ISBN 978-950-33-1673-3

Edición de
María Paula Buteler
Ignacio Heredia
Santiago Marengo
Sofía Mondaca

Filosofía de la Ciencia
por Jóvenes Investigadores

vol.2

Filosofía de la Ciencia por Jóvenes Investigadores

vol. 2

Edición de

María Paula Buteler
Ignacio Heredia
Santiago Marengo
Sofía Mondaca

Colecciones del CIFFyH

Filosofía de la Ciencia por Jóvenes Investigadores vol. 2 / Ignacio Heredia ... [et al.]; editado por María Paula Buteler... [et al.]. - 1a ed. - Córdoba: Universidad Nacional de Córdoba. Facultad de Filosofía y Humanidades, 2022.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-950-33-1673-3

1. Filosofía de la Ciencia. 2. Jóvenes. I. Heredia, Ignacio. II. Buteler, María Paula, ed.
CDD 121

Publicado por
Área de Publicaciones de la Facultad de Filosofía y Humanidades - UNC
Córdoba - Argentina
1º Edición

Área de
Publicaciones

Diseño de portadas: Manuel Coll
Diagramación: María Bella
Imagen de cubierta y contracubierta: Detalle del retrato de Carpenter (1836), autora: Margaret Sarah Carpenter. Imagen de dominio público editada por Martina Schilling.
Imagen de portads interiores: Retrato de Ada Lovelace, autore desconocide, circa 1840. Seis diseños en color por Ignacio Heredia.

2022

¿Qué es un diseño Top-Down en programación?

Xavier Huvelle*

El diseño *Top-Down* apareció a principios de la década de 1970 para el diseño de programas. La idea general detrás del diseño Top-Down es tener un proceso en el cual se parte de una idea general abstracta que se va refinando y concretizando en las etapas que les siguen. Este método no era novedoso para la época, podemos observar su uso en por lo menos varios autores como Simon (1947) o principalmente en cibernautas como Wiener y Rosenblueth (1945) y Ashby (1960). Tal proliferación permitió una rápida expansión del método en diversos ámbitos donde usan una u otra teoría del diseño. Dada esta diversidad, el principio general detrás del diseño Top-Down se fue también modificando en función de las interpretaciones que se fueron dando a tal procedimiento. En diseño de programas, un método notable fue el denominado método cascada que, como su nombre lo indica, parte desde una idea general abstracta que luego debe ser implementada tal como se lo especifica en la etapa de planificación. Benington (1983) y Royce (1970) describieron este método con fines de criticarlo, aunque, irónicamente, fueron considerados como los "padres" de dicho método. Benington dice claramente que al usar su método se buscaba seguir los lineamientos clásicos de la ingeniería y aplicarlos en el diseño de programas, por lo que el uso de métodos Top-Down y de programación estructurada le parecía natural. Evidentemente no todos lo interpretaron de este modo, por lo que el método cascada terminó siendo una mala aplicación del método Top-Down. El problema, como veremos a continuación, tiene que ver estrictamente con el estatuto que se le otorga a las especificaciones que debe tener el programa.

Por otro lado, la diferencia que existe entre los elementos constitutivos de un diseño Top-Down no ha sido muy trabajada. En efecto, no es lo mismo hacer uso de funciones, problemas, objetivos, o especificaciones al esbozar un esquema de un diseño. Estas distinciones pueden resultar en que las especificaciones que tenemos en la fase de planificación del diseño no se correspondan con la implementación. En particular debido a que

* FONCYT, UNC / xavier.huvelle@gmail.com

si pensamos el diseño en términos de funciones es muy probable que no podamos tener en claro todas las funciones que podrían implementarse en el programa antes de construirlo. Esta dificultad es lo que empujó a Baker (1975) a distinguir entre desarrollo Top-Down y programación Top-Down.

Este trabajo se encuentra dividido en dos partes. La primera describe las distintas características que posee el método Top-Down y las consecuencias que resultan en usar unas u otras características. La segunda concierne al método cascada y a cómo la interpretación de ciertas características del método Top-Down derivaron en malas prácticas en el diseño de programas. En el último apartado presentamos las conclusiones.

Método Top-Down para el diseño de programas

El método Top-Down en programación fue propuesto por Mills[1] (1971) y Wirth (1971) con la intención de ordenar el desarrollo de programas "amplios" y encontrar una manera de proceder ordenadamente. La amplitud de los programas surgió como problemática debido a que, a partir de los años 1950 e impulsado por los grandes trabajos en lógica simbólica, se supo que podíamos alcanzar grandes complejidades a partir de pequeños conjuntos de funciones. Por aquel entonces se creía que el programa tenía mayor complejidad mientras más cantidad de funciones tenga. Tanto el método Top-Down como la programación estructurada tenían el propósito de poner orden a esta profusión de código, pero también el de establecer que la cantidad de líneas no implican (tal como sugería la creencia popular) mayores funcionalidades.

Hamilton y Zeldin (1972) vieron muy rápidamente las ventajas que planteaba el uso de un método Top-Down para el diseño de programas, tanto así que este método fue usado para la codificación empleada en el programa APOLO (ver figura. 1). Para Hamilton y Zeldin un método Top-Down es un proceso organizacional que tiene el siguiente orden: (1) el concepto total es formulado, (2) la especificación funcional es designada, (3) la especificación funcional se encuentra refinada en cada paso

[1] Algunos afirman que la metodología Top-Down fue creada por Wirth (1971), a partir de su idea de refinamiento por etapas o por Dijkstra también alrededor de esta fecha, por el uso de su idea de descomposición jerárquica. En realidad fueron Mills y Wirth, quienes trabajaban en IBM, los creadores. Mills fue quien acuñó por primera vez el nombre del método, mientras que Wirth lo denominó "refinamiento por etapas".

intermediario y (4) el último refinamiento es producido para definir completamente el problema (Hamilton y Zeldin, 1972).

Figura 1. Hamilton al lado del código escrito a mano del programa APOLO.

Baker (1975), otro defensor del método Top-Down, nos precisa que el término es engañoso y no debe ser tomado literalmente; dado que con

este método se construye un sistema de una manera que idealmente elimina la necesidad de crear una codificación cuyo testeo dependa de otro código que todavía no haya sido escrito o de datos que no estén todavía disponibles. Esto requiere un cuidado particular durante la planificación ya que ciertas partes de los programas deberán ya estar parcialmente completados antes de poder iniciar la escritura de otras partes. Esta parcialidad[2] del código no es concebible en un método cascada ya que para el método cascada es necesario completar todo el módulo anterior para pasar al siguiente.

Baker plantea, además, que se debe distinguir entre programación Top-Down y desarrollo Top-Down. Esta distinción busca diferenciar la programación de un sistema a la programación de un programa que luego debe interactuar en un sistema. La programación Top-Down para Baker consiste entonces en programar un programa constituido por algunos módulos y por un número independiente de unidades compilables desarrollados por uno o varios programadores. La complejidad de problemas presentes en este nivel son para Baker mayoritariamente problemas de diseño y para solucionarlos se usan aproximaciones de la programación estructurada, como los "niveles de abstracción" de Dijkstra o el teorema de expansión de Mills. Para Baker, la programación estructurada permitió prácticas beneficiosas para los programadores tales como la mejora metodológica, o el mejoramiento en la estandarización y el control de la programación.

El desarrollo Top-Down para Baker es visto (ver figura 2) como una secuenciación del desarrollo del sistema de un programa para eliminar o evitar problemas principalmente de interfaces y consiste más bien en una metodología para gestionar el equipo de programadores. Su ventaja es que permite la integración y el desarrollo en paralelo y, en particular, una

[2] La idea de parcialidad tiene además otra particularidad que tiene que ver con ciertas cuestiones provenientes de la lógica simbólica. En ciertos desarrollos de paradigmas de lenguajes de programación, tales como los lenguajes declarativos o funcionales, se parte de "raíces simbólicas", por ejemplo, para adecuarlas con Inputs determinados. En Schwartz y Cocke (1970) se puede observar una metodología Top-Down para la confección de programas a partir de lenguajes de programación y su relación con la formalización BNF (Backus Normal Form). La formalización BNF es antes que nada un metalenguaje que consiste en un conjunto de reglas para la construcción iterativa de la familia de todas las oraciones gramaticales de un lenguaje específico. Entonces usando el método Top-Down uno puede partir de un símbolo cuya raíz es "a" para luego intentar generar una oración que se adecue con la oración del input. Esta forma de caracterizar métodos Top-Down contrasta con las otras posturas propuestas en este apartado y que conciernen principalmente a los lenguajes imperativos.

disponibilidad temprana de ciertos módulos del sistema. Dada su complejidad, es necesario tener una persona que cumpla el rol de administrar (*manager*) el progreso del sistema como un todo y que tenga en claro las especificaciones funcionales deseadas del sistema. Baker considera el rol del manager como más complejo, respecto de los otros roles, debido que el progreso como un todo no es determinable hasta la fase de integración *Bottom-Up*.

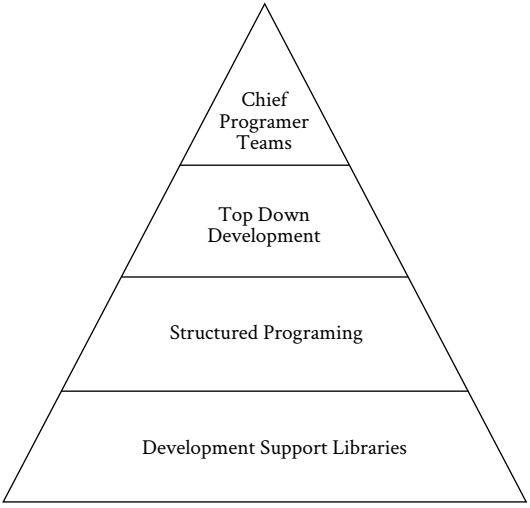

Figura 2. Jerarquías de técnicas para Baker en el diseño de un programa (Baker, 1975).

Por su lado, Hiemann (1975) acuerda también con lo propuesto por Baker, pero resalta otro elemento propio del diseño Top-Down: verlo desde un punto de vista "funcional" (1975, p. 16). Se definen las funciones básicas del programa para luego descomponerlas en sub-funciones que contengan mayores detalles y si continuamos el proceso hacia abajo llegamos a otros niveles que contengan mayores detalles hasta crear un árbol (figura 3). Para Hiemann, cuando terminamos el proceso de diseño Top-Down, tenemos conocimientos sobre todas nuestras interfaces, todas nuestras decisiones lógicas y en cómo los datos son estructurados.

Top-Down Design...

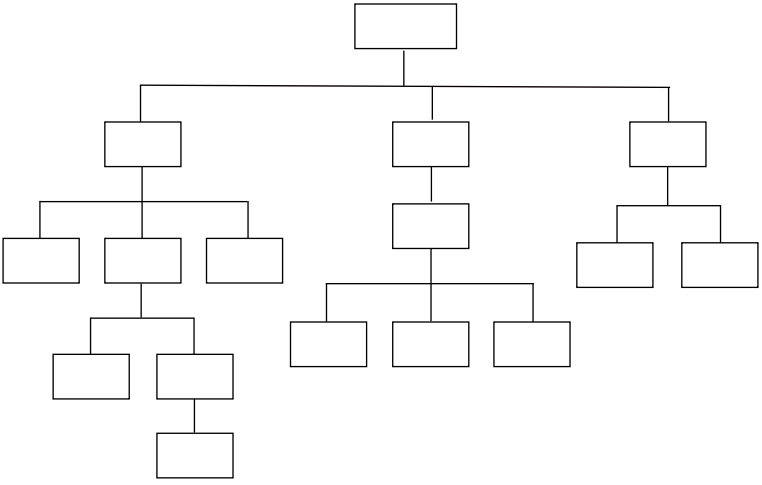

Figura 3. Representación de Hiemann de un diseño Top-Down (Hiemann, 1975).

Es importante notar que aquí nos encontramos con tres formas distintas de caracterizar al método Top-Down. Hamilton y Zeldin (1972) caracterizan al desarrollo Top-Down como un proceso en el cual se parte de un problema u objetivo general abstracto que se va concretando paso a paso en los niveles inferiores (ver figura 4).

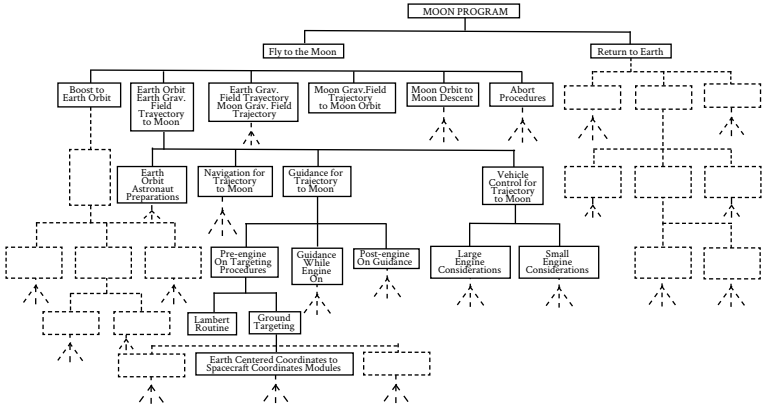

Figura 4. Ejemplo de árbol usado por Hamilton y Zeldin para el programa APOLO (Hamilton y Zeldin, 1972).

Por otro lado, el parecido de la figura 4 con el árbol funcional de Hiemann es muy impactante, pero, como podemos ver, se trata de un árbol funcional mientras que para Hamilton y Zeldin se encuentra guiado por objetivos o problemas. La diferencia radica en que partir de funciones implica un proceso operacional bien determinado, mientras que hacer uso de problemas u objetivos implica describir hacia qué apuntamos en esta parte del desarrollo. Esta distinción es, como hablamos en la introducción, de naturaleza parecida a la diferencia entre el deseo de producir un programa y el de realizar una computación. Baker, quien es una fuente conceptual de Hiemann, lo dice en otras palabras: para él no podemos tener una idea del proceso hasta una fase de integración Bottom-Up.[3] Plantear, entonces, las especificaciones como un proceso funcional representado como en el árbol de Hiemann es a lo que se desearía llegar, pero no es necesariamente lo alcanzado. Sin hablar del problema de si las especificaciones del sistema cumplen realmente con sus requisitos, lo cual implica otras dificultades como, por ejemplo, si el programa creado realmente es el deseado. En este sentido, se debe tener mucho cuidado cuando se habla de desarrollo Top-

[3] Es importante aclarar que aquí no se implementa el programa, es decir que Baker advierte el problema sin recurrir a la implementación.

Down desde una perspectiva funcional, pues la funcionalidad del programa parece ser solamente accesible una vez implementado o después de haber ejercido ciertos procesos Bottom-Up, no en una fase abstracta de diseño.

En Hamilton y Zeldin existe también un uso de procesos Bottom-Up donde el objetivo o el problema se encuentran descritos o especificados solamente en la fase final de su construcción. Para lograr este proceso se parte de las escrituras de una subrutina que va progresivamente alcanzando niveles superiores para luego definir el problema.

Nuestra pregunta entonces es ¿qué pasa cuando seguimos el planteo de Hiemann? Planificamos de antemano todas las funciones del programa, por ejemplo, y vamos siguiendo lo planificado. ¿Qué riesgos nacen de esta estrategia? Podemos decirlo sin muchas vueltas: nos arriesgamos a usar un método cascada.

Método cascada

El método cascada es una derivación de lo que más tarde se llamará método Top-Down. Es importante aclarar que el método cascada es de una u otra forma anterior a lo que hemos descrito por programación Top-Down, pero por otro lado no es anterior (usando la terminología de Baker) al desarrollo Top-Down. Tal como mencionamos en la introducción, algunas variantes de desarrollo Top-Down ya existían en el ámbito de la cibernética, tanto las defendidas por Ashley o Wiener y Rosenblueth. Entonces no es extraña la confusión que pudo existir en tratar de usar el desarrollo Top-Down como programación Top-Down. Este ha sido el error de Benington, quien es visto como el "padre" del método cascada por haberlo usado durante el desarrollo del sistema SAGE durante la década de 1950. Para Benington, (1983) tal hijo era indeseado y no se correspondía con el proceso que él consideraba haber usado, que se acercaba a la aplicación de estándares ingenieriles clásicos cuyos preceptos se adecuan más con una metodología Top-Down y de programación estructurada. De hecho, esto lo justifica por la exigencia de "racionalización" del proceso de diseño del sistema SAGE y sosteniendo que solo podrían haber sido producidos por el uso de ambas metodologías. Lo interesante es que Benington (1983) reconoce que le hubiera gustado usar una postura "evolutiva y progresiva que posea otro método, el método espiral" (1983, p. 352). Pero antes de

buscar explicar las causas del porqué del uso del método cascada se necesita describir en qué consiste.

Tradicionalmente se ha citado a Royce[4] (1970) como el "padre" del método cascada, pero en realidad fue el primero en formalizarlo y describir los pasos del método cascada para criticarlo. Royce lo descompuso en 5 etapas: análisis de los requisitos y definición[5], diseño del sistema y programa, implementación y unidades de testeos, integración y testeo de sistemas y finalmente operación y mantenimiento (Royce, 1970). Estas etapas deben ocurrir secuencialmente, en donde se espera la finalización de una para pasar a la siguiente. Otra característica importante del método cascada es que todas las especificaciones que corresponden a todas las funcionalidades del programa son establecidas con antelación a la escritura del programa. Además, se intenta producir una gran cantidad de documentación para describir las funcionalidades y los procesos que ocurren dentro del programa o de los resultados esperables. Algunos autores más recientes, como Sommerville (2016), un especialista en diseño de programas, advierte que este proceso -en el cual la fase anterior debe ser completada antes de iniciar la siguiente- tiene mucho sentido para el desarrollo de los componentes (hardware), pero para el desarrollo de programas (software) es contra-productivo, en particular por el papel que tiene la información en este proceso. Por ejemplo, durante el proceso de diseño se identifican problemas de requisitos y durante la codificación se encuentran problemas de diseño, por lo que cada etapa se encuentra interconectada con las etapas previas. El proceso de desarrollo de un programa no es lineal, por lo que si una nueva información aparece en una etapa posterior, esta necesita ser contemplada en la etapa previa. La dificultad es que, al elegir este método, los clientes y los desarrolladores congelan prematuramente las especificaciones del programa para luego no modificarlas más. Pero, como dice Sommerville, los problemas son dejados para ser resueltos posteriormente (hasta puede involucrar reescribir una buena parte o todo el programa), son ignorados, o se crea un desvío para evitarlos. Las consecuencias de tal práctica es tener un programa que no cumple con los deseos del usuario o que está mal estructurado ya que el diseño fue

[4] Royce no nombra al método cascada como "método cascada" sino como un tipo de práctica defectuosa.

[5] Sospechamos que esta noción de definición debe emparentarse con la noción de especificación de Turner. Ver Turner (2018).

probablemente resuelto mediante algunos "trucos" en la fase de implementación. En general, el método cascada suele ser usado como ejemplo de una mala práctica en programación si no se toma los cuidados suficientes. Tanto Royce como Benington plantean ciertos cuidados que se deben tener durante la fase de diseño y critican las malas interpretaciones o usos del método. ¿De dónde vienen esas malas interpretaciones?

El proyecto SAGE fue un proyecto no solamente importante por cuestiones de innovaciones tecnológicas, sino también por la cantidad de personas que trabajaron en el proyecto[6]. En 1953, se contabilizaron 1800 personas trabajando en el proyecto y la mayoría no sabía programar. Se requirió formar a estas personas para poder programar un instrumento como la computadora, totalmente novedoso para la época. Una de las mejores formas de coordinar y organizar un grupo tan grande era plantear objetivos claros y etapas temporales bien definidas. Por ello requería de herramientas muy formales[7] para lograr una cohesión lógica al código que solo algunos pocos podían manejar. Además, un nuevo integrante al proyecto era nada menos que Simon, especialista en toma de decisiones y resolución de problemas en grandes organizaciones. Si bien su trabajo fue estudiar la organización del proyecto SAGE, no podemos saber cuánto influenció en el proceso usado por Benington. Pero lo que está claro es que mucho del desarrollo de programas posterior fue influenciado fuertemente por el proyecto SAGE. Muchos de los programadores que pasaron por el proyecto fueron luego creando la industria de software, desarrollada durante las tres décadas siguientes. Lo más probable, al no haberse encontrado en las fases de establecimientos de las especificaciones para el diseño del proyecto SAGE, es que estos programadores hayan reproducido lo que habían aprendido. Esto es, seguir objetivos predefinidos anteriormente, proceder en etapas sin saltar o volver a otra y documentar con profusión lo que se va haciendo para mejorar la comunicación entre programadores. Este proceso de teléfono descompuesto fue empeorando la perspectiva de desarrollo Top-Down para terminar creando al método cascada. En-

[6] El sistema permaneció operativo hasta la década de 1980.

[7] El formalismo es importante dado el tenor matemático que requiere para ser válido. El proyecto SAGE al ser un sistema crítico para la defensa de los Estados Unidos, requería de muchos cuidados y certezas. El formalismo era por lo tanto una buena herramienta para lograr este compromiso. Al usar formalismo se requiere por lo tanto tener bien en claro todas las especificaciones con anterioridad para crear un sistema válido no solamente en un sentido gramatical del uso de un lenguaje de programación sino también lógico.

tonces entendemos el rechazo de Benington, o las críticas de Royce hacia un método que resultó ser en realidad una reproducción de prácticas que habían sido muy válidas para el desarrollo del proyecto SAGE, pero de ninguna forma para la industria. En particular, esto es así debido a que la industria en general no suele tener en claro los requisitos que necesita y los va modificando durante el desarrollo del programa, algo que la formalidad requerida en SAGE nunca lo hubiera permitido.

Conclusiones

Hablar de método Top-Down en programación requiere entonces tener en claro, por un lado, si hablamos de desarrollo Top-Down o de programación Top-Down, y por otro, cuáles son los elementos que componen uno y otro. En efecto, hemos observado a través de Baker que no podemos plantear una programación Top-Down en términos funcionales. Esta limitación se convierte en una ventaja para el desarrollo Top-Down, cuyos beneficios están claramente presentes en la tesis de Simon de 1947 y de la cibernética. Finalmente, hemos descrito al método cascada como un intento fallido de aplicar el desarrollo Top-Down en programación cuyo resultado fue la reproducción indeseada de malas prácticas principalmente en la industria de software.

Referencias Bibliográficas

Ashby, W.R. (1960). *Design for a brain: The origin of adaptive behaviour*. Chapman & Hall.

Baker, F.T. (1975). Organizing for Structured Programing. En C.E. Hackl (Ed.), *Programming methodology* (pp. 38–86). Springer.

Benington, H.D. (1983). Production of Large Computer Programs. *IEEE Annals of the History of Computing, 5(4)*, 350–361. En: https://doi.org/10.1109/MAHC.1983.10102

Boehm, B.W. (1988). A spiral model of software development and enhancement. *Computer, 21(5)*, 61–72. En: https://doi.org/10.1109/2.59

Cocke, J., y Schwartz, J.T. (1970). *Programming Languages and their compilers.* Courant Institute of Mathematical Sciences.

Hamilton, M., y Zeldin, S. (1972). *Top-down-Bottom-up structured programming and program structuring.* Charles Stark Draper Laboratory.

Hiemann, P. (1975). A new look at the Program Development Process. En C.E. Hackl (Ed.), *Programming methodology* (pp. 11–37). Springer.

Mills, H.D. (1971). Top Down programming In large systems. En R. Rustin (Ed.), *Debugging Techniques In Large Systems* (pp. 41–55). Prentice Hall.

Royce, D.W.W. (1970). Managing the Development of Large Software Systems. *Proceedings IEEE WESCON,* August, 1–9.

Simon, H.A. (1947). *Administrative behavior: a study of decision-making processes in administrative organization.* Macmillan Co.

Sommerville, I. (2016). *Software engineering.* Pearson.

Turner, R. (2018). *Computational artifacts: towards a philosophy of computer science.* Springer.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM, 14(4),* 221–227. En: https://doi.org/10.1145/362575.362577.