Aporte a la confiabilidad de programas C++ mediante una herramienta ad-hoc para implementar contratos

Garcia Justo[1], Insfrán Jordán Francisco[1], y Diaz Zamboni Javier Eduardo[1]

[1] Laboratorio de Informática y Computación Aplicada, Facultad de Ingeniería, Universidad Nacional de Entre Ríos, Oro Verde, Entre Ríos, Argentina
justo.garcia@ingenieria.uner.edu.ar
{jordan.insfran,javier.diaz}@uner.edu.ar

Resumen. El diseño por contrato es una metodología en ingeniería de software propuesta por Bertrand Meyer para mejorar la calidad de la construcción de software. A pesar de ser una metodología ampliamente reconocida, se observa que su puesta en práctica no tiene el mismo nivel de aplicación. Con el objetivo de aportar a mejorar la confiabilidad del software desarrollado en C++, en este trabajo se presenta el desarrollo y análisis de una biblioteca para aplicar contratos en dicho lenguaje. La herramienta no depende de extensiones por fuera del estándar de C++, presenta una sintaxis clara, una implementación eficiente y provee una integración con el mecanismo de manejo de excepciones. Está disponible para que equipos de desarrollo que buscan mejorar la confiabilidad de sus desarrollos incorporando diseño por contratos.

Palabras Clave: Contratos · C++ · Calidad del Software · Ingeniería de Software · Confiabilidad

1 Introducción

El diseño por contrato (DbC, por sus siglas en inglés) es una metodología en ingeniería de software, propuesta por Bertrand Meyer, pensada para mejorar la calidad de la construcción de software. Permite reducir la cantidad de código fuente en un programa, evitando el chequeo redundante de parámetros. Se basa en una especificación estricta de las interfaces, incluyendo la especificación de responsabilidades de chequeo por parte de los clientes. De esta manera, los contratos proveen una especificación precisa de la semántica de los componentes, [6,20].

A pesar de ser una metodología ampliamente reconocida, se observa que la puesta en práctica no tiene el mismo nivel de aplicación como lo tiene su reconocimiento. Esto puede deberse a varios factores, como la falta de soporte nativo para contratos en muchos lenguajes de programación, que la formación en programación no haya incorporado estos conceptos, la carencia de herramientas adecuadas para su implementación, la resistencia al cambio por parte de los

desarrolladores acostumbrados a otras metodologías, y la percepción de que la especificación de contratos introduce una carga adicional al proceso de desarrollo.

En un ejercicio de sistematización de las prácticas de diseño y desarrollo aplicadas a la docencia, investigación y desarrollo los autores del presente trabajo consideran fundamental la incorporación de los conceptos y la metodología de diseño por contratos. Esto se debe a que las aplicaciones, y las de los futuros graduados, podrían ser de carácter crítico. Entre ellas se pueden mencionar aplicaciones en dispositivos médicos que interactúan directamente con el organismo, sistemas complejos de procesamiento de datos bioinformáticos que contribuyen al diagnóstico, software de sistemas de transporte, o redes de Internet de las Cosas en hospitales o centros de salud, entre otras [11].

Dada la naturaleza crítica de estas aplicaciones, es importante contar con técnicas y métodos confiables para su desarrollo. Los contratos aportan en este sentido, dado que pueden contribuir significativamente a mejorar la vinculación con los requerimientos al definir claramente las expectativas y responsabilidades de cada componente del sistema, facilitando la detección y corrección temprana de errores. Además, si los contratos forman parte del código fuente y se expresan con una sintaxis diferenciada, se cuenta con los elementos necesarios para explotar la potencia de las pruebas de software, tanto empíricas como formales, [10].

El grupo de trabajo de los autores del presente artículo incorpora herramientas para aplicar la metodología de diseño por contratos en cada lenguaje de programación que utiliza. Esto permite aplicar herramientas de análisis automatizado y aproximaciones formales para la verificación de software, aportando así a la correctitud y robustez de sistemas críticos. Un trabajo desarrollado previamente por los autores del presente trabajo ha sido la implementación de contratos en Python con una biblioteca de terceros. El propósito de este trabajo es pedagógico, ya que exploramos cómo esta metodología puede transformar la experiencia didáctica del diseño y desarrollo de software biomédico crítico. Además, se ensaya cómo la programación basada en contratos puede mejorar el desarrollo de software biomédico al expresar los requerimientos como parte del código fuente y se proponen posibles estrategias para su incorporación en la enseñanza de programación en el área [11].

Centrando ahora la atención en C++, se destaca este lenguaje de programación porque es una elección muy frecuente para el desarrollo de diversas aplicaciones, debido a la eficiencia del código máquina que produce [3], lo cual es fundamental cuando se trabaja con recursos limitados o se requiere un rendimiento optimizado. Es un lenguaje de propósito general que enfatiza el diseño y uso de abstracciones ligeras y ricas en tipos [22]. Esto lo convierte en una plataforma ideal para aplicar metodologías formales que aporten a mejorar la confiabilidad del software, especialmente en aplicaciones críticas donde un fallo podría tener consecuencias graves [22,23]. Algunos de los sistemas más difundidos en la actualidad tienen sus componentes críticos escritos en C++. Entre ellos se encuentran la máquina virtual de Java, los intérpretes de JavaScript y Python, navegadores web y los sistemas operativos. C++ también se utiliza en aplicaciones cien-

tíficas y tecnológicas de carácter crítico, como la tomografía computada, los aceleradores lineales, los controladores de vuelo y controladores electrónicos en automóviles [23,2].

Con el objetivo de aportar a mejorar la confiabilidad del software crítico desarrollado en C++, este trabajo presenta el diseño, desarrollo y análisis de una biblioteca que implementa la programación por contratos en dicho lenguaje. Se exploran diversas formas de implementación mediante una combinación de clases, funciones y macros que permiten la declaración y verificación de precondiciones, postcondiciones e invariantes durante el desarrollo del software. Además, se realiza una comparativa entre estas implementaciones en términos de sintaxis y eficiencia para evaluar su rendimiento. Como resultado, la herramienta desarrollada es totalmente compatible con el estándar de C++ y se destaca por su sintaxis y semántica simples, lo que facilita su adopción y uso por parte de los desarrolladores.

2 Conceptos básicos de contratos

El diseño por contrato establece un marco en el que cada componente del sistema funciona bajo un contrato específico constituido por precondiciones, postcondiciones e invariantes. En esta metodología se debe definir claramente, de acuerdo a los datos disponibles, el comportamiento esperado de cada función, método o clase. En términos más concretos, el diseño por contrato, enfatiza el uso de los conceptos de precondiciones, postcondiciones e invariantes y los incorpora estratégicamente en el concepto de contrato [14,15,17,18]. La metodología propone hacer explícitas y sistemáticas las verificaciones del contrato. En este sentido, precondiciones y postcondiciones forman parte de la interfaz de las funciones dado que establecen propiedades lógicas que deben satisfacerse antes y después, respectivamente, de la ejecución de la tarea u operación que la función implementa, [17].

Las precondiciones son los criterios a cumplir al invocar una función, y las postcondiciones, al completarlas correctamente. En general, las funciones reciben argumentos con expectativas específicas. En parte, estas refieren a los tipos de datos de los argumentos y en C++ esto se valida durante la compilación, es decir, el compilador y el enlazador pueden asegurar que los argumentos sean del tipo correcto. Sin embargo, existen otras expectativas sobre los parámetros. Por ejemplo, el conjunto de posibles valores que los argumentos toman en el momento de la invocación de la función, y es responsabilidad del programador decidir qué hacer cuando las expectativas sobre los argumentos no se cumplen. Algo similar sucede con los valores de retorno. También se espera sobre ellos que, si la función se ejecuta correctamente, cumpla una expectativa lógica. Desde la perspectiva de C++, contar con información de precondiciones y postcondiciones es muy útil para el desarrollador, para los usuarios de la función y para los testers, [23,14,15].

El invariante es una evaluación lógica sobre el estado de una instancia de una clase que se debe satisfacer antes y después de la invocación de sus operaciones públicas. Es decir, en clases, existe un conjunto de expectativas respecto de lo

que siempre es verdadero para una instancia. En la programación en C++, en general, se asume que el invariante de clase es establecido por su constructor y es mantenido por todas sus funciones miembro con acceso a la representación interna del objeto hasta que este último es destruido, [23]. Sin embargo, si se decide definir el invariante, se sugiere hacerlo en un sitio concreto destinado a este fin cuando se define una clase, [17].

Otro aspecto central en los contratos tiene que ver con qué acciones se deben tomar en caso de incumplimiento del mismo. De acuerdo a B. Meyer [15], si una función o método falla en cumplir con su contrato este debe ser apropiadamente informado al usuario. Para lograr esto, el autor propone un conjunto de leyes que guían el diseño. La primera ley establece que una función puede terminar de dos maneras cumpliendo con su contrato o incumpliéndolo. En caso de incumplir con su contrato, la segunda ley establece que la función siempre debe causar una excepción. Finalmente, la tercera ley establece las estrategias de qué hacer durante el manejo de una excepción.

Desde el punto de vista de C++, la política sobre qué hacer si una función falla, se maneja a niveles de garantías de seguridad, [21,19,24]. Según Stourstrup [23], quien desarrolla una función tienen varias alternativas para tratar los argumentos de entrada, entre las que se incluyen: asegurarse de que cada entrada posee un valor válido, asumir que las precondiciones siempre se cumplen en la invocación, verificar las precondiciones y lanzar una excepción si no se cumplen, o verificar las precondiciones o finalizar el programa si alguna no se cumple. En el caso de postcondiciones en funciones con efectos secundarios recomienda considerarlos y documentar su efecto [23]. En [13], se plantean de forma general diversas acciones defensivas para manejar errores.

3 Propuestas de incorporación de contratos en C++

C++ es uno de los lenguajes más utilizados en sistemas críticos debido a su eficiencia y control sobre recursos de hardware. Sin embargo, carece de soporte nativo para DbC limitando su aplicación directa en la implementación de esta metodología. Los desarrolladores recurren a soluciones *ad hoc*, como documentación o bibliotecas para aplicar esta metodología, [20,8,9]. Particularmente, en [20] los autores tienen como objetivo establecer si el DbC puede ser emulado exitosamente en lenguajes que no proveen soporte nativo para contratos. Para ello desarrollan un framework al que le hacen diversas pruebas para dar respuesta a este planteo.

Por otra parte, se destaca el material producido por el subgrupo SG21 (The Contracts Study Group), que se encuentra avanzando en el estudio de un MVP (*minimum viable product*) para incorporar contratos de manera nativa en C++, [7]. En este grupo se pueden rastrear propuestas de incorporación de contratos al estándar de C++ que datan del año 2004.

En la práctica, se observa un nivel de uso de contratos en C++ muy bajo, y en aquellos que se utiliza, las precondiciones, postcondiciones e invariantes (de clase) suelen representarse como comentarios (por ejemplo "//pre: ... y //post: ...") al

iniciar la definición de una función, o indirectamente como código defensivo en el cuerpo de la función.

4 Métodos

El desarrollo y prueba de la biblioteca se llevó a cabo haciendo uso del framework de QT aprovechando la integración de herramientas y las opciones de configuración que provee para proyectos de este tipo. Se utilizó una versión de C++ del estándar 17, empleando para su compilación gcc en su versión 13. El modelado y documentación del diseño se realizó con UML. El control de cambios del código se gestionó con git y GitHub.

La forma de trabajo consistió en reuniones periódicas entre los integrantes del equipo de trabajo para discutir sobre aspectos relevantes de otras propuestas, y qué mejoras podría aportar la biblioteca desarrollada. Se consideró incorporar los conceptos centrales de DbC guiados por los recursos que provee el lenguaje, manteniendo una sintaxis y semántica simple.

5 Resultados y discusión

A continuación se describen los resultados obtenidos, presentando una discusión en relación con implementaciones propuestas por otros autores. El código fuente de la biblioteca, junto a un conjunto de pruebas unitarias, ejemplos de casos para discutir la metodología y su correcta aplicación, como el dilema del delegado dependiente y la bomba de infusión de insulina, además de un conjunto de comparativas de rendimiento (*benchmarks*) realizadas para evaluar la eficiencia [4]. Además, se puede encontrar la documentación de uso de la biblioteca desarrollada en formato web [1].

En la Figura 1 se muestra el diagrama de clases de la biblioteca para referencia y a continuación en las siguientes secciones se desarrolla cada elemento de la misma.

5.1 Implementación de precondiciones y postcondiciones

Para la incorporación de precondiciones y postcondiciones en la biblioteca se optó por desarrollar tres alternativas: clases, funciones y macros. El propósito de realizar tres desarrollos fue evaluar comparativamente dos aspectos de importancia: sintaxis y eficiencia. En las tres implementaciones, se manejan dos parámetros, una expresión lógica a evaluar y el mensaje que se incorpora a la excepción (ver en siguiente subsección) en caso que la evaluación del condicional de falso. En la Fig 1 sólo se representa la versión con clases.

Implementación con clases Tanto precondiciones y postcondiciones se implementaron como functors e instancias globales. Es decir, la biblioteca pone a disposición dos objetos función que se crean antes de que el programa principal se

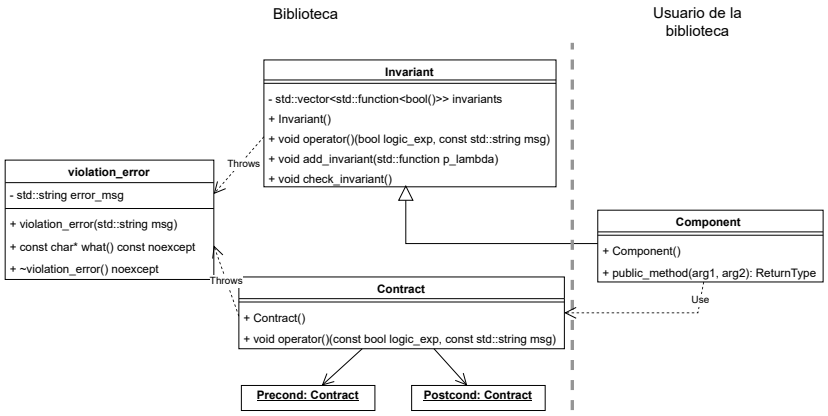

Fig. 1. Diagrama UML de la biblioteca

ejecute. Como se mencionó, estos functores reciben dos parámetros, una expresión lógica y un mensaje de error. Si la expresión lógica es verdadera, el functor no realiza ninguna acción y el código continúa ejecutándose. En el caso contrario, el resultado del functor es el lanzamiento de una excepción. A continuación se muestra el prototipo de invocación de una precondición y una postcondición.

```
precond(bool_exp, msg)
postcond(bool_exp, msg)
```

Implementación con funciones De manera similar se llevó a cabo la implementación con funciones. A continuación se muestra el prototipo de invocación de una precondición y una postcondición donde la diferencia en los nombres radica únicamente en la necesidad de diferenciarlas para las pruebas de la biblioteca.

```
precondition(bool_exp, msg)
postcondition(bool_exp, msg)
```

Implementación con macros En relación a las macros, se aconseja reemplazar su utilización por funciones para mantener la verificación de tipos que provee el estándar C++ así como el control de argumentos [25]. Sin embargo, consideramos su utilización dado que es una alternativa utilizada en bibliotecas de alto rendimiento, que también incorporan contrato de manera *ad hoc* [12].

Siguiendo la misma lógica que en las dos implementaciones anteriores se llevó a cabo la propuesta de tres macros. Dos macros son las que utiliza el usuario para expresar precondiciones y poscondiciones. La tercera macro `ASSERT` se emplea como subrutina de las dos primeras, y realiza la evaluación de la expresión y el lanzamiento de la excepción en caso que esta no se cumpla. A

continuación se muestra la manera de invocar las macros para una precondición y una postcondición, respectivamente.

```
#REQUIRE(bool_exp, msg)
#ENSURE(bool_exp, msg)
```

5.2 Incorporación de invariante de clase

Para representar el invariante de una clase específica, se implementó la clase `Invariant` (ver Fig 1). La clase cuenta con un atributo principal que consiste en un vector de funciones booleanas, que constituye el invariante en cada clase. Cada una de ellas es una expresión lógica que debe evaluarse como verdadera en los momentos de la vida de un objeto en el que se requiera su verificación que es, por lo general, en los métodos públicos de una clase.

Las clases que implementan invariantes deben heredar de `Invariant`. Se permite así que las clases derivadas utilicen directamente los métodos y estructuras para añadir y verificar su invariante. Esto contrasta con lo propuesto por [20] donde los autores plantean como desventaja de su framework que el invariante de una clase con miembros privados, no pueden ser chequeados de forma directa. Por lo tanto, una alternativa es hacer que la clase chequeadora del framework sea una clase amiga de la clase base. En esta propuesta, la jerarquía verifica los invariantes a nivel de la clase base, limitando la exposición de atributos privados.

Una dificultad potencial en el uso de esta clase es la necesidad de declarar funciones lambda al añadir un invariante. Para usuarios con un conocimiento actualizado de los estándares de C++, esto no representaría mayores inconvenientes [5]. Sin embargo, para proveer una sintaxis opcional, se definió una macro que facilite la creación de funciones booleanas para los invariantes. Esta macro toma una expresión booleana y la convierte en una función lambda que puede ser añadida directamente a los invariantes.

5.3 Manejo de fallas

Nuestra biblioteca incorpora una excepción (`violation_error`) que representa aquellas situaciones en las que un contrato no puede cumplirse. Esta fue desarrollada respetando la estructura jerárquica de excepciones en el estándar de C++ dotando así de una especificación clara para su interpretación y utilización.

5.4 Pruebas de rendimiento

Para analizar el costo computacional de incorporar contratos se desarrolló un conjunto de pruebas comparativas de tiempo de ejecución basado en el conteo de ciclos de máquina, empleando un método de integración numérica. El código de la prueba se encuentra en [4]. El método numérico se implementó mediante una clase que tiene como atributo los límites de integración y métodos para llevar a cabo la resolución propiamente dicha en base a la función que reciba.

Este método de integración se codificó ensayando la aplicación de las diferentes implementaciones de la biblioteca. Esto es, se ensayan las dos implementaciones de invariantes (con funciones lambda y con macros) y las tres implementaciones para precondiciones y postcondiciones (con funciones, clases y macros).

La figura 2 muestra los resultados de la medición de tiempos en ciclos de máquina del código que implementa el método de integración con las diferentes alternativas de aplicación de contratos. Las columnas verdes corresponden al código sin invariante. Y el grupo de tres columnas de la derecha corresponde al código que no implementa aserciones (precondiciones y postcondiciones) ni invariante por lo que constituyen la medición de tiempo control contra el que se realizaron las comparaciones. El análisis de esta gráfica devela que las macros ofrecen el mejor rendimiento en términos de ciclos, sin que se observe una diferencia significativa en comparación con la ausencia de chequeos. Por otro lado, la verificación de condiciones mediante clases y funciones requiere de dos a tres veces más ciclos que sin chequeos. Adicionalmente, las implementaciones con clases presentaron mejor rendimiento que la implementación con funciones.

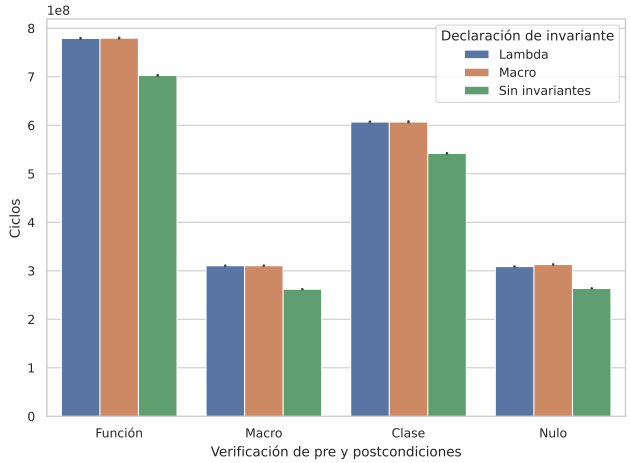

Fig. 2. Comparación de tiempos medidos con ciclos de máquina según implementación de aserciones (precondiciones y postcondiciones) e invariantes.

Para lograr una cobertura más amplia de casos de uso, se seleccionaron e implementaron aquellos relevantes encontrados en la bibliografía. Se optó por el "dilema del delegado dependiente" de Meyer [16] porque permite ensayar un hipotético caso de falla de invariante en una relación cliente-servidor.

Finalmente, pero no menos importante, se programa un conjunto de pruebas unitarias para verificar el comportamiento de la biblioteca. Se probaron las distintas implementaciones de precondiciones, postcondiciones e invariantes con

el objetivo de garantizar que se comportan según lo esperado. También, se comprobó la integridad de los mensajes de error y el tipo de excepción lanzada.

6 Conclusión

En este trabajo se propone una biblioteca para implementar la metodología de diseño por contratos en C++. La biblioteca permite definir y verificar precondiciones, postcondiciones e invariantes en tiempo de ejecución con una mínima huella (*footprint*) en el código fuente. La solución propuesta aporta a mejorar la confiabilidad y calidad de las aplicaciones de software crítico en C++ como lo hacen otros trabajos relacionados, pero mejorando la integración de la herramienta al evitar, por ejemplo, las clases y funciones amigas [9].

No obstante, algunos aspectos que podrían ser importantes para algunas aplicaciones particulares, como el desarrollo de drivers o bibliotecas de funciones, no se tuvieron en cuenta inicialmente en esta versión de la herramienta. Sin embargo, surgieron en la fase de prueba ya finalizada la primera iteración de diseño y desarrollo. Por ejemplo, mientras se probaba la biblioteca, en algunos casos surgía la necesidad de activar y desactivar las diferentes aserciones por lo que se consideró una característica importante a añadir, y que desarrollos de otros autores ya incorporan, [20]. Particularmente esta característica, aunque puede ser añadida fácilmente a la biblioteca, merece una evaluación más profunda, dado que no está lo suficientemente clara aun la forma de una solución general y apropiada para todas las situaciones.

Las pruebas y el análisis realizados sobre la biblioteca, validan su eficiencia y su integración sencilla en el flujo de desarrollo de software. Ofrece una sintaxis clara, funciones y métodos eficientes para asegurar el cumplimiento de contratos y gestionar los fallos mediante excepciones integradas a la jerarquía de excepciones del estándar C++. Esto resulta en una propuesta valiosa para equipos de desarrollo que buscan mejorar la confiabilidad de sus desarrollos incorporando diseño por contratos.

Reconocimientos. Este trabajo recibió apoyo financiero del Proyecto de Investigación Científica, Desarrollo e Innovación Tecnológica PID-UNER(6241), Argentina.

Declaración de Intereses. Los autores no tienen intereses contrapuestos que declarar que sean relevantes para el contenido de este artículo.

Referencias

1. DbC Library, `https://fiuner-lica.github.io/biblioteca-dbc-cpp/`
2. Sistema Integrado de Gestion de Proyectos, `https://proyectos.uner.edu.ar/aplicacion.php?ah=st66aa47087e0ee&ai=gestion_extinv`
3. TIOBE Index, `https://www.tiobe.com/tiobe-index/`
4. FIUNER-LICA/biblioteca-dbc-cpp (Jul 2024), `https://github.com/FIUNER-LICA/biblioteca-dbc-cpp`, original-date: 2024-06-11T12:36:23Z

5. Bengtsson, J., Hokka, H.: Analysing Lambda Usage in the C++ Open Source Community (2020)
6. Bourque, P., Fairley, R.E. (eds.): SWEBOK: guide to the software engineering body of knowledge. IEEE Computer Society, Los Alamitos, CA, 3.0 edn. (2014), oCLC: 880350861
7. Doumler, T., Ažman, G., Berne, J., Krzemieński, A., Voutilainen, V., Honermann, T.: Requirements for a Contracts syntax. Tech. Rep. P2885R1, SG21, EWG (Aug 2023)
8. Guerreiro, P.: Another mediocre assertion mechanism for C++. In: Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33 (Jun 2000). https://doi.org/10.1109/TOOLS.2000.848764
9. Guerreiro, P.: Simple support for design by contract in C++. In: Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39 (Jul 2001). https://doi.org/10.1109/TOOLS.2001.941656
10. Hakonen, H., Hyrynsalmi, S., Järvi, A.: Reducing the number of unit tests with design by contract. In: Proceedings of the 12th International CompSysTech. ACM, Vienna, Austria (2011). https://doi.org/10.1145/2023607.2023635
11. Insfrán, J.F., Diaz Zamboni, J.E.: Reflexiones sobre la enseñanza de confiabilidad y seguridad del software en bioingeniería con programación basada en contratos. Universidad Nacional del Comahue (2023)
12. King, D.E.: Dlib-ml: A Machine Learning Toolkit https://jmlr.csail.mit.edu/papers/volume10/king09a/king09a.pdf
13. McConnell, S.: Code complete. Microsoft Press, Redmond, Wash, 2nd ed edn. (2004)
14. Meyer, B.: Applying 'design by contract'. Computer **25**(10) (Oct 1992). https://doi.org/10.1109/2.161279
15. Meyer, B.: Design by Contract. Technical, Interactive Software Engineering Inc (1986)
16. Meyer, B.: The Dependent Delegate Dilemma. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems. Springer Netherlands, Dordrecht (2005). https://doi.org/10.1007/1-4020-3532-2_4
17. Meyer, B.: Object-oriented software construction. Prentice Hall PTR, Upper Saddle River, NJ, 2. ed., 16. print edn. (2009)
18. Meyer, B.: Touch of Class. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-92145-5
19. Monperrus, M.: Automatic Software Repair: A Bibliography. ACM Comput. Surv. **51**(1) (Jan 2018). https://doi.org/10.1145/3105906, number: 1
20. Nolle, L., Flechais, I.: On a C++ framework to support design by contract. In: 2016 7th IEEE ICSESS (2016). https://doi.org/10.1109/ICSESS.2016.7883011
21. de Pádua, G.B., Shang, W.: Studying the relationship between exception handling practices and post-release defects. In: Proceedings of the 15th International Conference on MSR. ACM (2018). https://doi.org/10.1145/3196398.3196435
22. Stroustrup, B.: Foundations of C++ (2012)
23. Stroustrup, B.: The C++ programming language. Addison-Wesley, Upper Saddle River, NJ, fourth edition edn. (2013)
24. Sutter, H.: When and How to Use Exceptions, http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836
25. TylerMSFT: Macros y C++ (Jun 2023), https://learn.microsoft.com/es-es/cpp/preprocessor/macros-and-cpp?view=msvc-170