SC313 004 Software Engineering

Chapter#10: Collaborative Software Development and Good Implementation Practices

By

Asst.Prof. Chitsutha Soomlek, Ph.D.

College of Computing, KKU

2/2567

February 3, 2025

Implementation

- Program code is created from designs.
- The main goal of the programmer is to translate the design into code that is correct, bug free, and maintainable.
- For OO, many of the classes-and perhaps the methods as well may already have been identified and defined by the time programming begins.

Agile and Non-Agile Approaches to Implementation

Agile

- Implementation is begun as soon as the first user story has been understood.
- Implementation is viewed not only as building the application but also as a process of understanding the requirements.
- As the implementation progresses, the process of refactoring is viewed as enabling developers to alter the design and implementation to suit evolving requirements.

Non-agile

- Requirements and a design (though not necessarily of the entire application) are in place when implementation begins.

Choosing a Programming Language

- The programming language selected for implementing a design is usually dictated by
  - the company
  - customer
  - the environment in which the application must run
- When the freedom exists to select a programming language, an identification and weighting of selection criteria can be used to assist the decision.

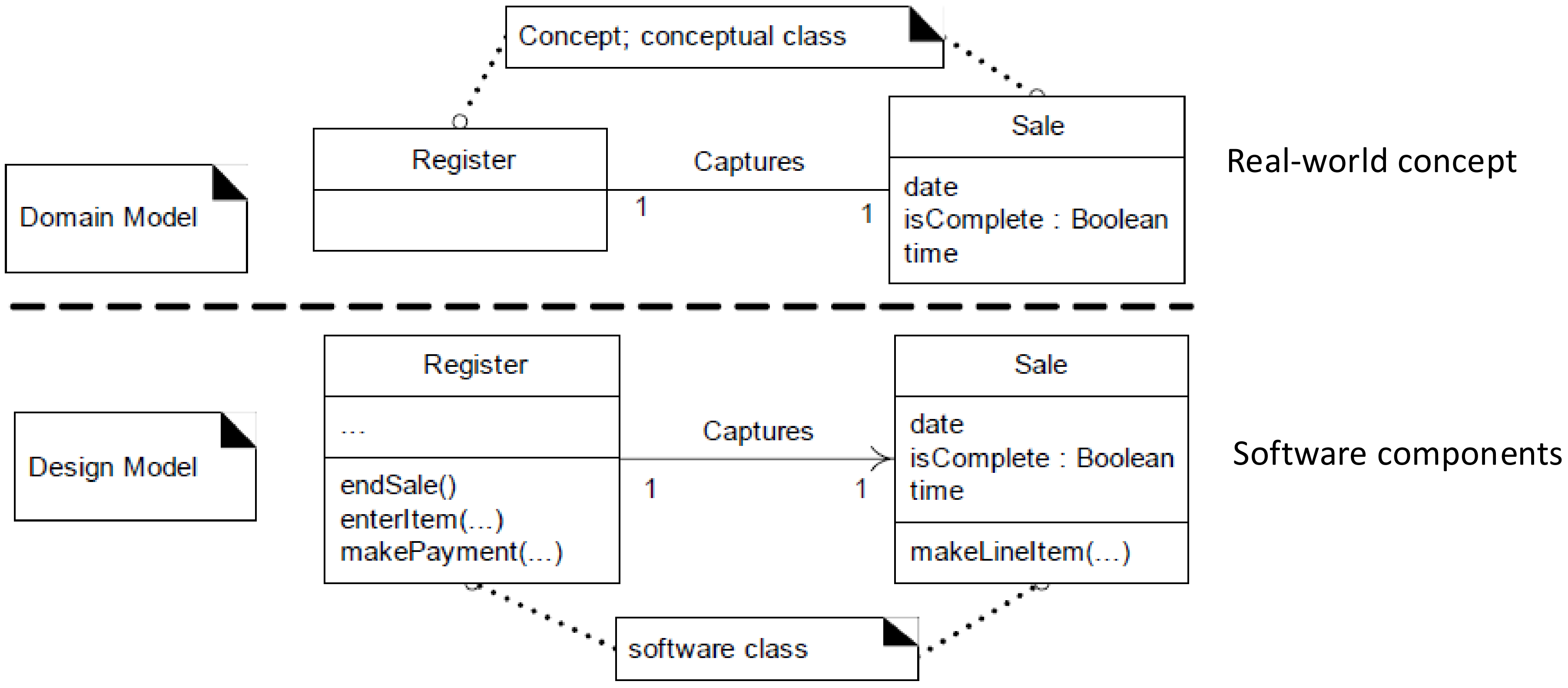
Selection Criteria

- Features of languages needed for the application constitute.
- The availability of useful libraries.
- The degree of software engineers' expertise in a language (and its weight is usually high ☺).

Identifying Classes

- Each class has one of three origins:
  - **Domain classes**
    - Corresponds to a requirements paragraph
  - **Design classes**
    - Specified in Software Design Document (SDD)
    - Not a domain class
  - **Implementation classes**
    - Too minor to specify in design

Examples


Concept; conceptual class
Domain Model
Register
Captures
1
1
Sale
date
isComplete : Boolean
time
Real-world concept
Design Model
Register
...
endSale()
enterItem(...)
makePayment(...)
Captures
1
1
Sale
date
isComplete : Boolean
time
makeLineItem(...)
Software components
software class

Writing a maintainable code!!

Coding Standards and Best Practices

- Coding standard is a well-defined and standard style of coding.
- Organizations usually make their own coding standards and guidelines.
- It is very important for a developer to maintain the coding standards, otherwise, the code will be rejected during code review.
- Example:
  PEP 8 – Style Guide for Python Code
  WordPress - Best practices

Coding Standards and Style Guide

- Applying coding standards across a team improves discipline, code readability, and code portability.
- Team member can be designated to draft conventions, e-mail them to the other members for comments, then have the choices finalized by the designated person with the approval of the team leader.
- Examples:
  - Scott Ambler
  - Sun Corporation's Java coding standard

Why is it important?

- Uniform appearance to the codes written by different people.
- Improve readability and maintainability of the code.
- Reduce ambiguity and complexity when reading the code.
- Increase code reuse and help in identifying errors in the code.
- Promote good programming practices and the efficiency of a programmer.

Example#1

```
// Function that takes a Foo* and may or may not assume
ownership of
// the passed pointer.
void TakeFoo(Foo* arg);

// Calls to the function don’t tell the reader anything about
what to
// expect with regard to ownership after the function returns.
Foo* my_foo(NewFoo());
TakeFoo(my_foo);
```

Example from T. Winters, T. Manshreck, and H Wright, “Software Engineering at Google: Lessons Learned from Programming Over Time, 1st ed., O’Reilly Media, 2020.

Example#2

```
// Function that takes a std::unique_ptr<Foo>.
void TakeFoo(std::unique_ptr<Foo> arg);

// Any call to the function explicitly shows that ownership is
// yielded and the unique_ptr cannot be used after the function
// returns.
std::unique_ptr<Foo> my_foo(FooFactory());
TakeFoo(std::move(my_foo));
```

Example from T. Winters, T. Manshreck, and H Wright, “Software Engineering at Google: Lessons Learned from Programming Over Time, 1st ed., O’Reilly Media, 2020.

Creating the Rules

- Rules must pull their weight
- Optimize for the reader
- Be consistent
- Aoid err-prone and surprising constructs
- Concrede to practicalities and necessary

What should be included?

- Limited use of global variables
- Standard headers and formats of different modules, e.g.,
  - Name of the module
  - Create date
  - Author
  - Modification history
  - Synopsis of the module/what the module does
  - Different functions supported in the module along with their input and output parameters
  - Global variables accessed or modified by the module

What should be included?

- Naming conventions
- Indentation
- Error return values and exception handling conventions
- Easily understandable code
- Avoid using an identifier for multiple purposes
- Well documented code
- Length of functions
- When to do code reviews

Java Coding Standards

- Code Conventions for the Java ^{TM} Programming Language

https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

- Google Java Style Guide

https://google.github.io/styleguide/javaguide.html

Naming Conventions in Java

- Name entities with concatenated words
  - E.g., *lengthCylinder*
- Begin class names with capitals to distinguish them from variables or start with capital C
  - E.g., *CCustomer*
- Name variables beginning with lowercase letters. Constants may be excepted.
- Name constants with capitals as in LAM_A_CONSTANT (use static final). lAMACONSTANT is hard to read

Naming Conventions in Java

- (Some organizations) begin the name of instance variables of classes with an underscore as in *_timeOfDay* to distinguish them from other variables or end with a suffix *I*, e.g., *timeOfDayI*.
- Uses the suffix *S* for static variables ,e.g., *numCarsEverBuiltS*
- Use *get* ... , *set* . .. , and *is* . .. for accessor methods as in *getName(), setName(), isBox()* (where the latter returns a boolean value) .

Naming Conventions in Java

- Augment these with standardized additional "getters" and "setters" of collections, for example *insertIntoName()*, *removeFromName()*, *newName()*.
- Consider a convention for parameters. One convention is to use the prefix a, as in sum(int *anInterger* , int *anInterger2*); uses the suffix P, as in sum(int *num1P*, int *num2P*).

Other Conventions

- Use a consistent standard for separation
- Single blank lines are useful for separating code sections within methods
- Use double blank lines between methods
- Within a method:
  - Perform only one operation per line
  - Try to keep methods to a single screen
  - Use parentheses within expressions to clarify their meaning, even if the syntax of the language makes them unnecessary
  - In naming classes, use singular names such as *Customer*, unless the express purpose is to collect objects

Comments

- Good comments should not simply repeat what is obvious from reading the code. They should provide meaning, explaining how and why a piece of code is doing something.
- Bad example:

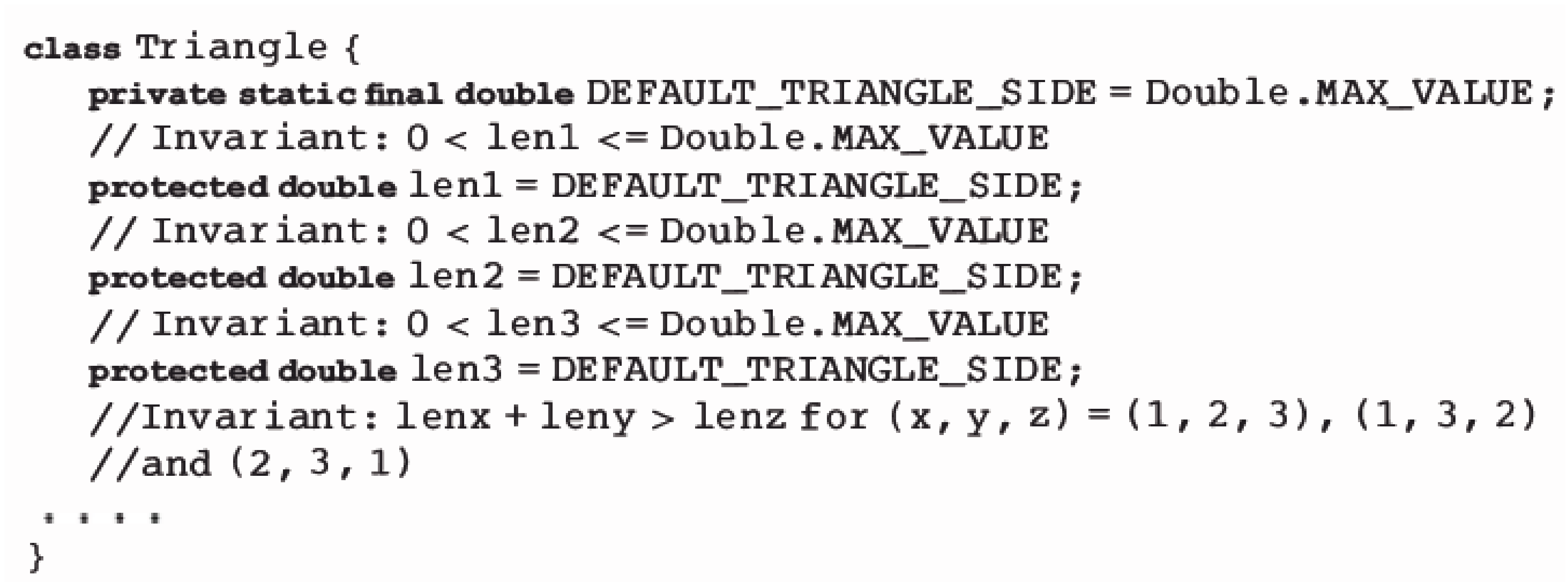
```
        i++;     //increment i
```

Example

```
// doIt - compute and return the area of rectangle
given its length and width
// a – length
// b – width
int doIt (int a, int b)
{
        int c;
        c = a * b;
        return c;
}
```

Documenting Attributes

- For each class attribute, state its purpose and provide all applicable invariants.



```
class Triangle {
   private static final double DEFAULT_TRIANGLE_SIDE = Double.MAX_VALUE;
   // Invariant: 0 < len1 <= Double.MAX_VALUE
   protected double len1 = DEFAULT_TRIANGLE_SIDE;
   // Invariant: 0 < len2 <= Double.MAX_VALUE
   protected double len2 = DEFAULT_TRIANGLE_SIDE;
   // Invariant: 0 < len3 <= Double.MAX_VALUE
   protected double len3 = DEFAULT_TRIANGLE_SIDE;
   //Invariant: lenx + leny > lenz for (x, y, z) = (1, 2, 3), (1, 3, 2)
   //and (2, 3, 1)
 ....
}
```

Defining Classes

- A class should have a name that makes sense to its audience.
  - **Good example: *BookStoreCustomer***
  - **Bad example: *StObsNobKI***
- Class invariants should be stated and observed by all of the methods of the class.
  - The constructors and nonstatic methods of the class

Example

/* Class invariant:

EITHER

the rental is inactive , id == ID_WHEN_RENTAL_NOT_IN_EFFECT ,

rentalCustomer == null , and rental Item == null

OR

ID_WHEN_RENTAL_NOT_ IN_EFFECT < id < =

HIGHEST_ALLOWABLE _RENTAL_ID ,

rentalCustomer ! = null , and rentalItem ! = null

* /

Good Practices

- The most important goal of a block of code is for it to satisfy its purpose. -> Correct
- Therefore,
  - the programmer knows what that purpose is
  - the programmer writes down that purpose precisely within the comments
  - the code implements the purpose
  - the programmer explains, formally or informally, why the code fulfills the purpose
- The *intent/preconditions/postconditions/return/inline* comment should be adopted.

Documenting Methods

- The work of an application is performed by its methods, procedures, or functions.
- Code comments can be effectively organized under categories such as:
  - Intent
  - Precondition
  - Postcondition
  - Return
  - Invariant
  - Exceptions
  - Known issues

Documenting Methods

- **Intent** - An informal statement of what the method is intended to do
  - Don't specify the details here: the other categories provide them.
- **Preconditions** - Conditions on nonlocal variables that the method assumes
  - Includes parameters.
  - Verification of these conditions not promised in method itself.
- **Postconditions** - Value of non-local variables after execution
  - Includes parameters.
  - Notation : x' denotes the value of variable x after execution.

Documenting Methods

- **Invariant -** Relationship among nonlocal variables that the method's execution leaves unchanged (The values of the individual variables may change, however.)
  – Equivalent to inclusion in both pre- and post-conditions.
  – There may also be invariants among local variables.
- **Return** – What the method returns
- **Known issues** - Honest statement of what has to be done, defects that have not been repaired, etc.
- **Exceptions** - These are often thrown when the preconditions are not met because this indicates an abnormality in execution .

Documenting Methods - Intent

- Programmers should provide an informal statement of what the method is intended to do.
- Example: *getNextRentalNumber()*

  Intent: Get the next available number for assigning a rental

Documenting Methods - Precondition

- Programmers should write the assumptions that the method makes about the value of variables external to the method, including the parameters, excluding the local variables.
- Example: *setId()*

Precondition: anId > RENTAL_ID_WHEN_NOT_IN_EFFECT && anId <= HIGHEST_ALLOWABLE _RENTAL_ID

Documenting Methods - Postcondition

- Every method has a return or a postcondition.
- More commonly, postconditions refer to variables whose values could change.
- Example: a method that displays the acknowledgement "DVD Rental Completed“

Postcondition: “Book Rental Completed" is presented on the monitor.

Documenting Methods - Postcondition

- Example: the constructor *Rental*

public Rental

(intanId , RentalCustomer aRentalCustomer , RentalItem, aRentalItem)

throws Exception

Postconditions:

(1) as for postconditions of setId (anId)

(2) RentalCustomer == aRentalCustomer

(3) RentalItem == aRentalItem

Documenting Methods - Postcondition

- When a method depends on another method from which preconditions or postconditions are to be repeated, it is preferable not to literally repeat the preconditions, but merely to reference the methods on which it depends.
- Example:

Postcondition: addend2' = addend1 + addend2

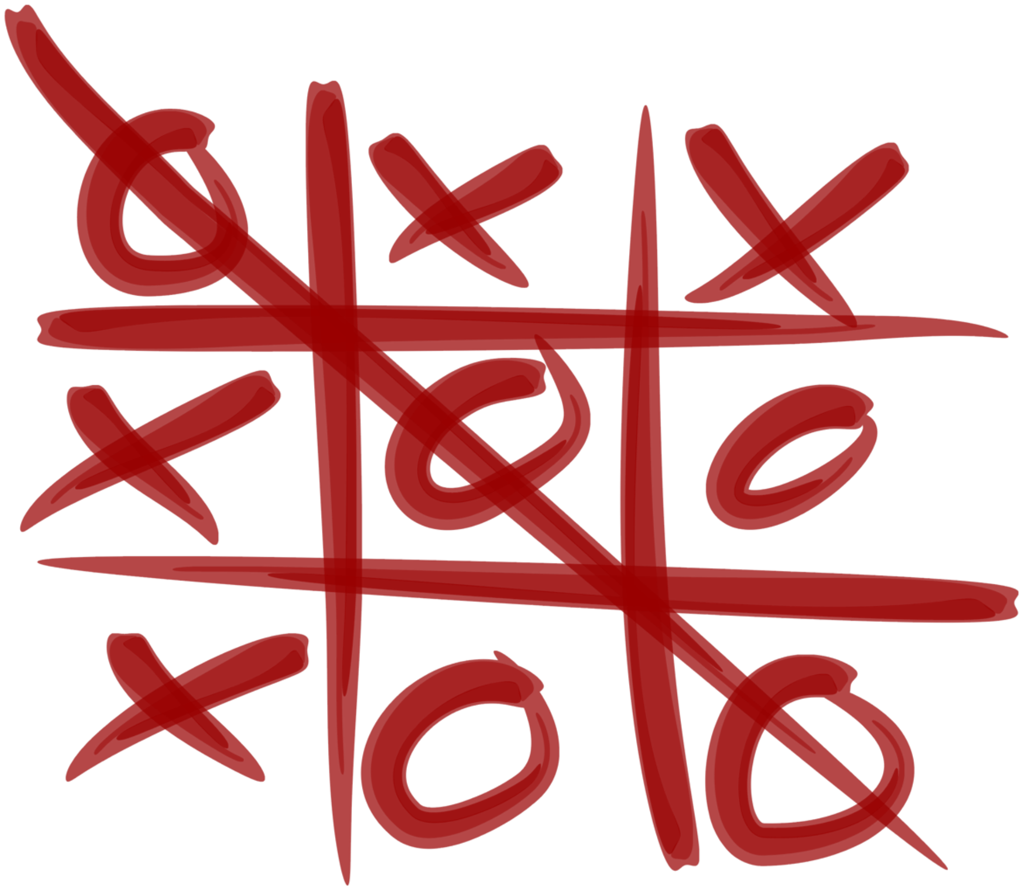
int weirdSum (int addend1 , int addend2)

Documenting Methods - Invariant

- The invariant specifies a relationship among the variables that the method <u>does not alter</u>.
- The return specifies the exact nature of what the method is intended to return in precise terms.
- Example:

```
class Rectangle { . . .
    public double area (double aLength , double aWidth)
}
. . . Rectangle.area ( l, w )….
```

Example



Example

```
/** Intent: Record anOorX at aRow/aCol if aRow/aCol blank; Return 'N' if
* not permitted; return anOorX if full row/column/diagonal
*
* Preconditions: anOorX='O' OR anOorX='X'; 1<=aRowO<=3; 1<=aCol<=3
*
* Postconditions (note use of x and x')
* Post0. All preconditions are met OR Exception thrown
* Postl. gameBoard' contains all non-blank elements of gameBoard
* Post2. gameBoard [aRow-l] [aCol-l] = ‘’ OR return = 'N'
* Post3. gameBoard[aRow-l] [aCol-l] != '' OR
* gameBoard' [aRow-l) [aCol-l) = anOorX
* Post4. There is no full line of anOorX in gameBoard' OR return = anOorX
* /
public static char makeMove( char anOorX, int aRow, int aCol ) throws Exception{
```

Using Expressive Naming

- When assigning names to variables, parameters, functions, classes, etc., the most important criteria are that the names are expressive and that they convey meaning.
- They should not include vague, ambiguous terms.

Example – Naming variables

- **Bad**

  manipulate(float aFloat, int anInt)

  run(double abc, int xyz)

- **Better**

getBaseRaisedToExponent(float aBase, int anExponent)

Example – Naming variables

```
/ / Example of poor use of naming
int DoIt (int a , int b)
{
        int c;
        c = a* b;
        return c;
}
```

- What does this function do?
- It multiplies the two parameters and returns the result, but what exactly is its purpose?
- It is hard to tell by the names of the function or the variables-they do not convey any meaning.

Example – Naming variables

```
//Better example
int ComputeRectangleArea (int length , int width)
{
        int area;
        area = length * width;
        return area;
}
```

Global Variables

- Avoid global variables: consider passing parameters instead
- But not when the number of parameters exceeds ± 7
- Bad

```
extract(int anEntry) { ... ... table = . . . . }
```

- Better

```
extract(int anEntry, EmployeeTable anEmployeeTable)
```

Function Parameters

- Introduce variables near their first usage
- Don't use parameters as working variables.
  - Parameters should only be used to pass information into and out of a function.
  - If a working variable is needed, it should be declared within the function.
- Otherwise, unintended errors can be introduced.
  - If an input parameter is used as a working variable, its original value may be modified. Then, if the parameter is used later in the function with the assumption that it contains its original value, an error will occur.

Function Parameters

- Limit the number of parameters to 6 or 7
  - It is hard to keep track of parameters and use them properly if there are more than 6 or 7.
  - The more parameters are used, the more tightly coupled the calling function is with the called function.
- If so much data need to be passed, reexamine the design and see whether the coupled functions should be combined, or if they belong in the same class, with the parameters becoming private class members.

Example

- Bad

```
    myMethod(int i)
    { . . . . ..
        for(i=0; . . .
```

Explicit Numbers

- It is not good practice to use explicit numbers in code.
- Bad example

```
        for(i = 0; i < 8927; ++i)
```

- Better example

```
const int NUM_CELLS = 8927;  // . . . .
for{cellCounter = 0; cellCounter < NUM _CELLS; ++cellCounter)
```

Variable Initialization and Declaration

- Initialize all variables
- Re-initialize where necessary to "reset“
- It is a good practice to declare variables as <u>close to their first use as possible</u>.
- The reason we initialize variables is to take control of them, avoiding default or garbage values that the system assigns.
  - To avoid potential errors where a variable is used and contains an unexpected value

Variable Initialization and Declaration

- It is good practice to initialize a variable when it is declared.
- Example:

```
float balance = 0;     // Initialize balance to 0
```

Loops

- Loops can be complicated and are common sources of serious failures.
- They are thus special targets of verification.
- Always check loop counters, especially for range correctness.
- Avoid nesting loops more than 3 levels -> introduce auxiliary methods to avoid
- Ensure loop termination

Defensive Programming

- Defensive programming is an effective practice for minimizing bugs by anticipating potential errors and implementing code to handle them.
- Common error sources
  - illegal data
    - a bad value in a function's input parameters
    - an external source such as a file, database
    - data communication line

Defensive Programming

- The bad data must be detected and a strategy employed to handle it.
- Example of defensive strategies:
  - ignoring the error
  - substituting a default value (set a default value)
  - if the error is from an external source, waiting for valid data
- Use exception handling
  - A mechanism that passes control to error handling code that knows how to respond to the error.

Error Handling

- A large fraction of programming goes toward the handling of errors.
- A disciplined approach is essential: *pick an approach*, *state it*, *and* *be sure everyone on the team understands and abides by it*.
- All possibility of errors must be dealt with -> how does one program a method to handle illegal input

Error Handling

- If the illegal data are from an external source such as a user interface, database, or communication device, then
  - Wait for a legal data value (A list box is also a good choice.)
  - The receiving method may be designed to expect certain data (The application should continue working, even when the data is illegal.)
  - Set a default value
  - Use the previous result
- Always check the "**business rules**" -> consistency and boundary checks

Error Handling

- **Log error** – store error information for later use
- **Throw an exception** – function where error occurred, call stack at time of error, register values, and so on. Throw an exception. Exceptions are a mechanism to handle unexpected program errors, including illegal data values. Languages such as C++ and Java have built-in exception support.
- **Abort** – any bad data are considered fatal and the system is aborted and reset.

Example

/** precondition: parameters are positive * /

int sum (int intlP, int int2P) {

/ / verification code for use in development: check parameters positive

/ / now do the work

. . . }

Example

/** precondition: parameters are positive * /

int sum (int intlP, int int2P) {

/ / verification code for deployed application: check parameters positive

/ / only if we have a clear philosophy of what to do if parameters not positive

/ / now do the work

. . . }

Exception Handling

- Languages such as C++ and Java have built-in support for exceptions.
- For those languages without explicit support, developers sometimes design and implement their own exception-handling code.
- When an error is detected an exception is thrown, meaning control is passed to that part of the program that knows how to deal with the error.
- You should catch those exceptions that you know how to handle.

Example

- A method handling an exception

```
ReturnType myMethod ( . . . ) { . . .
try { . . . / / call method throwing ExceptionX }
catch ( ExceptionX e ) ( . . . / / handle it )
. . . }
```

- A method <u>not</u> handling an exception

```
ReturnType myMethod ( . . . ) throws ExceptionX
{ . . . }
```

Guidelines for Implementing Exceptions

- If the present method cannot handle the exception
  - there has to be a handler in an outer scope that can do so.
- If you can handle part of the exception
  - handle that part and then rethrow the exception for handling within an outer scope.
- Make reasonable expectations about the ability of callers to handle the exception you are throwing.
  - Otherwise, find an alternative design since unhandled exceptions crash applications.

Buffer Overflow Prevention

- Buffer overflow occurs when one allows writing to memory that exceeds the space declared in the code.
- This can be prevented by checking variable size at key points (e.g., when a user provides input).

Enforce Intentions

- Enforcement of intentions makes roads safer
- If you intend something about how the code you are constructing is to be used by other parts of the application, then try to enforce this intention.
- Examples of enforcing intention principle:
  - Use qualifiers such as *final* and *const*
  - Use *abstract* and *final* classes
  - Make constants, variables, and classes as local as possible
  - Use the Singleton design pattern if there is to be only one instance of a class
  - Make attributes protected. Access them through more public accessor functions if required
  - Make methods private if they are for use only by methods of the same class

Key Takeaways

- Good habits for implementing code:
  - Write maintainable code
  - Use expressive naming
  - Minimizing the use of global variable
  - Dealing with function parameters
  - Explicit number
  - Initialization and declaration
  - Avoid nested loops and always check on loop termination
  - Write meaningful comments
  - Don’t forget to review your code (with your colleages)