1. Bevezető

1994-ben Erwin Unruh bemutatta a C++ szabvány bizottságnak bemutatta az első sablon-metaprogramot, amely, igaz nem fordult le, de viszont a hibaüzenetek prímszámok voltak. Felismervén a C++-ba beágyazott sablonmechanizmus Turing-teljességét, mint (fordítás idejű) nyelv, megfogant a sablon-metaprogramozás (template metaprogramming) fogalma és a generikus programozás paradigma.

A Graphical Template Library (GTL) [2], egy olyan kísérleti generikus C++ sablon-könyvtár, ami a C++ Standard Template Library (STL) alapötletét próbálja átvinni a grafikus feladatok körébe. A GTL-t 1999-ben készítette el Kisteleki Róbert (ELTE programtervező matematikus hallgató) és Dr. Porkoláb Zoltán.

Eredetileg, a feladat, egy rajzoló program megvalósítása lett volna, amely a GTL-re támaszkodik és a GTL által nyújtott alakzatokat tudja kirajzolni és ezeken a GTL algoritmusait, műveleteit tudja végrehajtani. A program fejlesztése során, kizárólag a generikus programozás eszközeire kellet használni, az öröklődést és a virtuális függvények használatát teljesen mellőzve. A program elkészítése folyamán, komoly problémákba ütköztünk, melyeket a GTL, vagyis általánosan, a generikus programozás korlátai okoztak. Így a feladat bővült, ezen problémák és korlátok feltárásával és annak a bizonyításával, hogy ezek a korlátok, az elvárt megszorítások (dinamikus polimorfizmus mellőzése) mellet nem kerülhetőek meg.

2. Egy kézenfekvő megoldás – öröklődés

Ha az ember geometriai alakzatokat rajzoló és manipuláló programot akar írni, akkor azt objektum orientált megközelítés, vagyis az öröklődés jó választásnak tűnik. A tipikus példa, az öröklődés és az objektum orientált programozás szemléltetésére pontosan az ilyen rajzoló programok implementációja. Egy `Shape` bázis-osztályból indulnánk ki és ebből származtatnánk a `Polygon`, `Circle` és hasonló osztályokat. A `Shape` osztályban deklaráljuk a virtuális `draw`, `move`, `rotate`, stb. metódusokat, majd a származtatott osztályokban definiáljuk őket és egy `Shape*` típusú, azaz alakzatokra mutató mutatókat tartalmazó tömbbe pakoljuk a rajzlap tartalmát. Ilyen megoldással erősen támaszkodunk az öröklődésre és a dinamikus polimorfizmusra.

A Java első kiadásai ezt a megközelítést alkalmazták a szabványos tárolók megvalósításához. Jávában minden típus az `Object` típusból származik, azaz Jávában minden objektum típusú [4]. Így kézenfekvő volt az a megoldás, hogy az `ArrayList`, a `List`, a `Set` és a `Map` konténerek `Object` típusú eleme-

ket (pontosabban referenciákat) tárolnak. Ennek egyértelmű hátránya volt, hogy a tárolók úgymond „nem tudtak" a tárolt elemek típusáról semmit. Továbbá amikor használták a konténer elemeit, akkor azok mivel `Object` típusúak voltak, ezért futásidejű, dinamikus típuskonverziót kellet rajtuk végrehajtani ami valamilyen mértékben rontotta a program teljesítményét. Ezt ki lehetett kerülni új, „típus-tudatos" osztályok létrehozásával. Ilyen lehetne például a `ShapeArray` osztály, amelyhez megírhatjuk a megfelelő `DrawAll(ShapeArray sa)` és `RotateAll(ShapeArray sa, float r)` függvényeket.

A konténereket és a rajtuk végrehajtandó algoritmusokat C++-ban nagyon elegáns módon oldották meg, generikus programozással, azaz sablon-osztályok és sablon-függvények segítségével. A C++ STL könyvtárában (Standard Template Library), a konténerek mint például a `vector`, a `list`, a `set` stb. mind osztály-sablonok, melyeket a tárolandó elemek típusával paraméterezünk. Ezek mellé, az STL még a megfelelő algoritmusokat is biztosítja, mint például a `sort` vagy a `find`. Ezek sablon függvényként vannak implementálva, és iterátorokon keresztül kommunikálnak a konténerekkel. Az STL-nek megközelítése számos előnye van az `Object`eket tároló konténerrel szemben:

- Típus biztonságos: A fordító mindent elvégez helyettünk. Fordításiidőben példányosítja a osztály-sablonokat és függvény-sablonokat, és a megfelelő típus ellenőrzéseket is elvégzi. Tehát kisebb a hiba lehetőség.
- Hatékony: A sablonok segítségével a fordító mindent a fordítási időben kiszámol, leellenőriz és behelyettesít – minden a fordításkor eldől semmit sem hagy futási-időre.
- Egyszerűbb bővíteni és programozni: akár új algoritmusokkal, akár új konténerekkel egyszerűen bővíthetjük a könyvtárat, csak a megfelelő iterátorokat kell implementálni és a megfelelő konvenciókat (concepteket) követni.

Az utobbiból az is adódik, hogy ha van m típusunk és n műveletünk, akkor az objektum orientál hozzáállással ellentétben, ahol minden típus minden (virtuális) metódusát, azaz $O(n*m)$ függvényt kell implementálni, generikus programozással viszont elég $O(n+m)$, vagyis külön-külön implementálnunk kell a típusokat (és iterátoraikat) és külön az algoritmusokat.

A GTL elkészítését az ösztönözte, hogy az alakzatok konténerekre, az transzformációs műveletek pedig algoritmusokra hasonlítanak. A GTL úgy van felépítve mintha pontokat (pontosabban `vect<T>` típusú elemeket) tároló konténerek lennének, és az alakzatok iterátorai ezeken a pontokon, azaz a csúcsokon iterálnak végig.

Hivatkozások

[1] Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu, 2001, [1305], ISBN 963 9301 17 5 ö

[2] Zoltán Porkoláb, Róbert Kisteleki: *Alternative Generic Libraries*, ICAI'99 4th International Conference on Applied Informatics, Ed: Emőd Kovács et al., Eger-Noszvaj, 1999, [79-86]

[3] David Abrahams, Aleksey Gurtovoy: *C++ Template Metaprogramming - Consepts, Tools, and Techniques from Boost and Beyond*, Addison Wesley Professional, 2004, ISBN 0-321-22725-5

[4] Bruce Eckel, *Thinking in Java (3rd Edition)* , Prentice Hall PTR, 2002, [1119], ISBN-10: 0131002872