San Pablo Catholic University (UCSP)
Undergraduate Program in
Computer Science
SILABO

CS113. Computer Science II (Mandatory)

2022-II

1. General information		
1.1 School	:	Ciencia de la Computación
1.2 Course	:	CS113. Computer Science II
1.3 Semester	:	3^{er} Semestre.
1.4 Prerrequisites	:	CS112. Computer Science I. (2^{nd} Sem)
1.5 Type of course	:	Mandatory
1.6 Learning modality	:	Face to face
1.7 Horas	:	2 HT; 2 HP; 2 HL;
1.8 Credits	:	4

2. Professors

3. Course foundation

This is the third course in the sequence of introductory courses in computer science. This course is intended to cover Concepts indicated by the Computing Curriculum IEEE (c) -ACM 2001, under the functional-first approach. The object-oriented paradigm allows us to combat complexity by making models from abstractions of the problem elements and using techniques such as encapsulation, modularity, polymorphism and inheritance. The Dominion of these topics will enable participants to provide computational solutions to design problems simple of the real world.

4. Summary

1. Fundamental Programming Concepts 2. Object-Oriented Programming 3. Algorithms and Design 4. Basic Analysis 5. Basic Type Systems 6. Fundamental Data Structures and Algorithms 7. Event-Driven and Reactive Programming 8. Graphs and Trees 9. Software Design 10. Requirements Engineering

5. Generales Goals

- Introduce the student in the fundaments of the paradigm of object orientation, allowing the assimilation of concepts necessary to develop an information system

6. Contribution to Outcomes

This discipline contributes to the achievement of the following outcomes:

a) An ability to apply knowledge of mathematics, science. (**Usage**)

b) An ability to design and conduct experiments, as well as to analyze and interpret data. (**Usage**)

d) An ability to function on multidisciplinary teams. (**Usage**)

7. Content

<table>
<tr><td colspan="2">UNIT 1: Fundamental Programming Concepts (5)</td></tr>
<tr><td colspan="2">Competences: a,b</td></tr>
<tr><td>Content</td><td>Generales Goals</td></tr>
<tr><td>

Basic syntax and semantics of a higher-level language
Variables and primitive data types (e.g., numbers, characters, Booleans)
Expressions and assingments
Simple I/O including file I/O
Conditional and iterative control structures
Functions and parameter passing
The concept of recursion

</td><td>

Analyze and explain the behavior of simple programs involving the fundamental programming constructs variables, expressions, assignments, I/O, control constructs, functions, parameter passing, and recursion. [Usage]
Identify and describe uses of primitive data types [Usage]
Write programs that use primitive data types [Usage]
Modify and expand short programs that use standard conditional and iterative control structures and functions [Usage]
Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing [Usage]
Write a program that uses file I/O to provide persistence across multiple executions [Usage]
Choose appropriate conditional and iteration constructs for a given programming task [Usage]
Describe the concept of recursion and give examples of its use [Usage]
Identify the base case and the general case of a recursively-defined problem [Usage]

</td></tr>
<tr><td colspan="2">Readings: stroustrup2013, Vandervoorde (2002), Lippman and E.Moo (2013)</td></tr>
</table>

<table>
<tr><th colspan="2">UNIT 2: Object-Oriented Programming (7)</th></tr>
<tr><th colspan="2">Competences: a,b</th></tr>
<tr><th>Content</th><th>Generales Goals</th></tr>
<tr><td>

Object-oriented design

Decomposition into objects carrying state and having behavior
Class-hierarchy design for modeling

Definition of classes: fields, methods, and constructors
Subclasses, inheritance, and method overriding
Dynamic dispatch: definition of method-call
Subtyping

Subtype polymorphism; implicit upcasts in typed languages
Notion of behavioral replacement: subtypes acting like supertypes
Relationship between subtyping and inheritance

Object-oriented idioms for encapsulation

Privacy and visibility of class members
Interfaces revealing only method signatures
Abstract base classes

Using collection classes, iterators, and other common library components

</td><td>

Design and implement a class [Usage]
Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses [Usage]
Correctly reason about control flow in a program using dynamic dispatch [Usage]
Compare and contrast (1) the procedural/functional approach—defining a function for each operation with the function body providing a case for each data variant—and (2) the object-oriented approach—defining a class for each data variant with the class definition providing a method for each operation Understand both as defining a matrix of operations and variants [Usage]
Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype) [Usage]
Use object-oriented encapsulation mechanisms such as interfaces and private members [Usage]
Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language [Usage]

</td></tr>
<tr><td colspan="2">Readings: stroustrup2013</td></tr>
</table>

UNIT 3: Algorithms and Design (5)	
Competences: a,b,d	
Content	**Generales Goals**

• The concept and properties of algorithms
  – Informal comparison of algorithm efficiency (e.g., operation counts)
• The role of algorithms in the problem-solving process
• Problem-solving strategies
  – Iterative and recursive mathematical functions
  – Iterative and recursive traversal of data structures
  – Divide-and-conquer strategies
• Fundamental design concepts and principles
  – Abstraction
  – Program decomposition
  – Encapsulation and information hiding
  – Separation of behavior and implementation
	• Discuss the importance of algorithms in the problem-solving process [Usage]
	• Discuss how a problem may be solved by multiple algorithms, each with different properties [Usage]
	• Create algorithms for solving simple problems [Usage]
	• Use a programming language to implement, test, and debug algorithms for solving simple problems [Usage]
	• Implement, test, and debug simple recursive functions and procedures [Usage]
	• Determine whether a recursive or iterative solution is most appropriate for a problem [Usage]
	• Implement a divide-and-conquer algorithm for solving a problem [Usage]
	• Apply the techniques of decomposition to break a program into smaller pieces [Usage]
	• Identify the data components and behaviors of multiple abstract data types [Usage]
	• Implement a coherent abstract data type, with loose coupling between components and behaviors [Usage]
	• Identify the relative strengths and weaknesses among multiple designs or implementations for a problem [Usage]

Readings: stroustrup2013, **Weert16**, Lippman and E.Moo (2013)	

UNIT 4: Basic Analysis (3)	
Competences: a,b	
Content	**Generales Goals**
• Differences among best, expected, and worst case behaviors of an algorithm • Asymptotic analysis of upper and expected complexity bounds • Big O notation: formal definition • Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential • Empirical measurements of performance • Time and space trade-offs in algorithms • Big O notation: use • Little o, big omega and big theta notation • Recurrence relations • Analysis of iterative and recursive algorithms • Master Theorem and Recursion Trees	• Explain what is meant by "best", "expected", and "worst" case behavior of an algorithm [Usage] • In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors [Usage] • Determine informally the time and space complexity of different algorithms [Usage] • State the formal definition of big O [Usage] • List and contrast standard complexity classes [Usage] • Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis Run algorithms on input of various sizes and compare performance [Usage] • Give examples that illustrate time-space trade-offs of algorithms [Usage] • Use big O notation formally to give asymptotic upper bounds on time and space complexity of algorithms [Usage] • Use big O notation formally to give expected case bounds on time complexity of algorithms [Usage] • Explain the use of big omega, big theta, and little o notation to describe the amount of work done by an algorithm [Usage] • Use recurrence relations to determine the time complexity of recursively defined algorithms [Usage] • Solve elementary recurrence relations, eg, using some form of a Master Theorem [Usage]
Readings: stroustrup2013	

<table>
<tr><td colspan="2">UNIT 5: Basic Type Systems (5)</td></tr>
<tr><td colspan="2">Competences: a,b</td></tr>
<tr><td>Content</td><td>Generales Goals</td></tr>
<tr><td>

A type as a set of values together with a set of operations

Primitive types (e.g., numbers, Booleans)
Compound types built from other types (e.g., records, unions, arrays, lists, functions, references)

Association of types to variables, arguments, results, and fields
Type safety and errors caused by using values inconsistently given their intended types
Goals and limitations of static typing

Eliminating some classes of errors without running the program
Undecidability means static analysis must conservatively approximate program behavior

Generic types (parametric polymorphism)

Definition
Use for generic libraries such as collections
Comparison with ad hoc polymorphism (overloading) and subtype polymorphism

Complementary benefits of static and dynamic typing

Errors early vs. errors late/avoided
Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections
Avoid misuse of code vs. allow more code reuse
Detect incomplete programs vs. allow incomplete programs to run

</td><td>

For both a primitive and a compound type, informally describe the values that have that type [Usage]
For a language with a static type system, describe the operations that are forbidden statically, such as passing the wrong type of value to a function or method [Usage]
Describe examples of program errors detected by a type system [Usage]
For multiple programming languages, identify program properties checked statically and program properties checked dynamically [Usage]
Give an example program that does not type-check in a particular language and yet would have no error if run [Usage]
Use types and type-error messages to write and debug programs [Usage]
Explain how typing rules define the set of operations that are legal for a type [Usage]
Write down the type rules governing the use of a particular compound type [Usage]
Explain why undecidability requires type systems to conservatively approximate program behavior [Usage]
Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections [Usage]
Discuss the differences among generics, subtyping, and overloading [Usage]
Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software [Usage]

</td></tr>
<tr><td colspan="2">Readings: stroustrup2013</td></tr>
</table>

<table>
<tr><td colspan="2">UNIT 6: Fundamental Data Structures and Algorithms (3)</td></tr>
<tr><td colspan="2">Competences: a,b,d</td></tr>
<tr><td>Content</td><td>Generales Goals</td></tr>
<tr><td>

Simple numerical algorithms, such as computing the average of a list of numbers, finding the min, max,
Sequential and binary search algorithms
Worst case quadratic sorting algorithms (selection, insertion)
Worst or average case O(N log N) sorting algorithms (quicksort, heapsort, mergesort)
Hash tables, including strategies for avoiding and resolving collisions
Binary search trees
Common operations on binary search trees such as select min, max, insert, delete, iterate over tree
Graphs and graph algorithms
Representations of graphs (e.g., adjacency list, adjacency matrix)
Depth- and breadth-first traversals
Heaps
Graphs and graph algorithms
Maximum and minimum cut problem
Local search
Pattern matching and string/text algorithms (e.g., substring matching, regular expression matching, longest common subsequence algorithms)

</td><td>

Implement basic numerical algorithms [Usage]
Implement simple search algorithms and explain the differences in their time complexities [Usage]
Be able to implement common quadratic and O(N log N) sorting algorithms [Usage]
Describe the implementation of hash tables, including collision avoidance and resolution [Usage]
Discuss the runtime and memory efficiency of principal algorithms for sorting, searching, and hashing [Usage]
Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data [Usage]
Explain how tree balance affects the efficiency of various binary search tree operations [Usage]
Solve problems using fundamental graph algorithms, including depth-first and breadth-first search [Usage]
Demonstrate the ability to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in a particular context [Usage]
Describe the heap property and the use of heaps as an implementation of priority queues [Usage]
Solve problems using graph algorithms, including single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm [Usage]
Trace and/or implement a string-matching algorithm [Usage]

</td></tr>
<tr><td colspan="2">Readings: stroustrup2013, Pai and Abraham (2018)</td></tr>
</table>

UNIT 7: Event-Driven and Reactive Programming (2)	
Competences: a,b	
Content	**Generales Goals**

• Events and event handlers
• Canonical uses such as GUIs, mobile devices, robots, servers
• Using a reactive framework
  – Defining event handlers/listeners
  – Main event loop not under event-handler-writer's control
• Externally-generated events and program-generated events
• Separation of model, view, and controller
	• Write event handlers for use in reactive systems, such as GUIs [Usage]
	• Explain why an event-driven programming style is natural in domains where programs react to external events [Usage]
	• Describe an interactive system in terms of a model, a view, and a controller [Usage]

Readings: stroustrup2013, Williams (2011)	

UNIT 8: Graphs and Trees (7)	
Competences: a,b,d	
Content	**Generales Goals**

• Trees
  – Properties
  – Traversal strategies
• Undirected graphs
• Directed graphs
• Weighted graphs
• Spanning trees/forests
• Graph isomorphism
	• Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each type of graph/tree [Usage]
	• Demonstrate different traversal methods for trees and graphs, including pre, post, and in-order traversal of trees [Usage]
	• Model a variety of real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology or the organization of a hierarchical file system [Usage]
	• Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting [Usage]
	• Explain how to construct a spanning tree of a graph [Usage]
	• Determine if two graphs are isomorphic [Usage]

Readings: Nakariakov (2013)	

<table>
<tr><td colspan="2">UNIT 9: Software Design (6)</td></tr>
<tr><td colspan="2">Competences: a,b</td></tr>
<tr><td>Content</td><td>Generales Goals</td></tr>
<tr><td>

System design principles: levels of abstraction (architectural design and detailed design), separation of concerns, information hiding, coupling and cohesion , re-use of standard structures
Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented
Structural and behavioral models of software designs
Design patterns
Relationships between requirements and designs: transformation of models, design of contracts, invariants
Software architecture concepts and standard architectures (e.g. client-server, n-layer, transform centered, pipes-and-filters)
The use of component desing: component selection, design, adaptation and assembly of components, component and patterns, components and objects (for example, building a GUI using a standar widget set)
Refactoring designs using design patterns
Internal design qualities, and models for them: efficiency and performance, redundacy and fault tolerance, traceability of requeriments
Measurement and analysis of design quality
Tradeoffs between different aspects of quality
Application frameworks
Middleware: the object-oriented paradigm within middleware, object request brokers and marshalling, transaction processing monitors, workflow systems
Principles of secure design and coding

Principle of least privilege
Principle of fail-safe defaults
Principle of psychological acceptability

</td><td>

Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation [Usage]
Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design [Usage]
Construct models of the design of a simple software system that are appropriate for the paradigm used to design it [Usage]
Within the context of a single design paradigm, describe one or more design patterns that could be applicable to the design of a simple software system [Usage]
For a simple system suitable for a given scenario, discuss and select an appropriate design paradigm [Usage]
Create appropriate models for the structure and behavior of software products from their requirements specifications [Usage]
Explain the relationships between the requirements for a software product and its design, using appropriate models [Usage]
For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system [Usage]
Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server [Usage]
Investigate the impact of software architectures selection on the design of a simple system [Usage]
Apply simple examples of patterns in a software design [Usage]
Describe a form of refactoring and discuss when it may be applicable [Usage]
Select suitable components for use in the design of a software product [Usage]
Explain how suitable components might need to be adapted for use in the design of a software product [Usage]
Design a contract for a typical small software component for use in a given system [Usage]
Discuss and select appropriate software architecture for a simple system suitable for a given scenario [Usage]
Apply models for internal and external qualities in designing software components to achieve an acceptable tradeoff between conflicting quality aspects [Us-

</td></tr>
</table>

<table>
<tr><td colspan="2">UNIT 10: Requirements Engineering (1)</td></tr>
<tr><td colspan="2">Competences: a,b</td></tr>
<tr><td>Content</td><td>Generales Goals</td></tr>
<tr><td>

Describing functional requirements using, for example, use cases or users stories
Properties of requirements including consistency, validity, completeness, and feasibility
Software requirements elicitation
Describing system data using, for example, class diagrams or entity-relationship diagrams
Non functional requirements and their relationship to software quality
Evaluation and use of requirements specifications
Requirements analysis modeling techniques
Acceptability of certainty / uncertainty considerations regarding software / system behavior
Prototyping
Basic concepts of formal requirements specification
Requirements specification
Requirements validation
Requirements tracing

</td><td>

List the key components of a use case or similar description of some behavior that is required for a system [Usage]
Describe how the requirements engineering process supports the elicitation and validation of behavioral requirements [Usage]
Interpret a given requirements model for a simple software system [Usage]
Describe the fundamental challenges of and common techniques used for requirements elicitation [Usage]
List the key components of a data model (eg, class diagrams or ER diagrams) [Usage]
Identify both functional and non-functional requirements in a given requirements specification for a software system [Usage]
Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements [Usage]
Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system [Usage]
Compare the plan-driven and agile approaches to requirements specification and validation and describe the benefits and risks associated with each [Usage]
Use a common, non-formal method to model and specify the requirements for a medium-size software system [Usage]
Translate into natural language a software requirements specification (eg, a software component contract) written in a formal specification language [Usage]
Create a prototype of a software system to mitigate risk in requirements [Usage]
Differentiate between forward and backward tracing and explain their roles in the requirements validation process [Usage]

</td></tr>
<tr><td colspan="2">Readings: stroustrup2013</td></tr>
</table>

8. Methodology

El profesor del curso presentará clases teóricas de los temas señalados en el programa propiciando la intervención de los alumnos.

El profesor del curso presentará demostraciones para fundamentar clases teóricas.

El profesor y los alumnos realizarán prácticas

Los alumnos deberán asistir a clase habiendo leído lo que el profesor va a presentar. De esta manera se facilitará la comprensión y los estudiantes estarán en mejores condiciones de hacer consultas en clase.

9. Assessment

Continuous Assessment 1 : 20 %

Partial Exam : 30 %

Continuous Assessment 2 : 20 %

Final exam : 30 %

References

Lippman, Stanley B. and Barbara E.Moo (2013). *C++ Primer*. 5th. O'Reilly. ISBN: 9780133053043.
Nakariakov, S. (2013). *The Boost C++ Libraries: Generic Programming*. CreateSpace Independent Publishing Platforml.
Pai, Praseed and Peter Abraham (2018). *C++ Reactive Programming*. 1st. Packt.
Vandervoorde, David (2002). *C++ Templates:The Complete Guide*. 1st. Addison-Wesley. ISBN: 978-0134448237.
Williams, Anthony (2011). *C++ Concurrency in Action*. 1st. Manning.