Coupling and Cohesion

Software Engineering

Module: Definition

- A logical collection of related program entities
- Not necessarily a physical concept, e.g., file, function, class, package
- Often requires multiple program entities to express:
  - Linked list module may require many class, e.g., list, node, iterators, etc.

Why Use Modules?

- Simplify testing
- Increase program understanding
- Increase reuse
- Reduce maintenance costs for fixes and enhancements
- Permit replacement

Desired Interaction

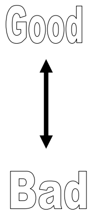
- Minimize external module interaction
  - modules can be used independently
  - easier to test
  - easier to replace
  - easier to understand
- Maximize internal module interaction
  - easier to understand
  - easier to test

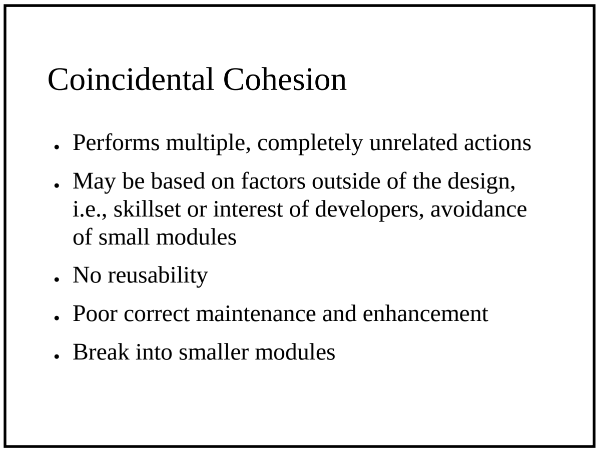
Characteristics

- *Cohesion* – Internal interaction of the module. Crisp abstraction of purpose
- *Coupling* – External interaction of the module with other modules
- *Action* – Behavior of the module. What it does
- *Logic* – How it performs the behavior. Algorithm used
- *Context* – Specific usage of the module

Cohesion

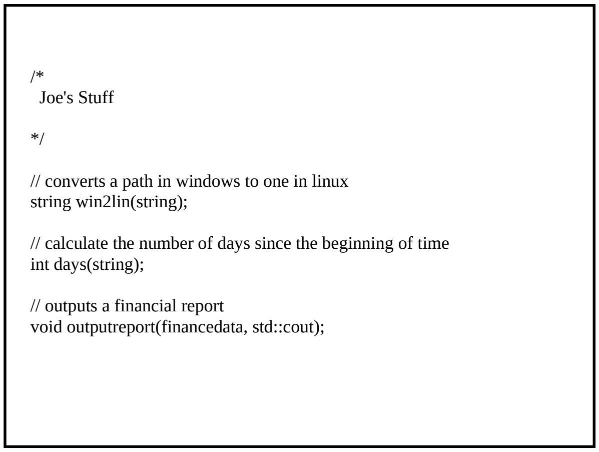
- In order from good (high) to bad (low)
  - Informational
  - Functional
  - Communicational
  - Procedural
  - Temporal
  - Logical
  - Coincidental



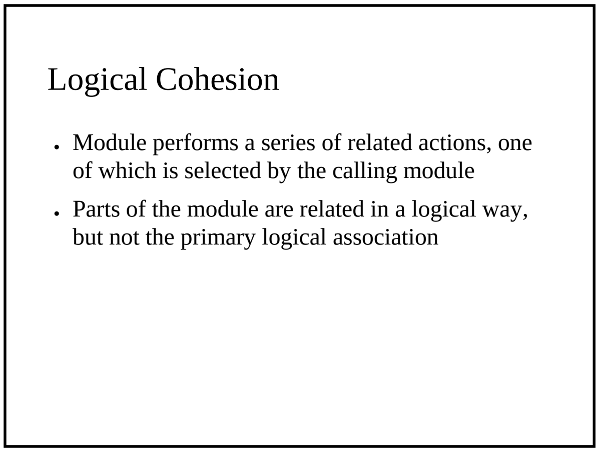


Coincidental Cohesion

- Performs multiple, completely unrelated actions
- May be based on factors outside of the design, i.e., skillset or interest of developers, avoidance of small modules
- No reusability
- Poor correct maintenance and enhancement
- Break into smaller modules

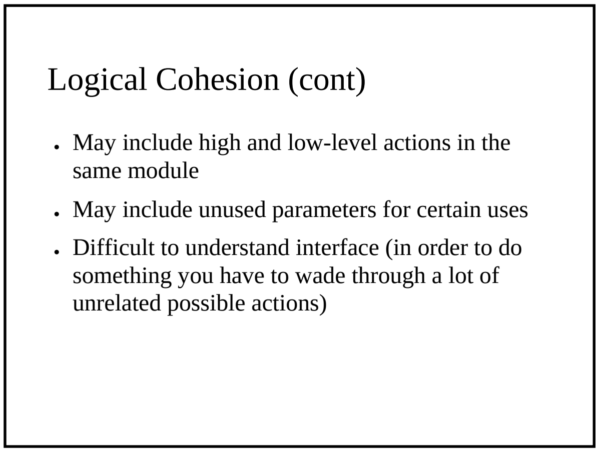


```
/*
  Joe's Stuff

*/

// converts a path in windows to one in linux
string win2lin(string);

// calculate the number of days since the beginning of time
int days(string);

// outputs a financial report
void outputreport(financedata, std::cout);
```



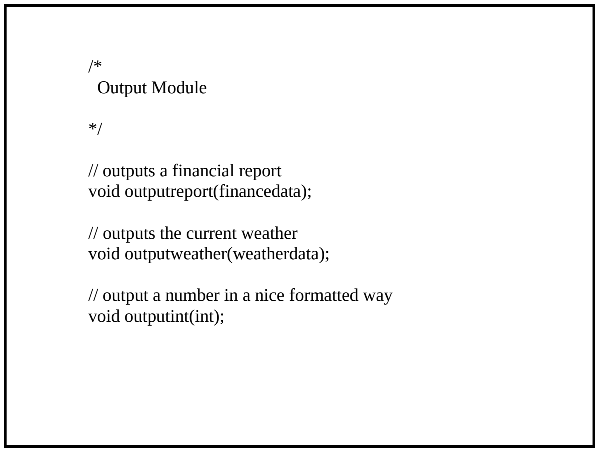
Logical Cohesion

- Module performs a series of related actions, one of which is selected by the calling module
- Parts of the module are related in a logical way, but not the primary logical association

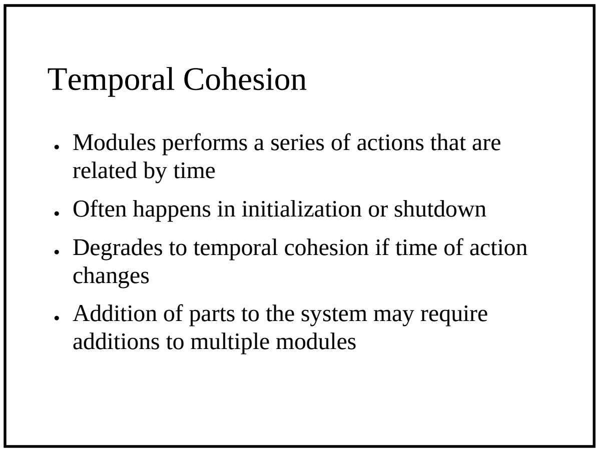


Logical Cohesion (cont)

- May include high and low-level actions in the same module
- May include unused parameters for certain uses
- Difficult to understand interface (in order to do something you have to wade through a lot of unrelated possible actions)

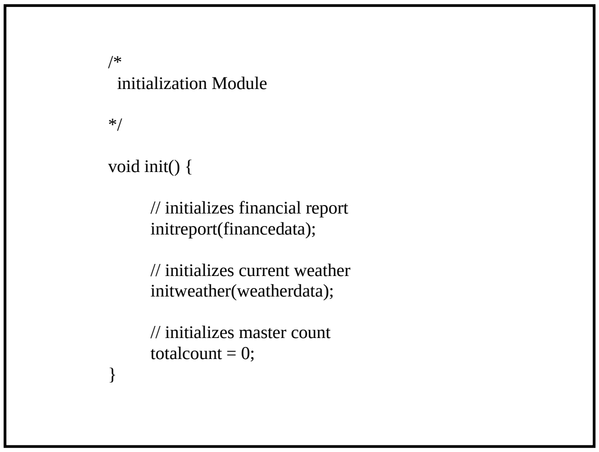


```
/*
  Output Module

*/

// outputs a financial report
void outputreport(financedata);

// outputs the current weather
void outputweather(weatherdata);

// output a number in a nice formatted way
void outputint(int);
```

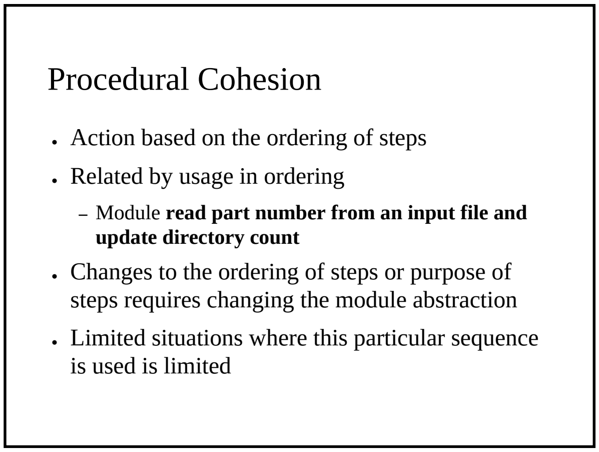


Temporal Cohesion

- Modules performs a series of actions that are related by time
- Often happens in initialization or shutdown
- Degrades to temporal cohesion if time of action changes
- Addition of parts to the system may require additions to multiple modules

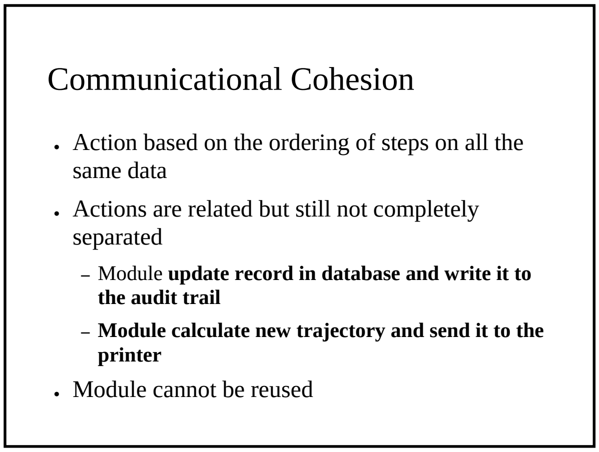


```
/*
  initialization Module

*/

void init() {

        // initializes financial report
        initreport(financedata);

        // initializes current weather
        initweather(weatherdata);

        // initializes master count
        totalcount = 0;
}
```



Procedural Cohesion

- Action based on the ordering of steps
- Related by usage in ordering
  - Module **read part number from an input file and update directory count**
- Changes to the ordering of steps or purpose of steps requires changing the module abstraction
- Limited situations where this particular sequence is used is limited



Communicational Cohesion

- Action based on the ordering of steps on all the same data
- Actions are related but still not completely separated
  - Module **update record in database and write it to the audit trail**
  - **Module calculate new trajectory and send it to the printer**
- Module cannot be reused

Functional Cohesion

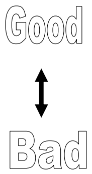
- Module that performs a single action or achieves a single goal
- Maintenance involves the entire single module
- Very reusable because the module is completely independent in action of other modules
- Can be replaced easily

Information Cohesion

- Performs a number of actions
- Each action has its own entry point and independent code
- All actions are performed on a shared data structure
- Object-Oriented

Coupling

- In order from good (low) to bad (high)
  - Data Coupling
  - Stamp Coupling
  - Control Coupling
  - Common Coupling
  - Content Coupling



Good

Bad

Content Coupling

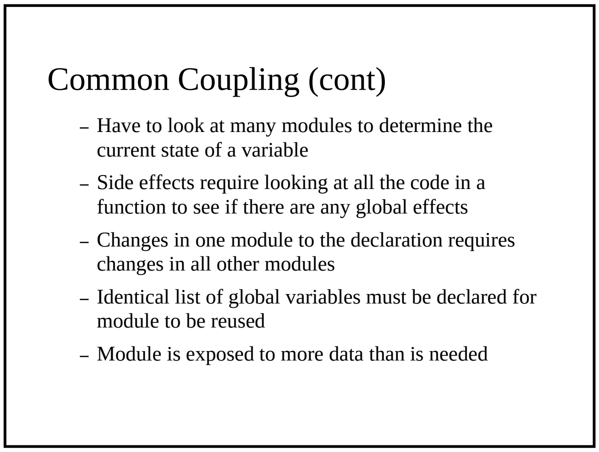
- A module directly references the content of another module
  - Module ***p*** modifies a statement of module ***q***
  - ***Module p refers to local data of module q (in terms of a numerical displacement)***
  - ***Module p branches to a local label of module q***

Content Coupling (cont)

- Content coupled modules are inextricably interlinked
  - Change to module **p** requires a change to module **q** (including recompilation)
  - Reusing module **p** requires using module **q** also

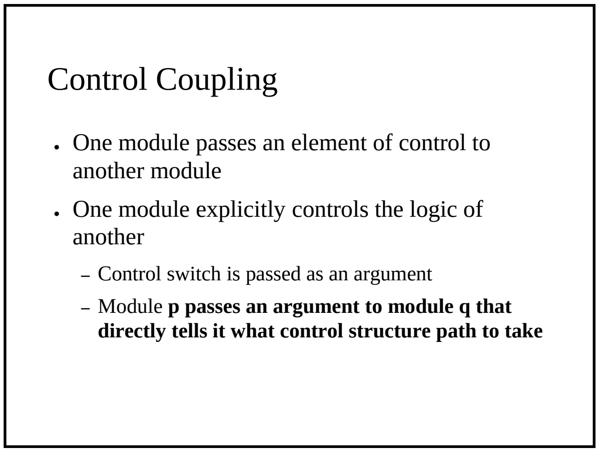
Common Coupling

- Using global variables
- All modules have read/write access to a global data block
- Modules exchange data using the global data block (instead of arguments)
- Single module with write access where all other modules have read access is not common coupling



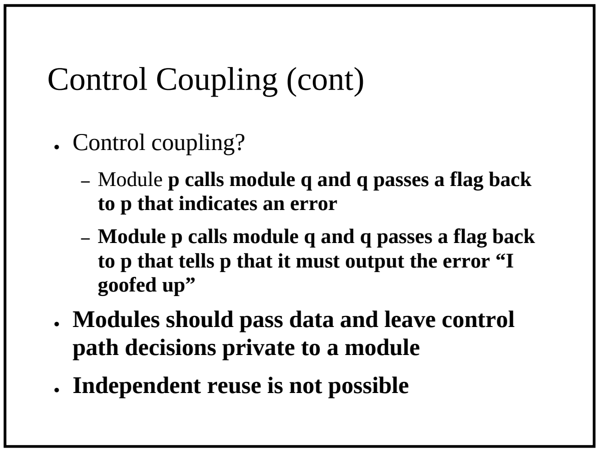
Common Coupling (cont)

- Have to look at many modules to determine the current state of a variable
- Side effects require looking at all the code in a function to see if there are any global effects
- Changes in one module to the declaration requires changes in all other modules
- Identical list of global variables must be declared for module to be reused
- Module is exposed to more data than is needed



Control Coupling

- One module passes an element of control to another module
- One module explicitly controls the logic of another
  - Control switch is passed as an argument
  - Module **p passes an argument to module q that directly tells it what control structure path to take**



Control Coupling (cont)

- Control coupling?
  - Module **p calls module q and q passes a flag back to p that indicates an error**
  - **Module p calls module q and q passes a flag back to p that tells p that it must output the error “I goofed up”**
- **Modules should pass data and leave control path decisions private to a module**
- **Independent reuse is not possible**

Stamp Coupling

- One module passes more data then needed to another module
  - void swap(int v[], int i, int j);
  - double calcsalary(Employee& e);
- Often involves records (structs) with lots of fields
- Entire record is passed, but only a few fields are used
- Efficiency considerations?

Data Coupling

- Only required data is passed from one module to another
- All arguments are homogenous data items
  - simple data type
  - complex data type, but all parts are used
- Holy grail
- Allows for reuse, maintenance, understanding, etc.