Programación y Tecnología: Un camino equivocado hacia la construcción de artefactos

CAMILO CHACÓN SARTORI

¿Por qué muchos programadores viven y sueñan con las nuevas tecnologías? Porque creen que eso es la computación, una visión errada, confusa y sin sentido. La programación es la actividad fundamental dentro de la computación, la que trata sobre la resolución de problemas usando como medio los algoritmos para expresarlos. Su definición, implementación y verificación. Las tecnologías computacionales son una caja pre-construida por otro programador que intenta expresar sus ideas para hacer el trabajo de los demás sea más fácil. No hay nada de malo en eso, pero cuando se vuelve el principio y fin de un programador es indudable que perdemos a un creador y obtenemos a un seguidor.

Additional Key Words and Phrases: informática, computación, programación, software, matemática, tecnología, artefactos

1 INTRODUCCIÓN

Hoy en día lo imperante son las apariciones de nuevas tecnologías. Vivimos dentro de una burbuja que nos impide crear, ya sea por una poca confianza personal o de una falta de utilidad inmediata. O, por las dos motivos en conjunto. La programación es una actividad que debe aspirar a la creatividad y rigurosidad. Donde los que la ejercen deben darle sentido a su trabajo y no ser simple seguidores de otros (o peor aún malos seguidores). Falta de rigurosidad es la piedra angular de los problemas de la programación. Y, una de sus aristas en el amor incrédulo hacia cualquier tecnología nueva —que al parecer por ser nueva es sinónimo de bueno—, que promueve un sesgo en pensar muchas veces que la novedad es mejor que lo pasado. Un claro error. Nos embarcamos a un nuevo viaje de nuevos «desafíos», para dominarla pero al final no dominamos nada. Solo aprendemos algo muy similar a lo anterior pero con una nueva sintaxis y semántica creada por los creadores de turno. Y, que en pocos casos representa un verdadero avance y mejora a algo anterior, pero ¿cómo podemos descubrir aquello si no sabemos mucho sobre computación? Porque muchos enamorados de lenguajes de programación o bibliotecas, conocen mucho de eso pero sin una visión global de la computación, los que los hace proclive a ser engañados y utilizar el nuevo *sticker* en su *laptop*[1].

Algunas nuevas tecnología han sido grandes avances, no es posible negarlo. Por ejemplo, muchas referentes a trabajar con tareas de *big data* y *machine learning*. Sin estas, no hubiera sido posible para un pequeño grupos de programadores poder iniciar proyectos. También todo lo referente a la *cloud* ha sido una revolución en la computación. Donde ya no es excusa el no tener infraestructura física para mantener una aplicación. Muchos avances pero no nos engañemos. Muchas de las nuevas tecnologías no trae nada nuevo, sino más bien, un desvió de lo importante: comprender los fundamentos de la computación. En este texto desarrollare este último punto, su importancia, la cual nos ayudara a seleccionar mejor una tecnología y por qué no, a ser un creador de la misma.

Este documento se divide en tres secciones que responde a cada una de estas preguntas: ¿Qué necesitamos previamente? ¿Qué estudiamos? ¿Cuál es el propósito?

2 LA PROGRAMACIÓN NECESITA RIGUROSIDAD

Escribir código, veamos que ocurre, ¿no funcionó?, modifiquemos algo y veamos que pasa —iterando una y otra vez—. ¿Cuántos no han realizado algo así durante su carrera profesional? El problema se genera cuando se vuelve un hábito,

[1]Existen alta probabilidad de encontrar a estos programadores en meetup de la herramienta de moda.

Author's address: Camilo Chacón Sartori, camilo.chacon.s@mail.pucv.cl.

casi como una forma aleatoria de programar, sin fundamento, que produce una forma desordenada de pensar. Un programador debería definir su algoritmo previamente a ser implementado en un lenguaje de programación. No al revés. No definirlo mientras se esta programando. Esto puede producir varios errores: (1) mayor probabilidad de *bug*; (2) no resolver lo que se requiere; (3) realizar la implementación más compleja de lo necesario. En general, un lenguaje de programación puede confundir más que ayudar en la definición de una solución de un problema. Es por esto que, similar a como menciona Lamport en su artículo «If You're Not Writing a Program, Don't Use a Programming Language»:

«He trabajado con varios ingenieros informáticos —tanto de hardware como de software— y he visto lo que sabían y lo que no sabían. He descubierto que la mayoría de ellos no entienden algunos importantes conceptos básicos. Estos conceptos son oscurecidos por los lenguajes de programación. Se entienden mejor mediante una forma simple y poderosa de pensar sobre la computación de forma matemática como se explica aquí.» (Lamport, 2018)

Su visión trata de la importancia de las matemáticas dentro de la programación, especialmente en la etapa de definición —previa a la implementación en un lenguaje de programación—, y para ésta Lamport propone usar el lenguaje de las matemáticas que, no cuenta con todas las dificultades de un lenguaje de programación. Un algoritmo implementado en un lenguaje de programación requiere que se piense en la cantidad de memoria a utilizar, definición de tipo para cada variable y seguir una sintaxis del determinado lenguaje (pudiendo ser fácil o compleja). Lo cual, te quita del foco de lo que realmente importa: resolver un problema en particular. Y, te debes centrar en características del propio lenguaje de programación (además si a esto le agregamos que se trate de un paradigma de programación[2] diferente al cual usas comúnmente, se vuelve aún más desafiante).

Un programador necesita conocer muy bien las matemáticas discretas: teoría de conjunto, lógica, grafos, combinatoria, teoría de número, entre otras. Estas son la base para cualquier conocimiento aplicado en computación, que trata por definición con estructuras discretas ya sean finitos o infinitos numerables (no contiene números reales). Existe una cierta percepción equivocada que un programador no requiere mucho conocimiento en matemáticas. Esto porque según se dice la programación es netamente práctica. ¿Pero acaso la programación no es lógica? Claro que sí, entonces tener conocimientos en matemática es fundamental no tan solo para construir algoritmos, sino también para analizarlos y proponer mejoras. Las matemáticas proporciona una mayor lógica. Esto quiere decir que, a mayor lógica podrás profundizar en temas mucho más avanzados del áreas que requiere de este conocimiento previo. Por ejemplo, el área de análisis de algoritmo es netamente matemáticas, donde se coloca el foco en el rendimiento de un algoritmo, ya sea para disminuir el tiempo de computación o reducir el uso de memoria ram. Sin matemáticas, no puedes hacer un análisis correcto porque no sabrías como realizar una demostración. Por lo tanto, tu implementación se ve limitada y solo escribes la solución que primero se te ocurra sin saber si es lo más óptimo para dicho problema.

3 ARTEFACTOS

Un artefacto computacional se divide en dos categorías: abstracto y físico[3]. En esta sección tratare de la primera, la cual hace referencia a cualquier construcción abstracta creada a través de código. Es decir, tiene un fin operativo y no es tangible. Por ejemplo, un algoritmo computacional, un sitio web, una base de dato, un sistema operativo, etc. son artefactos abstractos. Antes de crear un artefacto se necesita de una implementación, definición y un problema. (Lamport hace énfasis en las definiciones, que fue lo que comente en la sección anterior.) Posterior a eso, se necesita un medio (implementación), que puede venir dado por un lenguaje de programación. Por tanto, si tenemos un problema P, una definición D y una implementación I, podemos decir que: $I(D(P)) = A$, donde A es un artefacto.

[2]Cada lenguaje de programación tiene asociado una o más formas de expresar la computación.
[3]Hace referencia al hardware, ya sea: memoria ram, cpu, disco duro, keyboard, etc.

Dicha notación se puede leer como:

- Para crear una definición D se necesita de un problema P.
- Para crear una implementación I se necesita de una definición D.
- Para crear un artefacto A se necesita una implementación I.

Una visión similar fue propuesta por Turner en su artículo «Computational Artefacts» (Turner, 2014). Aunque él solo menciona tres etapas: definición, medio y artefacto (ver figura 1). Y, relaciona cada etapa de manera singular.

Fig. 1. Representación de la relación entre definición, implementación y artefacto propuesta por Turner.

No concuerdo con esa representación. Una definición *no* puede solo estar asociada a *una* implementación, ni un implementación puede estar asociada a solo *un* artefacto. Un problema que se trata de resolver de manera computacional —dado su complejidad— puede derivar en un conjunto de definiciones, medios y artefactos. (La solución a un problema puede provocar la creación de uno o más artefactos.) No se debe pensar como una secuencia de piezas unitarias enlazadas una detrás de otra. Por el contrario, son dinámicas y pueden variar una a otras. Una formalización más precisa podría ser $\{ x : D \mid P(x) \}$, donde cada x es parte del conjunto de definiciones D satisface el problema P. Similar ocurre con las implementaciones: $\{ y : I \mid D(y) \}$ y artefactos: $\{ z : A \mid I(z) \}$. Esta propuesta se puede ver en la siguiente figura:

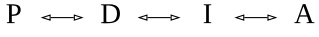

Fig. 2. Representación bidireccional en la relación entre problema, definición, implementación y artefacto.

Dado que D, I y A son cada uno un conjunto, se pueden derivar en sus elementos.

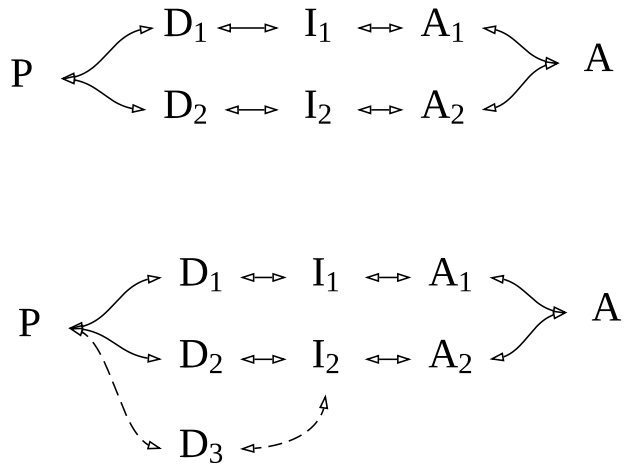

Fig. 3. Representación bidireccional de la expansión en la relación entre problema, definición, implementación y artefacto.

La figura 2 y 3 representan un básico diagrama llamado PDIA (problema, definición, implementación y artefacto). El cual como se puede apreciar es un grafo no-dirigido y cíclico (Es decir, existe una relación bidireccional entre sus elementos $[D, I, A]$ lo cual lo hace dinámica y factible de modificaciones para construir la solución a un problema P.) En la figura 3, un problema puede derivarse en múltiples sub-definiciones, sub-implementaciones y sub-artefactos. Donde existen restricciones de asociación. Por ejemplo, para una implementación I podría venir dada de una o múltiples D (pero no al revés). Lo mismo ocurre de con un artefacto que puede ser construido a través de una o más I. Al final cada sub-artefacto crea una agregación para formar un único artefacto A que viene a resolver el problema P. La linea punteada significa que en el caso que un problema necesite de una nueva definición, es posible respetando una restricción: estar asociada a una implementación, ya sea una existente o creando una nueva. Este diagrama PDIA expresa una idea central en el desarrollo de cualquier artefacto abstracto: se necesita una flexibilidad porque estos componente lo hacen humanos no máquinas; y lo extremadamente simple y rígido generalmente no funciona porque hay muchas más variables para controlar.

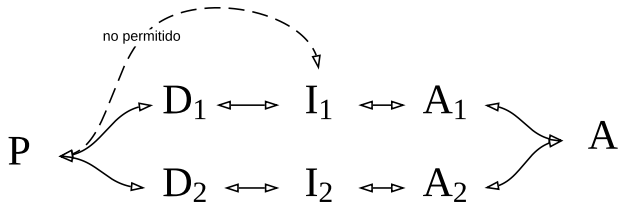

Fig. 4. Representación bidireccional en donde se omite la relación con la definición.

Por último, muchos programadores cometen el error de omitir la definiciones (ver figura 4), esto termina creando una débil implementación lo que se transforma en un artefacto defectuoso.

4 DOS TIPOS DE TECNOLOGÍAS

La falsa tecnología es una herramienta que presenta una forma diferente de hacer lo mismo, no es creativa y es la que no debería crearse ni menos usarse. La otra es la que realmente soluciona un problema correctamente y trae cosas nuevas. ¿Por qué se tiende a valorar más una herramienta que el conocimiento de conceptos? Porque los primeros ocultan los segundos, son aplicables, nos ahorran tiempo y son más rápidos de aprender. Una tecnología T nos ayuda a resolver un problema P, y dicho T nos proporciona un mecanismo para implementarlo. Pero nos omite la definición. Que es la zona más importante y teórica de las demás. Cuando una definición esta completa esta nos permite elegir de manera más eficiente que tecnologías es más acorde a nuestro problema. Es decir, la tecnología es solo una parte de la implementación la cual tiene dos posibles problemas: (a) la sobre estimación de los programadores sobre cualquier tecnología en particular dejando de lado la formalización; (b) la tecnología que no trae nada nuevo (esta es consecuencia de la primera).

Con respecto a la primera, la mayoría de los programadores omite la definición; que es algo riguroso por norma general. En cambio, sigue un mecanismo errado de prueba y error constante, donde a través de un conjunto de tecnologías se busca cual es la más acorde para solucionar su problema sin entender realmente cual es el problema! Es así, como se involucra en un ciclo de elección y descarte hasta encontrar lo que él cree que es mejor. Esto ocurre por dos motivos principales: (a) tiempos de entrega acotados; (b) incorrecta forma de pensar, donde se antepone la velocidad para llegar una solución antes de la calidad de la misma provocando mayor probabilidad de *bug*.

La segunda, desaparece cuando la primera es solucionada. Un programador que define correctamente un problema, ya sea usando las matemáticas para la definición de un algoritmo (como propone Lamport) o creando un documento detallado de especificación; tiene mayor conocimiento de lo que debe hacer. Por ende, su elección viene dada con mayor información.

Si vemos algunos repositorios en Github[4], algunos proyectos tienen muy buena documentación mientras otros ni siquiera la tienen. ¿Podría ocupar la segunda en un sistema de producción? Claro que no. Incluso el tener documentación no es suficiente para catalogar la madurez y calidad de dicha tecnología, es por esto, que se hace necesario hacer pruebas entre las que tengan un nivel mínimo. Pero cabe aclarar, que estas pruebas no son solo probar y descartar, sino más bien, crear experimentos estadísticos con cada una de estas y analizar los datos de los resultados para respaldar tu decisión. Y esta decisión, es más fácil cuando ya has trabajado un tiempo en la definición del problema. (Parece obvio pero lamentablemente no lo es.)

5 CONCLUSIÓN

Las tecnologías no están mal. Lo mal esta en creer que podemos omitir pasos que son muy importante en la solución de un problema porque creemos ingenuamente que dicha abstracción es la mejor opción. La rigurosidad le entrega a los programadores un mayor conocimientos de los conceptos de la computación, nos traslada del camino técnico a uno más completo con elementos científicos (como la verificación). La teoría nos brinda un mayor conocimiento global del área donde trabajamos, lo que se traduce en una mayor inteligencia en la elección de herramientas; que conlleva a una mejor construcción de artefactos.

REFERENCIAS

L. Lamport. If You're Not Writing a Program, Don't Use a Programming Language. *Eacts*, 2018. URL http://bulletin.eatcs.org/index.php/beatcs/article/view/539/532.

R. Turner. Computational artefacts. 07 2014.

[4]La plataforma más popular en la actualidad de software libre. https://github.com/