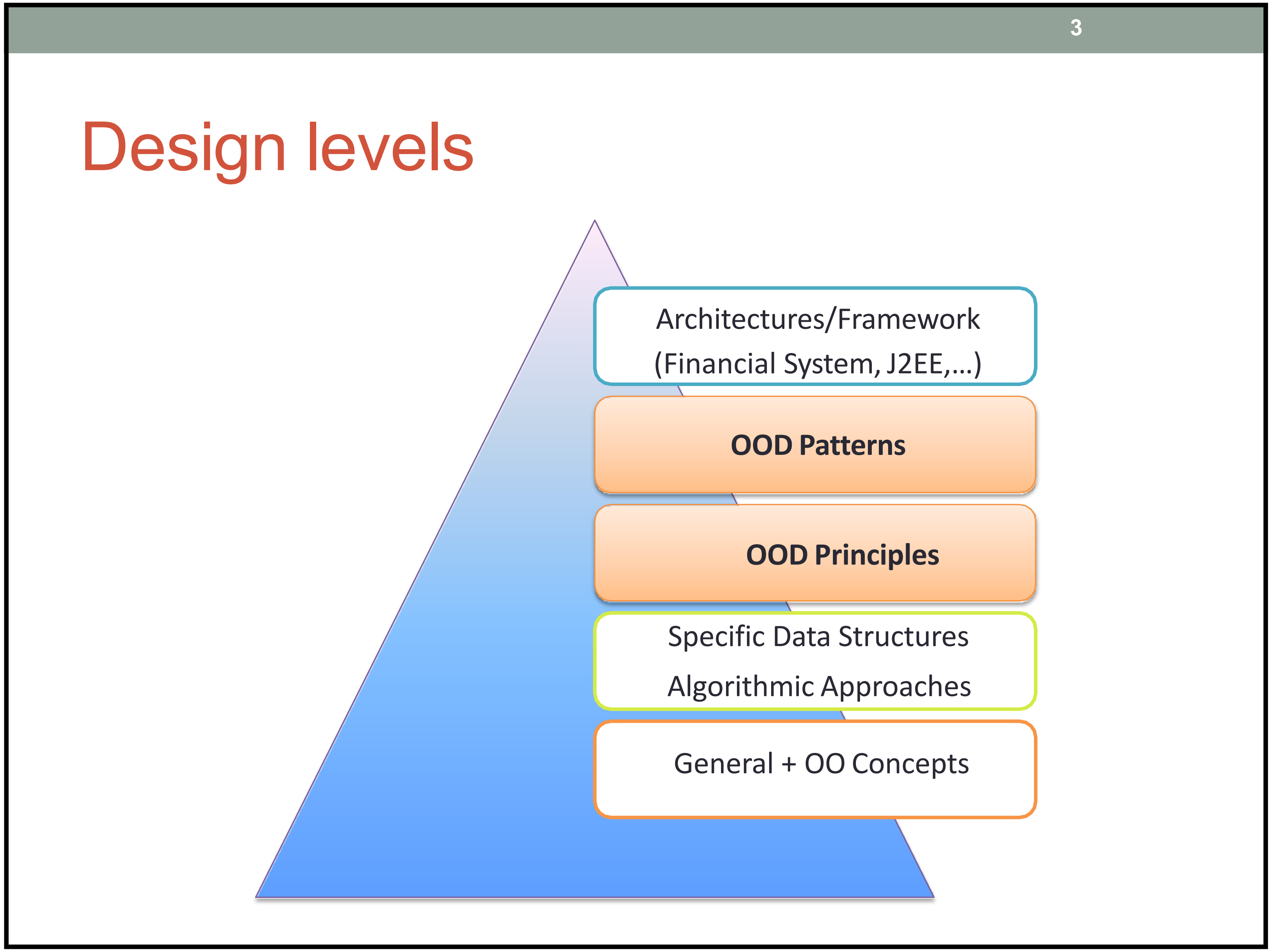
SOFTWARE DESIGN AND CONSTRUCTION

12. DESIGN CONCEPTS

Content



Design levels

The Process of Design

- Definition:
  - Design is a problem-solving process whose objective is to find and describe a way:
    - To implement the system's functional requirements...
    - While respecting the constraints imposed by the non-functional requirements...
      - including the budget
    - And while adhering to general principles of good quality

Design as a series of decisions

- A designer is faced with a series of design issues
  - These are sub-problems of the overall design problem.
  - Each issue normally has several alternative solutions:
    - design options.
  - The designer makes a design decision to resolve each issue.
    - This process involves choosing the best option from among the alternatives.

Making decisions

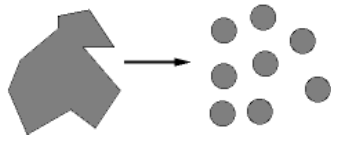
- To make each design decision, the software engineer uses:
  - Knowledge of
    - the requirements
    - the design as created so far
    - the technology available
    - software design principles and ‘best practices’
    - what has worked well in the past

Modules

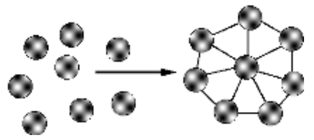
- A *module* is a relatively general term for a class or a type or any kind of design unit in software
- A *modular design* focuses on what modules are defined, what their specifications are, how they relate to each other, but not usually on the implementation of the modules themselves
- Modularity reduces the total complexity a programmer has to deal with at any one time assuming:
  - Functions are assigned to modules in away that groups similar functions together (Separation of Concerns), and
  - There are small, simple, well-defined interfaces between modules (information hiding)
- Overall, you've been given the modular design so far – and now you have to learn more about how to do the design

Ideals of modular software

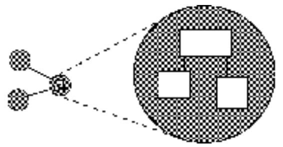
- Decomposable – can be broken down into modules to reduce complexity and allow teamwork



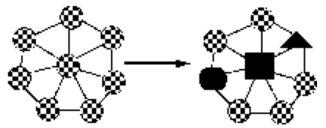
- Composable – “Having divided to conquer, we must reunite to rule [M. Jackson].”



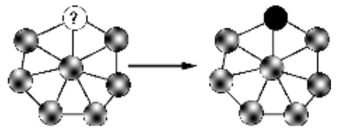
- Understandable – one module can be examined, reasoned about, developed, etc. in isolation



- Continuity – a small change in the requirements should affect a small number of modules



- Isolation – an error in one module should be as contained as possible



Top-down and bottom-up design

- Top-down design
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed decisions such as:
    - the format of particular data items;
    - the individual algorithms that will be used.

Top-down and bottom-up design

- Bottom-up design
  - Make decisions about reusable low-level utilities.
  - Then decide how these will be put together to create high-level constructs.

- A mix of top-down and bottom-up approaches are normally used:
  - Top-down design is almost always needed to give the system a good structure.
  - Bottom-up design is normally useful so that reusable components can be created.

Different aspects of design

- Architecture design:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- Class design:
  - The various features of classes.
- User interface design
- Algorithm design:
  - The design of computational mechanisms.
- Protocol design:
  - The design of communications protocol.

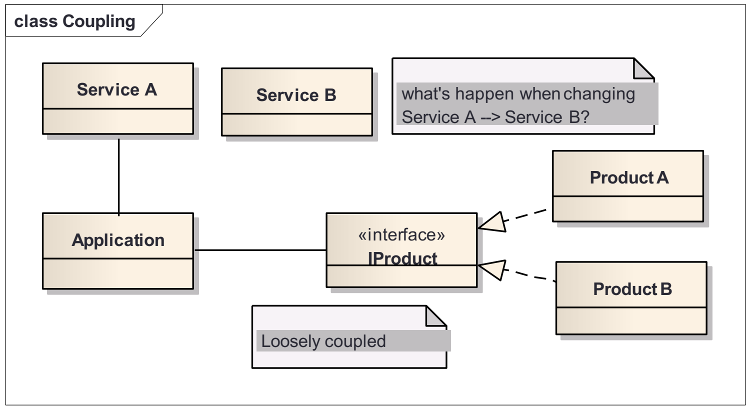
Good Design

- Overall goals of good design:
  - Increasing profit by reducing cost and increasing revenue
  - Ensuring that we actually conform with the requirements
  - Accelerating development
  - Increasing qualities such as
    - Usability
    - Efficiency
    - Reliability
    - Maintainability
    - Reusability
- The principles of cohesion and coupling are probably the most important design principles for evaluating the effectiveness of a design.

Cohesion and Coupling

❑ Coupling

Coupling or ***Dependency*** is the degree to which each module relies on each one of the other modules.



❑ Cohesion

Cohesion refers to the degree to which the elements of a module belong together. ***Cohesion*** is a measure of how strongly-related or focused the responsibilities of a single module are.

Quality Characteristics for Modules

- independent modules
  - avoid aggregation of multiple tasks per module
  - single-minded function

===> high COHESION

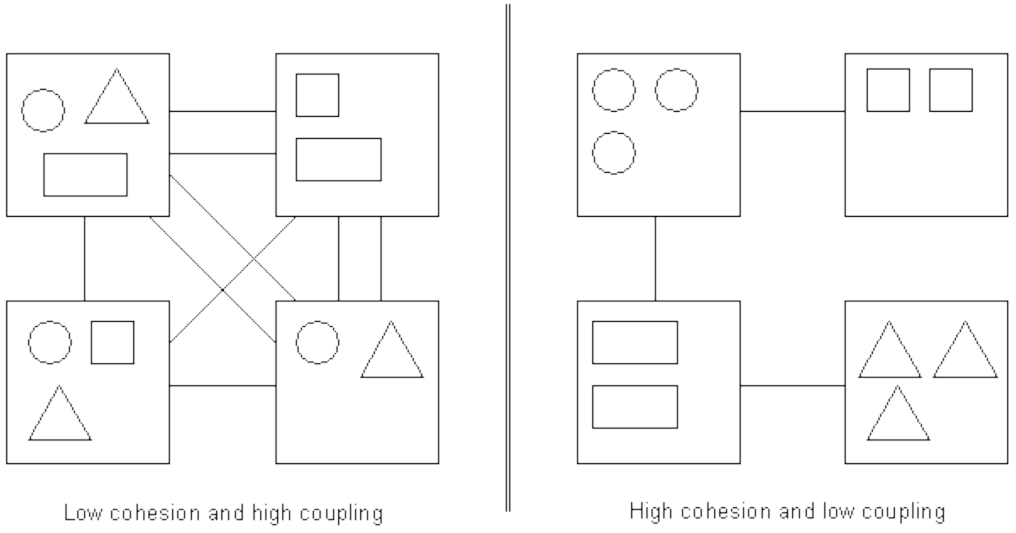
  - minimal interaction with other modules
  - simple interface

===> low COUPLING

[Steven, Myers, Constantine]

Cohesion and Coupling

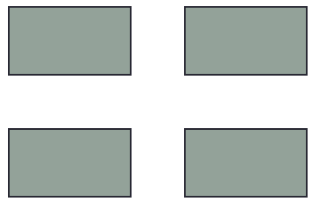
- The best designs have high cohesion (also called strong cohesion) within a module and low coupling (also called weak coupling) between modules.

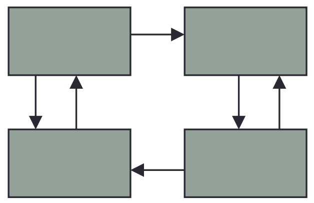

Low cohesion and high coupling
High cohesion and low coupling

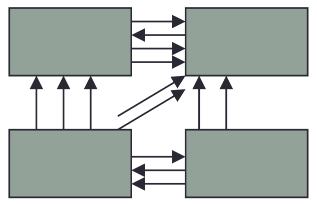
Content

1. How do you design?
2. Coupling
3. Cohesion

Coupling: Degree of dependence among components


No dependencies


Loosely coupled-some dependencies


Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

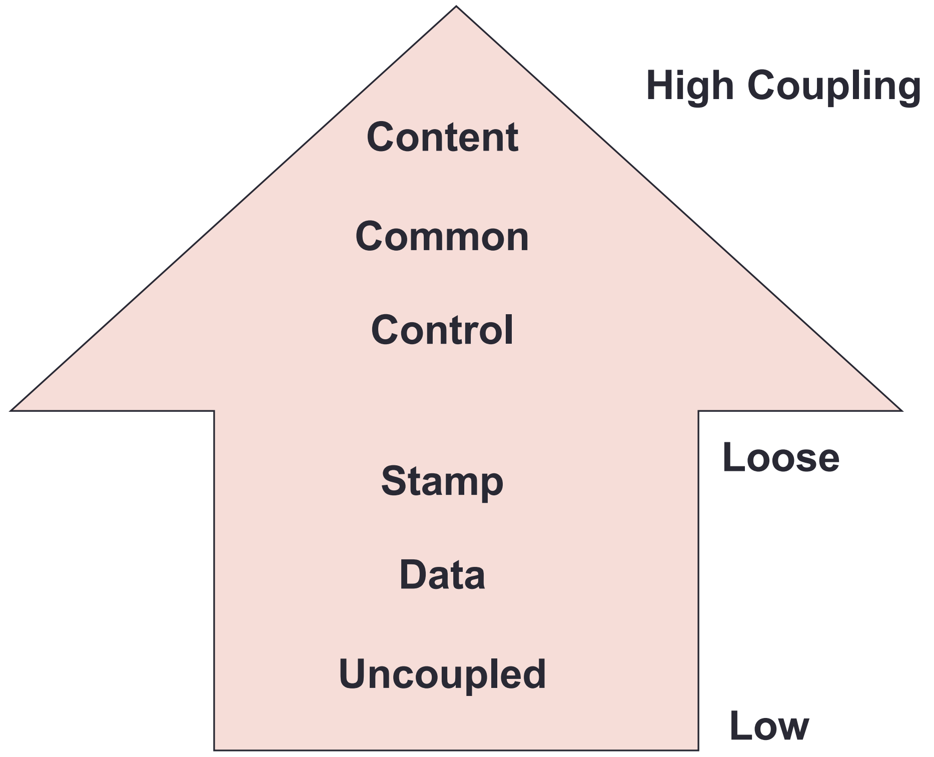
Coupling evaluation

- Degree
  - Degree is the number of connections between the module and others. With coupling, we want to keep the degree small. For instance, if the module needed to connect to other modules through a few parameters or narrow interfaces, then the degree would be small, and coupling would be loose.
- Ease
  - Ease is how obvious are the connections between the module and others. With coupling, we want the connections to be easy to make without needing to understand the implementations of the other modules.
- Flexibility
  - Flexibility is how interchangeable the other modules are for this module. With coupling, we want the other modules easily replaceable for something better in the future.

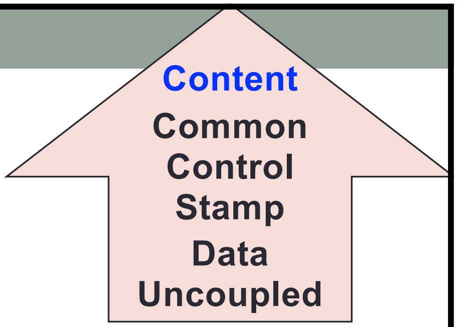
Disadvantages of tightly coupling

- A change in one module usually forces a ripple effect of changes in other modules.
- Assembly of modules might require more effort or time due to the increased inter-module dependency.
- A particular module might be harder to reuse or test because dependent modules must be included.

Range of Coupling



2.1. Content coupling



- Definition: One component references contents of another
- Example:
  - Component directly modifies another’s data
  - Component modifies another’s code, e.g., jumps into the middle of a routine

Content coupling:

- Occurs when one component surreptitiously modifies data that is **internal** to another component
  - To reduce content coupling you should therefore *encapsulate* all instance variables
    - declare them private
    - and provide get and set methods
  - A worse form of content coupling occurs when you directly modify an instance variable of an instance variable
- Disadvantage: if f changes g's local variables, then changing f often requires changing g or vice versa
  - Have to keep both in mind when changing either

Example of content coupling

- In C++, friend classes can access each other's private members

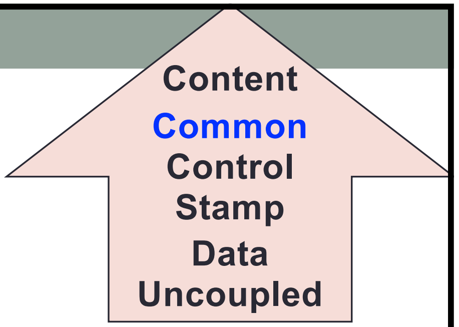
```
class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};
```

Example of content coupling

- "getters" and "setters" may not prevent content coupling if the calculations on this variable is done by another class and sets it back to the original class

```
public int sumValues(Calculator c){ // tight coupling
    int result = c.getFirstNumber() + c.getSecondNumber();
    c.setResult(result);
    return c.getResult();
}

public int sumValues(Calculator c){ // loose coupling
    c.sumAndUpdateResult();
    return c.getResult();
}
```

2.2. Common Coupling



- Definition: Several modules share an externally imposed data format, communication protocol, or device interface.
- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses

Example

- All the components using the global variable become coupled to each other: one sets variable, another refers to value
- This is certainly possible in O-O languages
  - One static object contains all “global data”
  - This object’s attributes are set and referred to
- Disadvantage: consider debugging task: e.g.
  - Global variable count has gotten some strange value
  - Both f and g have been called
  - Which changed it to the strange value?
  - Must flip back and forth between f and g to find out
  - Are there other functions which change count
  - Have to keep whole program in mind when debugging

Example

```
while( global_variable > 0 ) {
    switch( global_variable ) {
        case 1: function_a(); break;
        case 2: function_b(); break;
        ...
        case n:...

    }
    global_variable++;
}
```

2.3. Control Coupling

- Definition: Component passes control parameters to coupled components
  - Example: a method does different things depending on the value of a “flag” parameter.
- May be either good or bad, depending on situation
  - Bad when component must be aware of internal structure and logic of another module
  - Good if parameters allow factoring and reuse of functionality
- Example of good: Sort that takes a comparison function as an argument. The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.

Example 1 – Control Coupling

- In your video store, you might eventually create a method like this:
  - updateCustomer(int whatKind, Customer customer) where
    - whatKind takes on the values ADD, EDIT or DELETE, and
    - customer is used for EDIT, but is not used at all for ADD, and only the id is used for DELETE.

Example 1 – Hint for updateCustomer()

- Should be avoided, as it means the calling module must know how the logic of the called module is organized. This also may lead to gradually extent of param list when new logics defined
- The method becomes very complicated to maintain.
- How can we improve this situation?

- Often, control coupling can be reduced by:
  - Introduce different methods for different cases
  - Think of applying inheritance and polymorphism if possible

Example 2 – tight coupling

```
public void run() {
    takeAction(1);
}

public void takeAction(int key) {
    switch (key) {
    case 1:
        System.out.println("ONE RECEIVED");
        break;
    case 2:
        System.out.println("TWO RECEIVED");
        break;
    }
}
```

Example 2 – loose coupling

```
public void run() {
    Printable printable = new PrinterOne();
    takeAction(printable);
}
public void takeAction(Printable printable) {
    printable.print();
}


public interface Printable {
    void print();
}

public class PrinterOne implements Printable {
    @Override
    public void print() {
        System.out.println("ONE RECEIVED");
    }
}

public class PrinterTwo implements Printable {
    @Override
    public void print() {
        System.out.println("TWO RECEIVED");
    }
}
```

Example 3

```
int testStack(int kind, Stack S) {
    if (((kind == 1) && (S.num == 0)) || ((kind == 2) && (S.num == MAX)))
        return 1;
    else
        return 0;
}
```

- What does testStack do?
  - Seems to do two completely different things
- kind is a “magic” parameter
  - Selects from one of the two behaviours
  - Can’t tell what value it should be without looking at code for testStack

Example 3

- Better to have two functions and name them intuitively:

```
int emptyStack(Stack S) {
   if (S.num == 0)
      return 1;
   else
      return 0;
}

int fullStack(Stack S) {
   if (S.num == MAXSTACK)
      return 1;
   else
      return 0;
}
```

- Previous code worked fine (if called correctly) However, with new code:
  - It is clear from the function names what the functions do
  - Don’t have to guess at values of parameters

2.4. Stamp Coupling

- Definition: data are passed by parameters using a data structure containing fields which may or may not be used
- Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation)
- May be necessary due to efficiency factors: this is a choice made by insightful designer, not lazy programmer.
- Two ways to reduce stamp coupling,
  - using an interface as the argument type
  - passing simple variables

Example 1

- Example: code for finding how much income tax to deduct from employee’s salary:

```
int incomeTaxPayable(Person p) {
    /* code which refers to only p.salary */
}
```

- Function header leads us to believe that:
  - incomeTaxPayable uses all of p
  - We have to keep all of the Person fields in mind when debugging incomeTaxPayable

Example 1

- Better:

```
int incomeTaxPayable(int salary) {
    /* code which refers salary */
}
```

- Call would be incomeTaxPayable (employee.salary) rather than incomeTaxPayable(employee)
- Shows clearly that incomeTaxPayable is concerned only with the employee's salary
- If the calling function can extract all that is needed for the called function, it *should*

Example 2

```
public class Emailer {
   public void sendEmail(Employee e, String text) {
      ...
   }
   ...
}
```

- Using simple data types to avoid it:

```
public class Emailer {
   public void sendEmail(String name, String email, String text) {
        ...
   }
   ...
}
```

Example 2

- Using an interface to avoid it:

```
public interface Addressee {
    public abstract String getName();
    public abstract String getEmail();
}

public class Employee implements Addressee {...}


public class Emailer {
    public void sendEmail(Addressee e, String text) {
        ...
    }
    ...
}
```

2.5. Data coupling

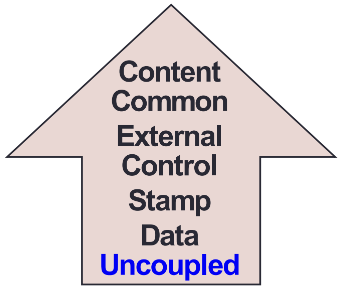
- Def: Component passes data (not data structures) to another component.
- Occurs whenever the types of method arguments are either primitive or else simple library classes
  - The more arguments a method has, the higher the coupling
    - All methods that use the method must pass all the arguments
  - You should reduce coupling by not giving methods unnecessary arguments

Data coupling

- Strengths of Data Coupling
  - A module sees only the data elements it requires.
  - It is the best (i.e., loosest) form of coupling.
- Weakness of Data Coupling
  - A module can be difficult to maintain if many data elements are passed.
  - Too many parameters can also indicate that a module has been poorly partitioned.
- There is a trade-off between data coupling and stamp coupling
  - Increasing one often decreases the other

Uncoupled

- Completely uncoupled components are not systems.
- Systems are made of interacting components.



Content

1. How do you design?
2. Coupling
3. Cohesion

3. Cohesion

- Definition1: The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.
- Definition 2: Cohesion represents the clarity of the responsibilities of a module.
- Internal glue with which component is constructed
- All elements of component are directed toward and essential for performing the same task
- High is good

Cohesion

- If our module performs one task and nothing else or has a clear purpose, our module has high cohesion.
- On the other hand, if our module tries to encapsulate more than one purpose or has an unclear purpose, our module has low cohesion.
- **Single Responsibility Principle** aims at creating highly cohesive classes.
- Cohesion is increased if:
  - The functionalities embedded in a class, accessed through its methods, have much in common.
  - Methods carry out a small number of related activities, by avoiding coarsely grained or unrelated sets of data

Advantages of high cohesion

- Reduced module complexity (they are simpler, having fewer operations).
- Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
- Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.

Example

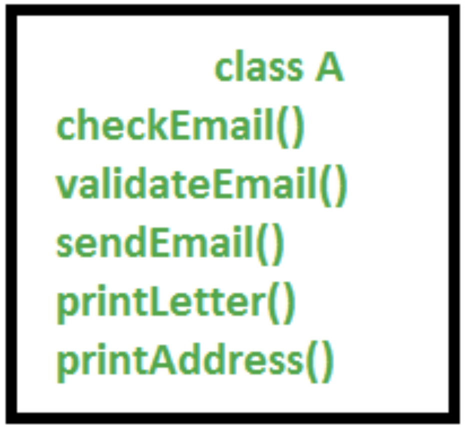


Fig: Low cohesion

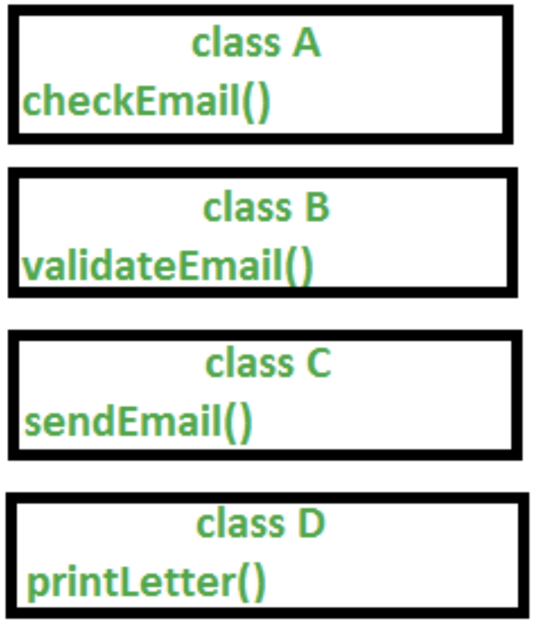


Fig: High cohesion

Example

- High cohesion is when we have a class that does a well defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
- High cohesion gives us better maintaining facility and Low cohesion results in monolithic classes that are difficult to maintain, understand and reduces re-usability

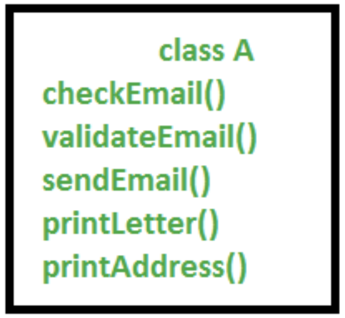


Fig: Low cohesion

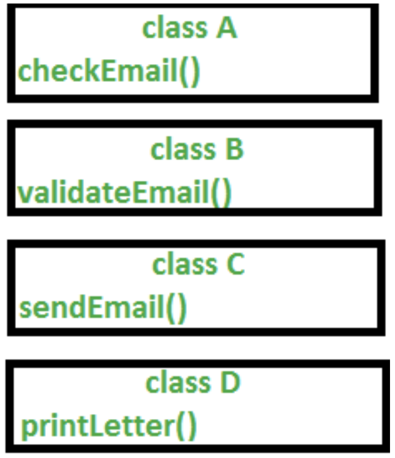
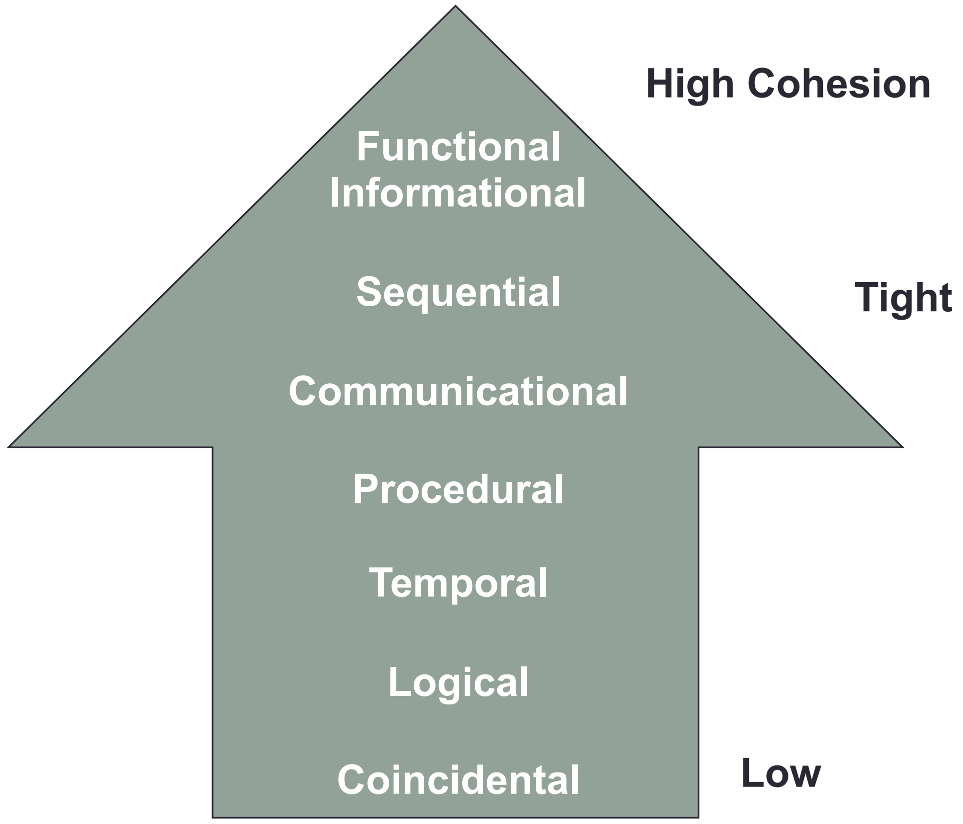


Fig: High cohesion

Range of Cohesion

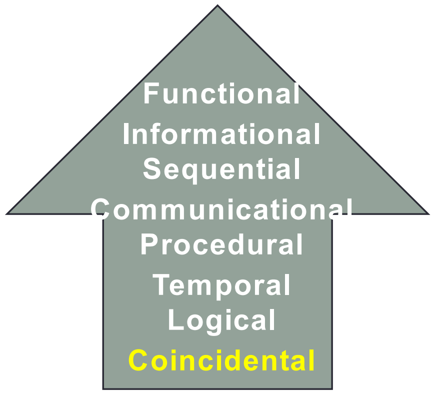


Range of cohesion

- Low Cohesion (Highly Undesirable)
  - Coincidental Cohesion
  - Logical Cohesion
  - Temporal Cohesion
- Moderate Cohesion (Acceptable)
  - Procedural Cohesion
  - Communicational Cohesion
  - Sequential Cohesion
- High Cohesion (Desirable)
  - Functional Cohesion

3.1. Coincidental Cohesion

- Definition: Elements of the component are only related by their location in source code
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form



Example 1

- A module supports the following tasks
  - Fix Car
  - Bake Cake
  - Walk Dog
  - Fill our Astronaut-Application Form
  - Have a Beer
  - Get out of Bed
  - Go the the Movies

Example 2

```
class Joe {
   // converts a path in windows to one in linux
   public String win2lin(String);

   // number of days since the beginning of time
   public int days(String);

   // outputs a financial report
   public void outputReport(FinanceData);
}
```

Example 3

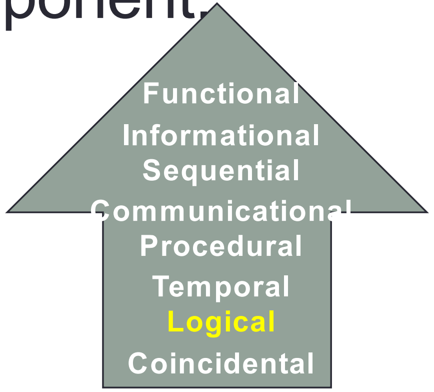
```
class Output {
   // outputs a financial report
   public void outputReport(FinanceData);

   // outputs the current weather
   public void outputWeather(WeatherData);

   // output a number in a nice formatted way
   public void outputInt(int);
}
```

Example 4

- (1) A programmer might take a 200 line program and break it into four procedures with 50 lines of code in each. Thus, lines 1-50 would be in procedure-1 and lines 51-100 would be in procedure-2, etc.
- (2) Recurring code which has **no complete function** might be spotted by the programmer and, in an effort to save memory space, placed that code in one procedure (a module). That procedure was then repeatedly called in the program
- (3) A package with non-related classes

3.2. Logical Cohesion

- Definition: Elements of component are related logically and not functionally (elements contribute to activities of the same general category)
- Several logically related elements are in the same component and one of the elements is selected by the client component.



Example 1

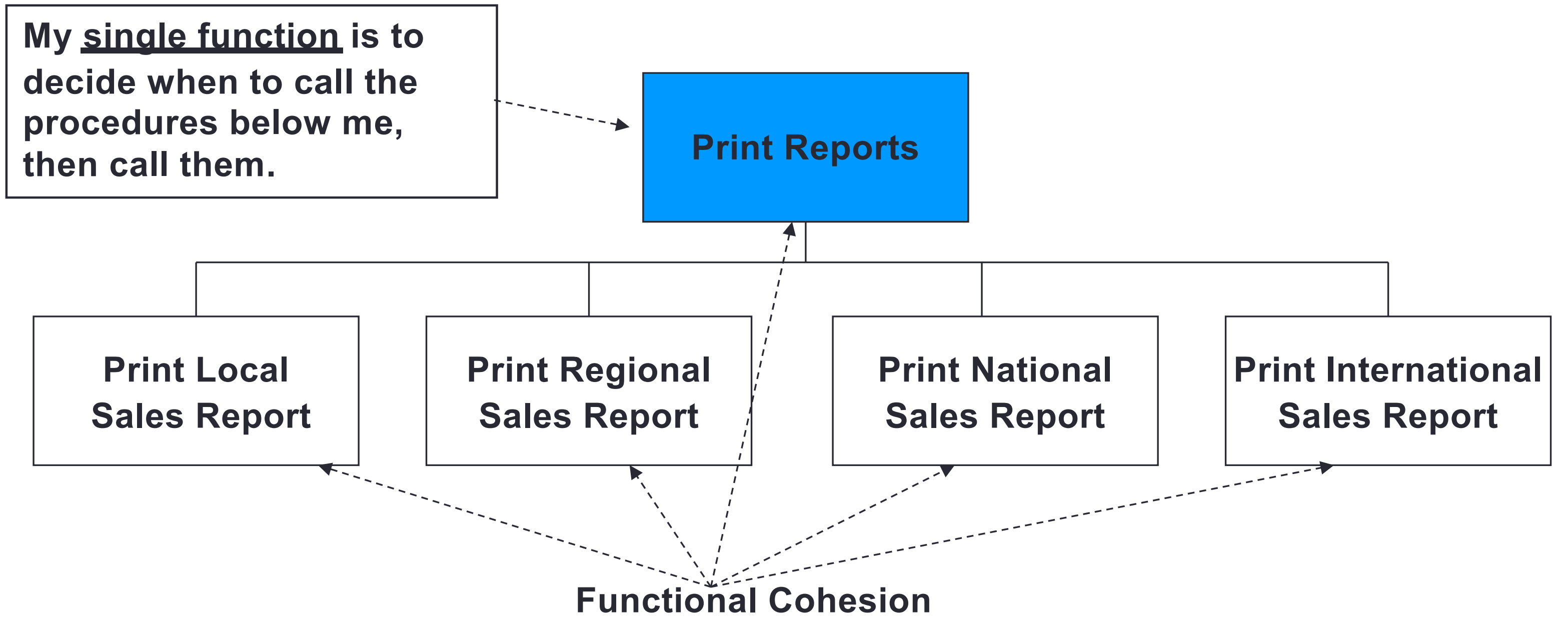
- Someone contemplating a journey might compile the following list:
  - Go by Car
  - Go by Train
  - Go by Boat
  - Go by Plane
- What relates these activities? They're all means of transport, of course. But a crucial point is that for any journey, a person must choose a specific subset of these modes of transport. It's unlikely anyone would use them *all* on any particular journey.

Example 2

- A "print all" procedure might consist of:
  - Printing a local sales report.
  - Printing a regional sales report.
  - Printing a national sales report.
  - Printing a international sales report A "print all" procedure might consist of:
- If the user, just wanted a national sales report, and not any of the others, a flag (a switch) would need to be passed to the procedure (using arguments and parameters) to tell the procedure which report to print.

Example 2

- You can eliminate the need for logical cohesion, by making each report above a separate procedure. Thus, there would be four separate procedures which could be called individually when needed by using buttons, check boxes, radio button, other procedures, etc...


My single function is to decide when to call the procedures below me, then call them.
Print Reports
Print Local Sales Report
Print Regional Sales Report
Print National Sales Report
Print International Sales Report
Functional Cohesion

Example 3

- A component reads inputs from tape, disk, and network. All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

```
public class Example {
    public void readDataFromTape(Tape t) {
    }
    public void readDataFromDisk(Disk d) {
    }
    public void readDataFromNetwork(Network n) {
    }
}
```

- => How to improve?

Example 3

- A component reads inputs from tape, disk, and network. All the code for these functions are in the same component. Operations are related, but the functions are significantly different.

=> Improvement and charging

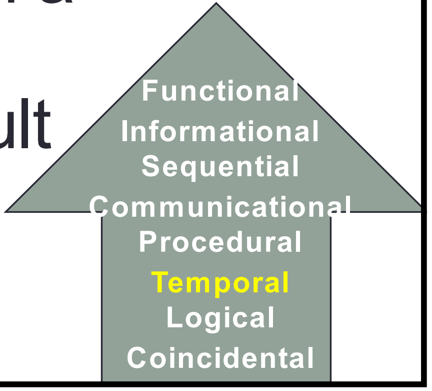
- A device component has a read operation that is overridden by sub-class components. The tape sub-class reads from tape. The disk sub-class reads from disk. The network sub-class reads from the network.

Example 3

```
public abstract class DataReader<T> {
    public void read(T t)
}

public class TapeDataReader extends DataReader<Tape>{
    @Override
    public void read(Tape t) {
    }
}
public class DiskDataReader extends DataReader<Disk>{
    @Override
    public void read(Disk d) {
    }
}
public class NetWorkDataReader extends DataReader<Network>{
    @Override
    public void read(Network n) {
    }
}
```

3.3. Temporal Cohesion

- Definition: Elements of a component are related by timing.
- Temporal cohesion is when parts of a module are grouped by when they are processed - the parts at a particular time in program execution
- Difficult to change because you may have to look at numerous components when a change in a data structure is made.
- Increases chances of regression fault
- Component unlikely to be reusable.



Example 1

- A module supports the following tasks
  - Put out Milk Bottles
  - Put out Cat
  - Turn off TV
  - Brush Teeth
- These activities are (only) related by the fact that you **do them all** late at night, just before you go to bed

Example 2

- A system initialization procedure: this procedure contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

```
class Init {
    // initializes financial report
    public void initReport(FinanceData);

    // initializes current weather
    public void initWeather(WeatherData);

    // initializes master count
    public void initCount();
}
```

- => How to improve?

Example 2

- A system initialization procedure: this procedure contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

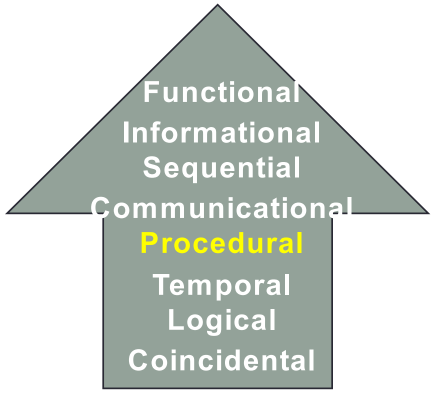
=> How to improvement?

- A system initialization procedure sends an initialization message to each component.
- Each component initializes itself at component instantiation time.

Remember: The goal is to produce procedures which can do their single function independently of other procedures.

3.4. Procedural Cohesion

- Definition: Elements of a component are related only to ensure a particular order of execution.
- Actions are still weakly connected and unlikely to be reusable



Example 1

- A *Prepare for Holiday Meal* module:
  - Clean Utensils from Previous Meal
  - Prepare Turkey for Roasting
  - Make Phone Call
  - Take Shower
  - Chop Vegetables
  - Set Table
- The module support different and possibly unrelated tasks, in which **control** passes from one activity to the next.
- This is a bit better than “temporal cohesion”, since we know that there's a fixed “linear ordering" of the activities. However, there's still not much reason for putting all these activities together into one module

Example 2

```
public class Example {
    public void readData {}
    public void sendEmail() {}
}
```

⇩

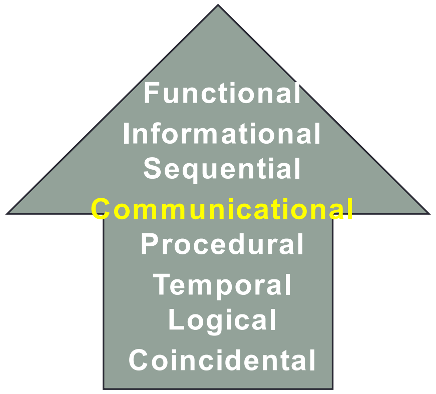
```
public class DataReader{
    public void readData(){}
}

public class Email{
    public void sendEmail() {}
}
```

Procedural cohesion

- Advantages
  - Elements are related to the program procedure and have a closer relationship to each other.
- Disadvantages
  - Elements have a closer relationship to the program procedure than the problem procedure.
  - Because these elements represent part of the functionality of the module it can lead to increased maintenance cost trough the ripple effect.

3.5. Communicational Cohesion

- Definition: Communicational cohesion is when parts of a module are grouped because they operate on the same data. (use the same input or output data - or access and modify the same part of a data structure)
- All of the data is passed among the elements of the module.

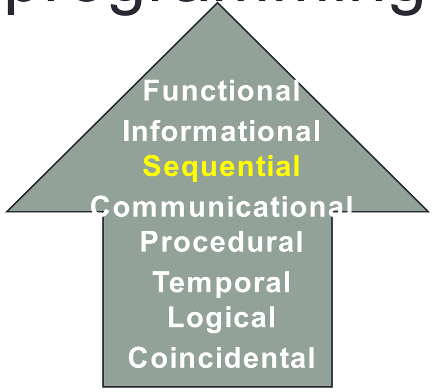


Example

```
public class Example {
    private Transaction trans;

    public readTransaction() {}
    public void sortTransaction() {}
    public void calculateTransactionMean() {}
    public void printTransaction() {}
    public void saveTransaction() {}
}
```

3.6. Sequential Cohesion

- A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next
- Occurs naturally in functional programming languages
- Good situation



Example

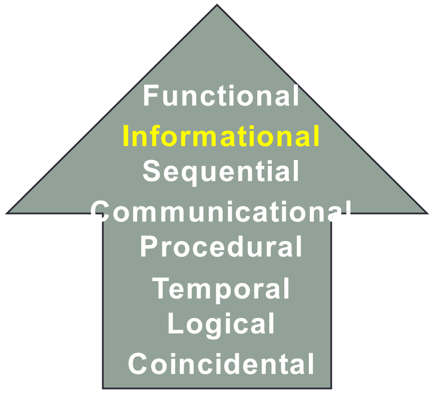
- A module supports the following activities
  - Clean Car Body
  - Fill in Holes in Car
  - Sand Car Body
  - Apply Primer
- The input car is being “passed" as a parameter from task to task

Sequential cohesion

- Example
  - Calculate the trajectory and print results.
- Advantage
  - Elements in a module with this type of cohesion are organized functionally providing a better isolation of functionality than in communicational cohesion.
- Disadvantage
  - Reuse is limited, because the module is only usable in an identical environment. For example, if there is a situation where there is a need to calculate the trajectory and write to a file or write to a display, it would be necessary to create a new module

3.7. Informational Cohesion

- Definition: Module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data.



Example 1

- A module performs all of the master file access operations. It reads, adds, updates and deletes master file records. Each of these functions has its own entry point.

```
void employeeAdd(int status, EmpRec rec);

void employeeUpdate(int status, EmpRec rec);

void employeeDelete(int status, EmpRec rec);
```

Example 2

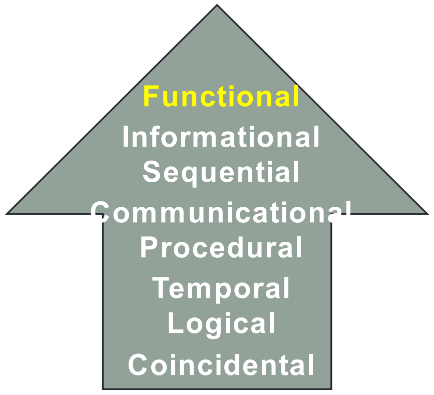
```
public class Example {
    public Transaction readTransaction() {}
    public void sortTransaction(Transaction trans) {}
    public void calculateTransactionMean(Transaction trans) {}
    public void printTransaction(Transaction trans) {}
    public void saveTransaction(Transaction trans) {}
}
```

Informational Cohesion

- Advantages
  - Informational cohesion provides the developer with the ability to "package" all of the functionality that relates to a given data structure. This can reduce maintenance cost if it is carried through the application by providing a single location for all of the functionality related to data structure.
- Disadvantages
  - If all of the functionality is not required in another application, it may increase the memory requirement of the application and there by limit the utility in another application.
- Note
  - Informational cohesion is the type of cohesion exhibited by the methods in an object or class. It ranks below functional cohesion in a structured development environment.

3.8. Functional Cohesion

- Definition: Every essential element to a single computation is contained in the component.
- Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function
- Ideal situation.



3.8. Functional Cohesion

- Advantage
  - Maximum possible reuse of the function in other applications.
  - Maximum isolation of functionality. This isolation reduces the the maintenance costs.

- Disadvantage
  - Using the functional cohesion results in a large number of modules that may necessitate the use of case tools like make and version control.

Determining Cohesiveness

- Write down a sentence to describe the function of the module
- If the sentence is compound,
  - It has a sequential or communicational cohesion.
- If it has words like “first”, “next”, “after”, “then”, etc.
  - It has sequential or temporal cohesion.
- If it has words like initialize,
  - It probably has temporal cohesion.

Difference between cohesion and coupling

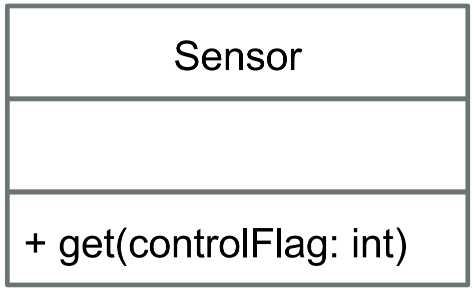
Cohesion

- Cohesion is the indication of the relationship within module
- Cohesion shows the module’s relative functional strength
- Cohesion is a degree (quality) to which a component / module focuses on the single thing
- While designing we should strive for high cohesion. Ex: cohesive component/module focus on a single task with little interaction with other modules of the system
- Cohesion is the kind of natural extension of data hiding, for example, class having all members visible with a package having default visibility
- Cohesion is Intra-Module Concept

Coupling

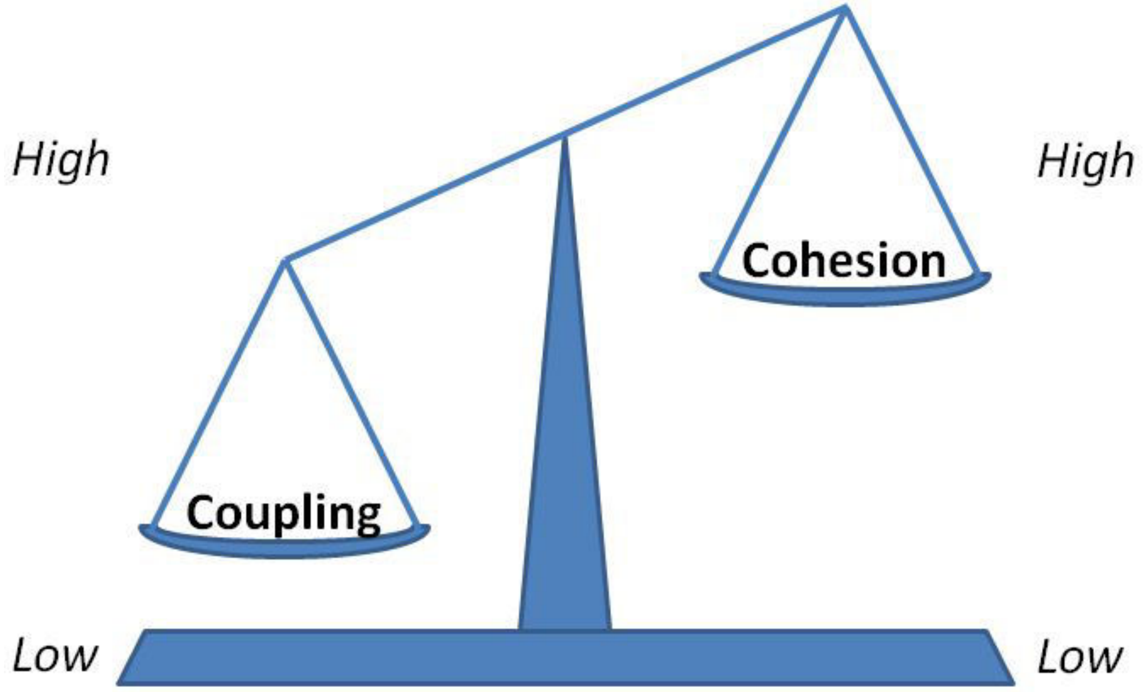
- Coupling is the indication of the relationships between modules
- Coupling shows the relative independence among the modules
- Coupling is a degree to which a component / module is connected to the other modules
- While designing we should strive for low coupling. Ex: dependency between modules should be less
- Making private fields, private methods and non public classes provides loose coupling
- Coupling is Inter-Module Concept

Bad Example



```
public void get (int controlFlag){
    switch (controlFlag) {
        case 0:
            return this.humidity;
            break;
        case 1:
            return this.temperature;
            break;
        default:
            throw new UnknownFlagException
    }
}
```

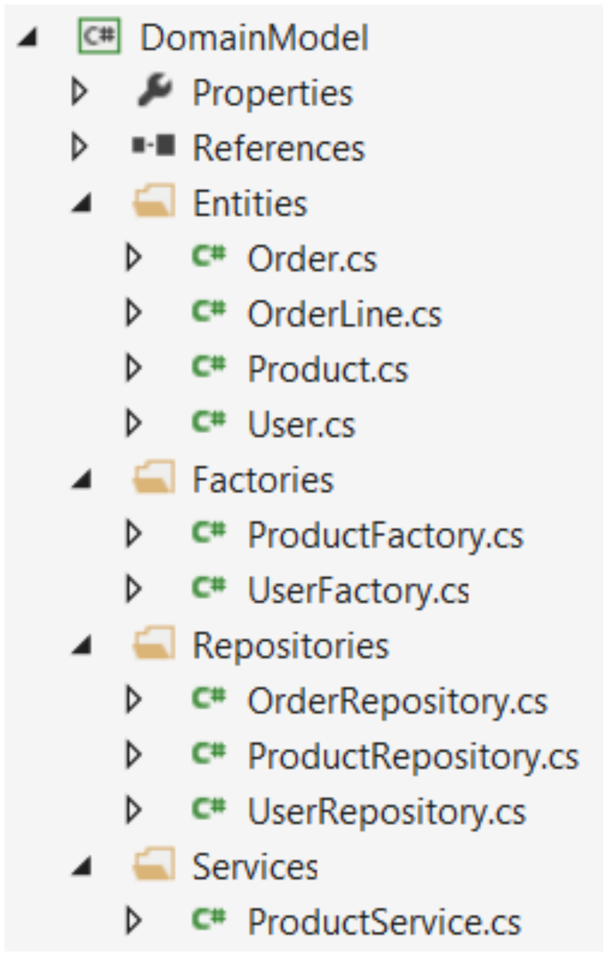
Relationship between Coupling and Cohesion



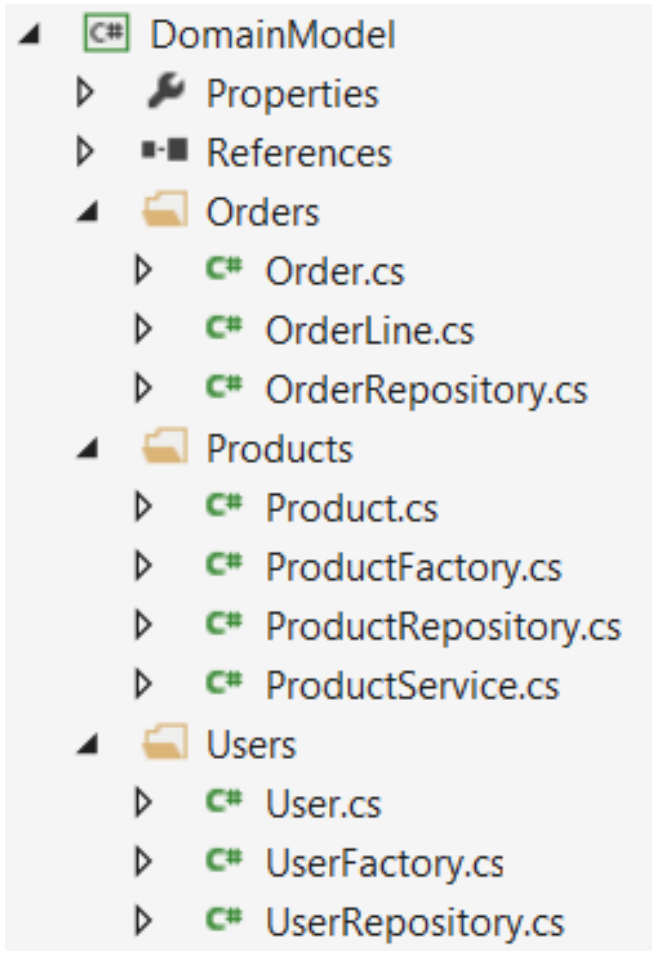
Cohesion and Coupling

- In general, there's a balance to be made between low coupling and high cohesion in your designs.
- For a complex system, the complexity can be distributed to between the modules or within the modules.
- As modules are simplified to achieve high cohesion, they may need to depend more on other modules thus increasing coupling.
- As connections between modules are simplified to achieve low coupling, the modules may need to take on more responsibilities thus lowering cohesion

Example



- DomainModel
  - Properties
  - References
  - Entities
    - Order.cs
    - OrderLine.cs
    - Product.cs
    - User.cs
  - Factories
    - ProductFactory.cs
    - UserFactory.cs
  - Repositories
    - OrderRepository.cs
    - ProductRepository.cs
    - UserRepository.cs
  - Services
    - ProductService.cs



- DomainModel
  - Properties
  - References
  - Orders
    - Order.cs
    - OrderLine.cs
    - OrderRepository.cs
  - Products
    - Product.cs
    - ProductFactory.cs
    - ProductRepository.cs
    - ProductService.cs
  - Users
    - User.cs
    - UserFactory.cs
    - UserRepository.cs