Recuerdo cuando trabajaba en VBA en Microsoft, teníamos debates extensos sobre la comprobación de tipos estáticos frente a la dinámica.

La "comprobación 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 comprobación de tipos estáticos, el compilador dirá: "¡Espera un momento! No puedes pasar una cadena a esa función porque espera un número", y tu programa no se compilará.

Esto es lo opuesto a la comprobación de tipos dinámicos en la que la verificación se realiza en tiempo de ejecución. Con la comprobación 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 te des cuenta del error hasta meses después cuando alguien realmente ejecute esa línea de código, especialmente si está en una función poco utilizada.

En el diseño de VBA, donde el objetivo original era proporcionar un lenguaje de secuencias de comandos para los usuarios de Excel, yo estaba fuertemente a favor del "tipado débil", porque es demostrablemente más fácil para los programadores no profesionales, quienes ya tienen suficientes dificultades para entender qué es una variable, y mucho menos qué es un tipo.

En mi bando, tenía a la comunidad de Smalltalk, que en aquellos días, presentaba el argumento bastante vago de que "aún descubrirás el problema en algún momento".

Sobre el problema, te das cuenta de él solo unos segundos más tarde..." 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, y de hecho más tarde llegó VBScript, que solo admitía variantes, así que debe haber sido una idea popular. Sin embargo, siempre supe en mi mente que la tipificación fuerte es una manera 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, nunca, nunca reciban un bono, puedes crear un sistema de tipos con gerentes y empleados y solo los gerentes tienen el método PayBonus(). Ahora, ¡voilà!, si tu programa se compila, puedes estar seguro de que solo los merecedores y nobles gerentes reciben bonos, no los codiciosos empleados. El problema es que crear tipos únicamente con el propósito de realizar más pruebas en tiempo de compilación es un poco incómodo. Los tipos solo pueden hacer un tipo de prueba, es decir, "¿Puedo hacer esta cosa 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 utilizar para verificar algún pequeño aspecto de la corrección del programa. Resulta que, si quieres asegurar 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 de la tipificación fuerte. Ahora, antes de pasarte a Bruce, debo advertirte que la tipificación dinámica tiene un inconveniente grave en cuanto al rendimiento. Debido a que los tipos deben evaluarse y verificarse 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. La tipificación dinámica obligatoria de Python lo convierte en un lenguaje muy lento. Utilizo 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 para esta agradable "tipificación dinámica". Si estás ejecutando una granja de servidores web, el uso de un lenguaje tipado dinámicamente puede significar que necesitas cinco o diez veces más servidores para atender al mismo número de clientes, lo que puede ser muy costoso. Así que usa tu propio criterio sobre qué tipo de rendimiento requiere tu aplicación, pero si tus pruebas unitarias proporcionan una buena cobertura de código, no te sientas demasiado paranoico por renunciar a la verificación de tipos en tiempo de compilación. - Ed.

En los últimos años, mi interés principal se ha centrado en la productividad del programador. Los ciclos de programador son caros, los ciclos de CPU son baratos, y creo que ya no deberíamos pagar por lo primero con lo segundo. ¿Cómo podemos obtener el máximo rendimiento 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 ocultar detalles innecesarios del programador. Sin embargo, siempre estoy atento a un pacto faustiano, especialmente uno que intente 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 irrelevantes de la construcción de un programa, pero la sintaxis ilegible (basada, lo sé, en la compatibilidad con herramientas Unix como awk, sed y grep) es un precio contraproducente que pagar. Los últimos años han aclarado este pacto faustiano en términos de lenguajes de programación más tradicionales y su orientación hacia la verificación estática de tipos. Esto comenzó con un romance de dos meses con Perl, que me dio productividad a través de un rápido desarrollo. (La relación se terminó debido al trato reprobable de Perl hacia las referencias y las clases; solo más tarde vi los verdaderos problemas con la sintaxis). Los problemas de tipificación estática frente a dinámica 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 mudarme a Python (gratis en www.Python.org) - un lenguaje que puede construir sistemas grandes y complejos - comencé a notar que a pesar de una aparente despreocupación por la verificación de tipos, los programas de Python parecían funcionar bastante bien sin mucho esfuerzo, y sin los tipos de problemas que esperarías de un lenguaje que no tiene la verificación estática de tipos que todos hemos llegado a "saber" que es la única forma correcta de resolver el problema de programación.

Esto era un rompecabezas: Si la verificación estática de tipos es tan importante, ¿por qué la gente puede construir programas grandes y complejos en Python (con mucho menos tiempo y esfuerzo que los equivalentes estáticos) sin el desastre que estaba tan seguro de que ocurriría?

Esto sacudió mi aceptación incuestionable de la verificación estática de tipos (adquirida al pasar de C previo a ANSI a C++, donde la mejora fue dramática) lo suficiente como para que la próxima vez que examiné el tema de las excepciones comprobadas en Java, pregunté "¿por qué?" lo que produjo una gran discusión en la que se me dijo que si seguía defendiendo las excepciones no comprobadas, las ciudades caerían y la civilización tal como la conocemos cesaría de existir. En "Thinking in Java", 3rd Edition (Prentice Hall PTR, 2002), seguí adelante y mostré el uso de RuntimeException como una clase de envoltura para "desactivar" las excepciones comprobadas. Cada vez que lo hago ahora, parece correcto (noto que Martin Fowler tuvo la misma idea aproximadamente al mismo tiempo), pero todavía recibo el ocasional correo electrónico que me advierte que estoy violando todo lo que es correcto y verdadero, y probablemente también el USA Patriot Act (¡hola, todos ustedes, del FBI! ¡Bienvenidos a mi weblog!).

Pero decidir que las excepciones comprobadas parecen ser más problemas de los que valen la pena (la comprobación, 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 fallos masivos?" Python y lenguajes similares tipados dinámicamente son muy laxos en cuanto a la comprobación de tipos. En lugar de imponer las restricciones más fuertes posibles sobre el tipo de objetos, lo antes posible (como hace Java), lenguajes como Ruby, Smalltalk y Python imponen las restricciones más laxas posibles sobre los tipos y solo evalúan los tipos si es necesario.

Esto produce la idea de tipificación latente o tipificación estructural, a menudo llamada informalmente "duck typing" (como en "Si camina como un pato, y suena como un pato, simplemente 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 de un tipo particular, como lo hace Java. Por ejemplo, si tienes mascotas que pueden hablar en Java, el código se vería así:

```java
// 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]);
    }
}
```

Nota que el método `command()` debe conocer el tipo exacto del argumento que va a aceptar, que es un `Pet`, y no aceptará nada más. Por lo tanto, debo crear una jerarquía de `Pet`, y heredar `Dog` y `Cat` para que pueda realizar un upcasting de ellos al método genérico `command()`.

Durante mucho tiempo, asumí que el upcasting era una parte inherente de la programación orientada a objetos, y encontraba irritantes las preguntas al respecto de ignorantes usuarios de Smalltalk y similares. Pero cuando comencé...

Trabajando con Python descubrí la siguiente curiosidad. El código anterior puede ser traducido directamente a Python:

```python
# Mascotas que hablan en Python:
class Mascota:
    def hablar(self):
        pass

class Gato(Mascota):
    def hablar(self):
        print("¡miau!")

class Perro(Mascota):
```

```
    def hablar(self):
        print("¡guau!")

def comando(mascota):
    mascota.hablar()

mascotas = [Gato(), Perro()]

for mascota in mascotas:
    comando(mascota)
```

Si nunca has visto Python antes, notarás que redefine el significado de un lenguaje conciso, pero de una manera muy buena. ¿Crees que C/C++ es conciso? ¡Vamos a tirar esas llaves—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 método.

La palabra clave `pass` dice "lo definiré más tarde", así que es una variación de una palabra clave abstracta.

Ten en cuenta que `comando(mascota)` simplemente dice que toma algún objeto llamado `mascota`, pero no da ninguna información sobre qué tipo de objeto es.

Debe ser así. Eso es porque no le importa, siempre y cuando puedas llamar a `hablar()`, o lo que sea que tu función o método quiera hacer. Esto es tipificación latente/pato, lo cual veremos más de cerca en un minuto.

Además, `comando(mascota)` es solo una función ordinaria, lo cual está bien en Python. Es decir, Python no insiste en que hagas todo un objeto, ya que a veces una función es lo que deseas.

En Python, las listas y los diccionarios (también conocidos como mapas o arreglos asociativos) son tan importantes que están integrados en el núcleo del lenguaje, así que no necesito importar ninguna biblioteca especial para usarlos. Puedes verlo aquí:

```python
```

mascotas = [Gato(), Perro()]

```

Se crea una lista que contiene dos nuevos objetos de tipo Gato y Perro. Los constructores son llamados, pero no se necesita "new" (y ahora volverás a Java y te darás cuenta de que tampoco se necesita "new" 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:

```python
for mascota in mascotas:
```

selecciona cada elemento de la lista en la variable `mascota`. 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 en Java, y puedes ver por qué a menudo se llama a Python "pseudocódigo ejecutable". No solo es lo suficientemente simple como para usarlo como pseudocódigo, tiene la maravillosa característica de que realmente se puede ejecutar. Esto significa que puedes probar rápidamente ideas en Python, y cuando encuentres una que funcione, puedes reescribirla en Java/C++/C# o en tu lenguaje de elección. O tal vez te des cuenta de que el problema está resuelto en Python, entonces ¿por qué molestarse en reescribirlo? (Eso suele ser tan lejos como llego). He empezado a dar pistas de ejercicios en Python durante seminarios, porque así no estoy revelando todo el panorama, pero la gente puede ver la forma que estoy buscando en una solución, para que puedan avanzar. Y puedo verificar que el pseudocódigo es correcto al ejecutarlo.

Pero la parte interesante es esta: debido a que el método `comando(mascota)` no se preocupa por el tipo que recibe, no tengo que realizar un upcast. Por lo tanto, puedo reescribir el programa de Python sin usar clases base:

```python
# Mascotas que hablan en Python, pero sin clases base:
class Gato:
    def hablar(self):
        print("¡miau!")
```
```

```
class Perro:
    def hablar(self):
        print("¡guau!")

class Bob:
    def hablar(self):
        print("¡hola, bienvenido al vecindario!")

def comando(mascota):
    mascota.hablar()

mascotas = [Gato(), Perro(), Bob()]

for mascota in mascotas:
    comando(mascota)
```

Dado que `comando(mascota)` solo se preocupa de poder enviar el mensaje `hablar()` a su argumento, he eliminado la clase base `Mascota`, e incluso he agregado una clase totalmente no relacionada con mascotas llamada Bob, que casualmente tiene un método `hablar()`, por lo que también funciona en la función `comando(mascota)`.

En este punto, un lenguaje de tipado estático estaría sputtering con rabia, insistiendo en que este tipo de descuido causará desastre y caos. Claramente, en algún momento se usará el "tipo incorrecto" con `comando()` o de alguna manera se deslizará a través del sistema. El beneficio de una sintaxis más simple y clara la expresión de conceptos simplemente no vale la pena el peligro, incluso si ese beneficio es un aumento de productividad de 5 a 10 veces mayor que el de Java o C++. ¿Qué sucede cuando ocurre un problema así en un programa de Python, cuando un objeto de alguna manera llega a donde no debería estar? Python informa todos los errores como excepciones, como lo hacen Java y C#, y como debería hacerlo C++. Entonces, descubres que hay un problema, pero casi siempre es en tiempo de ejecución.

"¡Ajá!" 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 "Pensando en C++, 1ª Edición" (Prentice Hall PTR, 1998), incorporé una forma muy rudimentaria de pruebas: 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 archivos de compilación que compilarían todo el código. De esta manera, podía garantizar que todo el código en mis libros se compilaba y, por lo tanto, razonaba, podía decir: "Si está en el libro, es correcto". Ignoré la voz que decía "Compilar no significa que se ejecute correctamente", porque fue un gran paso automatizar la verificación del código en primer lugar (como cualquiera que mire libros de programación sabe, muchos autores aún no dedican mucho esfuerzo a verificar la corrección del código). Pero naturalmente, algunos de los ejemplos no se ejecutaron correctamente, y cuando se reportaron 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 "Pensando en Java", escribí:

"Si no está probado, está roto."

Es decir, si un programa se compila en un lenguaje de tipado estático, solo significa que ha pasado algunas pruebas. Significa que la sintaxis está garantizada para ser 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 pase tu código. Si tu código parece ejecutarse, eso tampoco garantiza la corrección.

La única garantía de corrección, independientemente de si tu lenguaje es de tipado estático o dinámico, 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,

En la tercera edición de Thinking in Java, llené el libro con una especie de prueba unitaria, y estas pruebas dieron resultados 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 se estaba volviendo correcto 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 haces esto, puedes realizar cambios significativos (refactorizar el 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, igual que una compilación falla cuando hay un problema de sintaxis.

Pero sin un conjunto completo de pruebas unitarias (al menos), no puedes garantizar la corrección de un programa. Afirmar que las restricciones de comprobación de tipos estáticos en C++, Java o C#

te impedirán escribir programas defectuosos es claramente una ilusión (sabes esto por experiencia personal).

De hecho, lo que necesitamos es:

Pruebas sólidas, no tipificación fuerte.

Así que esto, afirmo, es un aspecto de por qué funciona Python. Las pruebas en C++ ocurren en tiempo de compilación (con algunos casos especiales menores). Algunas pruebas en Java ocurren en tiempo de compilación (verificación de sintaxis) y algunas ocurren en tiempo de ejecución (verificación de límites de matriz, por ejemplo). La mayoría de las pruebas en Python ocurren en tiempo de ejecución en lugar de en tiempo de compilación, pero ocurren, y eso es lo importante (no cuándo). Y debido a que puedo poner en marcha un programa de Python en mucho menos tiempo del que te llevaría escribir el equivalente en C++/Java/C#, puedo comenzar a ejecutar las pruebas reales más pronto: pruebas unitarias, pruebas de mi hipótesis, pruebas de enfoques alternativos, etc. Y si un programa de Python tiene pruebas unitarias adecuadas, puede ser tan robusto como un programa de C++, Java o C# con pruebas unitarias adecuadas (aunque las pruebas en Python serán más rápidas de escribir).

Robert Martin es uno de los habitantes de la comunidad de C++ desde hace mucho tiempo. Ha escrito libros y artículos, consultado, enseñado, etc. Un tipo de comprobación de tipos estáticos bastante duro. O eso pensaba, hasta que leí una entrada de weblog que hizo (en http://www.artima.com/weblogs/viewpost.jsp?thread=4639). Robert llegó más o menos a la misma conclusión que yo, pero lo hizo convirtiéndose primero en "infectado por las pruebas", luego comprendiendo que el compilador era solo una forma (incompleta) de hacer pruebas, luego comprendiendo que un lenguaje de tipos dinámicos podría ser mucho más productivo pero crear programas tan robustos como los escritos en lenguajes de tipado estático, proporcionando pruebas adecuadas.

Por supuesto, Martin también recibió los comentarios habituales de "¿Cómo puedes pensar esto?" Que es la misma pregunta que me llevó a comenzar a luchar con los conceptos de tipificación estática/dinámica en primer lugar. Y ambos comenzamos como defensores de la comprobación de tipos estáticos. Es interesante que se necesite una experiencia trascendental, como volverse "infectado por las pruebas" o aprender un tipo de lenguaje diferente, para causar una reevaluación de las creencias.