ITSS SOFTWARE DEVELOPMENT

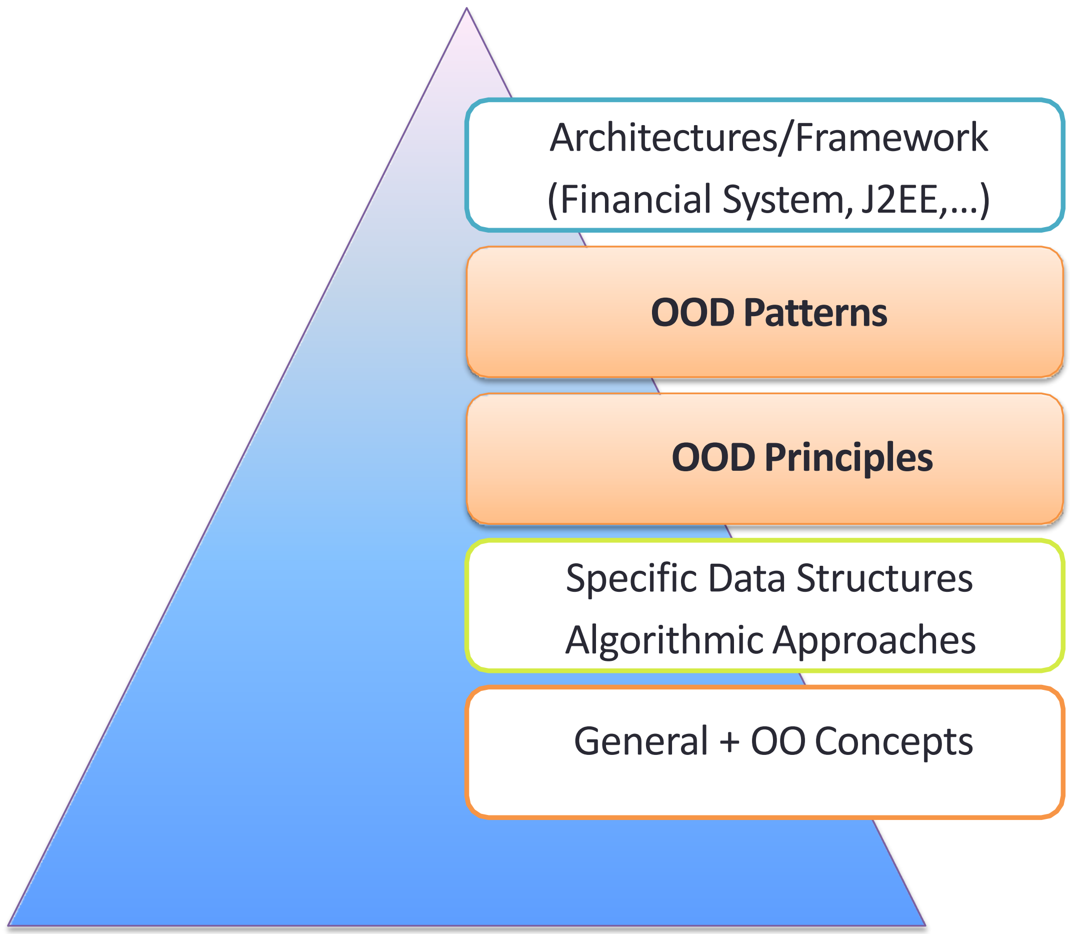
10. DESIGN CONCEPTS

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

Content

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

1.1. Design levels



Key design concepts

General

- Cohesion
- Coupling
- Information hiding
  - Encapsulation
  - Creation
- Binding time

OO Specific

- Behaviors follow data
- Class vs. Interface Inheritance
  - Class = implementation
  - Interface = type
- Inheritance / composition / delegation

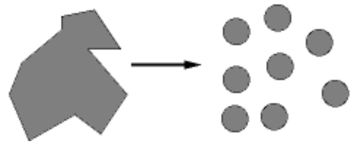
Modules

- 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

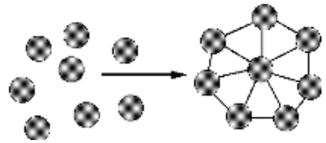
- Overall, you've been given the modular design so far – and now you have to learn more about how to do the design

Ideals of modular software

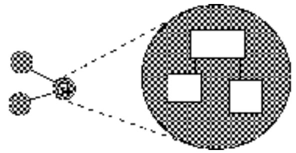
- Decomposable – can be broken down into modules to reduce complexity and allow teamwork



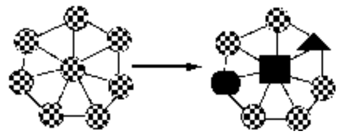
- Composable – “Having divided to conquer, we must reunite to rule [M. Jackson].”



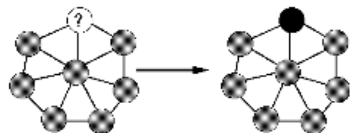
- Understandable – one module can be examined, reasoned about, developed, etc. in isolation



- 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



1.2. Good Design

- What's a design?
  - Express a idea to resolve a problem
  - Use for communications in the team members
- What's a good design?
  - Easy for Developing, Reading & Understanding
  - Easy for Communication
  - Easy for Extending (add new features)
  - Easy for Maintenance

Two general design issues

- *Cohesion* – why are sub-modules (like methods) placed in the same module? Usually to collectively form an ADT
- *Coupling* – what is the dependence between modules? Reducing the dependences (which come in many forms) is desirable

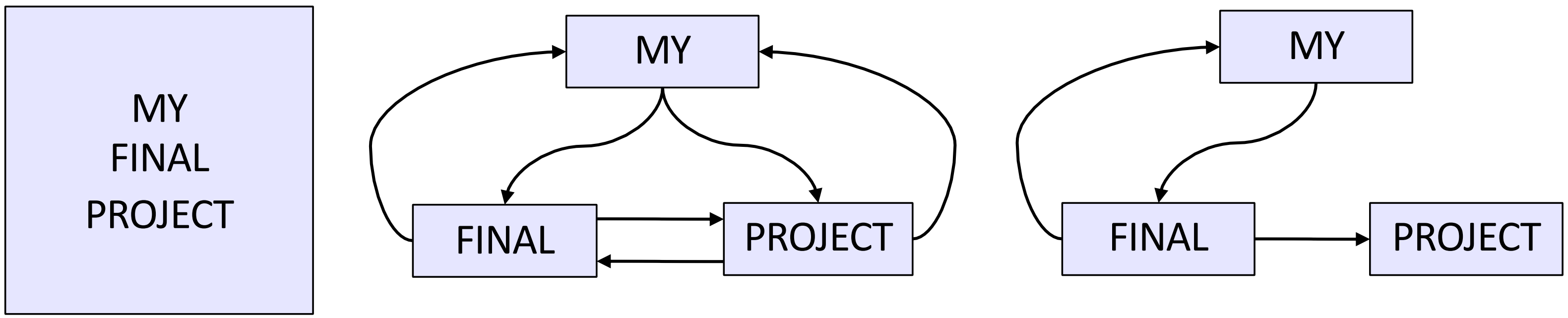
Cohesion

- The most common reason to put elements – data and behavior – together is to form an ADT
  - Sometimes may be other reasons, e.g. performance reasons: place together all code to be run upon initialization of a program
- The common design objective of separation of concerns suggests a module should address a single set of concerns
  - Should Item/DiscountItem know about added discount for purchasing 20+ items?
  - Should ShoppingCart know about bulk pricing?
  - Should BinarySearch know the type of the objects it is sorting?

Coupling

Roughly, the more coupled k modules are, the more one needs to think of them as a single, larger module

- How are modules dependent on one another?
  - Statically (in the code)? Dynamically (at run-time)? And more
  - Ideally, split design into parts that don't interact much



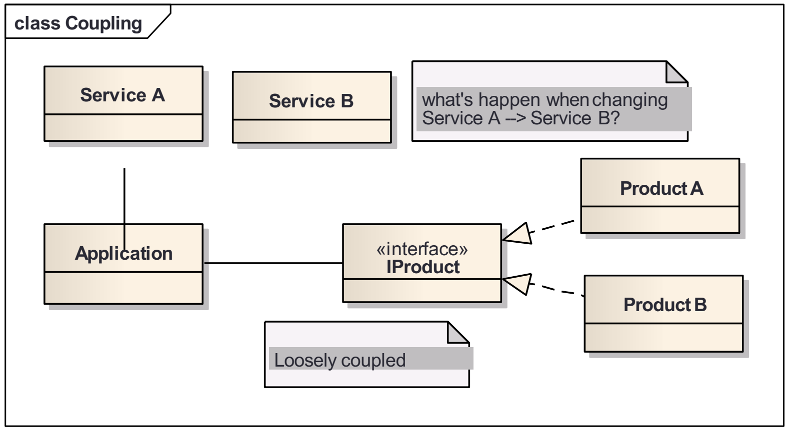
An application — *A poor decomposition (parts strongly coupled)* — *A better decomposition (parts weakly coupled)*

- An artist's rendition – to really assess coupling one needs to know what the arrows are, etc.

Cohesion and Coupling

❑ Coupling

Coupling or ***Dependency*** is the degree to which each program 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.

Good design

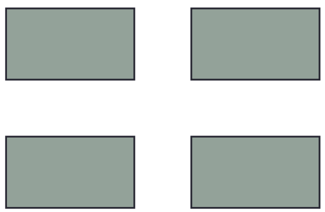
- Easy for Developing, Reading & Understanding
- Easy for Communication
- Easy for Extending (add new features)
- Easy for Maintenance

➔ “Loose coupling and high cohesion”

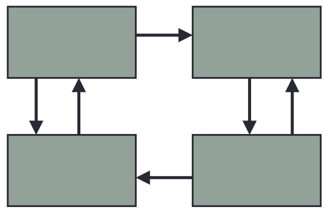
Content

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

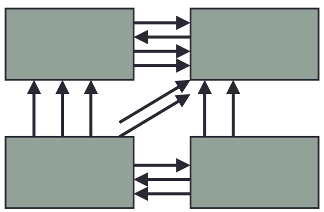
Coupling: Degree of dependence among components



No dependencies



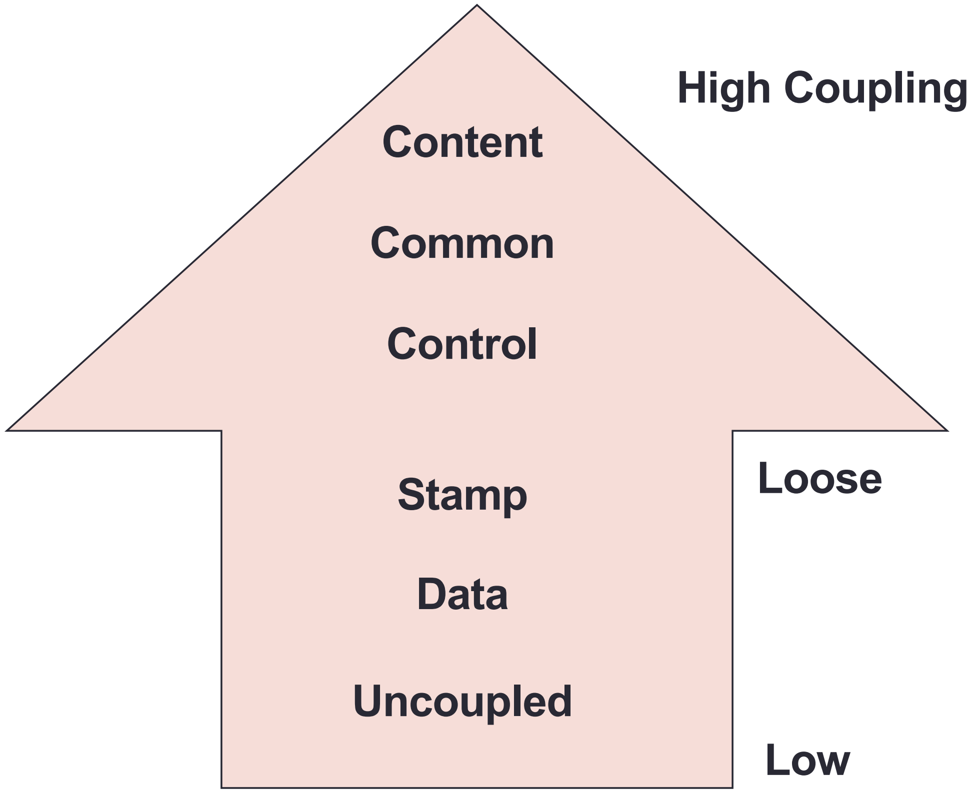
Loosely coupled-some dependencies



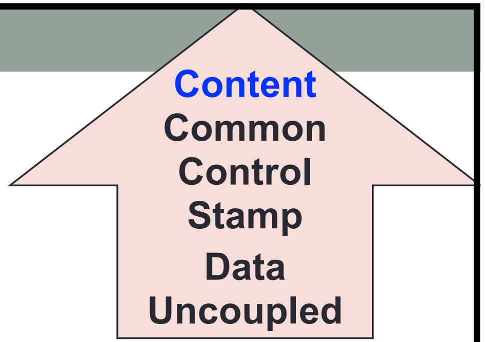
Highly coupled-many dependencies

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

Range of Coupling



2.1. Content coupling



- Definition: One component references contents of another
- Example:
  - Component directly modifies another's data
  - Component refers to local data of another component in terms of numerical displacement
  - Component modifies another's code, e.g., jumps into the middle of a routine

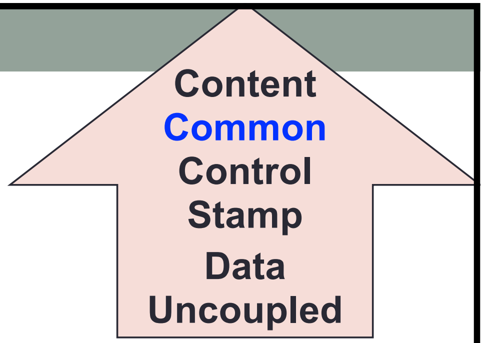
Exercise of Content Coupling

Part of program handles lookup for customer.

When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data

=> How to improve?

2.2. Common Coupling



- Definition: Two components share data
  - Global data structures
  - Common blocks
- 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

Exercise of Common Coupling

- Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks.
- Each source process writes directly to global data store. Each sink process reads directly from global data store.

=> How to improve?

2.3. Control Coupling

- Definition: Component passes control parameters to coupled components.
- 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 1

- Acceptable: Module p calls module q and q passes back flag that says it cannot complete the task, then q is passing data
- Not Acceptable: Module p calls module q and q passes back flag that says it cannot complete the task and, as a result, writes a specific message.

Exercise 2 – 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**.

Exercise 3 – Control Coupling

- In your video store, you might eventually create a method like this:
  - **editCustomer(int whatKind, Customer customer)** where **whatKind** takes on the values **RETAIL**, or **AGENCY**

2.4. Stamp Coupling

- Definition: Component passes a data structure to another component that does not have access to the entire structure.
- 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.

Example of Stamp Coupling

Customer billing system

The print routine of the customer billing accepts a customer data structure as an argument, parses it, and prints the name, address, and billing information.

=> How to improve?

2.5. Data Coupling

- Definition: Two components are data coupled if there are homogeneous data items.
- Every argument is simple argument or data structure in which all elements are used
- Good, if it can be achieved.
- Easy to write contracts for this and modify component independently.

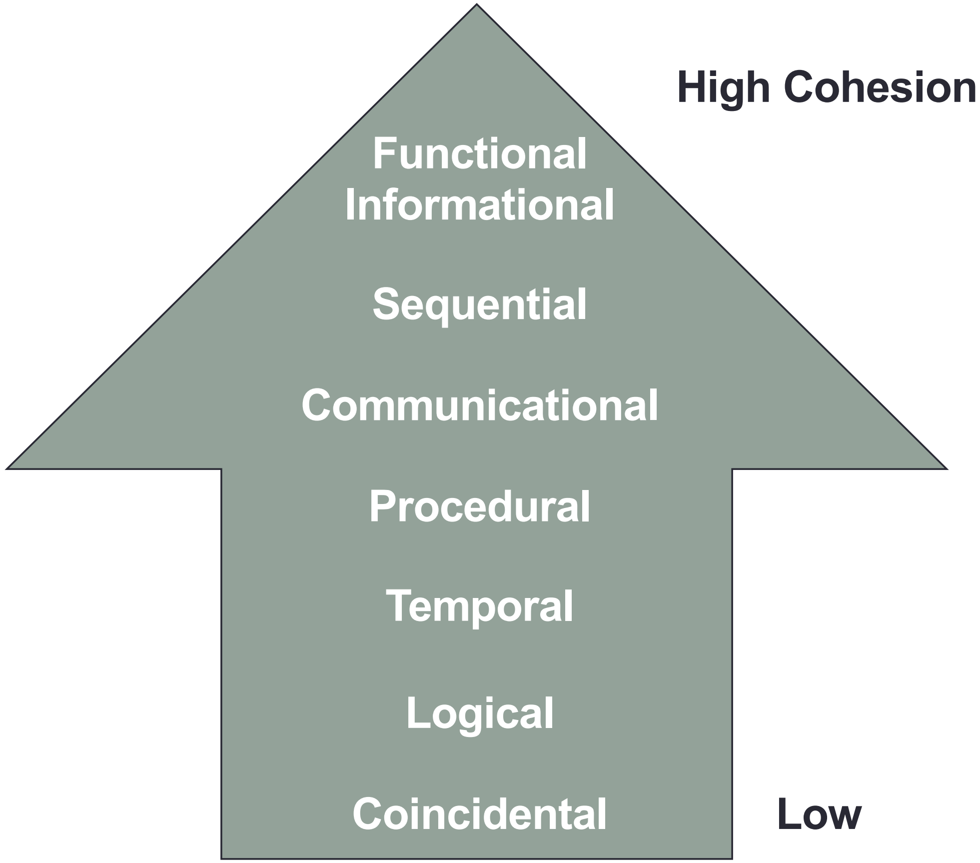
Content

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

3. Cohesion

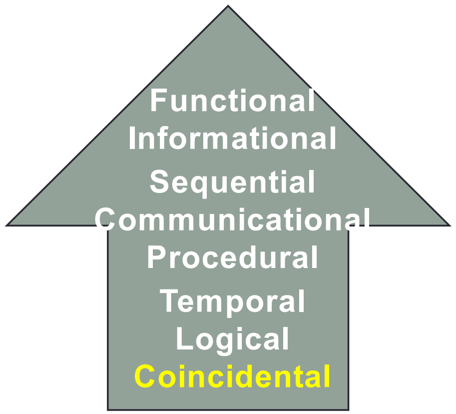
- Definition: The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.
- Internal glue with which component is constructed
- All elements of component are directed toward and essential for performing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

Range of Cohesion



3.1. Coincidental Cohesion

- Definition: Parts 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

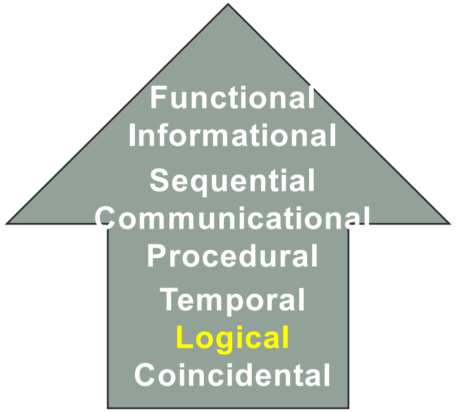


Example

- Print next line
- Reverse string of characters in second argument
- Add 7 to 5th argument
- Convert 4th argument to float

3.2. Logical Cohesion

- Definition: Elements of component are related logically and not functionally.
- Several logically related elements are in the the same component and one of the elements is selected by the client component.

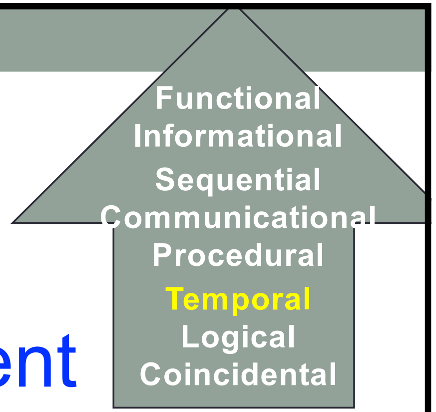


Example of Logical Cohesion

- 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.

- => How to improve?

3.3. Temporal Cohesion



- Definition: Elements of a component are related by timing.
- 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.

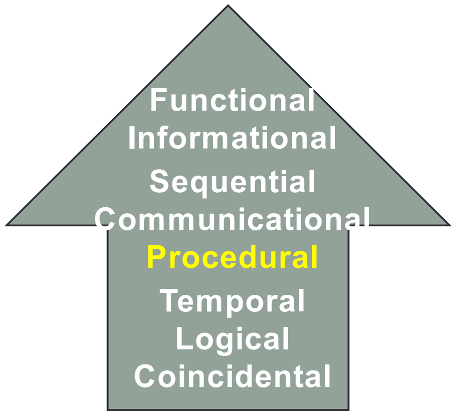
Example of Temporal Cohesion

- A system initialization routine: this routine 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 improve?

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

...

Read part number from database

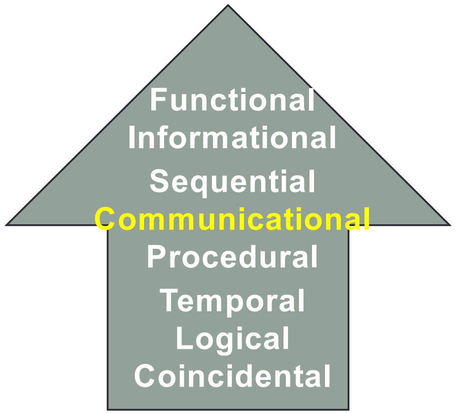
update repair record on maintenance file.

...

- May be useful to abstract the intent of this sequence. Make the data base and repair record components handle reading and updating. Make component that handles more abstract operation.

3.5. Communicational Cohesion

- Definition: Module performs a series of actions related by a sequence of steps to be followed by the product and all actions are performed on the same data



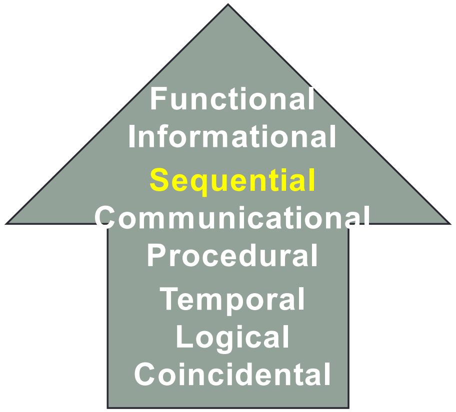
Example

- Update record in data base and send it to the printer.

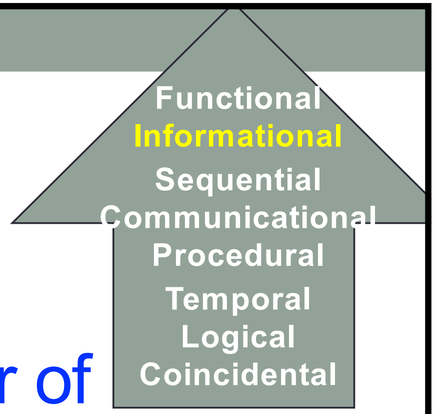
- database.Update (record).
- record.Print().

3.6. Sequential Cohesion

- The output of one component is the input to another.
- Occurs naturally in functional programming languages
- Good situation



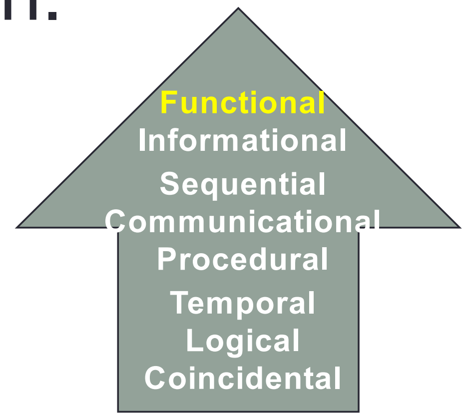
3.7. Informational Cohesion



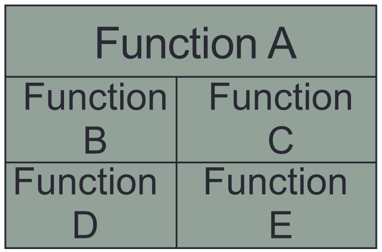
- Definition: Module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data.
- Different from logical cohesion
  - Each piece of code has single entry and single exit
  - In logical cohesion, actions of module intertwined
- ADT and object-oriented paradigm promote

3.8. Functional Cohesion

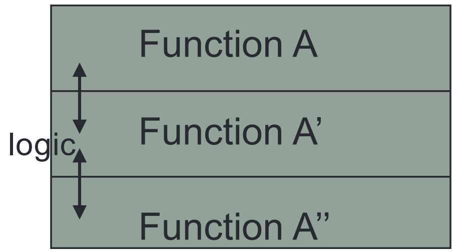
- Definition: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation.



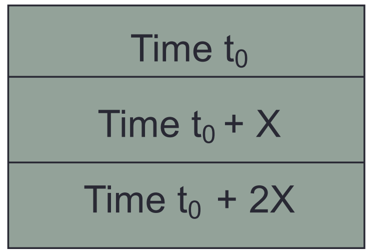
Examples of Cohesion



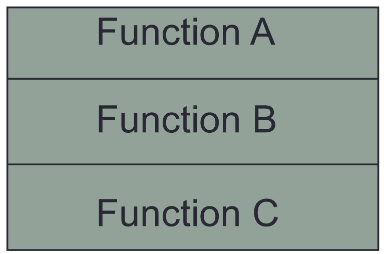
Coincidental
Parts unrelated



Logical
Similar functions

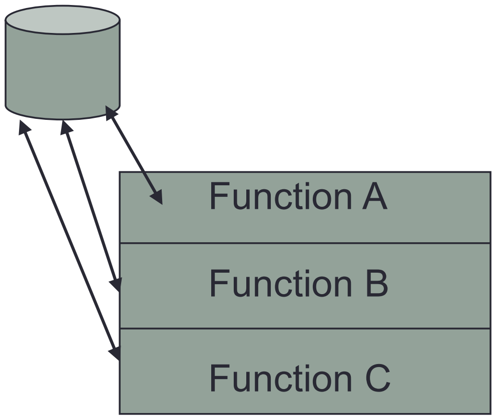


Temporal
Related by time

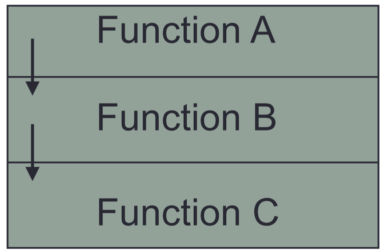


Procedural
Related by order of functions

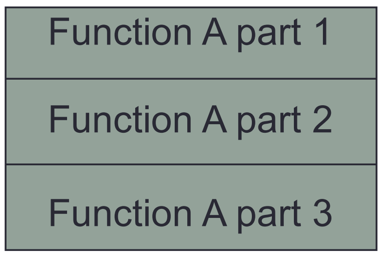
Examples of Cohesion-2



Communicational
Access same data



Sequential
Output of one is input to another



Functional
Sequential with complete, related functions

Different kinds of dependences

- Aggregation – "is part of" is a field that is a sub-part
  - Ex: A car has an engine
- Composition – "is entirely made of" has the parts live and die with the whole
  - Ex: A book has pages (but perhaps the book cannot exist without the pages, and the pages cannot exist without the book)
- Subtyping – "is-a" is for substitutability
- Invokes – "executes" is for having a computation performed
- In other words, there are lots of different kinds of arrows (dependences) and clarifying them is crucial

Law of Demeter
Karl Lieberherr ⓘ and colleagues

- Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts – a question of coupling
- Or… "only talk to your immediate friends"
- Closely related to representation exposure and (im)mutability
- Bad example – too-tight chain of coupling between classes

```
general.getColonel().getMajor(m).getCaptain(cap)
     .getSergeant(ser).getPrivate(name).digFoxHole();
```

- Better example

```
general.superviseFoxHole(m, cap, ser, name);
```

An object should only send messages to …
(More Demeter)

Guidelines: not strict rules! But thinking about them will generally help you produce better designs

- itself (`this`)
- its instance variables
- its method's parameters
- any object it creates
- any object returned by a call to one of `this`'s methods
- any objects in a collection of the above

- notably absent: objects returned by messages sent to other objects

Coupling is the path to the dark side

- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering

- Once you start down the dark path, forever will it dominate your destiny, consume you it will

God classes

- *God class*: a class that hoards too much of the data or functionality of a system
  - Poor cohesion – little thought about why all of the elements are placed together
  - Only reduces coupling by collapsing multiple modules into one (and thus reducing the dependences between the modules to dependences within a module)
- A god class is an example of an *anti-pattern* – it is a known bad way of doing things