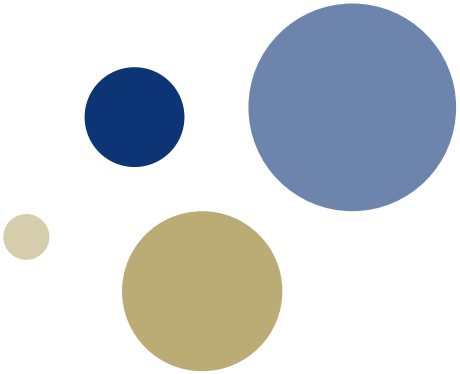
NTNU – Trondheim
Norwegian University of
Science and Technology



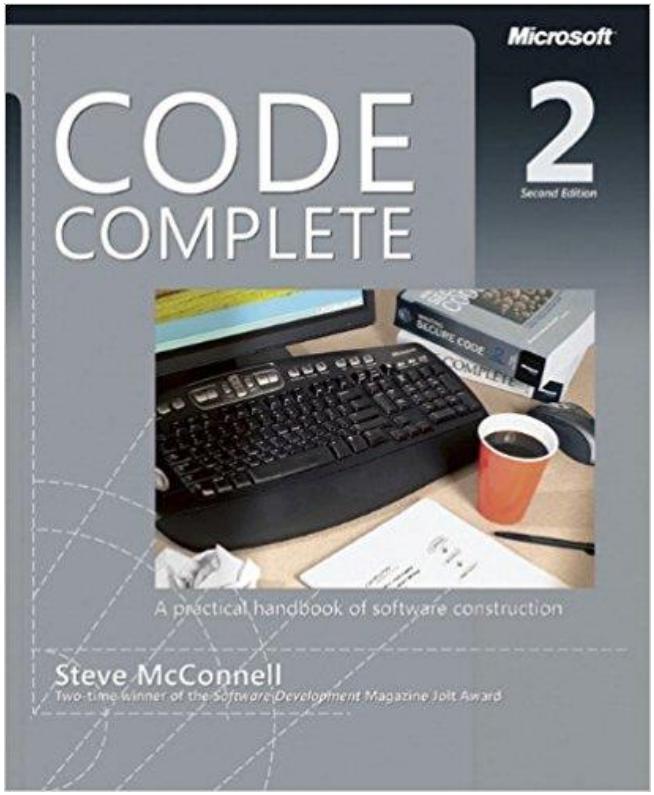
TTK4145 – Real-time Programming

Lecture 7 – Code Quality

Code Quality

What is Code Quality?

Which metric can we use to determine it?



Selected chapters available on Blackboard

Replace “Classes” with “Modules” and skip the subchapter about “Inheritance”.

Learning goals: Code Quality

- Be able to write software following selected Code Complete Checklists for modules, functions, variables and comments

- Be able to criticize program code based on the same checklists

Exam question on Code Quality from Continuation 2014

As the program grows it turns out that the combination of `printName()` and `printAddress()` occurs many places in the program (when printing name tags, letterheads, business cards, envelopes,...), and we decide to make the function `printNameAndAddress()` as a part of the module:

```
void printNameAndAddress(int i){
     sem_wait(personSem);
     printName(i);
     printAddress(i);
     sem_signal(personSem);
}
```

Look at the module interface; What do you think about this decision in a code quality perspective?

Exam question on Code Quality from Continuation 2014

Below is an excerpt from a C header file defining a Person module:

```
#ifndef PERSON_H
#define PERSON_H

typedef struct {
     char * firstName;
     char * lastName;
     char * street;
     int streetNumber;
} TPerson;

void reallocateArray(int newSize);

TPerson ** getArray();

void printName(int personNumber);
void printAddress(int personNumber);
void printNameAndAddress(int personNumber);
...
#endif
```

Criticize, from a code quality perspective, the inclusion of the type `TPerson`, and the two functions `reallocateArray()` and `getArray()`.

Exam question on Code Quality from Continuation 2014

In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is “cost”. Criticize (concisely, with bullet points) the design.

```
#ifndef lift_cost_h
#define lift_cost_h

int calculateCost(int currentFloor, int direction, int orderedFloor, int orderedDirection);

int downCost[MAX_ELEVATORS][N_FLOORS];
int upCost[MAX_ELEVATORS][N_FLOORS];

void fillCostArrays();
void clearCosts(void);

int lowestCostFloor(int elevator);
int lowestCostDirection(int elevator);

int findBestElevator(int floor, int direction);

void designateElevators();
void clearDesignatedElevator();

int designatedElevator[N_FLOORS][2];

#endif
```

What is Code Quality?

- Multiple definitions exist
  - Number of bugs per line of code
  - Number of lines per function
  - Average time to identify and fix bugs
  - Etc.

- In this course, your code has high Code Quality if it is maintainable

- Maintainable code can easily be
  - Read (the formatting is consistent and clear)
  - Understood (the intended functionality is easy to deduce)
  - Changed (local changes does not affect the entire program)
  - Expanded (flexible to allow for adding features, within reason)
  - Debugged (bugs are easy to identify, and quick to fix)
  - Documented (the code minimizes the need for external documentation)

Threshold for maintainability

- Every programmer has a threshold in terms of number of lines of code
- The number tends to increase with experience, but there is always an upper limit for each person
- Once your project exceeds your personal threshold, development becomes exponentially more difficult
- Leads to loss of Maintainability, and the Flying Spaghetti Monster usually decides to make a nest in your code base
- The elevator project might be close to (or exceed) your threshold.
- How can we ensure maintainability despite this?

The Flying Spaghetti Monster

The natural predator of maintainability

Modules

Using abstractions to simplify

What is a Module?

- A module is a partition of your code
  - Classes in C++
  - Packages in Ada
  - Header files in C
  - Lines of codes
  - Functions
  - Libraries
  - Etc.
- Modules increase maintainability by decreasing complexity, as long as you can
  - Maintain a module without knowing the rest of the system.
  - Use a module without knowing its internals.
- Modules allows for scalability through composition
  - Building super-modules from sub-modules
- A module acts as an abstraction for it’s underlying implementation

What is an Abstraction?

- The ability to view a complex operation in a simplified form
  - Higher-level programming languages compared to assembly
  - Mathematical functions such as `cos(), exp(), sqrt().`
- Code Complete introduces Abstract Data Types (ADTs) as a building block for modules
  - A collection of data and operations that work on that data
  - Examples such as graphics window, file and file operations, insurance-rates table
- The supplied drivers in the Elevator Project on GitHub is a good example of an abstraction
  - `void elevator_hardware_set_motor_direction(elevator_hardware_motor_direction_t dirn)`
  - `int elevator_hardware_get_button_signal(elevator_hardware_button_type_t button, int floor);`
- Abstractions can be interacted with through their Interface

What is an Interface?

- Interfaces are the set of data and/or routines the module exposes to the outside
- The interface hides implementation details of the module through a higher level abstraction
- All communication in and out of the module goes through the interface, defines the boundary of the module
- Common interfaces
  - Public member/methods in classes in C++ (implementation in private)
  - Header – files in C (implementation in .c)
  - Package specification in Ada (implementation in Body)
  - Routines

Interface considerations

- Interfaces should be complete and minimal
- Services often comes in pairs with their opposites: put/get, start/stop, init/shutdown, on/off...
- Choose the right level of abstraction
- Keep the level of abstraction consistent
- Make assumptions explicit in the code (”programmatic” rather than ”semantic”)
- Beware of erosion of the abstraction over time
- If your interface ends up not a great abstraction: Redesign!

Bad Interface (Lack of cohesion)

```
#ifndef COMMANDSTACK_H
#define COMMANDSTACK_H

void InitializeCommandStack();
void PushCommand( Command command );
Command PopCommand();
void ShutdownCommandStack();
void InitializeReportFormatting();
void FormatReport( Report report );
void PrintReport( Report report );
void InitializeGlobalData();
void ShutdownGlobalData();

#endif
```

Bad Interface
(Mixed level of abstraction)

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

void AddEmployee( Employee employee );
void RemoveEmployee( Employee employee );
Employee NextItemInList();
Employee FirstItem();
Employee LastItem();

#endif
```

Bad Interface
(Erosion of abstraction)

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

FullName GetName() const;
Address GetAddress() const;
PhoneNumber GetWorkPhone() const;
...
bool IsJobClassificationValid( JobClassification jobClass );
bool IsZipCodeValid( Address address );
bool IsPhoneNumberValid( PhoneNumber phoneNumber );

SqlQuery GetQueryToCreateNewEmployee() const;
SqlQuery GetQueryToModifyEmployee() const;
SqlQuery GetQueryToRetrieveEmployee() const;

#endif
```

Module Interfaces Checklist: Abstraction

- Does the module have a central purpose?
- Is the module well named, and does its name describe its central purpose?
- Does the module's interface present a consistent abstraction?
- Does the module's interface make it obvious how you should use the module?
- Is the module interface abstract enough that you don't have to think about how its services are implemented? Can you treat the module as a black box?
- Are the module's services complete enough that other modules don't have to meddle with its internal data?
- Has unrelated information been moved out of the module?
- Have you thought about subdividing the module into component modules, and have you subdivided it as much as you can?
- Are you preserving the integrity of the module's interface as you modify the module?

Module Interfaces Checklist: Encapsulation

- Does the module minimize accessibility to its data members?
- Does the module avoid exposing member data?
- Does the module hide its implementation details from other modules as much as the programming language permits?
- Does the module avoid making assumptions about its users?
- Is the module independent of other modules? Is it loosely coupled?

Module Interfaces Checklist: Other Implementation Issues

- Does the module contain about seven data members or fewer?
- Does the module minimize direct and indirect routine calls to other modules?
- Does the module collaborate with other modules only to the extent absolutely necessary?
- Is all member data initialized appropriately?

Module Interfaces Checklist: Language-Specific Issues

- Have you investigated the language-specific issues for modules in your specific programming language?

Key Points

- Module interfaces should provide a consistent abstraction.

- A module interface should hide something – a system interface, a design decision, or an implementation detail.

- Modules are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

Routines

The single greatest invention in computer science

What is a routine?

- An individual piece of code invokable for a single purpose that can be called by name

- An synonym for any of the following
  - Procedures
    - No return values
    - IN and OUT parameters (only side-effects)
    - Example: IO operations
  - Functions
    - A single (or few) return value(s)
    - Only IN parameters (no side-effects)
    - Example: Mathematical operations
  - Methods
    - Usually used in object-oriented programming, a function or procedure that is executed in the context of an object

Reasons to Create a Routine

- Reduce complexity
- Introduce abstraction
- Avoid duplicate code
- Encapsulate/hide something
- Support subclassing
- Hiding sequences
- Hide pointer operations
- Improve portability
- Simplify tests
- Improve performance

What is a High-Quality Routine?

Let's examine what a high-quality routine is NOT:

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
      double & estimRevenue, double ytdRevenue, int screenX, int screenY,
      COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
      int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
      inputRec.revenue[i] = 0;
      inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
      }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
      for ( i = 0; i < 12; i++ )
            profit[i] = revenue[i] - expense.type1[i];
      }
else if ( expenseType == 2 ) {
      profit[i] = revenue[i] - expense.type2[i];
      }
else if ( expenseType == 3 )
      profit[i] = revenue[i] - expense.type3[i];
      }
}
```

High-Quality Routines

- The routine has a good name
- The routine has a single, clearly defined purpose (high cohesion)
- The parameter list uses a consistent input-modify-output order for its parameters
- All parameters and variables are utilized, avoid reading and writing to global variables
- Input parameters are not repurposed inside the routine as working variables
- Parameter list includes less than 7 parameters
- The layout is readable and consistent, easy to see what the does routine does by reading it
- The routine is defensive, it protects itself from bad usage and bad parameters.
- All assumptions are programmatic rather than semantic
- The routine avoids using magic numbers, rather uses named constants

A good routine name

- Explicitly summarizes everything a routine does
- Is specific (avoid ”handle”, ”perform”, ”do”, ”process” etc.)
  - A function: describe the return value -> `printer_isready()`
  - A procedure: A verb and object - `print_document()`
- Is as long as necessary
- Uses opposites precisely
  - `start/stop, begin/end`
  - NOT `start/end, begin/stop`
- Standardizes naming and abbreviations
  - Establish a convention, avoid the following:

```
employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()
```

Routine Size

- How large can it be?
  - (...to not represent a maintenance problem in itself...)
  - Around 100-200 lines
- How small can it be?
  - (...and still represent useful abstraction, encapsulate something worth encapsulating, ...)
  - Can be 1 line, as long as the routine provides an abstraction
  - Set and Get functions

Routines Checklist: Big-Picture Issues

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- Does the routine's name describe everything the routine does?
- Have you established naming conventions for common operations?
- Does the routine have strong, functional cohesion-doing one and only one thing and doing it well?
- Do the routines have loose coupling-are the routine's connections to other routines small, intimate, visible, and flexible?
- Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

Routines Checklist: Parameter-Passing Issues

- Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- Are interface assumptions documented?
- Does the routine have seven or fewer parameters?
- Is each input parameter used?
- Is each output parameter used?
- Does the routine avoid using input parameters as working variables?
- If the routine is a function, does it return a valid value under all possible circumstances?

Routines: Key Points

- The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.
- Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
- You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.
- The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.
- Functions should be used only when the primary purpose of the function is to return the specific value described by the function's name.
- Careful programmers use macro routines with care and only as a last resort.

Variables Names

You can control a Demon by knowing its True Name

Variable names aid maintainability

- Poor variable names

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

- Good variable names

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

A good variable name

- Most important is that the name fully and accurately describe the entity the variable represents
- Length should be between 10 to 16 characters (depending on scope)
  - Local variables or loop variables can use shorter names
- Be conscious of the specific type of data you're naming
  - Boolean variables should have names that imply true/false
  - Temporary variables something other than `temp, x,` etc.
  - Consider naming loop variables, if the loops are long
  - Prefix enums with enum type, `Color_Red`
  - Constants should be named after the entity it represents, not the value
- Use opposites precisely, as with routines, naming conventions for opposites helps consistency and readability.
- Use naming conventions consistently to differ between
  - `TypeName`
  - `localVariable`
  - `routineParameter`
  - `RoutineName()`
  - `CONSTANT`
  - `g_GlobalVariable`
  - `p_Pointer`

Naming Variables Checklist: General Naming Considerations

- Does the name fully and accurately describe what the variable represents?
- Does the name refer to the real-world problem rather than to the programming-language solution?
- Is the name long enough that you don’t have to puzzle it out?
- Are computed-value qualifiers, if any, at the end of the name?
- Does the name use Count or Index instead of Num?

Naming Variables Checklist: Naming Specific Kinds of Data

- Are loop index names meaningful (something other than i, j, or k if the loop is more than one or two lines long or is nested)?
- Have all ”temporary” variables been renamed to something more meaningful?
- Are boolean variables named so that their meanings when they’re true are clear?
- Do enumerated-type names include a prefix or suffix that indicates the category - for example, Color for Color Red, Color Green, Color Blue, and so on?
- Are named constants named for the abstract entities they represent rather than the numbers they refer to?

Naming Variables Checklist: Naming Conventions

- Does the convention distinguish among local, class, and global data?
- Does the convention distinguish among type names, named constants, enumerated types, and variables?
- Does the convention identify input-only parameters to routines in languages that don't enforce them?
- Is the convention as compatible as possible with standard conventions for the language?
- Are names formatted for readability?

Naming Variables Checklist: Short Names

- Does the code use long names (unless it's necessary to use short ones)?
- Does the code avoid abbreviations that save only one character?
- Are all words abbreviated consistently?
- Are the names pronounceable?
- Are names that could be misread or mispronounced avoided?
- Are short names documented in translation tables?

Naming Variables Checklist:
Common Naming Problems: Have You Avoided...

- ...names that are misleading?
- ...names with similar meanings?
- ...names that are different by only one or two characters?
- ...names that sound similar?
- ...names that use numerals?
- ...names intentionally misspelled to make them shorter?
- ...names that are commonly misspelled in English?
- ...names that conflict with standard library routine names or with predefined variable names?
- ...totally arbitrary names?
- ...hard-to-read characters?

Variable Names: Key Points

- Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.
- Names should be as specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.
- Naming conventions distinguish among local, class, and global data. They distinguish among type names, named constants, enumerated types, and variables.
- Regardless of the kind of project you're working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.
- Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.
- Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time convenience

Self-documenting code

Code as if whoever maintains your program is a violent psychopath who knows where you live.

-Anonymous

Documentation

- External documentation
  - On large formal projects most of the documentation is outside the project
  - Tend to be at a high-level compared to the code
  - Low-level compared to problem definition, requirements, etc
- Internal documentation
  - Found within the program itself
  - The most detailed kind of documentation
  - Most likely to remain correct as code is modified
  - Main contributor is good programming style, NOT comments

Self-Documenting Code

- Rule 1: If something is not clear from the code: Improve Code!
- Choose better names, divide complex statements, move complex code into well-named functions; can you use layout or whitespace?
- Rule 2: If the code is still not clear: The code has a weakness! Improve it! Choose better abstractions.
- Comments add to the size of the project - comments must be maintained also.
- Use comments as headlines, to emphasize structure, give summaries or describe intent
- Make comments easy to maintain, avoid fancy styles
- Poor comments are worse than no comments

Types of comments

- Repeat of the Code
  - Useless, provides no additional information, only more reading
- Explanation of the Code
  - Sometimes useful, but rather a symptom that the code should be rewritten
- Marker in the Code
  - Indicates work left to be done, should never be in finished code
- Summary of the Code
  - Useful for others when they modify your code
- Description of the Code's Intent
  - Helps to clarify the programmer's intent, similar to summary
- Information that Cannot Be Expressed by the Code Itself
  - Copyright and other notices, notes about design, version numbers, reference to requirements, pointers to online references, etc

Good Commenting Technique Checklist : General

- Can someone pick up the code and immediately start to understand it?
- Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- Is the Pseudocode Programming Process used to reduce commenting time?
- Has tricky code been rewritten rather than commented?
- Are comments up to date?
- Are comments clear and correct?
- Does the commenting style allow comments to be easily modified?

Good Commenting Technique Checklist : Statements and Paragraphs

- Does the code avoid endline comments?
- Do comments focus on why rather than how?
- Do comments prepare the reader for the code to follow?
- Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- Are surprises documented?
- Have abbreviations been avoided?
- Is the distinction between major and minor comments clear?
- Is code that works around an error or undocumented feature commented?

Good Commenting Technique Checklist : Data Declarations

- Are units on data declarations commented?
- Are the ranges of values on numeric data commented?
- Are coded meanings commented?
- Are limitations on input data commented?
- Are flags documented to the bit level?
- Has each global variable been commented where it is declared?
- Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- Are magic numbers replaced with named constants or variables rather than just documented?

Good Commenting Technique Checklist : Control Structures

- Is each control statement commented?
- Are the ends of long or complex control structures commented or, when possible, simplified so that they don’t need comments?

Good Commenting Technique Checklist : Routines

- Is the purpose of each routine commented?
- Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

Good Commenting Technique Checklist : Files, Classes, and Programs

- Does the program have a short document, such as that described in the Book Paradigm, that gives an overall view of how the program is organized?
- Is the purpose of each file described?
- Are the author's name, e-mail address, and phone number in the listing?

Self-Documenting Code: Key Points

- The question of whether to comment is a legitimate one. Done poorly, commenting is a waste of time and sometimes harmful. Done well, commenting is worthwhile.
- The source code should contain most of the critical information about the program. As long as the program is running, the source code is more likely than any other resource to be kept current, and it's useful to have important information bundled with the code.
- Good code is its own best documentation. If the code is bad enough to require extensive comments, try first to improve the code so that it doesn't need extensive comments.
- Comments should say things about the code that the code can't say about itself - at the summary level or the intent level.
- Some commenting styles require a lot of tedious clerical work. Develop a style that's easy to maintain.

Keypoints to Exam questions on Code Quality from Continuation 2014

Keypoints to Exam question on Code Quality from Continuation 2014

As the program grows it turns out that the combination of `printName()` and `printAddress()` occurs many places in the program (when printing name tags, letterheads, business cards, envelopes,...), and we decide to make the function `printNameAndAddress()` as a part of the module:

```
void printNameAndAddress(int i){
     sem_wait(personSem);
     printName(i);
     printAddress(i);
     sem_signal(personSem);
}
```

Look at the module interface; What do you think about this decision in a code quality perspective?

- The interface is not minimal any more (which is a bad thing).
- Continuing this trend will lead to code duplication in the module. (More obscurely: We get dependencies between functions in the module which increases module complexity).
- But of course; we sometimes do make convenience functions, if the convenience is great enough.

Keypoints to Exam question on Code Quality from Continuation 2014

Below is an excerpt from a C header file defining a Person module:

```
#ifndef PERSON_H
#define PERSON_H

typedef struct {
      char * firstName;
      char * lastName;
      char * street;
      int streetNumber;
} TPerson;

void reallocateArray(int newSize);

TPerson ** getArray();

void printName(int personNumber);
void printAddress(int personNumber);
void printNameAndAddress(int personNumber);
...
#endif
```

Criticize, from a code quality perspective, the inclusion of the type `TPerson`, and the two functions `reallocateArray()` and `getArray()`.

- The type should not be a part of the module interface - breaks the encapsulation!
- returning the list in `getArray()` is *really* terrible! - breaks the encapsulation!
- `reallocateArray()` is inconsistent abstraction and breaks encapsulation revealing more of the implementation than should be necessary.

Keypoints to Exam question on Code Quality from Continuation 2014

In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is “cost”. Criticize (concisely, with bullet points) the design.

```
#ifndef lift_cost_h
#define lift_cost_h

int calculateCost(int currentFloor, int direction, int
orderedFloor, int orderedDirection);

int downCost[MAX_ELEVATORS][N_FLOORS];
int upCost[MAX_ELEVATORS][N_FLOORS];

void fillCostArrays();
void clearCosts(void);

int lowestCostFloor(int elevator);
int lowestCostDirection(int elevator);

int findBestElevator(int floor, int direction);

void designateElevators();
void clearDesignatedElevator();

int designatedElevator[N_FLOORS][2];

#endif
```

The main issue with this module is that this is not a module interface! It is (probably) a list of all functions in the module, including the variables!

- Ideally it should be clear from the interface exactly what the responsibility of the module is, and how it should be used correctly. It is not.
- There are (probably) nonobvious dependencies between the functions in the interface. (When do you need to call `clearCosts` for example?)
- The interface is not minimal. The functionalities of `calculateCost`, `findBestElevator`, `lowestCostFloor` and `lowestCostDirection` is overlapping. Also probably `clearCosts` and `fillCostArrays`.
- The abstraction is not consistent. The name “Cost” indicates that the module calculates or manages costs in some way. Either `calculateCost` or `findBestElevator` must be the main purpose of the module? But then there is the manipulation of some “`costArrays`” in addition - and keeping track of a “designatedElevator”?
- The data members are not encapsulated.
- (Having non-external variables in the headerfile like this is a bug, in addition to the fact that exporting the module’s data is bad form)

Possibly some of these thinks could have been mitigated by commenting, but this would still be a badly designed module interface.