Programación de Inteligencia Artificial

Conforme a contenidos del «Curso de Especialización en Inteligencia Artificial y Big Data»

Universidad de Castilla-La Mancha

Escuela Superior de Informática
Ciudad Real

Capítulo 1

Lenguajes de programación

David Vallejo Fernández, Santiago Sánchez Sobrino

Este capítulo sirve como punto introductorio a la programación, abordándola desde la perspectiva general del **proceso de desarrollo de software** y aumentando así el alcance que el primer término realmente tiene. El contexto para el cual se irá particularizando, no obstante, es el desarrollo de aplicaciones de IA (Inteligencia Artificial). Adicionalmente, se realiza un recorrido a alto nivel de aspectos fundamentales y buenas prácticas de desarrollo. En dicho recorrido, se discuten los conceptos generales de los lenguajes de programación compilados e interpretados, ofreciendo al lector una comparativa y estableciendo directrices que faciliten la toma de decisiones, a nivel de tecnología, cuando se aborda un proyecto de IA.

En este sentido, se discute un conjunto básico de las principales **características deseables de un lenguaje de programación para aplicaciones de IA**, con un especial énfasis en las que resultan relevantes para el programador. Posteriormente, y desde un punto de vista más concreto, se aborda la introducción de dos lenguajes de programación de referencia, **Python y R**, con respectivos casos prácticos que ejemplifican sus bondades.

Finalmente, el capítulo termina detallando aspectos más específicos que generalmente son transversales en el desarrollo de aplicaciones de IA, como el **procesamiento de datos** y el uso de lenguajes de marcas.

1.1. Introducción a la programación

Esta sección ofrece una una visión general del proceso de desarrollo de software, contextualizándolo para destacar la importancia que tiene más allá de la fase puramente vinculada a la programación. En este recorrido general y sintetizado, se abordará la relevancia de la capacidad de manejar abstracciones y de plantear

soluciones simples. Asimismo, esta visión se complementará con un listado acotado de buenas prácticas de desarrollo y generación de *clean code*. Por otro lado, esta sección resume las principales características de los lenguajes de programación compilados e interpretados, en el marco de herramientas para el desarrollo de proyectos de IA.

1.1.1. El proceso de desarrollo de software

La construcción de software es un proceso complejo y sofisticado que va más allá de lo que comúnmente conocemos como programación. En este apartado se ofrece una visión general de las fases más relevantes en las que este proceso se estructura. Asimismo, también se introducen dos aspectos que están relacionados con destrezas fundamentales a la hora de abordarlo: i) la capacidad de abstracción mediante el uso de metáforas y ii) la importancia de la simplicidad a la hora de plantear cualquier tipo de solución.

Fases principales

La industria del desarrollo o construcción de software ha madurado significativamente en los últimos 30 años, siendo posible identificar un conjunto de fases que conforman dicho desarrollo. Así, y desde una perspectiva de la ingeniería del software, las principales fases son las siguientes [McC04]:

- **Definición del problema**. En esta fase se especifica el prerrequisito más relevante de todo proyecto software, el cual consiste en una identificación clara y precisa del problema que se pretende resolver. En este punto, no se hace referencia a posibles soluciones. La salida esperada es un breve documento, de una o dos páginas, que defina el problema.
- **Especificación de requisitos**. Esta fase consiste en generar una descripción detallada acerca de lo que el sistema software debe hacer. Uno de los aspectos fundamentales de esta fase reside en garantizar que la funcionalidad del sistema esté guiada por el usuario final, y no por el programador. En otras palabras, un listado exhaustivo de requisitos representa el contrato entre el cliente y el desarrollador y permite que el cliente lo revise y valide antes de abarcar otras fases. Es importante prestar mucha atención a esta fase con el objetivo de minimizar el número de cambios sobre el sistema, especialmente en las etapas de diseño y desarrollo.
- **Planificación del desarrollo**. Esta fase tiene como objetivo detallar la planificación del resto de fases de la construcción de software. Existe una dependencia clara entre esta fase y la anterior, ya que el alcance del proyecto vendrá determinado por el número y complejidad de los requisitos previamente identificados.

- **Arquitectura software o diseño de alto-nivel**. Esta fase trata el diseño global de un proyecto software, entendido como la actividad que sirve de nexo entre la especificación de requisitos y la codificación. En proyectos complejos, la arquitectura software define el mapa global sobre el que se apoyará un conjunto de aspectos de diseño más específicos que, posteriormente, se puedan utilizar como punto de referencia a la hora de programar. En esta fase se manejan conceptos como módulos, subsistemas o paquetes.

- **Diseño detallado**. Esta fase persigue la especificación de un diseño más cercano al código, considerando las entidades que forman parte del mismo y sus relaciones. En esta fase se manejan conceptos como el tradicional diagrama de clases, en caso de utilizar un paradigma de programación orientada a objetos. Tanto en esta fase, como en la anterior, debe prestarse atención a características como la reducción de la complejidad, la facilidad de mantenimiento, el bajo acoplamiento, la extensibilidad o la capacidad de reutilización, entre otras.

- **Codificación y depuración**. Esta fase engloba tanto el proceso de programación en sí, entendido como la escritura de un conjunto de sentencias que persiguen la generación de un resultado, como el proceso de depuración o identificación y corrección de las causas que originan errores. En función del paradigma de programación empleado, el nivel de abstracción empleado será diferente. Por ejemplo, en la programación estructurada, el concepto de rutina o función representa un elemento fundamental, mientras que en la programación orientada a objetos la clase es la herramienta esencial para programar.

- **Pruebas unitarias**. Esta fase aborda la escritura de código que posibilite la prueba de cierto código, como por ejemplo una clase, de manera independiente al resto de código que conforma el sistema. La fase de pruebas tiene una gran relevancia para reducir el número de errores que se puedan producir en tiempo de ejecución. De hecho, existen procesos de desarrollo de software que giran alrededor del concepto de prueba, como por ejemplo TDD (Test-Driven Development) [Bec04].

- **Pruebas de integración**. Esta fase tiene como objetivo combinar la ejecución y pruebas de dos o más clases, módulos o subsistemas, de forma que se puedan evaluar las interacciones existentes entre ellas. Es importante realizar pruebas de integración desde el principio hasta el final del desarrollo del proyecto.

- **Integración**. Esta fase aborda la problemática de mezclar nuevo código con código existente. Al igual que ocurre con la fase de pruebas de integración, resulta recomendable abordar esta fase lo antes posible para mitigar la complejidad de la misma cuando el proyecto crece de tamaño. La idea de integración continua se ha acuñado en los últimos años como filosofía de trabajo [HF10].

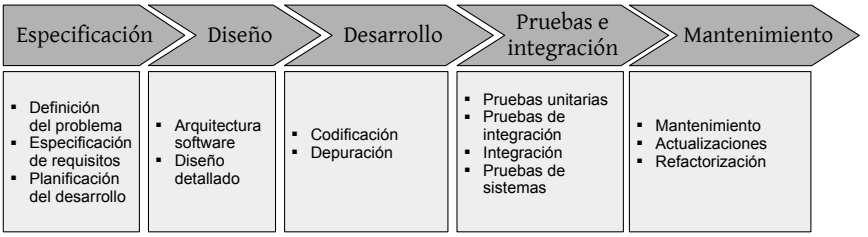

Figura 1.1: Fases globales del proceso de desarrollo de software.

- **Pruebas de sistemas**. Esta fase representa la última actividad relativa a pruebas de software antes de llevar a cabo el despliegue de un sistema, o una actualización del mismo, en producción.
- **Mantenimiento**. Esta última fase está relacionada con la actualización del código, e incluso en ocasiones del diseño, lo cual deriva en procesos de refactorización, que conforman el núcleo de un proyecto de desarrollo de software. Las actualizaciones de código son una consecuencia de los cambios introducidos en un proyecto después de su lanzamiento. Aunque pueda resultar sorprendente, los costes de mantenimiento suelen representar la mayor partida económica de un proyecto software a largo plazo.

Más allá del mantenimiento. De acuerdo a Fred Books, "el problema fundamental del mantenimiento de los programas es que arreglar un defecto tiene una probabilidad considerable (20-50 %) de introducir otro. Por lo tanto, todo el proceso consiste en dar dos pasos adelante y uno atrás".

Como puede apreciar, en el exhaustivo listado anterior se reflejan las 4 grandes fases generales que probablemente ya tuviera interiorizadas: i) especificación, ii) diseño, iii) desarrollo y iv) pruebas. La figura 1.1 muestra este listado de manera visual. Desafortunadamente, no es sencillo encontrar un equilibrio entre el nivel de formalidad del anterior listado y la posibilidad de relajar u obviar alguna de las fases mencionadas. De hecho, es bastante probable que, internamente, las haya agrupado alrededor del **concepto de programación**, especialmente si hasta ahora ha trabajado en proyectos personales o prototipos que no vayan a alcanzar un nivel significativo de madurez tecnológica.

Si bien la fase de programación o codificación representa una de las principales actividades en proyectos de prototipado rápido, es importante **conocer el contexto general del proceso completo de construcción de software**. Esto es especialmente relevante tanto para saber dónde poner el esfuerzo en cada momento como para no olvidar las dependencias, y consecuencias, existentes entre fases. A modo de ejemplo, un error en la fase de diseño puede acarrear consecuencias desastrosas en la fase de codificación y depuración, como el incremento de la complejidad de la solución o el aumento en el tiempo empleado en programar la misma.

Niveles de Madurez Tecnológica. Procedentes del término en inglés TRL (Technology Readiness Levels), representan una medida de la madurez de una tecnología, facilitando el uso de una nomenclatura consistente y uniforme. Los niveles TRL utilizados en la Unión Europea van de 1 a 9: 1) principios básicos observados, 2) concepto de tecnología formulado, 3) prueba experimental de concepto, 4) tecnología validada en laboratorio, 5) tecnología validada en un entorno relevante, 6) tecnología demostrada en un entorno relevante, 7) demostración del prototipo del sistema en un entorno operativo, 8) sistema completo y calificado, 9) sistema real probado en el entorno operativo.

Abstracción y uso de metáforas

Una de las capacidades más relevantes que un desarrollador debe adquirir es la **capacidad de abstracción**, es decir, la capacidad de generalizar y reutilizar ideas independientemente del contexto particular. En este sentido, el uso de metáforas ha resultado ser tremendamente útil conforme la industria del desarrollo de software ha ido evolucionando. De hecho, el concepto de modelado software está inherentemente relacionado con la aplicación de metáforas o analogías, las cuales facilitan la generación de resultados cuando se compara un tema o dominio que no se entiende del todo bien, al menos al principio, con otro que sí se entiende de manera efectiva.

Existe un amplio listado de metáforas que han servido para simplificar la complejidad asociada al desarrollo de software desde una perspectiva integral [McC04]. Quizá una de las más representativas viene derivada de la expresión *escribir código*, la cual sugiere que desarrollar un programa es similar a escribir una carta. Otra metáfora bastante conocida es la que compara el hecho de cultivar una cosecha con la creación de software. Esta se basa en la idea de plantear soluciones incrementales: diseño de una parte del sistema, codificación de esa parte, pruebas e integración. En esencia, plantear una solución que se va generando poco a poco minimiza el riesgo de encontrarse en una situación donde la complejidad subyacente es tal que dificulta enormemente la gestión de un proyecto.

La importancia de documentar código. En relación a la analogía entre escribir código y escribir una carta, en este punto se anima al lector a recordar la clásica afirmación *documentation is a love letter that you write to your future self*.

En el **contexto del desarrollo de aplicaciones de inteligencia artificial**, la utilización de metáforas es especialmente relevante. Considere, por ejemplo, que el concepto de red neuronal es en realidad una analogía, aunque inexacta, de cómo funcionan y se comunican las neuronas en un cerebro biológico. Hoy en día, las redes neuronales se emplean en programas capaces de reconocer patrones y aprender de forma automática, entre otras funciones, y representan el corazón del *deep learning*.

Otro caso relevante es el de los sistemas basados en reglas, ya sean definidas por un experto humano o aprendidas de manera automática, los cuales recrean la forma en la que los humanos solemos pensar: si se da una determinada situación, definida por un conjunto de condiciones, entonces se lanzan una o varias acciones. Una metáfora más reciente es la de los *web crawlers*, entendidos como programas que analizan y extraen enlaces o hipervínculos de páginas web, no dejan de ser una analogía de una telaraña explorada por una cazadora (araña) en búsqueda de presas (información). El concepto de *web scrapping*, relacionado con el anterior, se centra en la extracción de datos o contenido.

Incluso existen teorías científicas, como la lógica difusa o *fuzzy logic* [Zad99], que acercan la forma en la que nosotros razonamos a la definición de modelos matemáticos que permitan su uso por parte de máquinas o programas. En esencia, la lógica difusa es una forma de lógica multi-valuada en el que los valores de verdad de las variables pueden tomar valores en el rango [0, 1]. Por el contrario, en la lógica booleana las variables solo pueden tomar los valores enteros de 0 o 1. La lógica difusa se fundamenta en el hecho de que las personas toman decisiones atendiendo a información imprecisa o vaga (típicamente, no numérica). Así, surge la idea de conjunto difuso como medio matemático para representar dicha información. Una de las aplicaciones clásicas de la lógica difusa, combinada con los sistemas basados en reglas, está representada por los sistemas de control. De hecho, una parte significativa de los productos de consumo, como por ejemplo el controlador del aire acondicionado, se basan en lógica difusa.

Otra metáfora significativa es la de agente inteligente, definido en el ámbito de la IA como cualquier entidad capaz de percibir lo que ocurre en el medio o contexto en el que habita, mediante la ayuda de sensores, y actuar en consecuencia, mediante la ayuda de actuadores, generando normalmente algún tipo de cambio en dicho medio o contexto (ver figura 1.2). En este caso, la metáfora de agente se hace directamente con el concepto de ser humano o individuo. Cuatro son las características fundamentales de un agente inteligente: i) autonomía, de manera que un agente actúa sin la intervención directa de terceras partes, ii) habilidad social, de forma que los agentes interactúan entre sí y se comunican para alcanzar un objetivo común, iii) reactividad, de manera que un agente actúa en función de las percepciones del entorno, y iv) proactividad, de manera que un agente puede tomar la iniciativa en lugar de ser puramente reactivo. Los agentes inteligentes conforman sistemas multi-agente, los cuales están asociados al dominio de la inteligencia artificial distribuida e incluso al concepto de *programación orientada a agentes*.

La importancia de la simplicidad

Si la capacidad de pensar, modelar y desarrollar de forma abstracta es una de las habilidades más importantes de un ingeniero software, **la habilidad de plantear soluciones sencillas** quizá sea la que más impacto y alcance tiene cuando se aborda cualquier proyecto software. Piense que los problemas que tendrá que resolver, desde el punto de vista de la construcción de software, son lo suficientemente

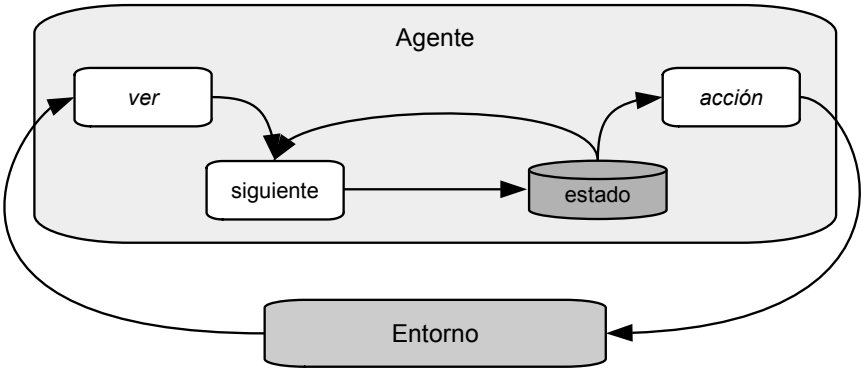

Figura 1.2: Visión abstracta del funcionamiento interno de un agente.

complejos como para que sus soluciones también lo sean. Recuerde, además, y como se ha mencionado anteriormente, que los costes de mantenimiento se suelen llevar la mayor parte de costes económicos de un proyecto software a largo plazo. Todo guarda relación con la simplicidad.

Por otro lado, el desarrollo de soluciones basadas en IA no deja de ser un caso particular de construcción de software, por lo que el planteamiento de soluciones simples sigue teniendo la misma o más importancia. De hecho, el desarrollo de soluciones basadas en IA, al menos en lo que se refiere a proyectos experimentales, está íntimamente relacionado con el **concepto de prototipado**, donde el pragmatismo y la simplicidad deberían ser protagonistas. Suele ser bastante común prototipar una solución en un lenguaje como Python o LUA para, posteriormente, generar una solución con más alcance en C++ (especialmente si existen restricciones en términos de eficiencia, como se discutirá más adelante).

Tomando decisiones. Ante un problema para el cual resulta posible aplicar diferentes soluciones, ¿cuánto tiempo dedica a evaluar qué solución es la más sencilla y, por lo tanto, ofrece mejores perspectivas de desarrollo y mantenibilidad de código?

Puede que el lector esté familiarizado con el principio KISS (Keep it Simple Stupid!), el cual pone de manifiesto que la mayoría de sistemas funcionan mejor si se mantienen lo más sencillos posible. En otras palabras, la simplicidad se convierte en un objetivo principal en el diseño, obviando cualquier atisbo de complejidad innecesaria.

Simplicidad como regla general. La navaja de Ockham es un principio metodológico y filosófico según el cual "en igualdad de condiciones, la explicación más sencilla suele ser la más probable". En el ámbito de desarrollo de software, cuando dos soluciones generan, potencialmente, el mismo resultado, el más sencillo tendría que prevalecer sobre el otro. Incluso Leonardo Da Vinci afirmó que "la simplicidad representa el máximo nivel de sofisticación".

1.1.2. Aspectos fundamentales de desarrollo

La elección del lenguaje de programación

En este punto se incluye una breve descripción de algunos de los lenguajes de programación que más protagonismo han tenido y que más se utilizan en la actualidad [McC04]. El objetivo que se persigue es el de ofrecer al lector una visión general de dichos lenguajes de programación. El listado que se presenta a continuación está ordenado alfabéticamente, y no por orden de importancia o relevancia.

- **Ada**. Lenguaje de programación de propósito general, orientado a objetos y con un soporte nativo para la construcción de sistemas concurrentes. Desarrollado originalmente por el Departamento de Defensa de los Estados Unidos, Ada está particularmente pensado para sistemas empotrados y sistemas de tiempo real. Una de las características más relevantes del lenguaje es el de la encapsulación de datos, forzando al programador a elegir entre aquellos elementos que son públicos o privados de cada clase o paquete. Actualmente, el uso principal de Ada está vinculado con los sectores militar y aeroespacial.
- **C**. El lenguaje C es considerado como un lenguaje de programación con un nivel de abstracción medio, originalmente asociado al desarrollo del sistema operativo UNIX. Sin embargo, C ofrece características de alto nivel, como las estructuras o las propias sentencias de control incluidas en el lenguaje.
- **C++**. Este lenguaje de programación orientado a objetos, compatible con C, ofrece características como el soporte a clases, polimorfismo, plantillas y una biblioteca estándar altamente potente y flexible. Es considerado como el estándar de facto en el ámbito de las aplicaciones gráficas interactivas.
- **C#**. Este lenguaje de programación, desarrollado por Microsoft, tiene una sintaxis similar a otros lenguajes populares, como C++ o Java, ofrece mecanismos de alto nivel, como la orientación a objetos, y proporciona herramientas de desarrollo especialmente pensadas para plataformas Microsoft.

- **Ensamblador**. Se trata de un lenguaje de bajo nivel caracterizado por establecer una relación directa entre instrucciones máquina y sentencias del propio lenguaje. Precisamente, debido a esta característica, un lenguaje ensamblador siempre estará vinculado inherentemente a un determinado procesador. Con carácter general, no se hará uso de lenguaje ensamblador salvo que existan unas restricciones enormes en lo que se refiere a optimización de código (tiempo de ejecución o tamaño del código).
- **Java**. Lenguaje de programación, similar en sintaxis a C y C++, creado por Sun Microsystems y que ofrece un entorno de desarrollo de alto nivel. Una de las principales características del lenguaje es la portabilidad, gracias al concepto de *Java Virtual Machine* y a la posibilidad de convertir el código fuente Java en *byte code*. Este código se puede ejecutar en cualquier plataforma que tenga disponible una máquina virtual. El auge de Java estuvo principalmente vinculado, en su momento, al desarrollo de aplicaciones web.
- **JavaScript**. Lenguaje de *scripting* interpretado que se ha utilizado principalmente para el desarrollo de la funcionalidad relativa a la parte del cliente en aplicaciones web.
- **PHP**. Lenguaje de *scripting* que se ha utilizado esencialmente para el desarrollo de la parte del servidor en aplicaciones web.
- **Python**. Lenguaje de alto nivel, versátil, interpretado y con soporte para la programación orientada a objetos, que se utiliza en un amplio rango de dominios, desde el desarrollo de la parte del servidor de aplicaciones web hasta el prototipado rápido de componentes de inteligencia artificial.
- **SQL**. Lenguaje estándar de referencia en lo que se refiere a consultas, actualización y gestión de bases de datos relacionales. De hecho, SQL es el acrónimo de *Structured Query Language*. Se trata de un lenguaje declarativo, es decir, que no se basa en el uso explícito de una secuencia de operaciones como tal, sino que está guiado por los resultados de las operaciones que se aplican.
- **Visual Basic**. Lenguaje de programación de alto nivel, orientado a objetos, con soporte a la programación visual y que deriva de la versión de BASIC creada para aplicaciones de escritorio en entornos Microsoft. Su acrónimo, BASIC, significa *Beginner's All-purpose Symbolic Instruction Code*.

Popularidad de los lenguajes de programación. Existen diversos índices que reflejan la demanda actual de los lenguajes de programación más representativos. Un ejemplo relevante es el del índice TIOBE, utilizado desde principios de siglo.

En el listado anterior, se ha mencionado el atributo *interpretado*, asociados al concepto de lenguaje de programación. Esta característica, junto al concepto de lenguaje de programación *compilado*, serán objeto de estudio más adelante, ya que dichas características resulta especialmente relevante a la hora de escoger un lenguaje de programación para el desarrollo de IA.

Convenciones de desarrollo

En el desarrollo de software debe existir una relación entre la arquitectura general propuesta para resolver un problema, materializada mediante el diseño, y la implementación final que le da soporte y que se realiza a través de un lenguaje de programación concreto. Esta coherencia también ha de reflejarse a nivel interno, de forma que la solución a nivel de código sea consistente. Precisamente, esta consistencia es la que idealmente ha de derivar en el uso de guías o convenciones de desarrollo a la hora de utilizar, por ejemplo, nombres para las clases, las funciones, las variables e incluso para los comentarios de código.

Piense en un proyecto en el que participan, concurrentemente, un equipo de 8 desarrolladores software. Resulta evidente asumir la existencia de una **guía que sirva como referencia para escribir un código que sea consistente**. De otro modo, cada ingeniero utilizaría, por ejemplo, una convención de nombrado diferente para su código. Esta situación dificultaría tanto la coordinación interna del equipo como el futuro mantenimiento del código.

Un ejemplo de guía de estilo para el desarrollo de código en C++ es la utilizada internamente por Google[1]. Otro ejemplo representativo, para el caso del lenguaje de programación Python, es la guía de estilo PEP (Python Enhancement Proposal) 8[2].

Visión general sobre buenas prácticas de desarrollo

En este apartado se pretende ofrecer al lector una visión general, y a muy alto nivel, de buenas prácticas de desarrollo cuando se aborda un proyecto. Debido a la profundidad de la temática, a continuación se esbozan algunas de las cuestiones más importantes [McC04].

A nivel de **desarrollo de código**:

- Identifique qué aspectos del diseño son susceptibles de abordar de manera previa a la codificación y cuáles se pueden trabajar en el momento de escribir el código.
- Utilice un convenio de nombrado de acuerdo al lenguaje de programación elegido.

[1] https://google.github.io/styleguide/cppguide.html
[2] https://www.python.org/dev/peps/pep-0008/

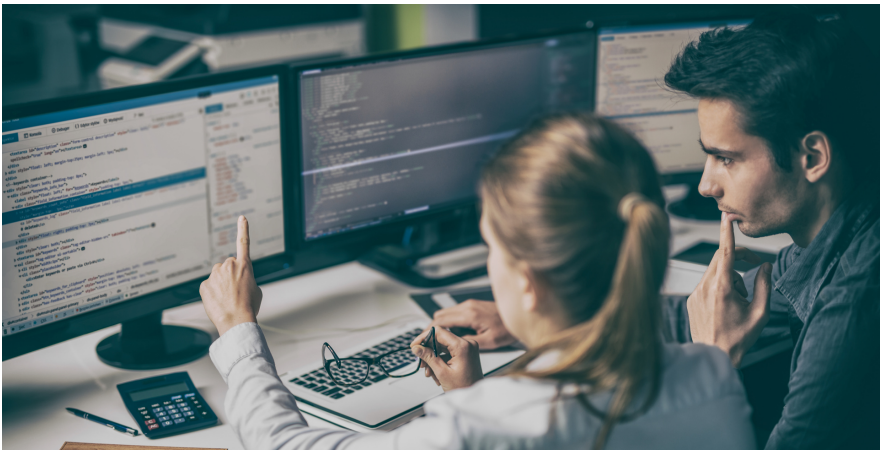

Figura 1.3: Paradigma de programación en parejas o *pair programming*.

- Adopte prácticas de programación que tengan en cuenta cómo se van a gestionar posibles errores o excepciones, cuestiones de seguridad de código y diseño de interfaces, entre otras.
- Evalúe el impacto del uso de tecnologías más maduras o más recientes en el contexto de su proyecto.

A nivel de **coordinación y trabajo en equipo**:

- Defina un esquema de integración de código, considerando los pasos que han de darse antes de desplegar código en un sistema en producción.
- Elija una técnica de desarrollo de software que encaje con su filosofía de trabajo, como por ejemplo *Pair Programming* [BC04] (ver figura 1.3).

A nivel de **pruebas y calidad**:

- Defina si los programadores usarán un modelo de desarrollo dirigido por tests, como TDD [Bec04], lo cual implica codificar primero las pruebas.
- Defina si los programadores codificarán pruebas unitarias.
- Defina si los programadores llevarán a cabo pruebas de integración antes de promocionar su código.
- Defina si los programadores realizarán una revisión cruzada de código.

A nivel de **herramientas y entorno de desarrollo**:

- Elija y use un sistema de control de versiones, como Git[3].

[3] https://git-scm.com/

- Elija lenguaje, versión del lenguaje y versión del compilador (o intérprete) del lenguaje.
- Elija y valore las ventajas de usar un *framework* de desarrollo concreto.
- Identifique y valore herramientas adicionales que puedan facilitar el proceso de desarrollo, como el depurador, el *framework* de pruebas o incluso la generación automática de esqueletos de código.

Principios de *Clean Code*

Si usted conoce el concepto de deuda técnica, probablemente también le resulte familiar el término *clean code*. Si no es así, probablemente intuya que una mala decisión en el diseño o incluso en la programación de código puede acarrear consecuencias que son difíciles de gestionar conforme pasa el tiempo. Implícitamente, ya hemos tratado el **concepto de deuda técnica** cuando comentamos que la etapa de mantenimiento de un proyecto software es la que suele estar relacionada con la mayor partida económica de un proyecto software a largo plazo. Incluso si se trata de un proyecto personal o de un prototipo interno, la diferencia entre un buen código y un mal código siempre marca la diferencia.

Una de las referencias bibliográficas más relevantes en el ámbito del clean es precisamente el libro que lleva su nombre: *Clean Code, A Handbook of Agile Software Craftsmanship* [Mar08]. Se anima al lector a revisar los principios fundamentales del código limpio y a descubrir la analogía existente entre ellos y la importancia de la simplicidad previamente expuesta.

Código limpio. La definición de *Clean Code* puede tener múltiples acepciones. Aquí se incluye la traducción de una de ellas, formulada por Bjarne Stroustrup, el creador del lenguaje de programación C++: "Me gusta que mi código sea elegante y eficiente. La lógica del código debería ser sencilla para dificultar la depuración de errores, con un número mínimo de dependencias para facilitar el mantenimiento, con una gestión de errores completa de acuerdo a una estrategia bien definida, y con un rendimiento cercano al óptimo para evitar que otros desarrolladores generen código complejo como consecuencia del primero. El código limpio hace una cosa y la hace bien."

1.1.3. Lenguajes de programación compilados e interpretados

Visión general

A la hora de abordar el diseño de una aplicación, tendrá que tomar la decisión de utilizar un lenguaje de programación compilado o interpretado para escribir su código fuente. Como puede imaginar, cada tipo de lenguaje tiene sus fortalezas y debilidades. Esencialmente, la decisión de emplear un lenguaje interpretado estará relacionada con las restricciones de tiempo existentes a la hora de abordar un

proyecto software y la facilidad de realizar futuros cambios en el mismo. La cara negativa de la moneda tiene que ver con el rendimiento. Así, cuando se emplea un lenguaje de programación interpretado, se está incurriendo en unos costes de ejecución elevados para emplear una herramienta que acelera la velocidad de desarrollo. En otras palabras, un lenguaje interpretado puede ser más adecuado para peticiones de desarrollo sobrevenidas, mientras que un lenguaje compilado sería una mejor opción, al menos a priori, para una petición predefinida.

Entrando más en detalle, un **lenguaje compilado** se basa en la escritura de programas que, una vez compilados, se expresan en instrucciones de una arquitectura máquina concreta. A modo de ejemplo, la operación de suma de dos valores numéricos en el lenguaje de programación C++ sería trasladada directamente en la operación ADD en código máquina. Si bien un programa se podría implementar utilizando directamente código máquina, también denominado ejecutable o binario, el nivel de abstracción tan bajo no haría práctico el desarrollo de proyectos complejos. Además, sería necesario realizar una implementación diferente para cada arquitectura hardware. Esta es una de las principales razones de la existencia de los compiladores. Un compilador es un programa que traduce código fuente, programado en un lenguaje de programación de alto nivel, en código máquina para una determinada arquitectura.

El proceso de compilación se estructura, a su vez, en varios pasos intermedios, tal y como se muestra en la figura 1.4. Se pueden distinguir dos fases principales. La fase de *frontend* realiza el análisis léxico, sintáctico y semántico de los ficheros de entrada que representan el código fuente del proyecto. La salida de esta fase es un *código intermedio* independiente de la arquitectura hardware final. La fase de *backend*, y particularmente el *optimizador*, toma como entrada dicho *código intermedio* y lo mejora mediante diversas estrategias, como la eliminación de código muerto. Posteriormente, el *generador de código* ofrece como salida el código binario vinculado a una determinada arquitectura, el cual también es optimizado por el generador.

Resulta muy común enlazar bibliotecas de código ya existente con el código binario generado por el compilador para aprovecharse de las ventajas de la reutilización de código. Así, el enlazador, otro componente de los compiladores modernos, posibilita el **proceso de enlazado** de código a través de dos formas generales: i) estático, donde las dependencias de código se resuelven en tiempo de enlazado, generando un ejecutable autocontenido que no requiere la instalación de bibliotecas externas en la máquina destino, y ii) dinámico, donde las dependencias de código se resuelven en tiempo de ejecución, generando un ejecutable final de menor tamaño pero que requiere la instalación previa de bibliotecas externas. En [VC15] se ofrece una descripción en detalle del proceso de compilación para el caso particular del conjunto de compiladores GCC (GNU Compiler Collection).

Por el contrario, un **lenguaje interpretado** es aquel en el que las instrucciones generadas no se ejecutan directamente sobre una máquina destino, sino que se leen y ejecutan por otro programa, normalmente escrito en el lenguaje de la máquina destino. Si retoma el ejemplo anterior de la suma de dos valores numéricos, en este

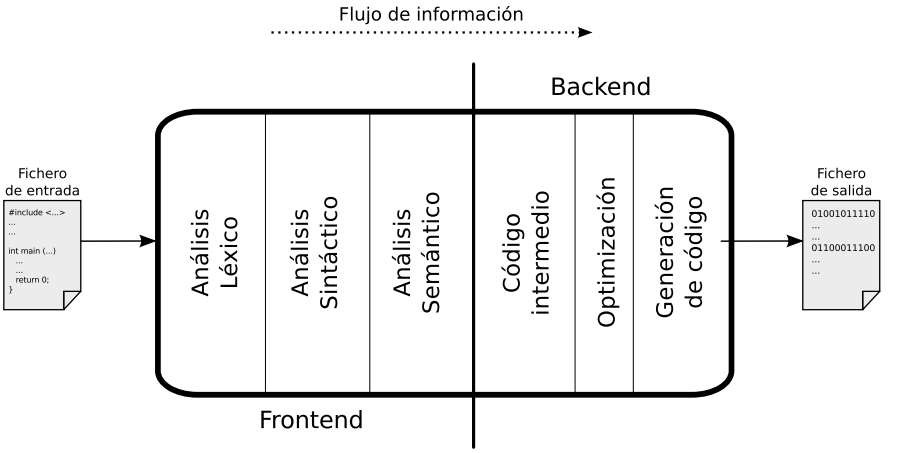

Figura 1.4: Fases del proceso de compilación [VC15].

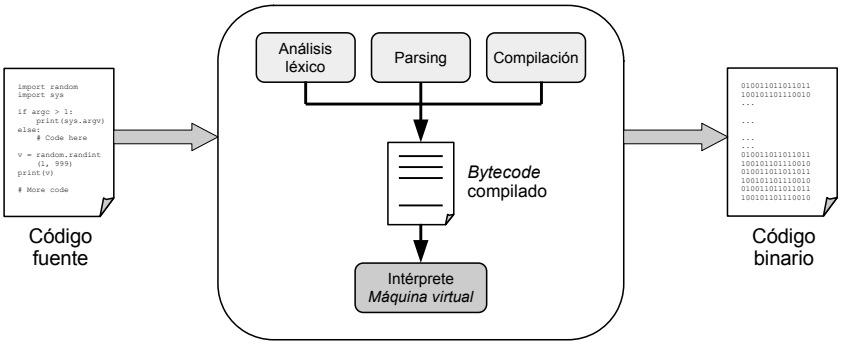

Figura 1.5: Visión general de las fases principales de un intérprete Python.

caso la misma operación de suma sería reconocida por el intérprete en tiempo de ejecución, el cual la traduciría a una función del tipo add(a, b) y que, posteriormente, ejecutaría a través de la instrucción máquina ADD. La figura 1.5 muestra, de manera gráfica, las principales fases abordadas por un intérprete Python.

El lenguaje de programación Java. Algunos autores afirman que los lenguajes de programación, en sí mismos, no se pueden clasificar en compilados o interpretados. Son las implementaciones específicas de un lenguaje las que realmente reflejan esta característica. En el caso de Java, están involucrados componentes como la máquina virtual de Java (*Java Virtual Machine*), compiladores nativos como *gcj* e intérpretes para el propio lenguaje, como *BeanShell*. Así pues, ¿qué tipo de lenguaje es Java?

Comparativa

En primer lugar, es importante destacar que, desde el punto de vista funcional, es posible implementar mediante un lenguaje interpretado cualquier programa que se pueda implementar con un lenguaje compilado, y viceversa. Sin embargo, cada opción, como se ha introducido anteriormente, tiene sus ventajas y desventajas. Estas se resumen a continuación.

Con **respecto a los lenguajes compilados**,

- Generación de código eficiente, que se puede ejecutar un número arbitrario de veces, para la máquina destino. En otras palabras, la sobrecarga incurrida por el proceso de compilación, una vez que el código ha sido validado, se reduce a un único proceso.
- Posibilidad de aplicar optimizaciones durante el proceso de compilación.
- Incremento del rendimiento con respecto a un lenguaje interpretado.

Con **respecto a los lenguajes interpretados**,

- Facilidad a la hora de implementar y prototipar código.
- Con carácter general, reducción de la complejidad a la hora de emplearlos como toma de contacto en relación a un lenguaje compilado.
- El código se puede ejecutar *al vuelo*, sin necesidad de un proceso previo de compilación.

Teniendo como referencia estas cuestiones, se establecen **dos escenarios generales que pueden servir como referencia** a la hora de elegir un lenguaje de programación compilado o interpretado:

1. Si el contexto del proyecto lo permite, un lenguaje interpretado resultaría más adecuado para llevar a cabo tareas de prototipado rápido o para generar una primera versión de un proyecto con agilidad. Como se ha introducido en este capítulo, las aplicaciones de IA en las que se necesiten probar y adaptar comportamientos con cierta rapidez representan un candidato ideal con respecto a dicha elección.
2. Las tareas más intensas, desde el punto de vista computacional, o aquellas que se ejecuten con más frecuencia dentro de un proyecto software, se podrían relacionar con un lenguaje compilado. Por el contrario, aquellas menos intensas, como por ejemplo una interfaz de usuario, se podrían relacionar con un lenguaje interpretado.

No todo es blanco o negro. Note que en un mismo proyecto software pueden convivir varios lenguajes de programación. Por ejemplo, se podría utilizar un lenguaje compilado para implementar el núcleo funcional del proyecto y un lenguaje interpretado para aquellas cuestiones menos relevantes desde el punto de vista computacional.

1.1.4. Resumen

En esta sección se ha ofrecido una visión general del proceso de desarrollo de software, entendido como un proceso complejo que va más allá de la programación. Particularmente, se han agrupado las fases identificadas en cuatro grandes etapas: especificación, ii) diseño, iii) desarrollo y iv) pruebas. Asimismo, se ha establecido la relación de este proceso con dos destrezas fundamentales que deberían interiorizarse por parte de cualquier ingeniero software: i) el uso de metáforas como mecanismo para manejar de manera eficaz la abstracción y ii) la importancia de la simplicidad como eje fundamental para diseñar, programar y mantener código. Esta introducción general ha quedado complementada con una visión general sobre buenas prácticas de desarrollo y referencias a cuestiones relacionadas con *clean code*.

Por otro lado, se ha ofrecido al lector una comparativa general de lenguajes de programación compilados e interpretados, relacionando la misma con su uso como herramienta para desarrollar proyectos de IA. En la siguiente sección se abordarán las características deseables en un lenguaje de programación para IA.

1.2. Programación de Inteligencia Artificial

La programación de aplicaciones de IA comparte la gran mayoría de aspectos y cuestiones a considerar por parte de otro tipo de proyecto donde el desarrollo software sea el principal protagonista. Sin embargo, es posible identificar un conjunto de características concretas que han de valorarse cuando se aborda un proyecto de IA. Particularmente, en esta sección se discute un listado de 5 características específicas a tener en cuenta a la hora de elegir un lenguaje de programación. Este listado se complementa con fragmentos de código y con un caso práctico diseñado para ilustrarlas y compararlas cuando se usan los lenguajes de programación C y Python, respectivamente.

Bibliografía

[BC04] K. Beck and Andres C. *Extreme Programming Explained: Embrace Change, 2nd Edition*. Addison-Wesley Professional, 2004.

[Bec04] K. Beck. *Test Driven Development: By Example, 1st Edition*. Addison-Wesley Professional, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[HF10] J. Humbel and .D Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[Mar08] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice-Hall, 2008.

[McC04] S. McConnel. *Code Complete: A Practical Handbook of Software Construction, 2dn Edition*. Microsoft Press, 2004.

[SHG+14] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D Ebner, C. Vinay, and M. Young. Machine learning: The high interest credit card of technical debt. *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, pages 1–9, 2014.

[SN19] Russell. S. and P. Norvig. *Artificial Intelligence: A Modern Approach, 4th US ed.* Prentice Hall, 2019.

[Str13] B. Stroustrup. *The C++ Programming Language, 4th Edition*. Addison Wesley, 2013.

[VC15] D. Vallejo and Martín C. *Desarrollo de Videojuegos. Un Enfoque Práctico. Módulo 1: Arquitectura del Motor*. Amazon CreateSpace, 2015.

[Zad99] L.A. Zadeh. Fuzzy logic = computing with words. *Computing with Words in Information*, Intelligent Systems 1:3–23, 1999.