Fecha: 14 de septiembre de 2020

COMISIÓN DE SELECCIÓN DE PROGRAMADORES

Convocatoria pública de 10 de junio de 2019 para la provisión de once plazas de Programador con destino en el Centro de Tecnologías de la Información y de las Comunicaciones de la Secretaría General del Congreso de los Diputados.

Cuarto Ejercicio

Nombre:	Firma:
Apellidos:	
DNI:	

Instrucciones:

1. No abra este cuestionario hasta que le sea indicado.
2. No escriba ni haga ninguna marca o alteración de los códigos de barras impresos en cada hoja del cuestionario.
3. El enunciado del ejercicio consta de una única página.
4. El tiempo de realización de este ejercicio es de **45 minutos**.
5. Rellene todos los datos de la portada y entréguela cuando se le solicite, antes de comenzar la realización del ejercicio.
6. Puede utilizar todas las hojas en blanco que considere necesario para la realización del ejercicio. Ponga en cada una de ellas el número de orden.
7. Al finalizar el ejercicio, deberá entregar las hojas de respuestas y el enunciado del ejercicio, los cuales serán grapados entre sí y guardados en un sobre cerrado.

If everyone hates it, why is OOP still so widely spread?

In the August edition of Byte magazine in 1981, David Robson opens his article, which became the introduction of ‘Object-Oriented Software Systems’ for many, by admitting up front that it is a departure from what many familiar with imperative, top-down programming are used to.

“Many people who have no idea how a computer works find the idea of object-oriented programming quite natural. In contrast, many people who have experience with computers initially think there is something strange about object oriented systems.”

It is fair to say that, generations later, the idea of organizing your code into larger meaningful *objects* that model the parts of your problem continues to puzzle programmers. If they are used to top-down programming or functional programming, which treats elements of code as precise mathematical functions, it takes some getting used to. After an initial hype period had promised improvements for modularising and organising large codebases, the idea was over applied. With OOP being followed by OOA (object-oriented analysis) and OOD (object-oriented design) it soon felt like everything you did in software had to be broken down to objects and their relationships to each other. Then the critics arrived on the scene, some of them quite disappointed.

Some claimed that under OOP writing tests is harder and it requires extra care to refactor. There is the overhead when reusing code that the creator of Erlang famously described as a case when you wanted a banana but you got a gorilla holding the banana. Everything comes with an implicit, inescapable environment.

Other ways of describing his new way of solving problems include the analogy between an imperative programmer as “a cook or a chemist, following recipes and formulas to achieve a desired result” and the object oriented programmer as “a greek philosopher or 19th century naturalist concerned with the proper taxonomy and description of the creatures and places of the programming world.”

Was the success just a coincidence?

OOP is still one of the dominant paradigms right now. But that might be due to the success of languages who happen to be OOP. Java, C++ and Kotlin rule mobile for Android and Swift and Objective-C for iOS so you can’t develop software for mobile unless you understand the object-oriented approach. For the web, it’s JavaScript, Python, PHP and Ruby.

Asking why so many widely-used languages are OOP might be mixing up cause and effect. Richard Feldman argues in his talk that it might just be coincidence. C++ was developed in the early 1980s by Bjarne Stroustrup, initially as a set of extensions to the C programming language. Building on C , C++ added object orientation but Feldman argues it became popular for the overall upgrade from C including type-safety and added support for automatic resource management, generic programming, and exception handling, among other features.