Best practices in programming

the things you don't want to hear, but that someone had to tell you

/> Albert Hernansanz, UPC

Atlas NTA3 · KU Leuven · Feb 2020

Nothing new

Index

1. There will be **nothing new**

2. Will be a so **boring** talk about a so boring **topic**

3. But, **it is a must**

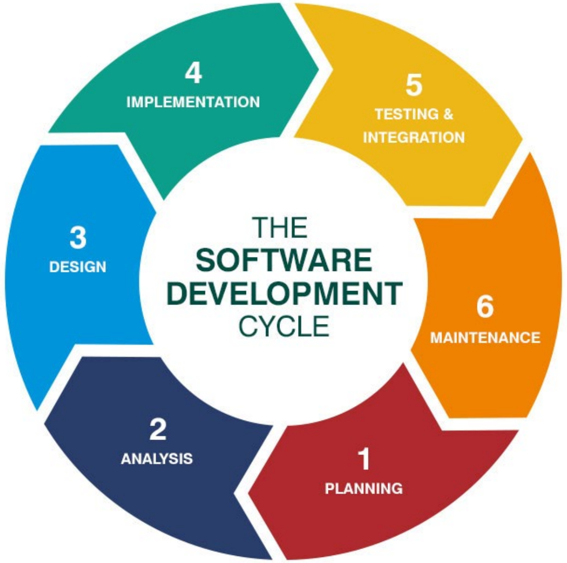
Nothing new

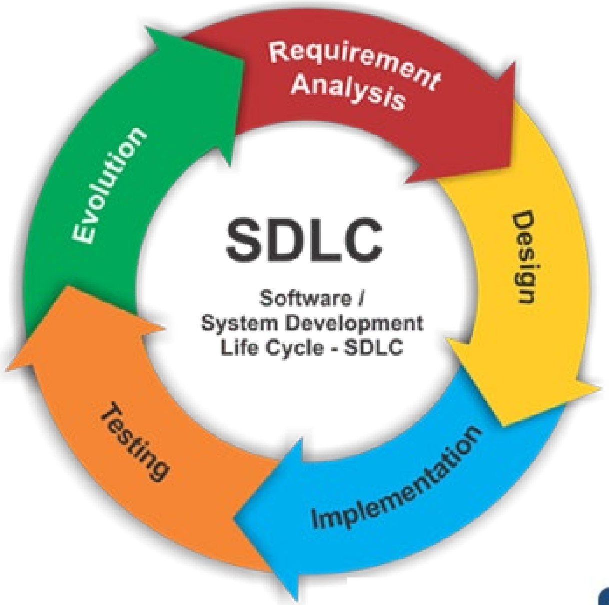
Index

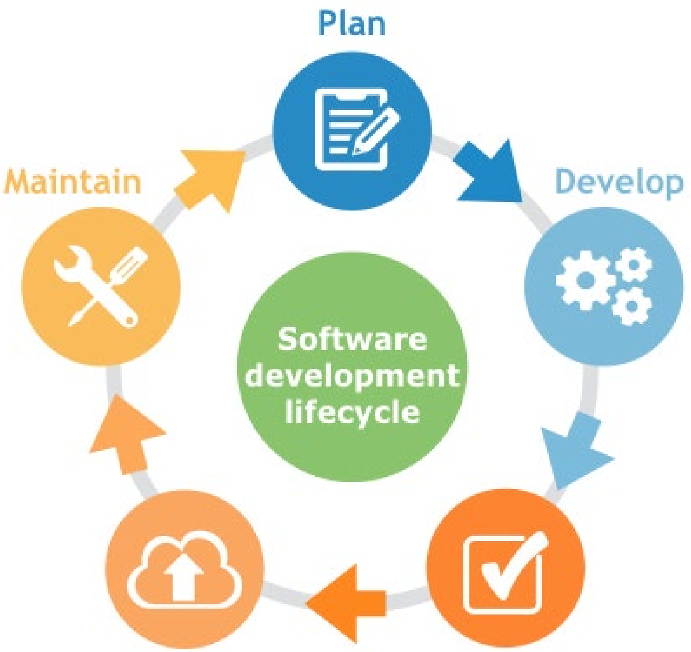
1. Software Development Phases
2. Software Quality
3. Recommendations for Codding Standards
4. Common programming mistakes
5. Version control systems
6. Hands On: Example of code commenting and documentation

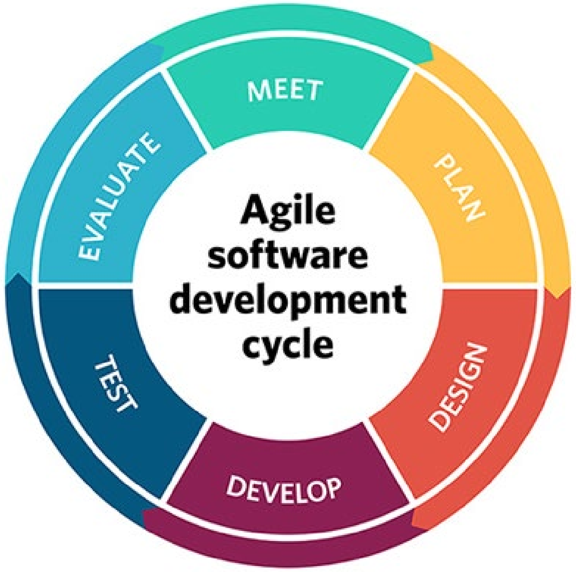
1. Software Development Phases

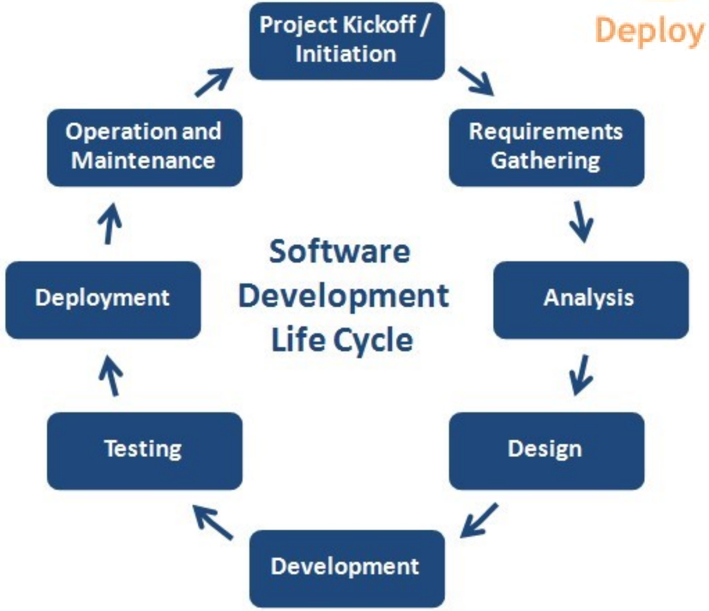
Software Development Phases





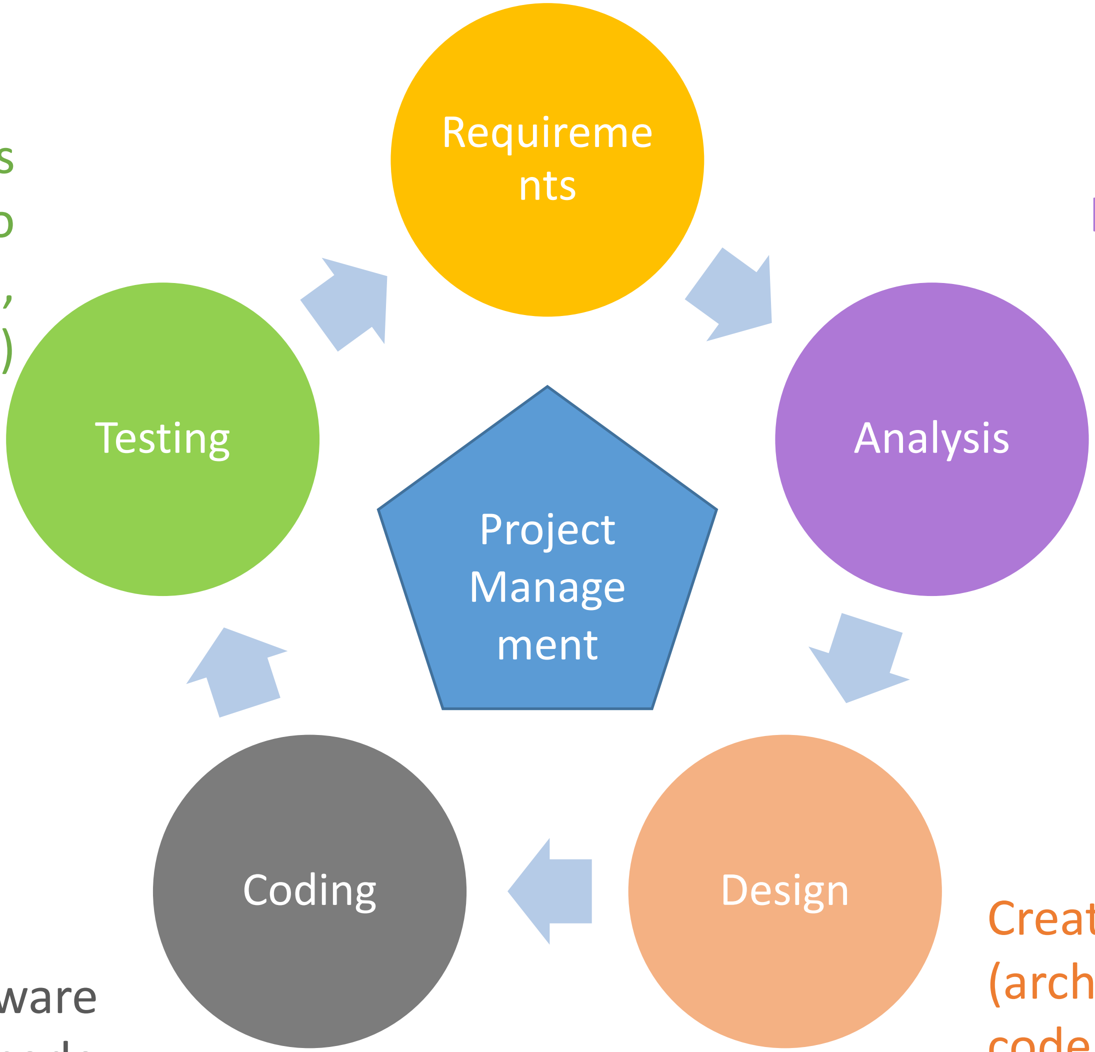





Software Development Life Cycle
Project Kickoff / Initiation
Requirements Gathering
Analysis
Design
Development
Testing
Deployment
Operation and Maintenance

Software Development Phases

What needs to be done



Requirements

Analysis

Design

Coding

Testing

Project Management

How it should be done

Create a software structure (architecture) around which code will be build

Fill in the software structure with code

Devise a plan, manage resources, costs, time, …

Check that the code does what it is supposed to (functionality, performance, reliability, …)

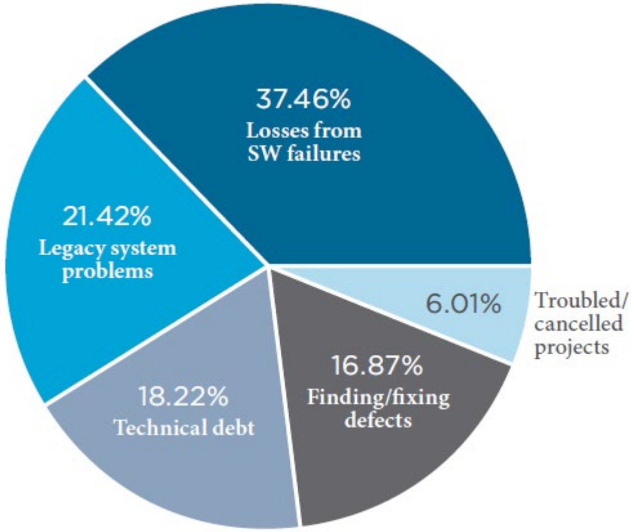
2. Software Quality

Software Quality

Cost of Software Quality (CoSQ) model identifies the component costs of quality and how those add up to form a notional total.

The cost of poor quality software in the US in 2018 is approximately **$2.84 trillion**

FIGURE 1: AREAS OF COST RELATING TO POOR IT/SOFTWARE QUALITY IN THE US



The Cost of Poor Quality Software in the US: A 2018 Report.
Herb Krasner
Member, Advisory Board
Consortium for IT Software Quality (CISQ)
www.it-cisq.org
Hkrasner@utexas.edu
Date: September 26, 2018

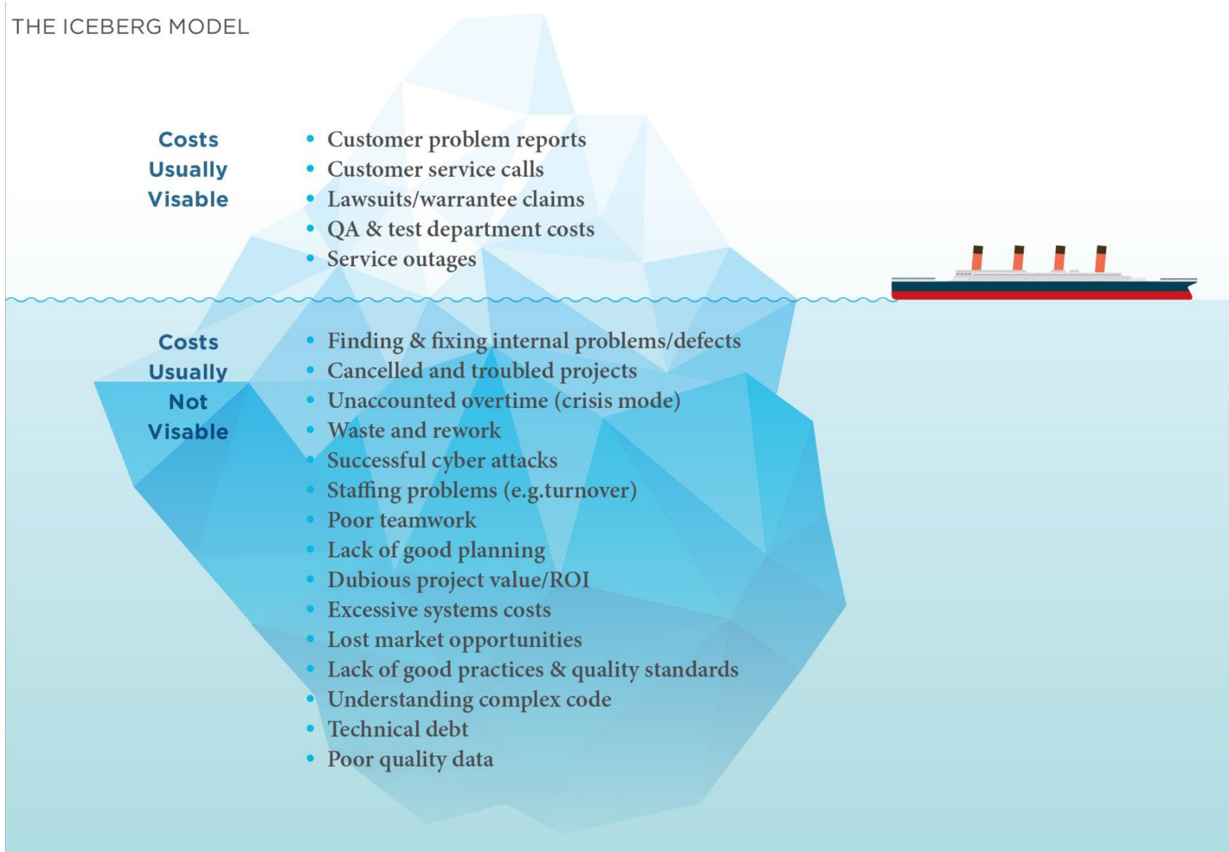
Software Quality

The Iceberg Model

Many of the costs of poor IT software quality are hidden and difficult to identify with formal measurement systems.

The iceberg model illustrates this concept:

- Only a minority of the costs of poor software quality are obvious
- There is a huge potential for reducing costs under the waterline.



THE ICEBERG MODEL

Costs Usually Visable

- Customer problem reports
- Customer service calls
- Lawsuits/warrantee claims
- QA & test department costs
- Service outages

Costs Usually Not Visable

- Finding & fixing internal problems/defects
- Cancelled and troubled projects
- Unaccounted overtime (crisis mode)
- Waste and rework
- Successful cyber attacks
- Staffing problems (e.g.turnover)
- Poor teamwork
- Lack of good planning
- Dubious project value/ROI
- Excessive systems costs
- Lost market opportunities
- Lack of good practices & quality standards
- Understanding complex code
- Technical debt
- Poor quality data

The Cost of Poor Quality Software in the US: A 2018 Report.
Herb Krasner
Member, Advisory Board
Consortium for IT Software Quality (CISQ)
www.it-cisq.org
Hkrasner@utexas.edu
Date: September 26, 2018

Minimize human factor

Enforce Programming Standards to Eliminate Human Error

> Implementing programming standards by using automated tools goes a long way to eliminating the human errors that cause fatal catastrophes.
>
> Jay Thomas | Nov 14, 2014

Minimize human factor

Human factor is a focus of errors and mistakes

- Control the software engineering tools used during all development phases
- Define formal specifications
- Formal design techniques
- Formal techniques to prove correctness
- Use programming standards
- Develop systematic testing
- Etc.

The bug

Software bug:

- A problem causing a program to crash or produce invalid output.

- The problem is caused by insufficient or erroneous logic.

Bug can be: an error, mistake, defect or fault, which may cause failure or deviation from expected results.

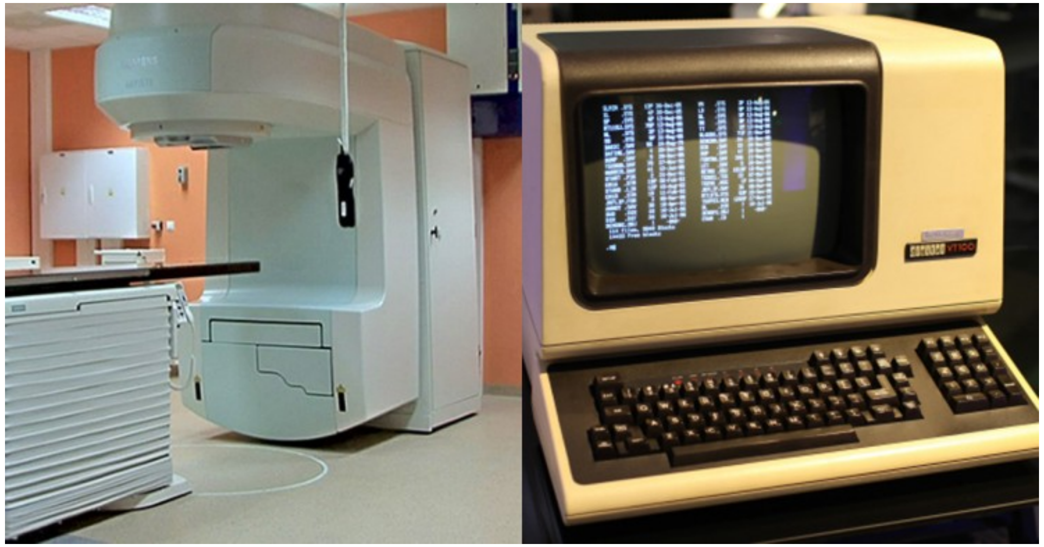
- Most bugs are due to human errors in source code or its design.
- Some bugs might not have serious effects on the functionality of the program and may remain undetected for a long time.
- A program might crash when serious bugs are left unidentified.

Source: Techopedia

The bug

Some of the worst bugs in history include:

- In the 1980s, bugs in the code controlling the machine called Therac-25, used for radiation therapy, lead to patient deaths.



techopedia

The bug

Some of the worst bugs in history include:

- In 1996, the $1.0 billion rocket called Ariane 5 was destroyed a few seconds after launch due to a bug in the on-board guidance computer program.

techopedia

The bug

Some of the worst bugs in history include:

- Software bug led to death in Uber’s self-driving crash
- Sensors detected Elaine Herzberg, but software reportedly decided to ignore her (software classified her as a "false positive" and decided it didn't need to stop for her)

Defect-Related Definitions

The term defect generally refers to some problem with the software, either with its external behavior or with its internal characteristics. The IEEE Standard 610.12 (IEEE 1990) defines the following terms related to defects:

- **Error**: A human action that produces an incorrect result
- **Fault**: An incorrect step, process, or data definition in a computer program
- **Failure**: The inability of a system or component to perform its required functions within specified performance requirements

Software Quality

Defect-Related Definitions

- **Error**: Mistake made in translation or interpretation
- **Fault**: Manifestation of the error in implementation
- **Failure**: Observable deviation in behavior of the system

Example: (Requirement) print speed, defined as distance divided by time

s = d/t; print s;

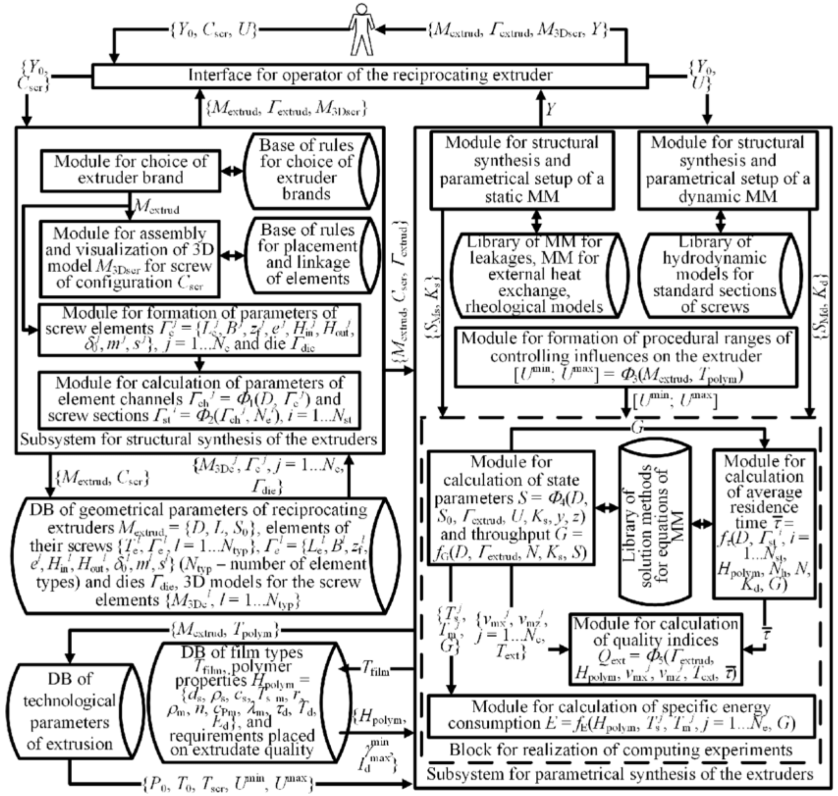
- Error: Account for t=0 (divided by zero error)
- Fault: Not catching t=0
- Failure: exception is thrown

Software Quality

Simplicity

- Unnecessary complexity generates errors P(error) = f(complexity)

  - Complex functions, modules and programs:
    - Difficult their understanding
    - Difficult to follow the execution thread
    - Difficult maintenance and contain more bugs

  
  $\{Y_0, C_{scr}, U\}$
$\{M_{extrud}, \Gamma_{extrud}, M_{3Dscr}, Y\}$
$\{Y_0, C_{scr}\}$
Interface for operator of the reciprocating extruder
$\{Y_0, U\}$
$\{M_{extrud}, \Gamma_{extrud}, M_{3Dscr}\}$
Y
Module for choice of extruder brand
Base of rules for choice of extruder brands
M_{extrud}
Module for assembly and visualization of 3D model M_{3Dscr} for screw of configuration C_{scr}
Base of rules for placement and linkage of elements
Module for formation of parameters of screw elements $\Gamma_e^j = \{L_e^j, B^j, z_f^j, e^j, H_{in}^j, H_{out}^j, \delta_0^j, m^j, s^j\}$, $j = 1...N_e$ and die Γ_{die}
Module for calculation of parameters of element channels $\Gamma_{ch}^j = \Phi_1(D, \Gamma_e^j)$ and screw sections $\Gamma_{st}^i = \Phi_2(\Gamma_{ch}^j, N_e^j)$, $i = 1...N_{st}$
Subsystem for structural synthesis of the extruders
$\{M_{extrud}, C_{scr}, \Gamma_{extrud}\}$
Module for structural synthesis and parametrical setup of a static MM
Module for structural synthesis and parametrical setup of a dynamic MM
Library of MM for leakages, MM for external heat exchange, rheological models
Library of hydrodynamic models for standard sections of screws
$\{S_{Ms}, K_s\}$
$\{S_{Md}, K_d\}$
Module for formation of procedural ranges of controlling influences on the extruder $[U^{min}; U^{max}] = \Phi_3(M_{extrud}, T_{polym})$
$[U^{min}; U^{max}]$
$\{M_{extrud}, C_{scr}\}$
$\{M_{3De}^j, \Gamma_e^j, j = 1...N_e, \Gamma_{die}\}$
DB of geometrical parameters of reciprocating extruders $M_{extrud} = \{D, L, S_0\}$, elements of their screws $\{T_e^l, \Gamma_e^l, l = 1...N_{typ}\}$, $\Gamma_e^l = \{L_e^l, B^l, z_f^l, e^l, H_{in}^l, H_{out}^l, \delta_0^l, m^l, s^l\}$ (N_{typ} – number of element types) and dies Γ_{die}, 3D models for the screw elements $\{M_{3De}^l, l = 1...N_{typ}\}$
G
Module for calculation of state parameters $S = \Phi_4(D, S_0, \Gamma_{extrud}, U, K_s, y, z)$ and throughput $G = f_G(D, \Gamma_{extrud}, N, K_s, S)$
Library of solution methods for equations of MM
Module for calculation of average residence time $\overline{\tau} = f_\tau(D, \Gamma_{st}^i, i = 1...N_{st}, H_{polym}, N_h, N, K_d, G)$
$\{T_s^j, T_m^j, G\}$
$\{v_{mx}^j, v_{mz}^j, j = 1...N_e, T_{ext}\}$
Module for calculation of quality indices $Q_{ext} = \Phi_5(\Gamma_{extrud}, H_{polym}, v_{mx}^j, v_{mz}^j, T_{ext}, \overline{\tau})$
$\overline{\tau}$
$\{M_{extrud}, T_{polym}\}$
DB of technological parameters of extrusion
DB of film types T_{film}, polymer properties $H_{polym} = \{d_s, \rho_s, c_s, T_s, T_m, r, \rho_m, n, c_{Pm}, \lambda_m, \tau_d, T_d, E_d\}$, and requirements placed on extrudate quality
T_{film}
$\{H_{polym}, \gamma^{min}, I_d^{max}\}$
Module for calculation of specific energy consumption $E = f_E(H_{polym}, T_s^j, T_m^j, j = 1...N_e, G)$
Block for realization of computing experiments
Subsystem for parametrical synthesis of the extruders
$\{P_0, T_0, T_{scr}, U^{min}, U^{max}\}$

  - Complex data is usually
    - Hard to operate with
    - Difficult to understand and debug
    - Inefficient (computation and memory)
    - Dynamic memory allocation is dangerous and require strict access control

Common mistakes programming

Simplicity

Complexity is synonym of problems. Be simple when programming

> *“Ironically, writing simple code is neither easy nor simple and, at times, it may actually be quite complex to simplify a logic or a piece of code”*
> ***mitendra mahto*** *@mitendra_mahto*

Simplicity in Data Structures

Complex data is not a good idea, but, if required:

- Each class should mean something by itself
- Structure the data in a logical way (make sense)
- Group in classes protecting data integrity and generating access methods

```
int GetPointCoord(int idPoint, pointCoord pointCoord);
int SetPointCoord(int idPoint, pointCoord point);
```

- Test parameters

```
if ( (idPoint < 0) || (idPoint > PolygonCoord.size()) )
{
        return ERR_PARAMETER_OUT_OF_RANGE;
}
```

Simplicity in Data Structures

Dynamic data

- Better static data structures than dynamic data structures
- Use methods to prevent data access error (e.g. vector of *n* elements, access to *n+1*)
- Static data structures are usually:
  - Safer (no undesired memory access)
  - Faster (consecutive memory allocation)

Example: dynamic vector
Use standard data structures and methods

Do it by my self: Core Dump/Segmentation/memory access
std::vector : Release memory addressing to a well proven method
std::vector + test of range (v[i] just when *i < length(v)*)

Software Quality. Testing

Unit Tests

- Ensure each “module” works properly before integrating modules together.
- Easier to test modules of a system rather than debugging the entire
- executable.
- Good for catching “rare” bugs that only occur on unusual conditions.

- Black Box tests:
  - Reviewing only the functionalities of an application, i.e. *if it does what it is supposed to, no matter how it does it*.

- White Box tests:
  - reviewing the functioning of an application and its internal structure, its processes, rather than its functionalities.

- Both types of testing are needed

Unit Tests: Coverage metrics

- **Statement Coverage:** Execution of every statement at least once.
- **Branch/Decision Coverage:** Decision points are globally tested TRUE/FALSE

  IF (A OR B) THEN
  Two tests: (A=TRUE, B=TRUE), (A=FALSE, B=FALSE).
- **Condition Coverage:** Decision points based on multiple decisions are evaluated for all possibilities of its components.

  IF (A OR B) THEN
  Four tests: (A=T, B=T), (A=F, B=F), (A=T, B=F), (A=F, B=T).
- **Path Coverage:** Each possible path from beginning to the end is evaluated.
- **Function Coverage:** Each function is executed during test execution.

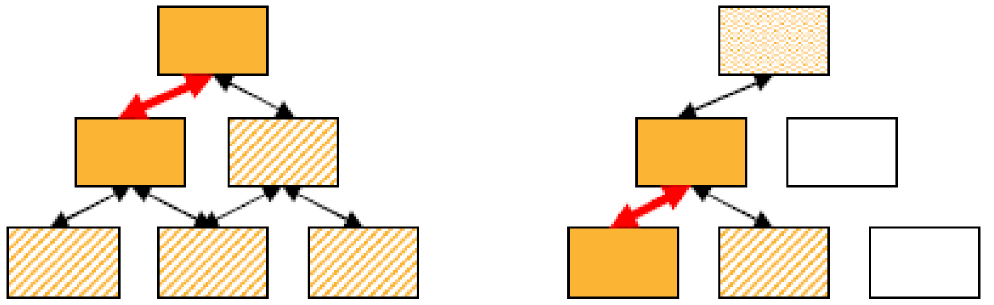
Software Quality. Testing

Integration tests

The goal of integration tests is testing interfaces between modules and communication mechanisms (design).

- Two approaches:
  - Top-down: Test of interfaces of high level modules is done first.
  - Bottom-up: Test of interfaces of lower layers is done first.

- Use of stubs / drivers for the modules not being integrated.



Top-down Strategy

Bottom-up Strategy

NOTE: The use of proven HW and SW

Critical applications require from highly tested HW and SW

- The continuous use detects HW and SW bugs
- Follow all threads in complex SW and HW is impossible.

Redundancy

Critical applications require from redundancy to increase safety

- If a system fails, there is another identical system running. The process do not stops
- Two complete control systems acting on the same plant (e.g. airplane). If the output of both systems is not identic -> release control to human (something is wrong on automatic control)

3. Recommendations for Codding Standards

Software Quality. Common Mistakes

Metrics systems:

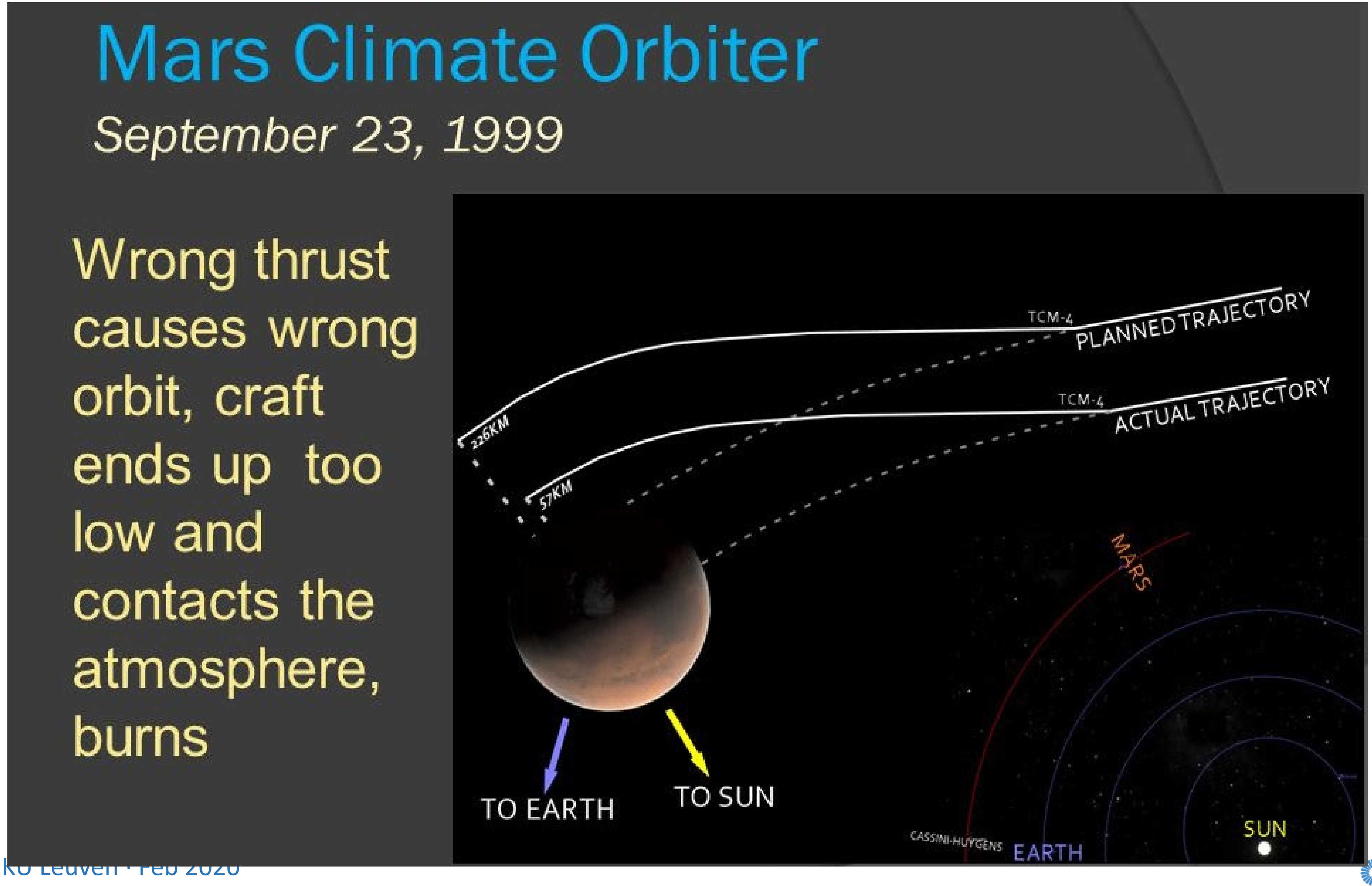
Define a common metric system

- In robotics: mm or m, deg or rad, …
- Programming: double or float, …

The Metric System and NASA's Mars Climate Orbiter

- cost of $125 million and 338-kilogram robotic space probe
- launched by NASA on December 11, 1998
- Study the Martian climate, atmosphere, and surface changes.
- The navigation team at the Jet Propulsion Laboratory (JPL) used the metric system of millimeters and meters
- Lockheed Martin Astronautics (designed and built the spacecraft), provided crucial acceleration data in the English system of inches, feet, and pounds.
- NASA review board: the problem was in the sw controlling the orbiter’s thrusters. The sw calculated the force that the thrusters needed to exert in pounds of force. A second piece of code that read this data assumed it was in the metric unit—“newtons per square meter”

Metrics systems:



Mars Climate Orbiter

September 23, 1999

Wrong thrust causes wrong orbit, craft ends up too low and contacts the atmosphere, burns

Recommendations for Codding Standards

Recommendations for Codding Standards

By *NOAA National Weather Service NWS/OHD General Software Coding Standards and Guidelines*

3.1 Indentation

Proper and consistent indentation is important in producing easy to read and maintainable programs.

Indentation should be used to:

- Emphasize the body of a control statement such as a loop or a select statement
- Emphasize the body of a conditional statement
- Emphasize a new scope block

```
/* Indentation used in a loop construct. Four spaces are used for indentation. */
for ( int i = 0 ; i < number_of_employees ; ++i )
{
    total_wages += employee [ i ] . wages ;
}
```

Recommendations for Codding Standards

3.2 Inline comments

Inline comments explaining the functioning of the subroutine or key aspects of the algorithm shall be frequently used.

- Inline comments promote program readability.
- Allow a programmer not familiar with the code to more quickly understand it.
- Helps the programmer who wrote the code to remember details forgotten over time.
- Reduces the amount of time required to perform software maintenance tasks.

Think what you will need if you have to review the code after 4 years

Recommendations for Codding Standards

3.3 Structured Programming

- Structured (modular) programming techniques are a MUST.
- Group code in functions or modules that do something by themselves.
- Structured programs help programming, debugging and maintenance tasks.

Think what you will need if you have to review the code after 4 years

Recommendations for Codding Standards

3.4 Classes, Subroutines, Functions and Methods

- Keep reasonably sized.
- 1 module does just 1 “thing”
- Too long: programmer is trying to do too many actions at one time

3.5 Source Files

The name of the source file or script shall represent its function. All of the routines in a file shall have a common purpose.

3.6 Variable Names

Variable shall have mnemonic or meaningful names that convey to a casual observer, the intent of its use. Variables shall be initialized prior to its first use.

Recommendations for Codding Standards

3.7 Use of Braces

- More readable
- Less programming errors
- Better control

Even for a single statement in the control block!:

```
if (j == 0)
            printf (“j is zero.\n”);
```

```
if (j == 0)
{
            printf (“j is zero.\n”);
}
```

3.8 Compiler Warnings

- Compilers generate: Warnings and Errors
- **Compiler and linker warnings shall be treated as errors and fixed.**

↓

raise your hand who does

(I have to confess, I do not…)

4. Common programming mistakes

Common mistakes programming

Some basic rules for coding

We always forget some basic rules when coding.

- **Internalize good programming practices**
- **Test each piece of code (results and interaction with the rest of the software)**
- **Be systematic during coding, documenting and testing phases**
- **Commenting and documenting is a must**
- **Use of standardized tools**

Common mistakes programming

Programming style

Set of rules or guidelines used when writing the source code for a computer program.

Good programming style helps to:

- Reading and understanding source code
- Reduce the risk of introducing faults

Some basic rules for a correct style

- Comment each function
- Don’t write deeply nested code
- Don’t write very large modules or functions
- Don’t write very long lines
- Don’t optimize code
- Eliminate side effects
- Write deterministic code
- Use device drivers to isolate hardware interfaces
- Do and undo things in the same function
- Etc…

Common mistakes programming

Naming

Naming is not only a marketing concept

- **Name of functions, classes, data structures and variables must be self descriptive**

This means nothing, this is so confusing and focus of errors

```
int function f (int b, int c)
{
    int a = b + c;
    return a;
}
```

Common mistakes programming

NaN, Divided by Zero, …

- **Name of functions, classes, data structures and variables must be self descriptive**

```
double function divide (double numerator, double denominator)
{
    double result = numerator / denominator;
    return result;
}
```

Before dividing, check that denominator is not zero

Common mistakes programming

Use of correct datatypes

- double in for statements
- Use of epsilon for checking
- Remember to use fabs/abs

```
double function divide (double numerator, double denominator)
{
    double result = numerator / denominator;
    return result;
}
```

Before dividing, check that denominator is not zero:

```
Double epsilon = 0.0001;
If ( fabs(denominator) < epsilon)
…
```

Shared resources: access control

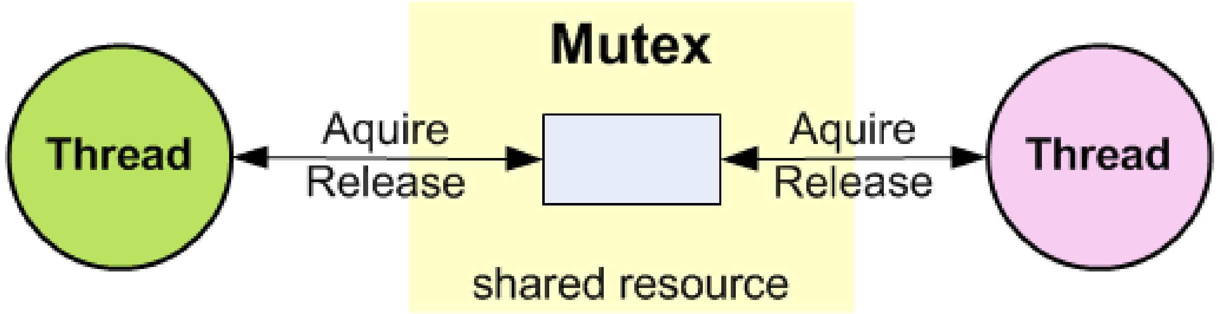
Access control methods in programming

- Use data access control for shared variables

- ***mut**ual **ex**clusion object (mutex)* is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.

- Mutex example in C++:

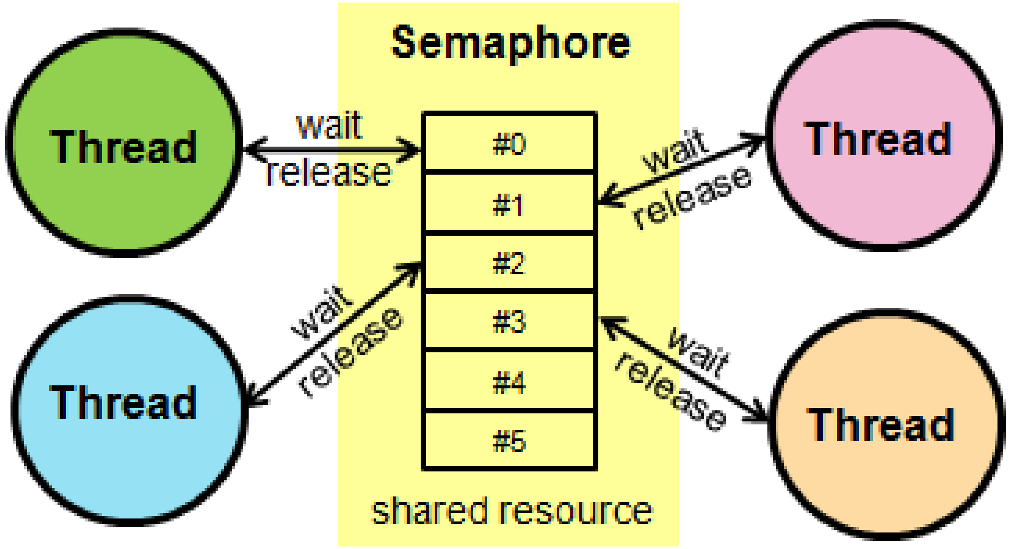
```
std::mutex mu;

void shared_cout(std::string msg, int id)
{
        mu.lock();              //Better
        std::cout << msg << ":" << id << std::endl;
        mu.unlock();
}
```



Shared resources: access control

Access control methods in programming

- **Semaphore**: A semaphore does the same as a mutex but allows x number of threads to enter.



5. Version control systems

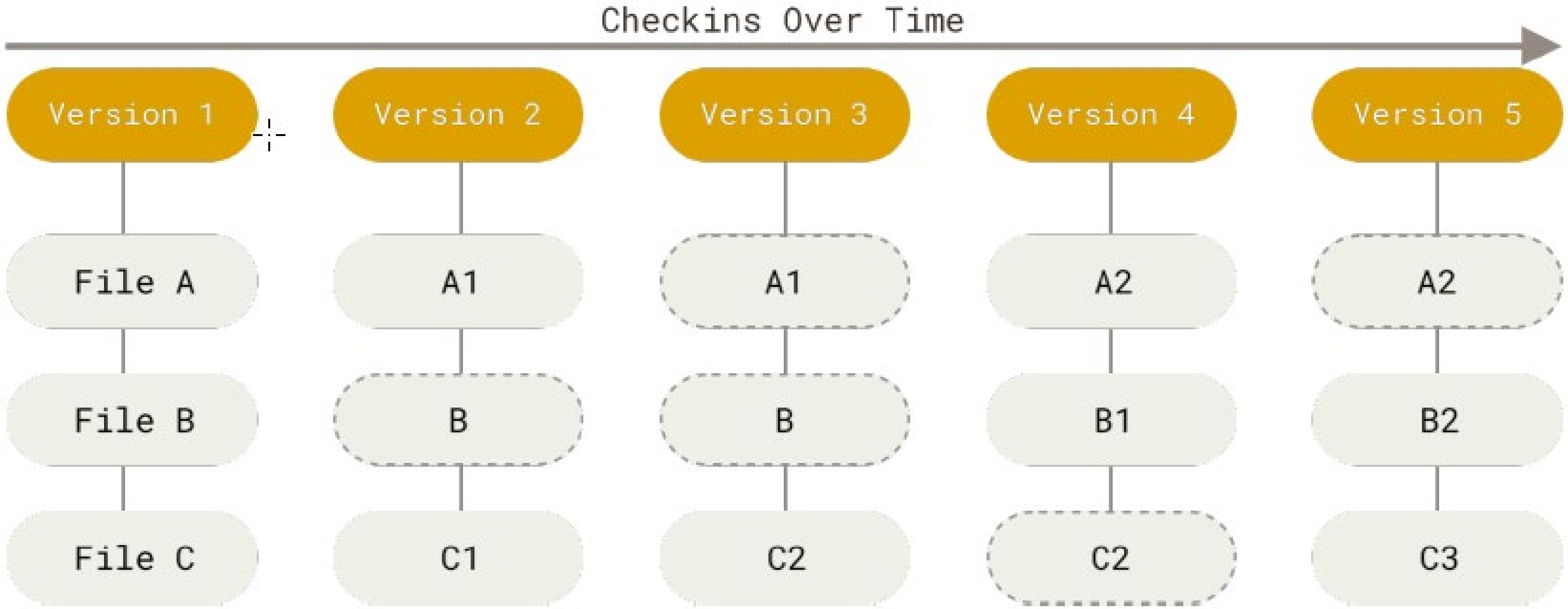
Version Control Systems

Version Control System

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later

Types:

1. Delta (files with modifications from previous to current version) e.g. SVN
2. Snapshot (store all files at each version) e.g. Git



Version Control Systems

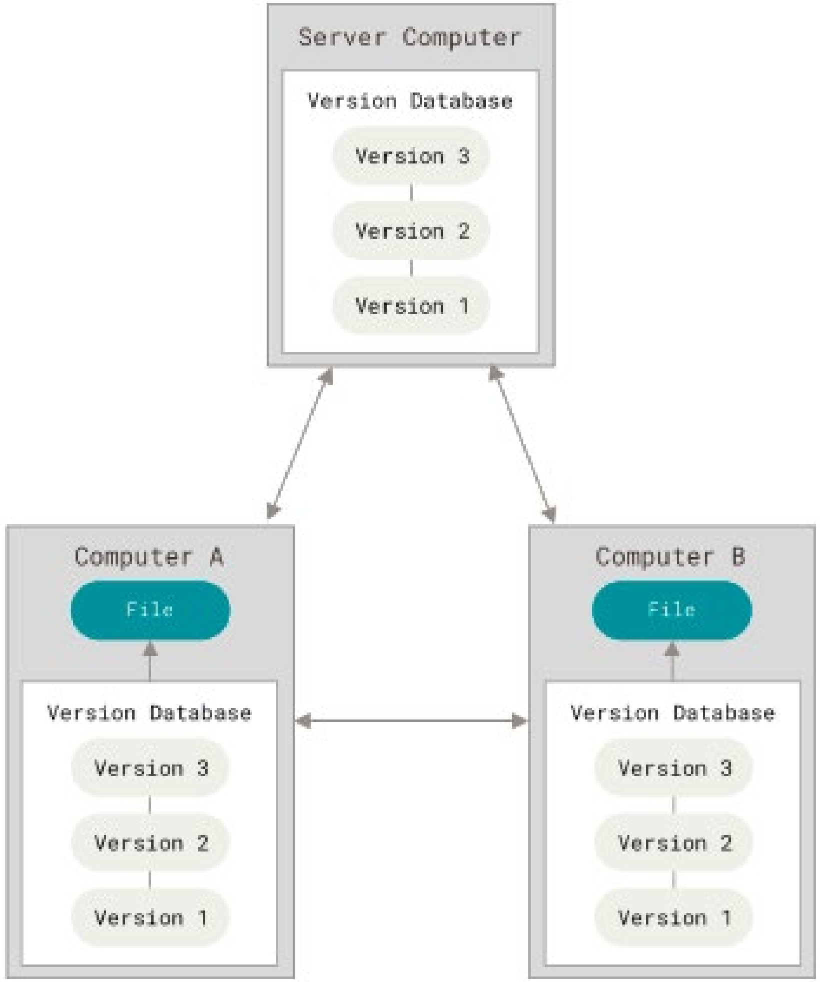
Version Control System

- Current configuration: distributed and redundant system
- Local and remote servers (free and payed)
- Remote servers for free:

GitLab

Github

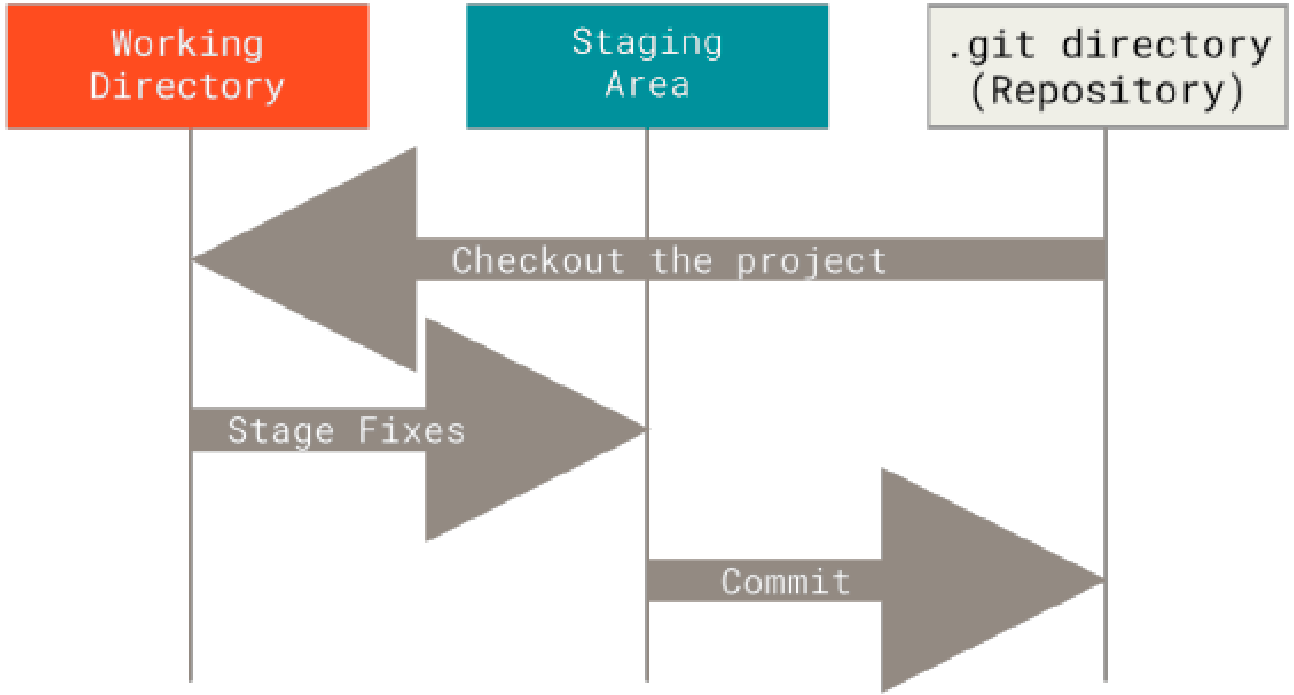
SOURCEFORGE



Version Control Systems

Git: The three states

- **Modified** means that you have changed the file but have not committed it to your database yet.
- **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed** means that the data is safely stored in your local database.

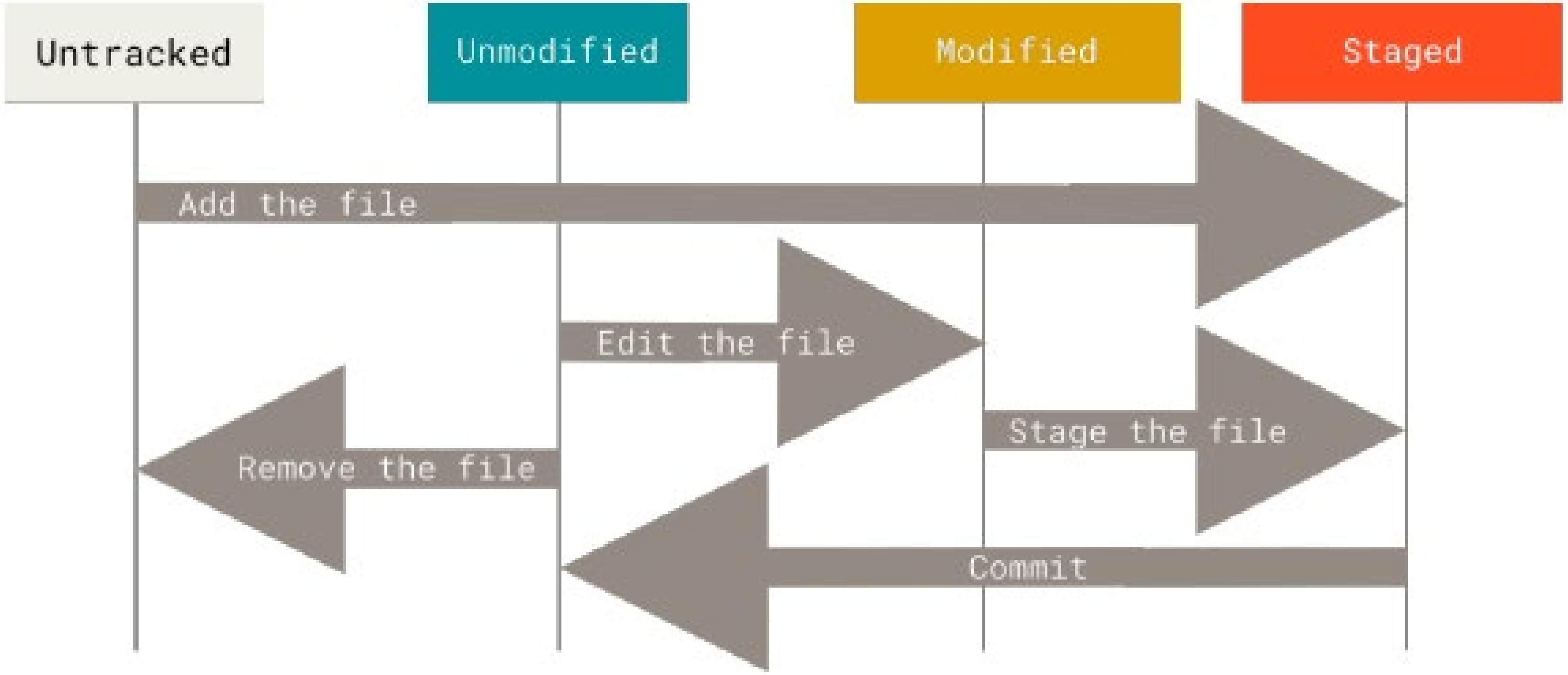


Version Control Systems

Git Workflow

- A file inside working directory can be in state:
    - Tracked: files that were in the last snapshot
    - Not Tracked: any files in working directory that were not in your last snapshot

Life cycle of any file in Git



Version Control Systems

Git basic instructions: easy to understand.

- *git add* begin tracking a new file (new file is in modified state, not in commited)
- *git status* gives a general view (all updated, files out of date, …)
- *git diff* shows you the exact lines added and removed
- *git commit* upload all added files (from modified to commited)
- *git rm* remove file from Git
- *git log* shows commit history

- *etc*

6. Hands On: Example of code commenting and documentation

Version Control Systems

Tools

- Programming IDE: *Visual Studio with C++*
- Automatic comments generator: *GhostDoc Community Edition* (free)
- Documentation: Doxygen & Doxywizard