Bruce Eckel TIPADO FUERTE VS. PRUEBAS FUERTES1 Recuerdo cuando trabajaba en VBA en Microsoft y teníamos debates extensos sobre la verificación de tipos estáticos versus dinámicos. "Verificación de tipos estáticos" es cuando el compilador, en tiempo de compilación, verifica que todas tus variables sean del tipo correcto. Por ejemplo, si tienes una función llamada log() que espera un número, y la llamas así: log("foo"), pasando una cadena, bueno, con la verificación de tipos estáticos, el compilador dirá: "¡Espera un minuto! No puedes pasar una cadena a esa función porque espera un número", y tu programa no se compilará. Esto es lo opuesto a la verificación de tipos dinámicos, en la que la comprobación se realiza en tiempo de ejecución. Con la verificación de tipos dinámicos, log("foo") se compilaría bien, pero en tiempo de ejecución generaría un error. La desventaja de esto es que es posible que no descubras el error hasta meses después, cuando alguien realmente ejecute esa línea de código, especialmente si está en una función raramente utilizada. En el diseño de VBA, donde el objetivo original era proporcionar un lenguaje de scripting para los usuarios de Excel, yo estaba fuertemente a favor del "tipado débil", porque es demostrablemente más fácil para programadores no profesionales, que ya tienen suficientes problemas entendiendo qué es una variable, y mucho menos qué es un tipo. En mi bando, tenía a la comunidad de Smalltalk, que, en esos días, argumentaba de manera bastante vaga que "todavía te darás cuenta del problema, simplemente te darás cuenta unos segundos después..." Lo cual a menudo es cierto, pero no siempre. Eventualmente, gané el debate interno en Microsoft, y se agregó el tipo de datos "Variant" -una estructura que puede contener valores de cualquier tipo- a VBA y COM. De hecho, más tarde apareció VBScript, que solo admitía variantes, así que debe haber sido una idea popular. Sin embargo, siempre supe en el fondo de mi mente que el tipado fuerte es una forma inteligente de hacer que el compilador verifique muchos tipos de errores, y de hecho, en C++, siempre usé el sistema de tipos extensamente para verificar errores de todo tipo. Por ejemplo, si quieres asegurarte absolutamente de que los empleados nunca, jamás, reciban un bono, puedes crear un sistema de tipos con gerentes y empleados, y solo los gerentes tienen el método PayBonus(). Ahora, ¡presto!, si tu programa se compila, puedes estar seguro de que solo los gerentes dignos y nobles obtienen bonificaciones, no los codiciosos empleados. El problema es que crear tipos únicamente con el propósito de hacer más pruebas en tiempo de compilación es un poco incómodo. Los tipos solo pueden hacer un tipo de prueba, es decir, "¿Puedo hacer esta acción con ese objeto?" No pueden probar "¿Esta función realmente devuelve 2.12 cuando los valores de entrada son 1, 32 y 'aardvark'?" En efecto, es un rompecabezas para el programador idear algún tipo de esquema de tipos ingenioso que se pueda usar para verificar algún pequeño aspecto de la corrección del programa. Resulta que, si deseas garantizar la corrección del programa, tenemos una herramienta más directa y poderosa: las pruebas unitarias. Así que me intrigó mucho la idea de Bruce Eckel de pruebas fuertes como sustituto del tipado fuerte. Ahora, antes de pasar la palabra a Bruce, debo advertirte que el tipado dinámico tiene una desventaja seria en rendimiento. Debido a que los tipos deben evaluarse y verificar en tiempo de ejecución, los lenguajes tipados dinámicamente siempre serán más lentos que los lenguajes tipados estáticamente. Esto puede estar bien o no, dependiendo de la aplicación. El tipado dinámico obligatorio de Python lo convierte en un lenguaje muy lento. Uso un filtro de spam escrito en Python que a menudo me hace esperar varios segundos para marcar un solo mensaje como spam, así que cuando necesito marcar 10 o 20 mensajes como spam, estoy pagando algo así como un minuto o dos por esta agradable "característica de tipado dinámico". Así que utiliza tu propio juicio sobre qué tipo de rendimiento necesita tu aplicación, pero si tus pruebas unitarias ofrecen una buena cobertura de código, no te sientas demasiado paranoico al renunciar a la verificación de tipos en tiempo de compilación. - Ed. En los últimos años, mi principal interés se ha centrado en la productividad del programador. Los ciclos de programador son costosos, los ciclos de CPU son baratos, y creo que ya no deberíamos pagar por lo primero con lo segundo. ¿Cómo podemos obtener un apalancamiento máximo en los problemas que intentamos resolver? Cada vez que aparece una nueva herramienta (especialmente un lenguaje de programación), esa herramienta proporciona algún tipo de abstracción que puede o no ocultar detalles innecesarios al programador. Sin embargo, siempre estoy atento a un pacto faustiano, especialmente uno que trate de convencerme de ignorar todos los obstáculos que debo superar para lograr esta abstracción. Perl es un excelente ejemplo de esto: la inmediatez del lenguaje oculta los detalles sin sentido de construir un programa, pero la sintaxis ilegible (basada, lo sé, en la compatibilidad hacia atrás con herramientas de Unix como awk, sed y grep) es un precio contraproducente a pagar. Los últimos años han aclarado este pacto faustiano en términos de lenguajes de programación más tradicionales y su orientación hacia la verificación de tipos estáticos. Esto comenzó con un romance de dos meses con Perl, que me proporcionó productividad a través de un tiempo de respuesta rápido. (La relación se rompió debido al tratamiento reprobable de Perl hacia las referencias y las clases; solo más tarde vi los verdaderos problemas con la sintaxis). Los problemas de la verificación de tipos estáticos versus dinámicos no eran visibles con Perl, ya que no puedes construir proyectos lo suficientemente grandes como para ver estos problemas, y la sintaxis oculta todo en programas más pequeños. Después de pasar a Python (gratis en [www.Python.org](http://www.python.org/)), un lenguaje que puede construir sistemas grandes y complejos, comencé a notar que a pesar de una aparente falta de cuidado con la verificación de tipos, los programas Python parecían funcionar bastante bien sin mucho esfuerzo y sin los problemas que esperarías de un lenguaje que no tiene la verificación de tipos estáticos que todos creemos que es la única forma correcta de resolver el problema de la programación. Esto fue un rompecabezas: si la verificación de tipos estáticos es tan importante, ¿por qué las personas pueden construir programas grandes y complejos en Python (con mucho menos tiempo y esfuerzo que sus contrapartes estáticas) sin el desastre que estaba seguro de que ocurriría? Esto sacudió mi aceptación sin cuestionamientos de la verificación de tipos estáticos (adquirida al pasar de C pre-ANSI a C++, donde la mejora fue dramática) lo suficiente como para que la próxima vez que examiné el problema de las excepciones verificadas en Java,2 pregunté "¿por qué?", lo que produjo una gran discusión3 en la que se me dijo que si seguía abogando por excepciones no verificadas, las ciudades caerían y la civilización tal como la conocemos dejaría de existir. En "Thinking in Java", 3ra Edición (Prentice Hall PTR, 2002), procedí a mostrar el uso de RuntimeException como una clase envoltorio para "desactivar" las excepciones verificadas. Cada vez que lo hago ahora, parece correcto (noto que Martin Fowler tuvo la misma idea más o menos al mismo tiempo), pero aún recibo ocasionalmente correos electrónicos que me advierten que estoy violando todo lo que es correcto y verdadero, y probablemente también la Ley Patriota de los Estados Unidos (¡hola, todos ustedes de la FBI! ¡Bienvenidos a mi blog!). Pero decidir que las excepciones verificadas parecen más problemas de los que valen (la verificación, no la excepción; creo que un mecanismo de informe de errores único y consistente es esencial) no respondió a la pregunta "¿Por qué Python funciona tan bien, cuando la sabiduría convencional dice que debería producir fallas masivas?" Los lenguajes tipados dinámicamente como Python son muy laxos con la verificación de tipos. En lugar de poner las restricciones más fuertes posibles en el tipo de objetos, lo más temprano posible (como hace Java), lenguajes como Ruby, Smalltalk y Python ponen las restricciones más laxas posibles en los tipos y evalúan los tipos solo si es necesario. Esto produce la idea de un tipado latente o tipado estructural, a menudo llamado informalmente "tipado pato" (como en "Si camina como un pato y habla como un pato, podemos tratarlo como un pato"). Esto significa que puedes enviar cualquier mensaje a cualquier objeto, y el lenguaje solo se preocupa de que el objeto pueda aceptar el mensaje. No requiere que el objeto sea un tipo particular, como hace Java. Por ejemplo, si tienes mascotas que pueden hablar en Java, el código se vería así: // Mascotas parlantes en Java: interface Pet { void speak(); } class Cat implements Pet { public void speak() { System.out.println("¡miau!"); } } class Dog implements Pet { public void speak() { System.out.println("¡guau!"); } } public class PetSpeak { static void command(Pet p) { p.speak(); } public static void main(String[] args) { Pet[] pets = { new Cat(), new Dog() }; for(int i = 0; i < pets.length; i++) command(pets[i]); } } Observa que command() debe conocer el tipo exacto del argumento que va a aceptar, un Pet, y no aceptará nada más. Por lo tanto, debo crear una jerarquía de Pet e heredar Dog y Cat para que pueda hacer un upcast a el método genérico command(). Durante mucho tiempo, asumí que el upcasting era una parte inherente de la programación orientada a objetos, y encontraba molestas las preguntas al respecto de ignorantes Smalltalkers y similares. Pero cuando empecé a trabajar con Python, descubrí la siguiente curiosidad. El código anterior se puede traducir directamente a Python:

**Mascotas parlantes en Python:**

class Pet: def speak(self): pass class Cat(Pet): def speak(self): print "¡miau!" class Dog(Pet): def speak(self): print "¡guau!" def command(pet): pet.speak() pets = [ Cat(), Dog() ] for pet in pets: command(pet) Si nunca has visto Python antes, notarás que redefine el significado de un lenguaje lacónico, pero de una manera muy buena. ¿Piensas que C/C++ es lacónico? Tiremos esos corchetes -la sangría ya tiene significado para la mente humana, así que la usaremos para indicar el alcance en su lugar. ¿Tipos de argumentos y tipos de retorno? ¡Deja que el lenguaje lo resuelva! Durante la creación de clases, las clases base se indican entre paréntesis. "def" significa que estamos creando una definición de función o método. Por otro lado, Python es explícito acerca del argumento "this" (llamado "self" por convención) para las definiciones de métodos. La palabra clave "pass" significa "Lo definiré más tarde", así que es una variación de una palabra clave abstracta. Observa que command(pet) solo dice que toma algún objeto llamado pet, pero no da información sobre qué debe ser el tipo de ese objeto. Eso es porque no le importa, siempre y cuando puedas llamar a speak(), o cualquier otra cosa que tu función o método desee hacer. Esto es tipado latente o tipado pato, que veremos más de cerca en un minuto. Además, command(pet) es simplemente una función ordinaria, lo cual está bien en Python. Es decir, Python no insiste en que todo sea un objeto, ya que a veces una función es lo que necesitas. En Python, las listas y los diccionarios (también conocidos como mapas o matrices asociativas) son tan importantes que se incorporan al núcleo del lenguaje, por lo que no necesito importar ninguna biblioteca especial para usarlos. Puedes ver esto aquí: pets = [ Cat(), Dog() ] Se crea una lista que contiene dos nuevos objetos de tipo Cat y Dog. Se llaman a los constructores, pero no es necesario un "new" (y ahora volverás a Java y te darás cuenta de que tampoco es necesario allí, es solo una redundancia heredada de C++). Iterar a través de una secuencia también es lo suficientemente importante como para ser una operación nativa en Python: for pet in pets: selecciona cada elemento en la lista en la variable pet. Mucho más claro y directo que el enfoque de Java, creo, incluso en comparación con la sintaxis "foreach" de J2SE5. La salida es la misma que la versión de Java, y puedes ver por qué Python a menudo se llama "pseudocódigo ejecutable". No solo es lo suficientemente simple como para usarlo como pseudocódigo, sino que tiene la maravillosa característica de que realmente se puede ejecutar. Esto significa que puedes probar rápidamente ideas en Python y, cuando encuentras una que funciona, puedes reescribirla en Java/C++/C# o en el lenguaje que elijas. O tal vez te des cuenta de que el problema está resuelto en Python, así que ¿por qué molestarse en reescribirlo? (Eso suele ser tan lejos como llego). He adoptado la costumbre de dar pistas de ejercicios en Python durante los seminarios, porque así no estoy revelando todo el panorama, pero la gente puede ver la forma que estoy buscando en una solución, así que pueden avanzar. Y puedo verificar que el pseudocódigo es correcto ejecutándolo. Pero la parte interesante es esta: dado que el método command(pet) no le importa el tipo que recibe, no tengo que hacer un upcast. Así que puedo reescribir el programa de Python sin usar clases base:

**Mascotas parlantes en Python, pero sin clases base:**

class Cat: def speak(self): print "¡miau!" class Dog: def speak(self): print "¡guau!" class Bob: def bow(self): print "gracias, gracias!" def speak(self): print "hola, bienvenido al vecindario!" def drive(self): print "beep, beep!" def command(pet): pet.speak() pets = [ Cat(), Dog(), Bob() ] for pet in pets: command(pet) Dado que command(pet) solo se preocupa de que pueda enviar el mensaje speak() a su argumento, he eliminado la clase base Pet, e incluso he agregado una clase totalmente no relacionada llamada Bob, que tiene un método speak(), por lo que también funciona en la función command(pet). En este punto, un lenguaje tipado estáticamente estaría sputtering con rabia, insistiendo en que este tipo de descuido causará desastres y caos. Claramente, en algún momento se usará el tipo "incorrecto" con command() o de alguna manera se colará en el sistema. El beneficio de expresar conceptos de manera más simple y clara simplemente no vale el peligro, incluso si ese beneficio es un aumento de productividad de 5 a 10 veces en comparación con Java o C++. ¿Qué sucede cuando ocurre ese problema en un programa de Python, es decir, cuando un objeto llega de alguna manera a donde no debería estar? Python informa todos los errores como excepciones, al igual que lo hacen Java y C#, y como debería hacer C++. Así que te enteras de que hay un problema, pero virtualmente siempre es en tiempo de ejecución. "Aha!" dices, "Ahí está tu problema: no puedes garantizar la corrección de tu programa porque no tienes la verificación de tipos en tiempo de compilación necesaria." Cuando escribí "Thinking in C++", 1ra Edición (Prentice Hall PTR, 1998), incorporé una forma muy primitiva de prueba: escribí un programa que extraería automáticamente todo el código del libro (usando marcadores de comentarios colocados en el código para encontrar el principio y el final de cada listado), y luego construiría makefiles que compilarían todo el código. De esta manera, podría garantizar que todo el código en mis libros se compilaba y, por lo tanto, razoné, podría decir "Si está en el libro, es correcto". Ignoré la voz molesta que decía: "Compilar no significa que se ejecute correctamente", porque era un gran paso automatizar la verificación de código en primer lugar (como cualquiera que mire libros de programación sabe, muchos autores todavía no ponen mucho esfuerzo en verificar la corrección del código). Pero naturalmente, algunos de los ejemplos no se ejecutaron correctamente, y cuando se informaron suficientes de estos a lo largo de los años, comencé a darme cuenta de que ya no podía ignorar el problema de las pruebas. Llegué a sentirme tan fuertemente al respecto que en la tercera edición de "Thinking in Java", escribí: Si no está probado, está roto. Es decir, si un programa se compila en un lenguaje tipado estáticamente, simplemente significa que ha pasado algunas pruebas. Significa que se garantiza que la sintaxis sea correcta (Python también verifica la sintaxis en tiempo de compilación, simplemente no tiene tantas restricciones de sintaxis). Pero no hay garantía de corrección solo porque el compilador apruebe tu código. Si tu código parece funcionar, eso tampoco garantiza la corrección. La única garantía de corrección, independientemente de si tu lenguaje es tipado estática o dinámicamente, es si pasa todas las pruebas que definen la corrección de tu programa. Y debes escribir algunas de esas pruebas tú mismo. Estas, por supuesto, son pruebas unitarias, pruebas de aceptación, etc. En "Thinking in Java", 3ra Edición, llené el libro con una especie de prueba unitaria, y estas pruebas se pagaron una y otra vez. Una vez que te vuelves "infectado por las pruebas", no puedes volver atrás. Es muy parecido a pasar de C pre-ANSI a C++. De repente, el compilador estaba realizando muchas más pruebas por ti y tu código estaba siendo corregido más rápido. Pero esas pruebas de sintaxis solo pueden llegar hasta cierto punto. El compilador no puede saber cómo esperas que se comporte el programa, así que debes "ampliar" el compilador agregando pruebas unitarias (independientemente del lenguaje que estés usando). Si lo haces, puedes realizar cambios drásticos (refactorizar código o modificar el diseño) de manera rápida porque sabes que tu conjunto de pruebas te respaldará, y fallará de inmediato si hay algún problema, al igual que una compilación falla cuando hay un problema de sintaxis. Pero sin un conjunto completo de pruebas unitarias (como mínimo), no puedes garantizar la corrección de un programa. Afirmar que las restricciones de verificación de tipos estáticos en C++, Java o C# te impedirán escribir programas defectuosos es claramente una ilusión (lo sabes por experiencia personal). De hecho, lo que necesitamos es: Pruebas sólidas, no tipos sólidos. Si observas el código Python en el ejemplo anterior, puedes ver que cada vez que la función command() realiza una tarea que le pide a su argumento (pet), se está moviendo hacia adelante, y no le importa cómo llegó allí, o qué tipo es. (En las listas de lectura de código, a veces es obvio cuál es el tipo correcto, pero en la escritura no siempre es tan claro, por lo que tiendo a ser implícitamente tipado). Esto significa que puedo agregar nuevas clases y funciones a medida que avanza mi proyecto, y no tengo que volver y cambiar las funciones antiguas (excepto para las ocasiones en que puedo hacerlo para hacerlas más generales). En otros lenguajes, especialmente en lenguajes tipados estáticamente, cada vez que agrego una nueva clase o función a mi proyecto, tengo que volver y cambiar todas las funciones antiguas que podrían ser capaces de tratar con esa nueva clase o función. Entonces, aunque hay un precio en el rendimiento por la pérdida de la verificación de tipos en tiempo de compilación, la recompensa es que el código es más sencillo y directo, y puedo cambiar y extender más fácilmente. La clave para entender cómo esto puede funcionar es comprender que las pruebas sustituyen completamente la verificación de tipos en tiempo de compilación. La verificación de tipos solo puede decir: "Sí, puedo llamar a ese método", mientras que las pruebas pueden decir: "No solo puedo llamar a ese método, sino que sé que siempre devolverá un valor entre 1 y 10". Entonces, ¿por qué las pruebas son más poderosas que la verificación de tipos? Porque las pruebas prueban la semántica, mientras que la verificación de tipos solo prueba la sintaxis. Considéralo de esta manera: puedes escribir una función que toma dos enteros y devuelve su suma. La verificación de tipos en tiempo de compilación puede garantizar que siempre recibirás dos enteros y siempre devolverás un entero. Pero no puede garantizar que la suma de esos dos enteros esté en el rango correcto, o que siempre hagas la suma correctamente. Solo las pruebas pueden hacer eso. Si bien no puedes garantizar la corrección de un programa solo con verificación de tipos, puedes garantizar la corrección de un programa solo con pruebas. Esas pruebas pueden incluir pruebas unitarias, pruebas de aceptación, pruebas de rendimiento y más, pero al menos debes tener pruebas unitarias. Dado que las pruebas sustituyen completamente a la verificación de tipos en tiempo de compilación, cualquier beneficio que obtengas de la verificación de tipos estática es automáticamente un beneficio que obtendrás de las pruebas unitarias. Y, como he argumentado, obtienes más beneficios de las pruebas que de la verificación de tipos. Si bien las pruebas unitarias son absolutamente esenciales para garantizar la corrección de un programa, hay algunas desventajas. Uno es que, aunque pruebas la semántica del programa, no puedes probar la semántica de todas las situaciones posibles. Es decir, tu conjunto de pruebas siempre es finito y, por lo tanto, siempre hay situaciones posibles que no has probado. La verificación de tipos en tiempo de compilación también tiene la propiedad de que siempre está presente y siempre se aplica, lo que significa que siempre se aplicará en todas las situaciones posibles. Así que la verificación de tipos tiene una ventaja en términos de cobertura completa. Sin embargo, la ventaja de la cobertura completa es contrarrestada por dos desventajas en la verificación de tipos. La primera es que, debido a que la verificación de tipos se realiza en tiempo de compilación, no puede tener acceso a toda la información disponible en tiempo de ejecución. Un ejemplo simple es que la verificación de tipos no puede garantizar que una lista esté vacía antes de llamar a list.get(0) (esto sería simplemente "un índice fuera de rango"), pero un conjunto de pruebas puede garantizarlo fácilmente. En el fondo, la verificación de tipos en tiempo de compilación se basa en la suposición de que si un programa compila, entonces es correcto. Pero la corrección de un programa no se trata simplemente de tener la sintaxis correcta; se trata de cumplir con las expectativas del usuario. Y, a menudo, estas expectativas se refieren al estado del programa en tiempo de ejecución. La verificación de tipos estática simplemente no puede garantizar nada sobre el estado del programa en tiempo de ejecución. Además, el precio que pagas por la cobertura completa en la verificación de tipos estáticos es que debes hacer concesiones en la simplicidad y flexibilidad del código, ya que debes satisfacer las restricciones del compilador. Estas concesiones pueden hacer que el código sea más complicado de lo necesario, y la complicación es a menudo la causa de los errores. La flexibilidad también es importante porque los programas siempre cambian; si tu código es demasiado rígido, será difícil de cambiar y mantener con el tiempo. Otra desventaja es que, debido a que la verificación de tipos estática se realiza en tiempo de compilación, todo el código que usa un tipo debe tener acceso a la definición del tipo en tiempo de compilación. En la mayoría de los lenguajes, esto significa que todos los tipos que usa un programa deben estar definidos en el mismo archivo, o deben ser explícitamente incluidos en cada archivo que use ese tipo. Esto limita la capacidad de dividir un programa en módulos independientes que se pueden compilar y probar por separado. El resultado es que, a menudo, los programas de lenguajes tipados estáticamente son más difíciles de dividir y probar, y la división y prueba son dos de las técnicas más efectivas para mejorar la calidad del código. Entonces, dado que las pruebas prueban la semántica del programa y proporcionan una cobertura completa para las situaciones posibles (dentro de las limitaciones de la finitud), son más poderosas que la verificación de tipos en tiempo de compilación. Dado que las pruebas tienen acceso a toda la información en tiempo de ejecución, pueden garantizar más aspectos del estado del programa que la verificación de tipos estáticos. Dado que las pruebas no imponen restricciones en la simplicidad y flexibilidad del código, permiten un código más simple y flexible. Dado que las pruebas no requieren que todos los tipos estén definidos en tiempo de compilación, permiten una división más efectiva y la prueba de programas más grandes y complejos. Así que Bruce argumenta que, aunque la verificación de tipos estáticos puede tener sus ventajas, las pruebas unitarias son más poderosas y proporcionan más beneficios en términos de simplicidad, flexibilidad, cobertura y capacidad para mejorar la calidad del código.