graph
n = number of nodes of the graph
p = number of disconnected parts
of the flow graph

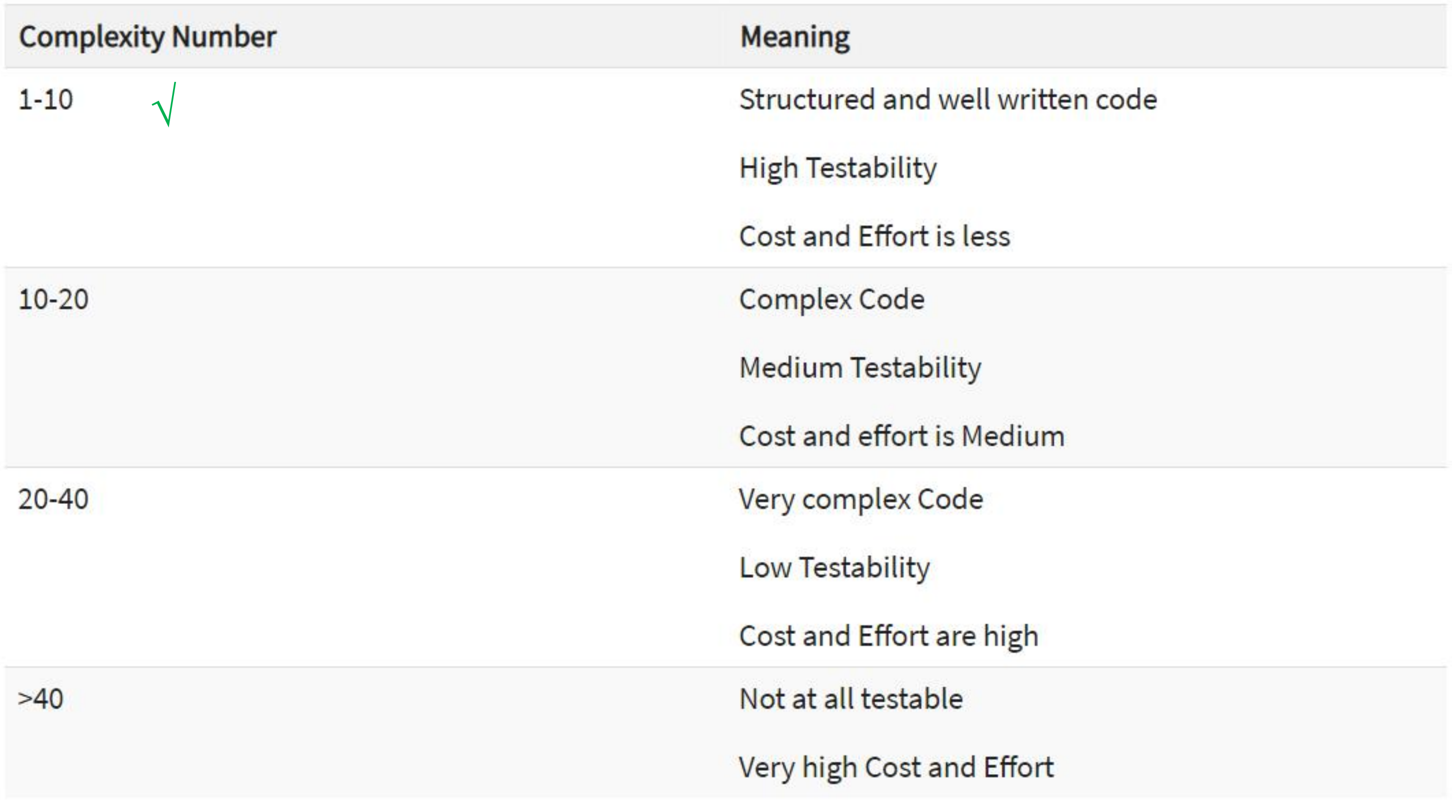
for one function：
M = E − N + 2

```
IF A = 10 THEN
 IF B > C THEN
   A = B
 ELSE
   A = C
 ENDIF
ENDIF
Print A
Print B
Print C
```



V(G) = E − N + 2 = 8 − 7 + 2*1 = 3

14.2.4 Complexity



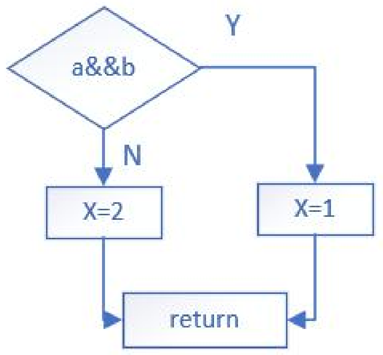
Complexity Number	Meaning
1-10 √	Structured and well written code High Testability Cost and Effort is less
10-20	Complex Code Medium Testability Cost and effort is Medium
20-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

https://www.guru99.com/cyclomatic-complexity.html

14.2.4 Complexity

Please calculate McCabe’s cyclomatic complexity :

```
void foo(void)
{
    if (a && b)
        x=1;
    else
        x=2;
}
```



V(G) = 4− 4 + 2*1 = 2

Could you write an automated tool to calculate McCabe’s cyclomatic complexity?

Take a break

Five minutes

14.3 Component-Level Design

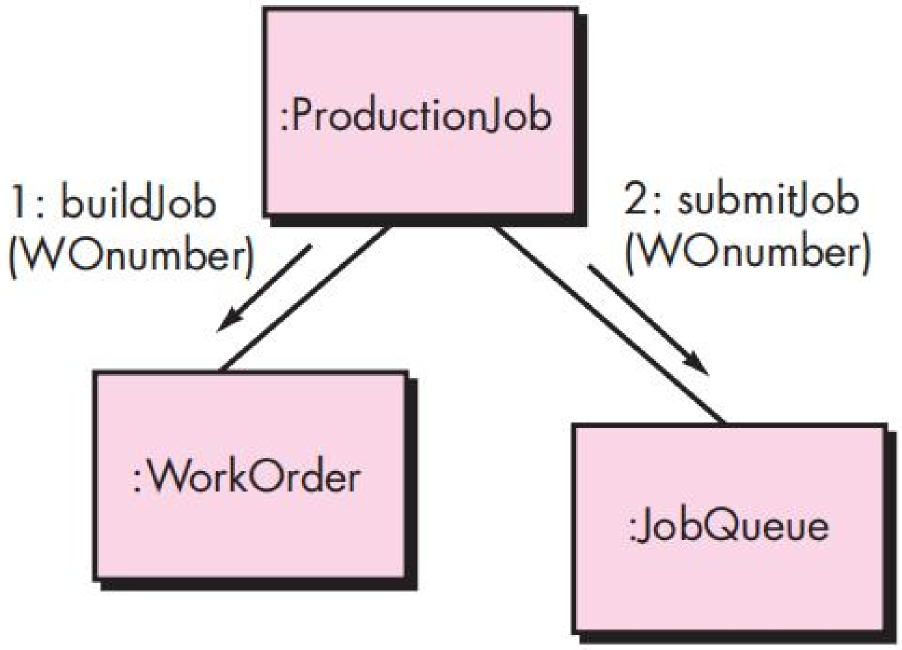
- **Step 1**. Identify all design classes that correspond to the problem domain. (elaborate analysis class and architectural component)

- **Step 2**. Identify all design classes that correspond to the infrastructure domain (GUI, OS, object and data management).

- **Step 3**. Elaborate all design classes that are not acquired as reusable components (consider component cohesion and coupling).

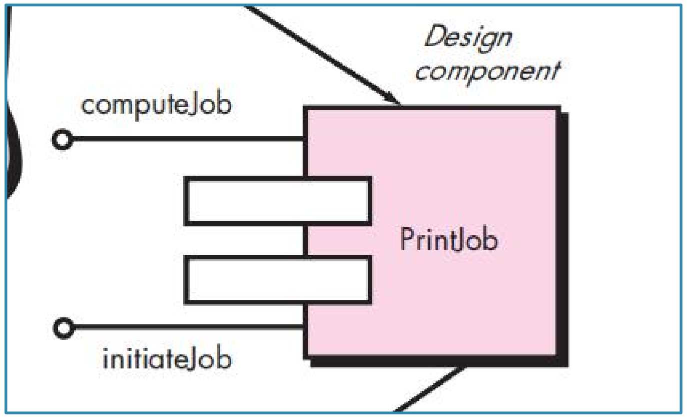
14.3 Component-Level Design

- **Step 3a**. Specify message details when classes or component collaborate.

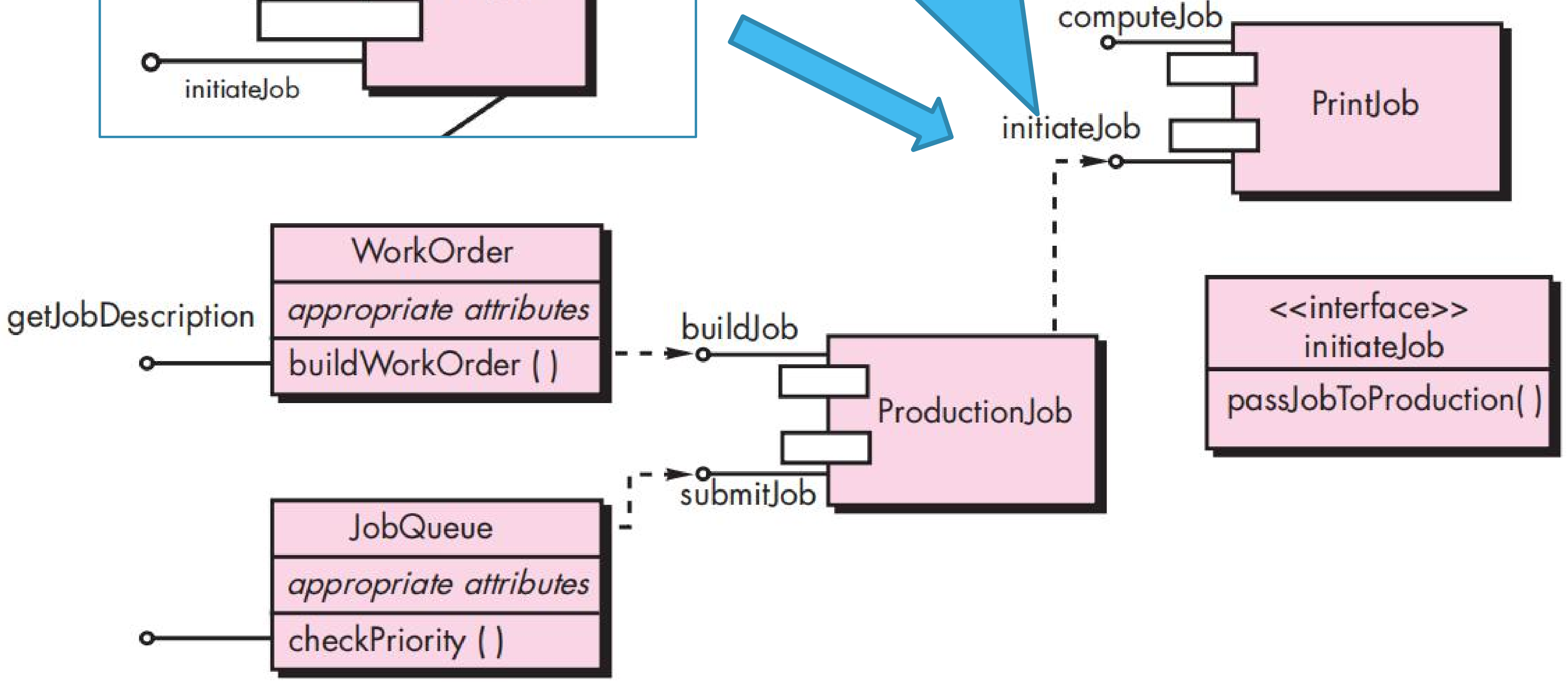


14.3 Component-Level Design

Step 3b. Identify appropriate interfaces for each component.

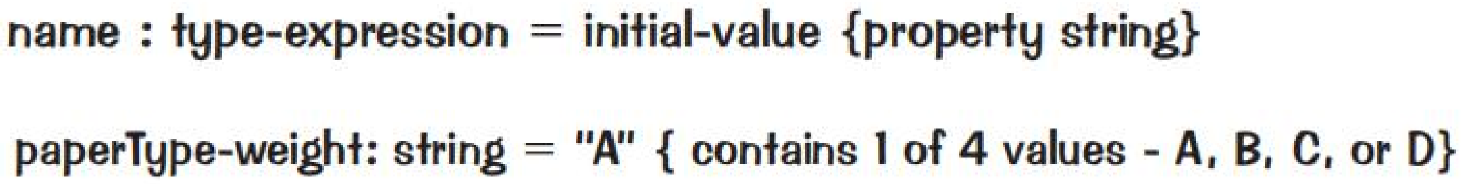


building a work order,
checking job priority,
passing a job to production



14.3 Component-Level Design

- **Step 3c.** Elaborate attributes and define data types and data structures required to implement them.

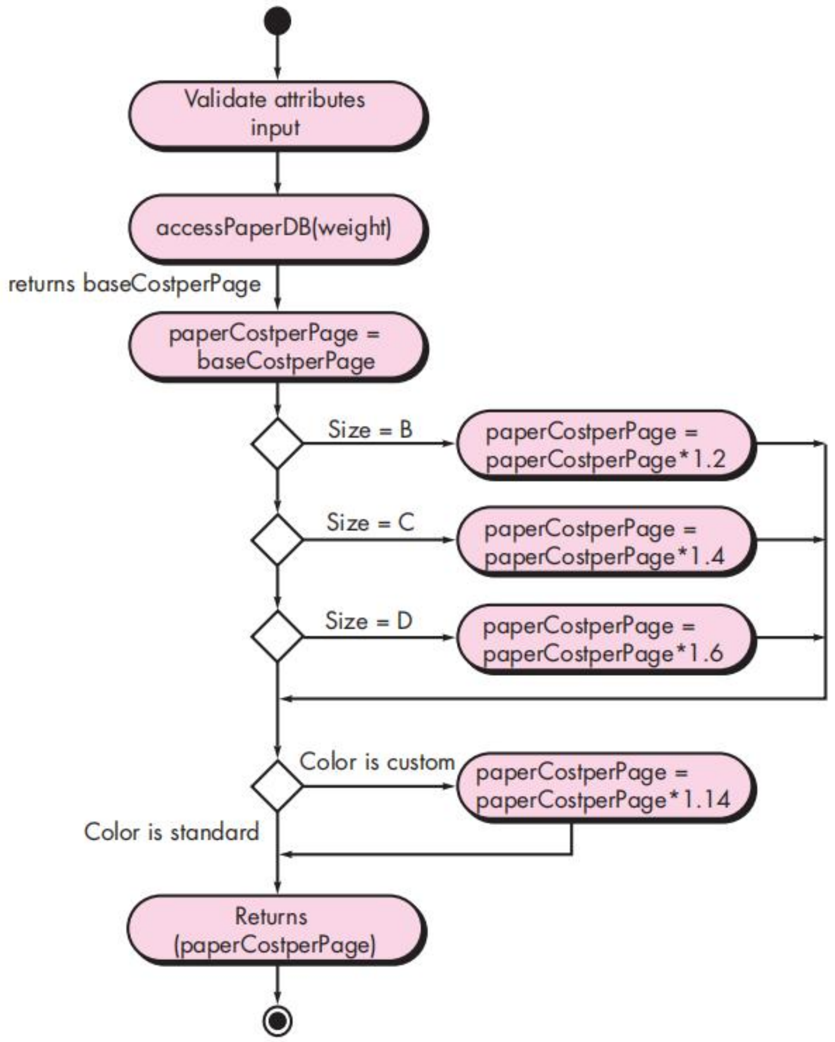


name : type-expression = initial-value {property string}

paperType-weight: string = "A" { contains 1 of 4 values - A, B, C, or D}

14.3 Component-Level Design

- **Step 3d.** **Describe processing flow within each operation in detail.**

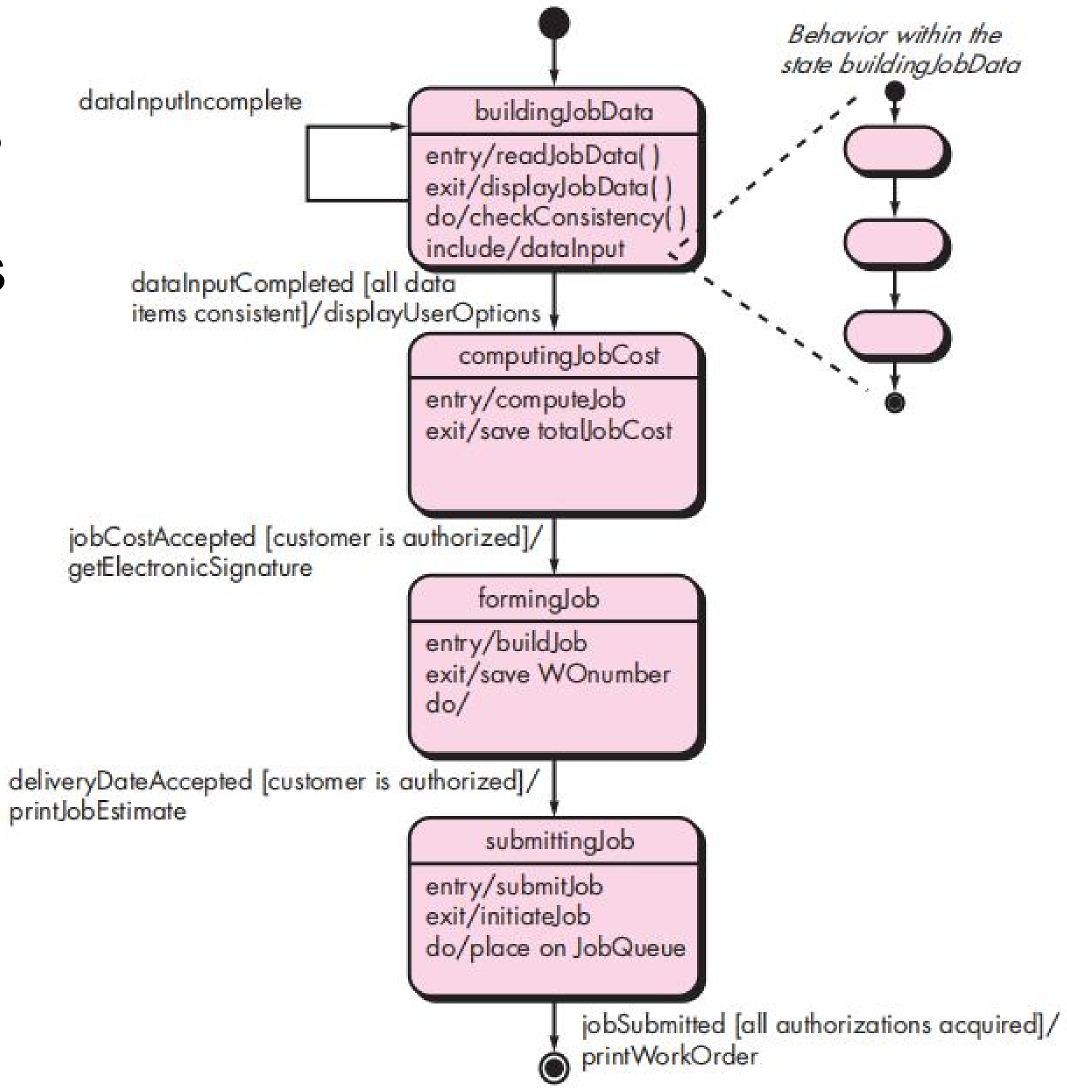


Validate attributes input
accessPaperDB(weight)
returns baseCostperPage
paperCostperPage = baseCostperPage
Size = B
paperCostperPage = paperCostperPage*1.2
Size = C
paperCostperPage = paperCostperPage*1.4
Size = D
paperCostperPage = paperCostperPage*1.6
Color is custom
paperCostperPage = paperCostperPage*1.14
Color is standard
Returns (paperCostperPage)

14.3 Component-Level Design

- **Step 4.** Describe persistent data sources (databases and files) and identify the classes required to manage them.

- **Step 5.** Develop and elaborate behavioral representations for a class or component.



Behavior within the state buildingJobData
dataInputIncomplete
buildingJobData
entry/readJobData( )
exit/displayJobData( )
do/checkConsistency( )
include/dataInput
dataInputCompleted [all data items consistent]/displayUserOptions
computingJobCost
entry/computeJob
exit/save totalJobCost
jobCostAccepted [customer is authorized]/ getElectronicSignature
formingJob
entry/buildJob
exit/save WOnumber
do/
deliveryDateAccepted [customer is authorized]/ printJobEstimate
submittingJob
entry/submitJob
exit/initiateJob
do/place on JobQueue
jobSubmitted [all authorizations acquired]/ printWorkOrder

14.3 Component-Level Design

- **Step 6.** Elaborate deployment diagrams to provide additional implementation detail.
- **Step 7.** Factor every component-level design representation and always consider alternatives.

Take a break

10 seconds

14.4 Component Design for WebApps

- WebApp component is
  - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user
  - (2) a cohesive package of content and functionality that provides end-user with some required capability.

- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

14.4 Component Design for WebApps

- Web-based video surveillance capability within **SafeHomeAssured.com**
  - potential content components:
    - the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
    - the collection of thumbnail video captures (each a separate data object)
    - the streaming video window for a specific camera.
  - Each of these components can be separately named and manipulated as a package.

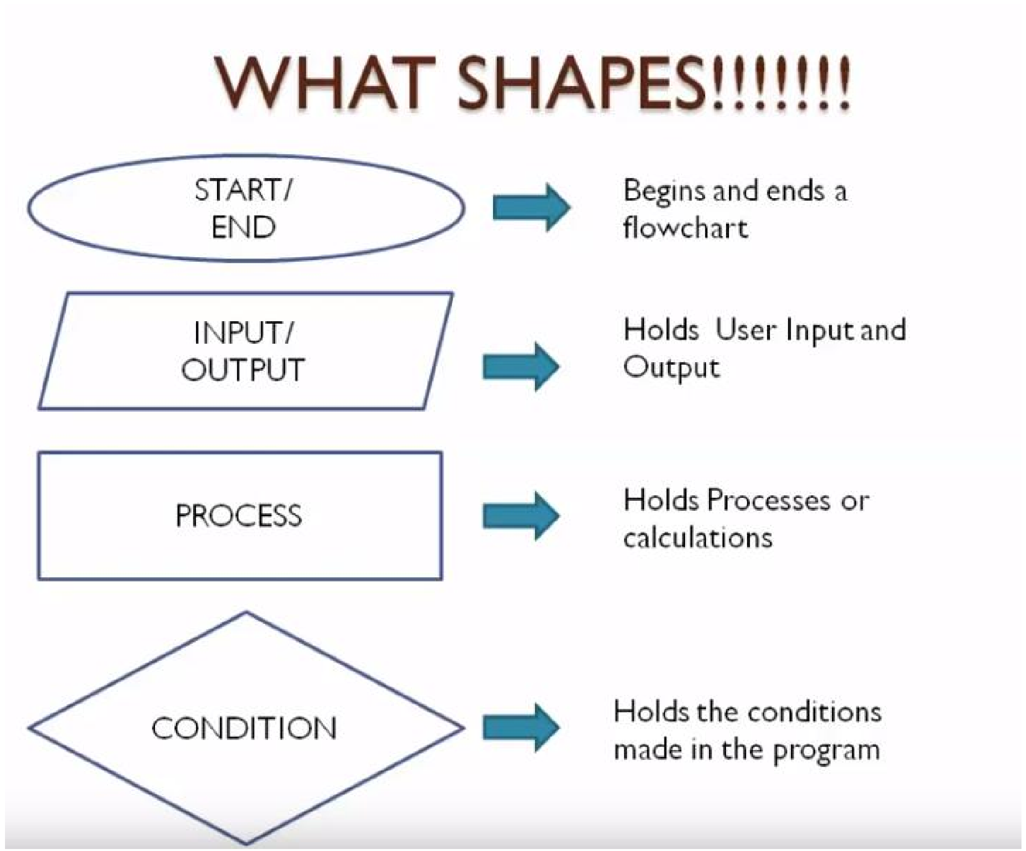
14.5 Component Design for Mobile Apps

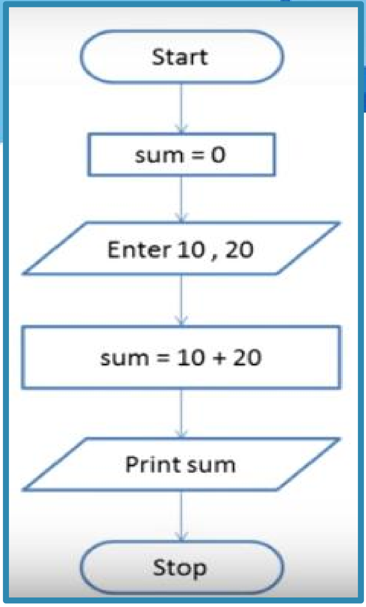
- Thin web-based client
  - Interface layer only on device
  - Business and data layers implemented using web or cloud services
- Rich(Thick) client (powerful client)
  - All three layers (interface, business, data) implemented on device
  - Subject to mobile device limitations

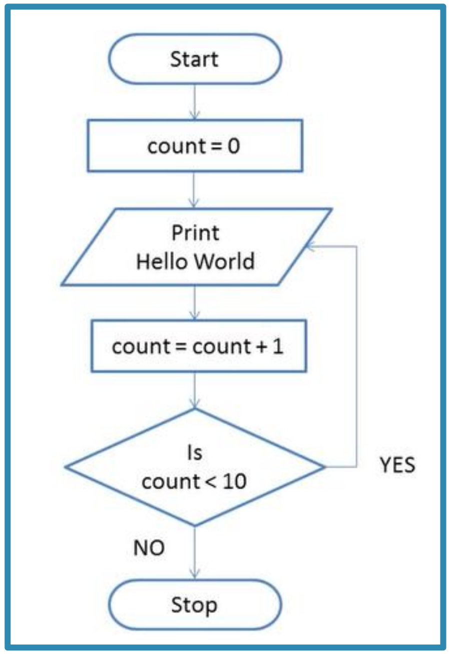
14.6 Designing Conventional Components

- The design of processing logic is governed by the basic principles of algorithm design and structured programming
- The design of data structures is defined by the data model developed for the system
- The design of interfaces is governed by the collaborations that a component must effect

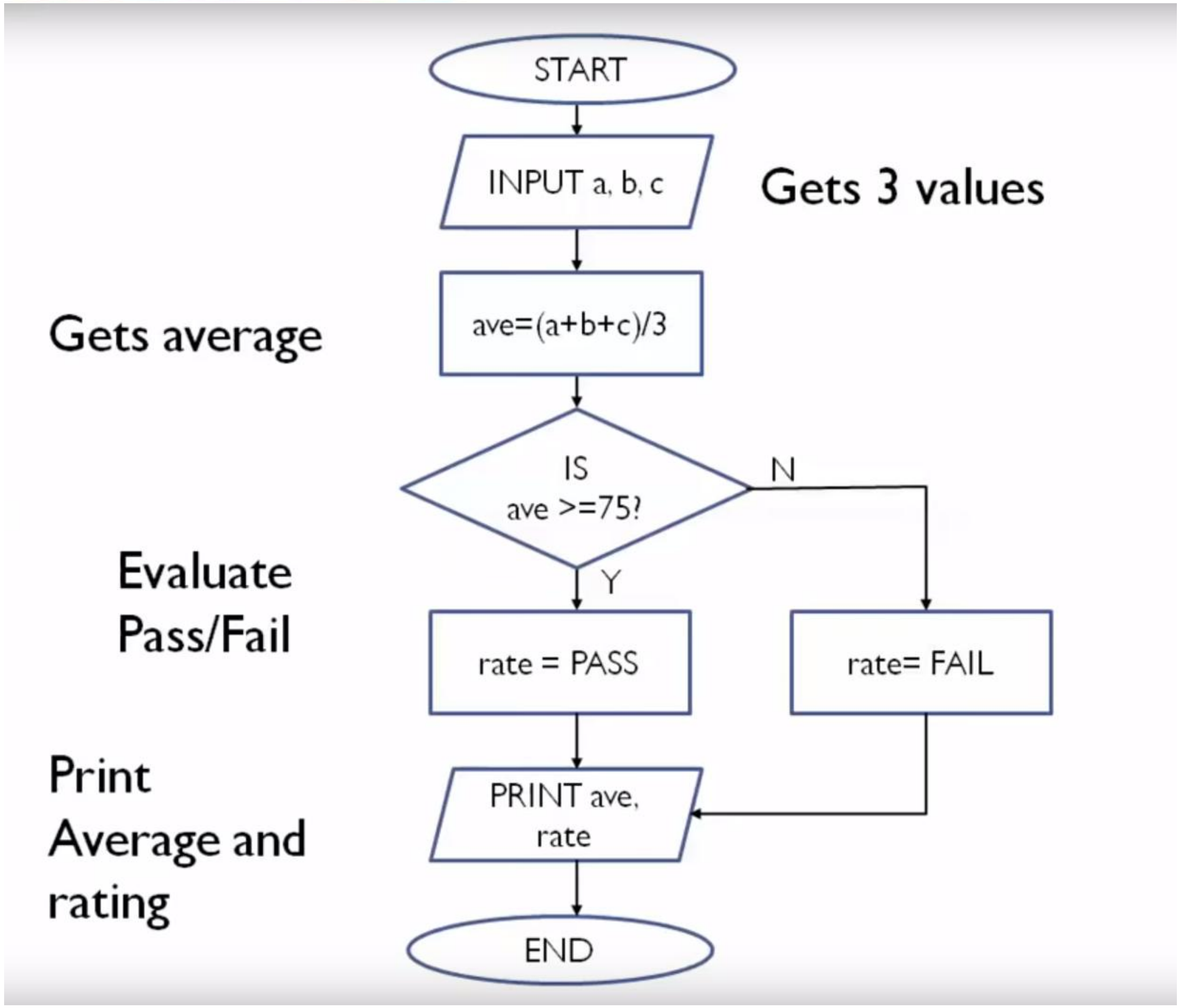
14.6 Flow chart


WHAT SHAPES!!!!!!!
START/ END
Begins and ends a flowchart
INPUT/ OUTPUT
Holds User Input and Output
PROCESS
Holds Processes or calculations
CONDITION
Holds the conditions made in the program





14.6 Flow chart



14.7 Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

14.7 Component-Based SE

- a library of components must be available
- components should have a consistent structure
- a standard should exist, e.g.,
  - OMG/CORBA
  - Microsoft COM
  - Sun JavaBeans

14.7 CBSE(Component-based SE) Activities

- Component qualification
- Component adaptation
- Component composition
- Component update

14.7 Composition

- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage
  - Underlying object model

14.7 Sun JavaBeans

- The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language.
- The JavaBeans component system encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to
  - analyze how existing Beans (components) work
  - customize their behavior and appearance
  - establish mechanisms for coordination and communication
  - develop custom Beans for use in a specific application
  - test and evaluate Bean behavior.

Summary

- What is component ?
- Component design principles
  1. TheSingle responsibility principle (SPR)
  2. The Open-Closed Principle (OCP)
  3. The Liskov Substitution Principle (LSP)
  4. The Dependency Inversion Principle (DIP)
  5. The Interface Segregation Principle (ISP)
- Component design steps
  - elaborate all classes (attribute and method)
  - message / data
  - activity diagram/ flowchart /state diagram
  - deployment
- Coupling , Cohesion, Complexity

Assignment3:

Deadline: 20 March

Find one project or part of your project you finished before, analyze the cohesion and coupling, complexity (draw flowchart) about your modules, then give some advice if need improvement.

THE END