

TDK-dolgozat

Baráth Áron

Tervminta felismerés C++ programokban ¹

Készítette: Baráth Áron
Programtervező Informatikus, MSc.
ELTE

Témavezető: Sike Sándor
egyetemi docens
ELTE, Informatikai Kar,
Programozáselmélet és Szoftvertchnológia Tanszék



2012. június 23.

¹Készült a KMOP-1.1.2-08/1-2008-0002, az Európai Regionális Fejlesztési Alap (ERFA) és az Ericsson Magyarország támogatásával

Absztrakt

Az objektumorientált szoftvertervezés egyik fontos eleme a tervminták felhasználása és alkalmazása. A tervminták előre kidolgozott sablonok, melyek alkalmazásával megkönnyíthető a tervezés fázisa. Kész programokban is fellelhetőek ezek a minták, és a kód megértését nagyban segítik. Gyakran előfordul, hogy egy program nincs kellően dokumentálva, és szükséges megérteni a működését. Ekkor jól jön egy eszköz, amellyel meglévő kódban lehet tervmintákat keresni.

A dolgozatom témája tervminták felismeréséről szól. A felismerés absztrakt szintaxis fa alapján működik, és kapcsolódik hozzá egy prototípus program, ami C++ programokban keres tervmintákat.

Tartalomjegyzék

Bevezetés	2
1. A tervmintákról általában	4
1.1. A tervminta	4
1.2. Jelentősége	5
1.3. Felismerés	5
2. Absztrakt felismerés	7
2.1. Singleton	8
2.2. Template method	10
2.3. State	12
2.4. Composite	15
2.5. Factory method	17
2.6. Összefoglalás	20
3. Tervminták felismerése Clang-gal	21
3.1. Előkészületek	22
3.2. Megfeleltetések	28
3.3. Tervminta felismerés	29
4. Összefoglaló	40
4.1. Értékelés	40
4.2. Továbbfejlesztési lehetőségek	41

Bevezetés

Gyakran előfordul a szoftverfejlettszókkel, hogy a program kódot tovább kell fejleszteni, és hiányos vagy nem tér ki bizonyos részekre a dokumentáció; vagy nincsen dokumentáció. Ekkor a szoftverfejlesztőnek meg kell értenie a program működésének legalább az alapelveit, a részletekbe már csak akkor kell elmerülnie, ha szükséges.

Vannak olyan segédeszközök, amelyekkel programszöveget statikus elemzéssel lehet feldolgozni, megjelölni az érthető és homályos kódrészeket. Azonban ezekkel nagy mennyiségű kódot elemezni túl sokáig tart. Néhány eszköz segítségképpen magasabb szintű egységeket is képes meghatározni.

Motiváció. A kód megértésének szempontjából magasabb szintű egység a tervezési minta (*Design pattern*) is. Ha a szoftverfejlesztő maga előtt látná a kódban használt összes tervmintát, akkor lényegesen egyszerűbb dolga lenne, mert a tervmintákat nem lehet „össze-vissza” alkalmazni. Ezeknek a mintáknak céljaik vannak. A célok ismeretében pedig következtetni lehet a program viselkedésére, így a megértése is gyorsabban megy. Ezzel egyidőben az osztályok kapcsolatáról is képet kap.

A problémát az okozza, hogy a kész kódból a tervmintákat felismerni nem egyszerű feladat. Hibás vagy hiányos algoritmus téves felismeréseket is adhat eredményül. Ez pedig a megértést akadályozza, mert félrevezeti a szoftverfejlesztőt. A jelenleg elérhető eszközökben a tervminta felismerés nem áll olyan fejlettségi szinten, hogy megbízhatóan lehessen alkalmazni.

Dolgozatom célja az Eötvös Lóránd Tudományegyetemen az Ericsson-nal közösen fejlesztett „*gRocker*” nevű C++ forráskód megértését segítő eszközbe integrálható tervminta felismerő modul felvázolása, és a prototípus fejlesztése során eddig elért eredmények és tapasztalatok ismertetése. Az eszköz képes grafikusán is vizualizálni osztályokat, és a megjelenített osztályokhoz kiegészítő információként megjeleníthető az, hogy mely tervmintá(k)hoz csatlakozik az adott osztály.

1. fejezet

A tervmintákról általában

A tervminták objektumelvű rendszerek tervezéséhez nyújt segítséget újrafelhasználható tervek formájában. A rendszertervezők általában bevált tervekből építkeznek, és ezeket terveket újra és újra felhasználják. Annak érdekében, hogy könnyen be lehessen építeni egy tervet az adott rendszerbe, ismerni kell a terv absztrakt felépítését, tulajdonságait, valamint azt, hogy milyen esetekben célszerű használni azt. Ezeket a terv leírásokat nevezzük *tervmintáknak* (design patterns) [1, 2, 3]. A tervminta az újrafelhasználhatóság egyik eszköze. Az újrafelhasználhatóság jellemzői:

- meg kell felelnie a konkrét probléma sajátosságainak;
- elég általánosnak kell lennie, hogy más felmerülő feladatok megoldása során is alkalmazható legyen;
- ne kelljen, vagy minimálisan kelljen újratervezni.

1.1. A tervminta

A tervminta a következő elemekből épül fel:

Név: Ezzel a névvel hivatkozunk az adott tervmintára. Rövid, egy-két szóból álló kifejezés, amely utal a funkciójára. A tervezés során

ezeket a neveket adjuk meg, és kibővítjük az osztályokat néhány új adattaggal és metódussal. Egy tervmintának több neve is lehet.

Feladat: Ez a leírás tartalmazza azt, hogy az adott tervminta mire való, milyen esetekben lehet alkalmazni. Tartalmaznia kell a problémát, és a környezetét, valamint az esetleges feltételeket.

Megoldás: Osztálydiagram, az osztályok kapcsolatai, és az együttműködés leírása. Az osztálydiagram nyelvfüggetlen, és ezzel csak az objektumorientáltságot követeljük meg. Az általánosság érdekében nem tartalmaz semmilyen implementációt.

Következmények: A tervminta alkalmazása során kompromisszumok alakulhatnak ki (futási idő, tárkapacitás). Ezeknek a leírását kell itt megadni.

1.2. Jelentősége

Meglévő kódokat lehet úgy is ellenőrizni, hogy megkeressük benne, hogy a programozó milyen tervmintákat használt fel. Minél több tervminta van jelen a kódban, annál könnyebb következtetni a kód minőségére. Ugyanis az ismert tervminták bizonyítottan jók.

A tervminták jelenléte sajnos nem feltétlen jelenti azt, hogy a rendszer tökéletes. Igazán jó a rendszer akkor, ha a tervmintákat tudatosan használta a fejlesztő. Ezt azonban nehéz ellenőrizni a kód megértése nélkül.

1.3. Felismerés

A cél olyan algoritmus elkészítése, amely adott tervmintákat képes felismerni objektumorientált nyelven írt program kódban. Alapvetően az absztrakt szintaxis fa alapján lehet megfogalmazni feltételeket. Ilyen feltételeket általánosan is meg lehet fogalmazni, de a konkrét megvalósítás C++ kódokra működik.

Ahhoz, hogy C++ programban lehessen tervmintákat keresni, ismerni kell a tervminták általános leírását, felépítését. Az ehhez kapcsolódó fogalmakat nyelvfüggetlen módon adom meg az absztrakt szintaxis fa alapján. Feltételezve, hogy léteznek olyan fogalmak a fában, mint osztály, adattag, metódus – és ezeknek az elemeknek a tulajdonságait is lehet vizsgálni, például: típusok egyezése, dekorálása; nevek vizsgálata; paraméterek típusa és neve. Majd a konkrét megvalósítást mutatom be, amely a Clang által felépített szintaxis fát vizsgálja, és azonosít néhány tervmintát. Az absztrakt szintaxis fában a típusok egyezése egy magasabb szintű fogalom, Clang-ban ez esetenként egyszerűen adódik. De vannak olyan szituációk – mint például az aktuális template paraméterek esete – amikor nehéz eldönteni azt, hogy a két típus valamilyen módon hasonlít egymásra.

2. fejezet

Absztrakt felismerés

Célom tervminták felismerése az absztrakt szintaxis fa (AST) alapján, és a tervminták jellemzőinek meghatározása. Ezen jellemzők az osztálydiagram és a felépítés tanulmányozásával megadhatók.

A következőkben néhány példát mutatok, hogyan lehet nyelvfüggetlen módon felismerni az egyes tervmintákat. Ebben a fejezetben csak az általános, objektumorientált nyelvekre jellemző tulajdonságokra támaszkodva adjuk meg a tervminták felismeréséhez szükséges információkat.

A jellemzőkben a feltételek olvashatóak, melyek alapján az adott tervmintát felismerő algoritmus elkészíthető. A jellemzők megadásánál feltételezem az absztrakt szintaxis fáról, hogy létezik benne osztály típusú csomópont, aminek le lehet kérdezni a metódusait és adattagjait. A metódusokat reprezentáló csomópontok információt szolgáltatnak arról, hogy mi a visszatérési értékük, mik paraméterek nevei és típusai. Az alosztály-ősosztály viszony a fából természetes módon kinyerhető, ha alosztálynak van éle az ősosztály felé, illetve ha ez az él kétirányú, akkor egy osztály származtatott osztályait is fel lehet deríteni.

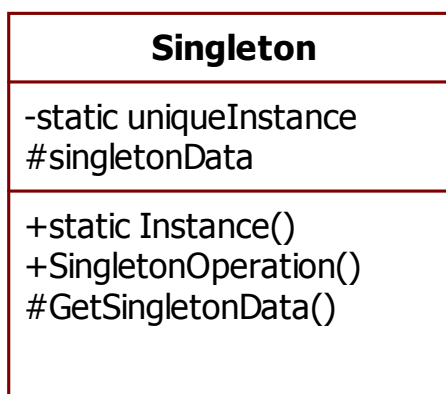
2.1. Singleton

Vannak olyan esetek, amikor biztosítani kell azt, hogy egy adott objektumból legfeljebb csak egy létezzen. Ilyen lehet például egy fájlrendszer kezelő objektuma, vagy egy ablakkezelő rendszer.

Megoldás lenne, hogy globális változóként hozzuk létre az objektumot, de semmilyen garancia nincs arra, hogy több példányt nem lehet létrehozni. Az objektumorientált felfogás szerint egy megadott objektum sémából tetszőleges számú objektum hozható létre. Az egyetlen felső korlátot a memória véges kapacitása határozza meg.

Ezt a gyakorlatban gyakran előforduló esetet a singleton (egyke) tervmintával oldhatjuk meg. Ebben a megoldásban maga az osztály felel az egyetlen példány létrehozásáért. Továbbá garantálja, hogy egynél több példány sose jöjjön létre.

2.1.1. Osztálydiagram



2.1. ábra. A Singleton osztálydiagramja

Az osztálydiagramban szereplő adattagok és metódusok:

uniqueInstance: Ez egy statikus adattag. Az egykének legfeljebb csak

1 példány lehet. Ezt a példányt tárolja ez a mező. Kezdetben az értéke az üres referencia.

singletonData: Az egyke tárolhat egyéb, futás során változtatható értékeket is, ez az adattag ezeket tárolja el. Ez csak jelölésbeli adattag, helyette természetesen több adattag is lehet.

Instance(): *Ez egy statikus metódus.* Ezzel a metódussal lehet lekérdezni a létrehozott példány referenciáját. Első híváskor (amikor a példány referenciája üres) hozza létre a példányt.
`return uniqueInstance;`

SingletonOperation(): Az egykének lehetnek egyéb metódusai is, ami a konkrét funkcionalitást biztosítja. Ezek a metódusok módosíthatják az egyke belső állapotát (*singletonData*), de ebből az egykéből új példányt már nem hozhat létre.

GetSingletonData(): Az egyke belső állapotának egy részét le is lehet kérdezni. A lekérdező metódusokat reprezentálja ez a metódus. Az általános konvenció szerint ezek a lekérdező függvények nem módosíthatják a belső állapotot.

2.1.2. Jellemzők

Az osztálydiagram alapján megadható néhány jellemző, amit egy osztálynak teljesíteni kell ahhoz, hogy egykének nevezzük.

1. Van egy olyan statikus adattagja, amelynek a típusa referencia (vagy mutató) a saját osztályra. A nevére általános követelményt nem lehet adni. Azonban erre a névre nincs feltétlenül szükség.
2. A `Instance()` metódus visszatérési értéke az előző pontban meghatározott tulajdonságokkal rendelkező statikus adattagnak kell lennie. Amennyiben csak egy ilyen adattag létezik, akkor nyilvánvaló a választás, de ha több is van, akkor a nevek alapján ki lehet választani a megfelelőt. Ami a választást befolyásolja, az az éppen

használt elnevezési konvenció, valamint a kódolási nyelv (angol, magyar, német, ...) is.

3. A Singleton osztály konstruktora nem publikus – azaz vagy privát, vagy védett. Erre azért van szükség, hogy ne lehessen osztályon kívüli kódból példányosítani az osztályt, vagyis nyelvi eszközzel védjük az osztályt.

Lehet több konstruktora is, de a egyik konstruktor sem lehet publikus.

2.1.3. Megjegyzés

A garancia, hogy csak egyetlen példányt lehessen létrehozni:

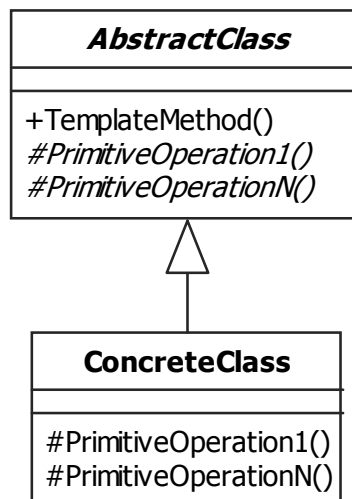
- A konstruktor privát vagy védett hozzáférésű, tehát csak az osztály metódusai férhetnek hozzá.
- A származtatott objektumok nem férhetnek hozzá a privát adattagokhoz és metódusokhoz. Így a privát konstruktorokat sem hívhatják meg. Csak a védett konstruktorokat tudják hívni, de ezek azok a konstruktorok, melyeket engedélyez az osztály.
- A konstruktort egyedül az `Instance()` metódus hívja meg akkor, ha a `uniqueInstance` üres referencia.

2.2. Template method

Egy algoritmusnak csak a vázát készítjük el úgy, hogy néhány lépést felüldefiniálható függvényként használunk. Így a származtatott osztályok a szerkezet megváltoztatása nélkül módosíthatják és kiegészíthetik az algoritmus bizonyos lépéseit. Az ősoosztály ebből a szemszögből nézve csak egy sablont jelent, amit származtatással lehet konkretizálni. Ezért a magyar megfelelője, a *Sablon művelet* egyértelműen leírja, hogy mit is várhatunk ettől a tervezési mintától.

Felhasználható olyan esetekben, amikor egy algoritmus állandó részeit külön is lehet implementálni. Ezzel elkerüljük a kód ismétlését, és a változó részek lokalizálva vannak a származtatott osztályokban. Ha változtatunk az algoritmuson, akkor nem kell számtalan helyre átvezetni, hanem elég az őszosztályban lévő kódot átírni. Végül a származtatott osztályok kiterjesztését is kontroll alatt tartja, nem lehet akárhol kiegészíteni, csak azokon a pontokon, amelyeket a sablon művelet definiált – vagyis a felüldefiniálható metódusokon keresztül.

2.2.1. Osztálydiagram



2.2. ábra. A Template method osztálydiagramja

2.2.2. Jellemzők

1. A tervminta felismerésének a menete az absztrakt őszosztályból indul. Ennek az osztálynak kell nézni az összes származtatott osztályát. Amennyiben egyetlen ilyen osztály sincs, akkor ez nem lehet sablon tervminta.
2. Az előző pontban kiválogatott osztályok közül csak azok jöhetnek

szóba, melyekben az őszosztálybeli metódusoknak van megvalósításuk. A továbbiakban csak ezeket az osztályokat kell vizsgálni.

3. Az őszosztályból ki kell gyűjteni azokat a metódusokat, amelyek nem publikus, absztrakt (vagy felüldefiniálható) metódusok. Ezek lényegében azok a metódusok lesznek, amelyeket a származtatott osztályok felüldefiniáltak.
4. Az őszosztály törzzsel rendelkező publikus metódusában kell keresi tagfüggvény hívásokat.
5. Az előző pontban kigyűjtött kifejezéseket szűrni kell úgy, hogy csak azokat kell megtartani, melyekre az alábbi tulajdonságok teljesülnek:
 - A függvényhívás az őszosztálynak (a saját objektumnak) egyik metódusáról szól.
 - A hívott metódus szerepel a 3. pontban.

2.3. State

Célja, hogy egy objektum futási időben megváltoztathassa a viselkedését. A viselkedés megváltoztatását rendszerint az objektum állapotának megváltozása okozza, ezért hívják a tervmintát *állapotnak* (ez az angol state magyar megfelelője). Az állapotokat általában felsorolással adják meg.

Ennek a tervmintának nem objektumorientált környezetben néhány függvény mutató felülírása, vagy – mivel az állapotokat felsorolással adják meg – egy terjengős és gyakran átláthatatlan switch-case lenne. Az ilyen struktúrájú felépítés bővítés szempontjából nem rugalmas.

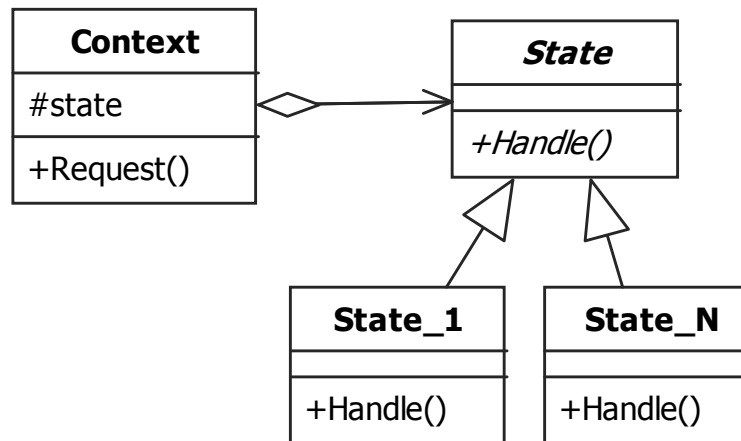
A tervminta 3 fontos szerepű osztályból áll:

- Az egyik a tényleges objektum, ami az elvárt viselkedést produkálja. Ennek az osztálynak az állapota megváltozhat. Ezt az osztályt jelöli a **Context**.

- A második egy absztrakt osztály, amely a viselkedést befolyásoló absztrakt metódusokat tartalmazza. Ennek az osztálynak a neve **State**. Ennek feladata, létrehozzunk belőle származtatott osztályokat, ahol a műveleteket megvalósítjuk. A **Context** osztály ennek az osztálynak a metódusait fogja meghívni. Továbbá az osztály feladata az is, hogy a **Context** osztályban elhelyezhessünk egy referenciát. Így az állapot megváltoztatása annyiból fog állni, hogy ezt a referenciát átállítjuk egy másik osztály példányára, melynek az őse a **State**.
- A harmadik szerep pedig több olyan osztály, amelynek a **State** osztály az őse. Ezeket az osztályokat **State_1 ... State_N** nevekké azonosítjuk. Ezeknek az osztályoknak nagyon egyszerű feladatuk van: az adott állapothoz tartozó műveletek megvalósítását tartalmazzák. Hivatkozhatnak a **Context** osztály adattagjaira, de saját attribútumokkal is rendelkezhetnek.

A TCP protokoll egy kiváló példa az állapot tervmintára, mert sok állapota van. Ezekben az állapotokban a kapcsolat felépítésétől kezdve az adatküldésen át a kapcsolat „szabályos” bontásáig különböző tevékenységeket kell elvégeznie. Például C-ben a fogadott csomagok feldolgozását végző függvény (ami tartalmazza az állapotok elválasztását is), körülbelül 2000 soros.

2.3.1. Osztálydiagram



2.3. ábra. A State osztálydiagramja

2.3.2. Jellemzők

Egy potenciális **State** osztályból kiindulva.

1. Ki kell gyűjteni az osztály származtatott osztályait. Ezek közül azokat kell megtartani, amelyek legalább egy őosztálybeli (**State**-beli) metódust megvalósítanak. Ha a kigyűjtött osztályok száma nulla, akkor nem kell folytatni a felismerést.
2. Az összes osztály közül ki kell válogatni azokat, amelyeknek van **State** referencia típusú adattagjuk. Ezek közül fog kikerülni a **Context** osztály, így a válogatás eredménye nem lehet üres. Ha üres lenne, akkor a kiválasztott **State** osztály nem illeszkedik a tervmintába.
3. A **State** felüldefiniálható (vagy felüldefiniálandó) metódusait kell kigyűjteni. Ha üres lenne az eredmény halmaz, akkor nem illeszkedik a tervmintába.
4. Ki kell gyűjteni azokat az osztályokat a 2. pontban jellemzett osztályok közül, amelyekben van publikus metódus. Ezek közül azokat

kell megtartani, amelyek törzsében van hivatkozás a 3. pontban kiválogatott metódusok egyikére. Ennek a válogatásnak az eredménye nem lehet üres.

5. Ha az eddigi kiválogatások sikeresek voltak, akkor meg lehet jelölni a **State** osztályt, mint egy állapot tervminta része. További részei az 1. pont eredménye, valamint a 4. pont eredménye. Fontos megjegyezni, hogy ez a megoldás több **Context** osztályt enged meg – ennek az az oka, hogy az elkészített állapotokat akár több osztály is felhasználhatja.

2.4. Composite

Vannak olyan felhasználási módok, amikor új objektumokat már meglévők segítségével – vagyis objektumok összetételével (kompozícióval → *composite*) hozunk létre. Ezt rendszerint fa szerkezetben tárolják. Cél, hogy az összetétellel létrehozott hierarchiákat egységesen lehessen kezelni úgy, mint az önálló objektumokat.

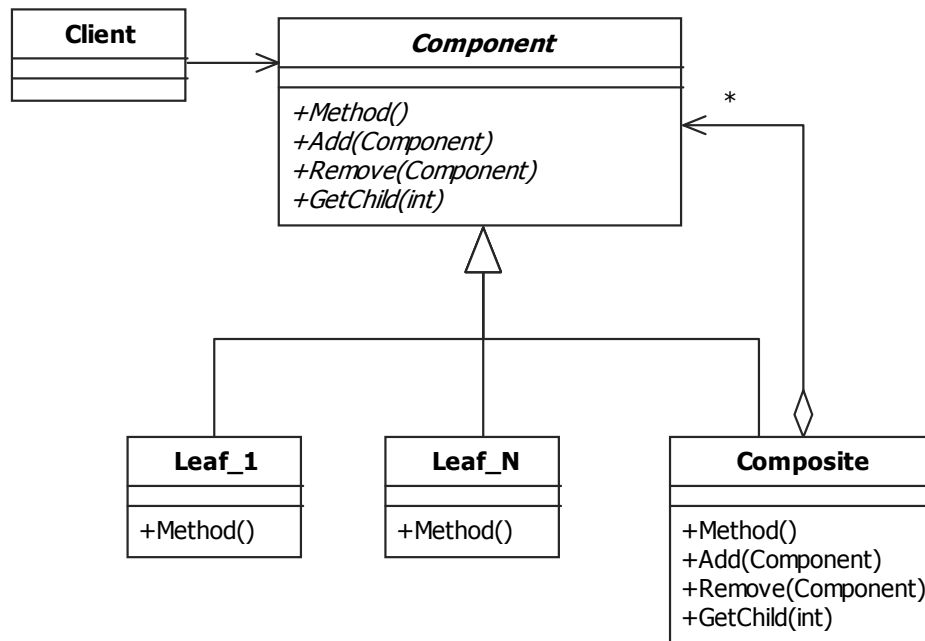
A tervminta elemei:

Component: Egy absztrakt osztály, ennek a megvalósításai a levél objektumok és az összetétel is. Ez biztosítja a levelek és az összetétel egységes kezelőfelületét.

Composite: Őse a **Component**, ez az összetétel megvalósítása, lényegében egy **Component** osztályokból álló gyűjtemény, amely a metódusok egy részét átirányítja a gyűjteménybeli objektumokra.

Leaf_i: A tervminta szempontjából levél objektumok, ezek az összetétel elemi objektumai.

2.4.1. Osztálydiagram



2.4. ábra. A Composite osztálydiagramja

2.4.2. Jellemzők

Egy potenciális **Component** osztályból kiindulva.

1. Meg kell keresni a **Component** osztály összes absztrakt metódusát. Ha nincs absztrakt metódusa, akkor nem tekintjük az összetétel tervminta **Component** osztályának.
2. Ki kell válogatni azokat a származtatott osztályokat, amelyeknek van legalább egy **Component**-beli metódusnak megvalósítása. Ennek az eredménye nem lehet üres.
3. A 2. pontban kigyűjtött osztályok közül ki kell gyűjteni azokat, amelyekben van valamilyen **Component** osztályokat tartalmazó gyűjtemény. Üres nem lehet, de általában csak egy ilyen osztály van.

4. A 3. pontban kigyűjtött osztályokat egyesével kell megvizsgálni:
- (a) Ki kell gyűjteni az osztály metódusainak a törzséből a tagfüggvény hívás kifejezéseket.
 - (b) Amennyiben a tagfüggvény hívások bármelyike szerepel az 1. pontban kigyűjtött absztrakt metódusok között, akkor az osztály benne van az eredményben.
- Az eredmény nem lehet üres.
5. A felismert tervminta a **Component** osztályból, és annak származtatott osztályaiból áll.

2.5. Factory method

Objektum vagy objektum hierarchia létrehozási felületének meghatározása. A konkrét objektumok példányosításáról a származtatott osztályok döntenek attól függően, hogy melyik alosztályról van szó. Abban az esetben is hasznos ez a tervminta, ha nem is tudjuk megadni előre, hogy mik lesznek a konkrét osztályok, így a létrehozás lokalizálttá válik. Ezzel a technikával felelősséget is tovább lehet adni a származtatott osztályok felé.

A tervminta legfontosabb eleme a *FactoryMethod* nevű metódus (a tervminta nem mondja ki, hogy csak egy metódus lehet – több objektum létrehozó metódus is lehet, akár paraméterekkel ellátva is), ami egy absztrakt metódus. Ez hozza létre a konkrét objektumot. A megoldás lényege, hogy a származtatott osztályok valósítják meg ezt a metódust.

A tervminta elemei:

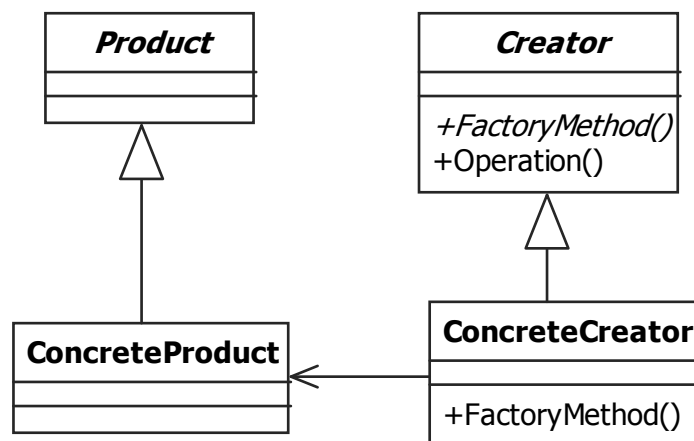
Creator: Egy absztrakt osztály, ennek a megvalósításai tartalmazzák a *FactoryMethod* implementációját. Tekinthető a tervminta csomópontjának is.

ConcreteCreator: Őse a **Creator**, a konkrét objektumot hozza létre.

Product: A létrehozandó objektumok őse, a **Creator** ezzel az absztrakt osztállyal hivatkozik a létrehozott konkrét objektumokra.

ConcreteProduct: Őse a **Product**, a konkrét objektum implementációja a tervminta szempontjából. Elég, ha a **ConcreteCreator** osztály ismerni ezt az osztályt.

2.5.1. Osztálydiagram



2.5. ábra. A Factory method osztálydiagramja

2.5.2. Jellemzők

A **Creator** osztályból kiindulva.

1. Ki kell válogatni a **Creator** osztály összes absztrakt metódusát. Ezek között kell lennie a *FactoryMethod* metódusnak, ezért ennek a válogatásnak az eredménye nem lehet üres.
2. A **Creator** származtatott osztályai közül ki kell gyűjteni azokat, amelyek megvalósítják legalább az egyik 1. pontban kiválogatott metódust. Az eredmény nem lehet üres.

3. A **Creator** osztály törzzsel rendelkező metódusaiban meg kell keresni az osztály hivatkozásokat. A keresés célja, hogy legyen egy alaphalmaz, amiből a **Product** osztály(ok) kerül(nek) ki. Ezen az osztályok közül azokat kell csak megtartani, amelyek szerepelnek visszatérési értéként¹ bármely 1. pontban kiválogatott absztrakt metódusban.
4. A 2. pontban kiválogatott osztályok metódusai közül azokra van szükség, amelyek megvalósításai a **Creator** osztály absztrakt metódusainak. Ezeket tovább kell szűkíteni a visszatérési érték ellenőrzésével: csak a 3. pontban kiválogatott osztály lehet visszatérési érték. A megmaradt metódusok törzseiben kell keresni osztály létrehozási hivatkozásokat. Az érintett osztályok képezik az eredményt.
5. A **ConcreateProduct** osztálynak megfelelő osztályokat úgy kell meghatározni, hogy a 4. pontban kapott osztályok halmazából metszetet kell képezni a következő osztályhalmazok uniójával:
 - A 3. pontban kapott osztályok.
 - A 4. pontban kapott osztályok és azok származtatott osztályai.A metszet nem lehet üres. Ha üres lenne, akkor nem része a gyártó művelet tervmintának.
6. A tervminta által érintett osztályok:
 - A **Creator** osztály.
 - A 2. pontban kapott osztályok.
 - Az 5. pontban kapott osztályok.

¹Lehet referencia és mutató típusú is.

2.6. Összefoglalás

A fenti leírásokban használt egyszerű műveletek² és komplex lekérdezések:

1. Alapvető halmaz műveletek (keresés, hozzáadás, unió, metszet).
2. Ősosztályok és származtatott osztályok kigyűjtése.
3. Egy osztály metódusainak és adattagjainak meghatározása.
4. Metódus és adattag láthatósága (public, protected, private).
5. Metódusok tulajdonságai (absztrakt, felüldefiniálható, nem felüldefiniálható, konstans).
6. Adattagok típusa; metódusok visszatérési értéke és paraméterei³.
7. Típusok összehasonlítása, akár konstansság ellenőrzése nélkül.
8. Típus dekorálása (referencia, mutató, konstans).
9. Osztályok, metódusok és adattagok szűrése megadott feltétellel.
10. Tagfüggvény hívások kigyűjtése egy adott metódus törzséből, a tagfüggvény hívás részeihez való hozzáféréssel együtt (hívott függvény, és az objektum).
11. Metódust megvalósító osztályok meghatározása egy metódushalmaz alapján.
12. Metódus törzsében hivatkozott/létrehozott objektumok; eredmény egy osztályhalmaz.

²Azokat tekintem egyszerű műveletnek, amelyek a szintaxis fa alapján adódnak. A művelet várható futási ideje nincs beleszámolva.

³Paraméter: típus és paraméter neve együtt.

3. fejezet

Tervminták felismerése Clang-gal

A tervminta felismerő prototípus program a C++ forráskódból a Clang [4, 5] segítségével olvassa be a memóriába a szintaxis fát. A szintaxis fa csomópontjait C++ osztályok reprezentálják, és a csomópont tulajdonságait (neve, típus információk, paraméterek, őssztály, ...) függvényekkel lehet lekérdezni. Lehetőség van a szintaxis fa átalakítására¹ is, de erre nem volt szükségem, mert a C++ kódot csak elemzés céljából töltöm be.

A szintaxis fa teljesen dinamikus², ezért minden csomópont egy mutató a leíró osztályra. A mutatók egyértelműen azonosítanak egyetlen csomópontot. Ezt a lehetőséget kihasználva a feldolgozás során elég csak a mutatókat tárolni valamilyen adatszerkezetben³.

A Clang úgy különbözteti meg a statikus és nem statikus adattagokat, hogy az egyiket *Variable*-nek (ez a statikus adattag), a másikat *Field*-nek (ez a nem statikus adattag) hívja.

Néhány fontosabb csomópont típus a Clang-ban:

CXXRecordDecl: C++ osztály csomópont. Ennek az osztálynak sok lekérdező függvénye van, amivel az osztály egyes részeihez férhe-

¹Például: új adattagok felvétele, új metódusok definiálása, új osztály létrehozása.

²Dinamikusan foglalható, és módosítható.

³A leggyakoribb a `std::set` és a `std::map`.

tünk hozzá [6].

Megfelelője a 2.6. fejezet 3. pontjának.

CXXMethodDecl: Metódus (függvény) deklarációs csomópont. Gyakori lekérdezések: a paraméter lista és a visszatérési érték. Ezekkel azonosítom a függvény deklarációkat [7].

Megfelelője a 2.6. fejezet 4 és 5. pontjának.

CXXMemberCallExpr: Tagfüggvény hívás kifejezés. Ez a csomópont fontos szerepet játszik majdnem az összes tervminta felismerésében [8].

Gyakran használt Clang-beli osztályok, melyek nem csomópontok:

QualType: (A *Qualified Type* rövidítése.) Ez az osztály írja le a típusokat a Clang-ban. Vannak metódusai, amivel lehet dekorálni⁴ és összehasonlítani [9].

Megfelelője a 2.6. fejezet 7. és 8. pontjának.

3.1. Előkészületek

Ebben a fejezetben a prototípus programhoz szükséges eszközöket ismertetem, ezek közül a legfontosabb osztály a Context osztály.

A gyorsabb fejlesztés érdekében létrehoztam néhány segédeszközt és adatszerkezetet. Ezek az eszközök a Clang-ra épülnek, azaz csak a Clang-ból elérhető információkkal dolgoznak. Ezért a használatukhoz nincs szükség semmilyen további információra.

A tervminta felismerő rendszer teljes egészében a Pattern névtérben található. A program legfontosabb osztálya a Context osztály, ahol a létrehozott adatszerkezeteket tárolom. A felhasználó (kliens) kód ezt az egyetlen osztályt látja, és ennek a szolgáltatásait használhatja.

A Context osztály műveletei:

⁴A prototípus programban csak a mutatóvá alakítást használom.

isPatternSupported(): Ellenőrzi, hogy a paraméterül kapott tervminta felismerése implementálva van-e. Ha igen, akkor a visszatérési értéke `true`, ha nem, akkor `false`.

Jelenleg támogatott tervminták:

- Singleton
- Template
- State
- Composite⁵

getSupportedPatterns(): Megadja az összes implementált felismerhető tervmintának a nevét egy listában. A tesztprogram ezt a függvényt fogja használni, hogy ne kelljen egyesével megadni az összes tervminta nevét. Továbbá a bővíthetőség szempontjából sem szerencsés, ha kézzel kell beírni a tesztprogramba az újonnan elkészített tervminta nevét.

getPattern(): Paraméterül kapja egy tervmintának a nevét⁶, előkeresi az azt felismerni képes osztályt, és lefuttatja a Context-ben fellelhető összes osztályra. A visszatérési értéke egy `pattern_t` típusú asszociatív tömb.

dumpClasses(): Kiírja a paraméterül kapott kimeneti folyamra (`std::ostream`) a Context-beli osztályokat. Ez a függvény hibakereséshez használatos.

A feldolgozó kódban az összes általam készített osztály és függvény is a `Pattern` névtérben található. Ebben a névtérben típusok is vannak, melyek gyakran előfordulnak. A tényleges típusok nevei túl hosszúak ahhoz, hogy kényelmesen lehessen velük dolgozni, ezért a C++ `typedef` szolgáltatásával készítettem néhány szinimomát az alábbi típusokról:

⁵A Composite tervminta implementációja nem teljes. Lásd: 3.3.4. alfejezet.

⁶Például az egyiket azok közül, melyeket a `getSupportedPatterns()` függvény adott vissza.

classes_t: Ez egy halmaz, amiben Clang-os osztály csomópontokat azonosító mutatók gyűjthetők.

```
std::set<clang::CXXRecordDecl*>
```

dercls_t: Ez egy asszociatív tömb, ami egy osztályhoz hozzárendel egy halmazt, melyben a megadott osztály származtatott osztályai szerepelnek.

```
std::map<clang::CXXRecordDecl*, classes_t>
```

pattern_t: A felismert tervmintákat egy asszociatív tömbben gyűjtöm össze. Minden mintát egy jellemző osztályával⁷ azonosítok. Cél volt egy olyan adatszerkezet alkalmazása, amit futás során kis költségű művelettel lehet módosítani, így később is lehet bővíteni a felismert tervmintát alkotó osztályokat.

```
std::map<clang::CXXRecordDecl*, classes_t>
```

3.1.1. Osztályok

A beolvasott C++ forrásokból ki kell gyűjteni az összes osztályt, mert nagyon sok helyen szükség van erre az információra. Az osztályokat a Context osztály konstruktora gyűjti ki. Ezt a `classes_t` típusú `Context::_classes` adattag tárolja.

Továbbá a Context osztály konstruktora építi fel azt az asszociatív tömböt, amely egy adott osztályhoz rendeli hozzá a származtatott osztályok halmazát (`classes_t`). Ezt az információt a Context osztály `dercls_t` típusú `_dercls` adattagja tárolja.

Ez megfelelője a 2.6. fejezet 2. pontjának.

A programban gyakori művelet egy adott osztály összes származtatott osztályának a legkérdezése. Ez az információ ilyen alakban nem létezik a Clang-ban, valamint redundáns lenne, és költséges lenne a karbantartása is.

Ezzel a megoldással egy probléma merül fel: nem minden osztálynak van származtatott osztálya, így ezek az osztályok nem szerepel-

⁷Ez általában egy absztrakt ősoosztály.

nek a `_dercls`-ben. Ezért a `getDerivedClasses()` függvény törzse nem olyan egyszerű⁸. Az `std::map` osztályban található `find()` műveletet kell meghívni az osztály azonosítójával (vagyis a mutatójával), melynek visszatérési értéke egy iterátor (pontosabban `const_iterator`).

Ha az iterátor értéke megegyezik a `_dercls.end()` értékével, akkor az osztálynak nincsenek származtatott osztályai, és egy üres halmazt kell visszaadnia a függvénynek. Ehhez szükséges a `Context`-ben található `_empty` nevű adattag, amely egy osztályokat tartalmazó üres halmaz. Ugyanis a `getDerivedClasses()` visszatérési értéke egy `const classes_t &`.

Ha az iterátor nem a végét jelölő iterátor, akkor az iterátor által mutatott pár második komponensét kell visszaadni (ez egy osztályokat tartalmazó nem üres halmaz – azért nem üres, mert ha üres lenne, akkor nem is szerepelne a `_dercls`-ben).

3.1.2. Látogató

A Clang biztosít egy látogatót⁹ (*Visitor*) a szintaxis fához, de ennek a használata nem minden esetben egyszerű. Ezért a látogató létrehozásához készítettem egy egyszerűsített látogatót. A látogató a `Pattern::Tools` névtérben található.

Ez a látogató nem csak egyszerűbben használható, hanem gyorsabban működik is, mert csak a deklarációkat nézi végig, míg a Clang látogatója a függvények törzsébe is eljut – ezáltal egy teljes részfát bejárni, amire semmi szükség nincs. A saját látogató pedig a Clang-beli `CXXRecordDecl` osztály `decls_begin` és `decls_end` által elérhető deklarációkkal foglalkozik (ebben a felsorolásban található az osztály összes deklarációja).

```
class SimplifiedVisitor
{
public:
```

⁸Nem használható az `std::map` indexelő operátora (`operator[]`).

⁹A látogató is egy tervminta.

```

virtual void var(const VarDecl* v) { } // static attr.
virtual void field(const FieldDecl* f) { } // instance attr.
virtual void method(const CXXMethodDecl* m) { }
virtual void ctor(const CXXConstructorDecl* c) { }
virtual void dtor(const CXXDestructorDecl* d) { }
};

```

Ebből látogatóból célszerű származtatni, és a származtatott osztályba tenni adattagokat, illetve új függvényeket. A fenti 5 virtuális (felüldefiniálható) függvényre vonatkozó információk:

var(): Ez a függvény az osztály összes statikus adattagjára pontosan egyszer hívódik meg.

field(): Ez a függvény az osztály összes nem statikus adattagjára pontosan egyszer hívódik meg.

method(): Ez a függvény az osztály összes metódusára (statikus és nem statikus metódusokra egyaránt) pontosan egyszer hívódik meg.

ctor(): Ez a függvény az osztály összes konstruktorára pontosan egyszer hívódik meg.

dtor(): Ha van az osztálynak destruktora, akkor ez a függvény pontosan egyszer meghívódik. Ha csak ezt függvényt kell felüldefiniálni, akkor célszerűbb ehelyett a `CXXRecordDecl` osztály `getDestructor()` metódusát meghívni¹⁰.

Ez az egyszerűsített látogató nem a Clang által biztosított látogatót használja, mert a Clang-os látogató a teljes részfat bejárja, amit a `CXXRecordDecl` meghatároz. Ehelyett jóval optimálisabb a saját látogató megoldása, hiszen ez csak az osztály deklarációit nézi át. Így a felsorolásban csak azok a csomópontok szerepelnek, melyek valóban szóba jöhetnek.

¹⁰Ha van az osztálynak destruktora, akkor a `getDestructor()` visszatérési értéke nem a nulla mutató.

3.1.3. Halmazok és szűrések

A programban a legtöbbet használt adatszerkezet a halmaz. Ezek a halmazok a `std::set` példányosított változatai. Például: osztályok (`CXXRecordDecl`) halmaza, metódusok (`CXXMethodDecl`) halmaza. Megfelelője a 2.6. fejezet 1. pontjának.

Egy adott osztálynak le lehet kérdezni az összes, megadott tulajdonsággal rendelkező statikus adattagjainak, nem statikus adattagjainak és metódusainak a halmazát. Ezek a segédfüggvények a `Pattern::Tools` névtérben találhatók:

`findVarsWithType()`: Kiválogatja a paraméterül kapott osztályból azokat a statikus adattagokat, melyeknek a típusa a paraméterül kapott típus.

`collectFields()`: Kiválogatja a paraméterül kapott osztályból a megadott feltételt kielégítő nem statikus adattagokat.

`findFieldsWithType()`: Kiválogatja a paraméterül kapott osztályból azokat a nem statikus adattagokat, melyeknek a típusa a paraméterül kapott típus.

`findMethodsWithResultType()`: Kiválogatja a paraméterül kapott osztályból azokat a metódusokat, melyeknek a visszatérési értéke a paraméterül megadott típus.

`collectMethods()`: Kiválogatja a paraméterül kapott osztályból a megadott feltételt kielégítő metódusokat.

`collectMemberCalls()`: Paraméterül kap egy metódusokat tartalmazó halmazt, és a visszatérési értéke egy `CXXMemberCallExpr`-eket tartalmazó halmaz. A kimenetben csak azok az adattag hívások fognak szerepelni, melyek a paraméterül kapott metódusok törzsében találhatóak.

Megfelelője a 2.6. fejezet 10. pontjának.

collectMethodImplementerClasses(): Ez egy osztályokat tartalmazó halmazt szűr. A bemenetként kapott halmazból csak azokat fogja a kimenetbe rakni, melyek a paraméterül kapott osztálynak legalább egy virtuális vagy absztrakt metódusát megvalósítják. Megfelelője a 2.6. fejezet 11. pontjának.

A leggyakoribb szűrési feladat a `classes_t` típusú halmazok szűrése. Ehhez hoztam létre a `filter()` függvényt. Paraméterei:

s_: Ez egy osztályokat tartalmazó halmaz, ebből a halmazból veszi egyesével az osztályokat, megvizsgálja, hogy a megadott feltétel igaz-e. Ha igaz, akkor szerepelni fog a visszatérési értékben.

filt_: Szűrő feltétel, amelyet meghív az `s_` összes elemére. Ennek a függvénynek a visszatérési értéke legyen `true`, ha a paraméterül kapott osztályra szükség van a kimenetben, és legyen `false`, ha erre az osztályra nincs szükség.

Paraméterei:

item: Ezt az osztályt kell vizsgálni. (Az `s_` halmaz egyik eleme.)

userdata: Ez ugyanaz a mutató, amit a felhasználó adott meg a `filter()` függvény hívásakor.

userdata_: Ez egy tetszőleges adat lehet, semmilyen megkötés nincs arra, hogy mi lehet. Ezt a paramétert átadja a `filt_` függvénynek.

A visszatérési értéke egy másik halmaz, melyben csak azok az osztályok szerepelnek, melyekre teljesült a `filt_` szűrő feltétel.

Megfelelője a 2.6. fejezet 9. pontjának.

3.2. Megfeleltetések

Az absztrakt szintaxis fa alapján definiált műveleteknek egy kivételével mindnek van megfelelője a prototípus programban. Ezeket a műveleteket a Clang, és a C++ eszközeivel sikerült megvalósítani.

A hiányzó művelet a 2.6. fejezet 12. pontjában leírt művelet. Azért nem szerepel az implementációban, mert erre a műveletre nem volt szükségem a prototípus programban egyik tervmintához sem¹¹, így nem is foglalkoztam ennek megvalósításával.

3.3. Tervminta felismerés

3.3.1. Singleton

Az egyke tervmintát felismerő függvény megegyezik az absztrakt algoritmussal, semmilyen kiegészítésre nem volt szükség.

Ennek a tervmintának a szerkezetét sikerült olyan szinten leírni, hogy az absztrakt algoritmus megvalósítása a prototípus programban kiemelkedő módon viselkedik. Kipróbáltam sok egykének látszó – amik valójában nem egykék – osztállyal, és nem ismerte fel egykének. Csak azokat, amik tényleg egykék.

3.3.1.1. Példa: felismert egyke tervminta

```
class Singleton
{
public:
    static Singleton* Instance() { return mInst; }
protected:
    Singleton() { }
private:
    int my_field;
    static Singleton* mInst;
};

Singleton* Singleton::mInst = 0;
```

¹¹A *Factory method* tervmintához kellene ez a lekérdezés, de ennek a tervmintának az implementációja még nem készült el.

```

int main()
{
    Singleton* x = Singleton::Instance();

    return 0;
}

```

A fenti Singleton osztály majdnem azonos az absztrakt leírásban megadott osztállyal, csak néhány átnevezésben tér el. Ezt a prototípus program felismerte egykének.

3.3.1.2. Példa: nem egyke tervminta

```

class FalseSingleton3
{
public:
    static FalseSingleton3* Instance()
        { return 0/*mInst*/; }
protected:
    FalseSingleton3() { }
private:
    /*static*/ FalseSingleton3* mInst;
};

//FalseSingleton3* FalseSingleton3::mInst = 0;

```

A fenti FalseSingleton3 osztály nem egyke, mert nincs statikus adattagja, amiben a példány referenciáját (mutatóját) tárolja. A prototípus program ezt nem ismeri fel egykének, ami helyes eredmény.

3.3.2. Template method

A sablon művelet megvalósítása megegyezik az absztrakt algoritmussal.

Ezt a tervmintát felismerő absztrakt algoritmus implementálása is jó eredményeket produkált. Azonban ennél a tervmintánál már adódhatnak olyan szituációk, amikor felismer egy osztálycsoportot sablon műveletnek akkor, amikor valójában nem annak készült. Ezt nem az algoritmus hibájának tekintem, hanem ez egy természetesen adódó, gyakran használt konstrukció, aminek formálizálása a sablon művelet.

3.3.2.1. Példa: felismert sablon metódus tervminta

```
class TemplBase
{
public:
    void method() { op1(); op2(); }
protected:
    virtual void op1() = 0;
    virtual void op2() { }
};

class TemplDerived1 : public TemplBase
{
public:
    TemplDerived1() : x(0), y(0) { }
protected:
    virtual void op1() { ++x; }
    void op2() { ++y; }
private:
    int x, y;
};

class TemplDerived2 : public TemplBase
```

```

{
public:
    TemplDerived2() : xx(0), yy(0) { }
protected:
    virtual void op1() { ++xx; }
    void op2() { ++yy; }
private:
    int xx, yy;
};

```

A fenti kód részlet egy egyszerű sablon metódus tervminta, alapja az absztrakt leírásban található osztálydiagram. A prototípus program ezt a tervmintát sikeresen felismeri.

3.3.2.2. Példa: nem sablon metódus tervminta

```

class TemplFalseBase
{
public:
    void method() { /*op1(); op2();*/ }
protected:
    virtual void op1() = 0;
    virtual void op2() { }
};

class TemplFalseDer1 : public TemplFalseBase
{
public:
    TemplFalseDer1() : x(0), y(0) { }
protected:
    virtual void op1() { ++x; }
    void op2() { ++y; }
private:

```

```

    int x, y;
};

class TemplFalseDer2 : public TemplFalseBase
{
public:
    TemplFalseDer2() : xx(0), yy(0) { }
protected:
    virtual void op1() { ++xx; }
    void op2() { ++yy; }
private:
    int xx, yy;
};

```

Ez a példa egyetlen különbségtől eltekintve megegyezik a 3.3.2.1. alfejezetben látott kódtól¹². A különbség a `TemplFalseBase` osztály `method()` metódusában van, mert nincs benne hivatkozás virtuális függvényre.

Ezt a kódot a prototípus program nem ismeri fel sablon metódusnak, mert nem is az.

3.3.3. State

Az állapot megvalósítása megegyezik az absztrakt algoritmussal.

Ennek a tervmintának a felépítése elég szigorú ahhoz, hogy a téves felismerések száma alacsony legyen. A prototípus programban a felismerési arány nagyon magas, néhány kivételes konstrukció esetén előfordulhat téves felismerés, vagy nem tekinti állapot tervmintának az adott osztályokat. A felismerő algoritmus ugyan megfelelő megszorításokat tesz, ennek ellenére képes téves felismeréseket tenni. Ennek oka az, hogy nagyon hasonlít a *stratégia* tervmintára, de csak egy árnyalat-

¹²Átnevezés is történt, de az nem folyásolja be a felismerést. Minden osztály nevébe belekerült a *False* szó.

nyi különbség van. Az algoritmus ezt az apró különbséget nem minden esetben képes kiszűrni.

3.3.3.1. Példa: felismert állapot tervminta

```
class StateBase
{
public:
    virtual void handle() = 0;
};

class StateContext
{
public:
    void request() { state->handle(); }

private:
    StateBase* state;
};

class StateDer1 : public StateBase
{
public:
    void handle() { x = 1; }
private:
    int x;
};

class StateDer2 : public StateBase
{
public:
    void handle() { y = 2; }
private:
```

```
    int y;  
};
```

A fenti példa az absztrakt leírásban található osztálydiagram alapján készült, ez egy egyszerű állapot tervminta. Két levél (**Leaf_i**) osztály szerepel a kódban, de elég lenne egy is. A prototípus program ezt felismeri állapot tervmintának.

3.3.3.2. Példa: nem állapot tervminta

```
class StateFalseBase  
{  
public:  
    virtual void handle() = 0;  
};  
  
class StateFalseContext  
{  
public:  
    void request() { /*state->handle();*/ }  
  
private:  
    StateFalseBase* state;  
};  
  
class StateFalseDer1 : public StateFalseBase  
{  
public:  
    void handle() { x = 1; }  
private:  
    int x;  
};
```

```
class StateFalseDer2 : public StateFalseBase
{
public:
    void handle() { y = 2; }
private:
    int y;
};
```

Ez a példa nem sorolható az állapot tervminták közé, mert a `StateFalseContext` osztály `request()` metódusa nem hivatkozik a `StateFalseBase` egyetlen metódusára sem. A fenti konstrukciót a prototípus program nem ismeri fel állapot tervmintának.

3.3.4. Composite

Az összetétel tervmintát felismerő kód elkészült része megegyezik az absztrakt algoritmussal, de hiányzik belőle a konténert felismerő kód. Az implementációban szerepel egy kezdetleges konténer felismerés, de nem teljes:

- Nem ismer fel minden konténert, és az is előfordulhat, hogy konténernek jelez valamit, ami nem konténer.
- Az implementációja nem teljes. Az osztály aktuális `template` paramétereit összehasonlító kód nem készült el¹³.

A konténerek azonosítására az alábbi feltételt dolgoztam ki: A konténer egy `template` osztály, és az aktuális `template` paraméterek között szerepelnie kell a tárolt objektumok típusának¹⁴.

¹³A problémát az okozza, hogy a Clang az aktuális `template` paramétereket olyan módon tárolja, amiből nehéz `QualType` objektumot létrehozni.

¹⁴Vagy a tárolt objektumok típusának valamilyen dekorált változatának.

3.3.4.1. Példa: felismert összetétel tervminta

```
template<typename T>
class FakeContainer
{
public:
    FakeContainer() : mSize(0) { }
    void insert(const T & t) { ++mSize; }
    void erase(const T & t) { --mSize; }
    T & at(int i) { return mT; }
    int size() const { return mSize; }

private:
    int mSize;
    T mT;
};

class CompositeComponent
{
public:
    virtual void method() = 0;

    virtual void add(CompositeComponent* c) { }
    virtual void remove(CompositeComponent* c) { }
    virtual CompositeComponent* get(int i) { return 0; }
};

class CompositeLeaf1 : public CompositeComponent
{
public:
    void method() { int i = 0; }
};
```

```

class CompositeLeaf2 : public CompositeComponent
{
public:
    void method() { double d = 0.; }
};

class CompositeComposite : public CompositeComponent
{
public:
    void method()
    {
        for(int i=0,n=mChildren.size();i<n;++i)
            mChildren.at(i)->method();
    }

    void add(CompositeComponent* c)
        { mChildren.insert(c); }

    void remove(CompositeComponent* c)
        { mChildren.erase(c); }

    CompositeComponent* get(int i)
        { return mChildren.at(i); }

private:
    FakeContainer<CompositeComponent*> mChildren;
};

```

Ez a példa megfelel az összetétel tervminta jellemzőinek, azonban a konténer adatszerkezet jelentősen le lett egyszerűsítve, hogy könnyebben lehessen ellenőrizni a prototípus program viselkedését¹⁵.

¹⁵A tervminta felismerés nem rendelkezik a konténer helyességéről.

A fenti konstrukciót a prototípus program felismeri összetétel tervmintának, azonban az implementáció nem teljes, így könnyen adható olyan példa, amit hibásan felismerne összetétel tervmintának. Az implementált változat gyengepontja a konténerek azonosítása. A konténerek azonosításának problémáját a 4.2.1. fejezetben bővebben kifejtem.

4. fejezet

Összefoglaló

4.1. Értékelés

A dolgozatomban ismertettem a tervminták felépítését, és szerepüket a programkód megértésében. Bemutattam néhány tervminta részletes, felismerésükhöz használható jellemzését. A jellemzőkben használt lekérdezésekről írtam egy összefoglalót, és megadtam azok Clang-beli megfelelőit. Készítettem a segédfüggvényekről egy rövid leírást, és azok használatát. Az absztrakt szintaxis fában megadott jellemzők, és a megfeleltetések felhasználásával elkészítettem a prototípus programot.

Az absztrakt szintaxis fa alapján felállított szabályrendszer elég precíz ahhoz, hogy a tervmintákat nagy biztonsággal azonosítani lehessen velük. Ezt a sikert a prototípus program eredménye is mutatja. A prototípus programot interfész módosítással be lehet építeni a „gRocker” nevű C++ forráskód megértését segítő eszközbe.

Az elkészített prototípus programot kipróbáltam egy nagy, 100 ezer soron programon, és a benne szereplő összes egyke, sablon művelet, állapot és összetétel¹ tervmintát felismerte. A programban található 50 osztály közül 35 volt része valamilyen tervmintának.

Néhány tervminta felépítésében csak árnyalatnyi különbség van. Ez bizonyos szempontból hátrány, de vannak előnyei is. Hátrány az, hogy

¹Tekintettel arra, hogy az összetétel tervmintát felismerő algoritmus nem teljes.

így nehéz különbséget tenni a felismert tervmintákban, és ez hibás eredményt produkálhat – például a Állapot és a Stratégia tervminták. Az előny azonban csak implementálásnál mutatja meg magát: hasonló a felismeréshez szükséges kód, rengeteg közös kód van, amiket ki lehet emelni segédfüggvényekbe. Mivel nagyon sok kód kiemelhető, de ezek jórészt egyformák, ezért aránylag kevés függvény jön létre. Ezeket a függvényeket kiválóan lehet optimalizálni, és nem fáradságos munka, hiszen nem kell sok függvénnyel foglalkozni.

További előny, hogy a reprezentáció lecserélése így egyszerű, mert a reprezentációtól függő kód nagy része átkerült a segédfüggvényekbe. A Clang szintaxis fájától nagyban eltérő adatbázisos reprezentáció az általam használt lekérdezésekre jóval optimálisabb megoldást kínál.

4.2. Továbbfejlesztési lehetőségek

A prototípus program futási sebessége ugyan nem a legjobb, de a segédfüggvények nincsenek optimalizálva. Érdekes feladat lehet szintaxis fában megfogalmazott lekérdezések optimalizálása, valamint ezek párhuzamosítása. Mivel a különböző tervminták feldolgozása teljesen függetlenül elvégezhető, így a tervminták keresését több szálon is el lehetne indítani, és az I/O műveletek okozta szabad processzor időt fel lehet használni a memóriában tárolt adatok feldolgozására – vagyis másik szálon futó tervminta feldolgozását fogja futtatni a processzor az üresjárat helyett.

Az általam használt lekérdezéseket sokkal hatékonyabban meg lehetne fogalmazni SQL lekérdezések formájában. Ilyen lekérdezéseket a Clang szintaxis fájában nem lehet feldolgozni, ezért felmerülhet továbbfejlesztési lehetőségként a szintaxis fa átalakítása olyan formára, amit már el lehet tárolni adatbázis táblákban.

4.2.1. Konténerek

A prototípus programban a **Composite** tervminta felismerése nem teljes, mert hiányzik belőle a konténerek felismerése. A tervmintákhoz hasonlóan a konténerekre is adhatók hasonló jellemzők. Például a konténerekre jellemző műveletek (hozzáadás, törlés, keresés) nevét illetve paraméterezését lehet összehasonlítani egy megadott sémával.

Példa névre: *add*, *add_item*, *insert*, *push*, *push_back*, *push_front*. Ez a halmaz bővíthető más nyelvű megfelelőekkel is. Az ilyen nevű metódusok törzsét ellenőrizni lehet, hogy az osztály belső adatszerkezeteit miként változtatja, például van-e benne memória foglalás, vagy darabszámot azonosító egész szám növelése (ha `std::vector`-szerű a belső adatszerkezet). A belső adatszerkezet lehet még láncolt ábrázolású is, ezt is lehetne figyelni, és megfelelő jellemzők vizsgálatával ellenőrizni.

Irodalomjegyzék

- [1] "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison Wesley,
1994.
- [2] Design Patterns,
http://en.wikipedia.org/wiki/Design_Patterns
2012.
- [3] Design pattern,
http://en.wikipedia.org/wiki/Design_pattern
2012.
- [4] Clang,
<http://clang.llvm.org>
2012.
- [5] Clang online dokumentáció,
<http://clang.llvm.org/doxygen/index.html>
2012.
- [6] CXXRecordDecl,
http://clang.llvm.org/doxygen/classclang_1_1CXXRecordDecl.html
2012.

- [7] CXXMethodDecl,
http://clang.llvm.org/doxygen/classclang_1_1CXXMethodDecl.html
2012.
- [8] CXXMemberCallExpr,
http://clang.llvm.org/doxygen/classclang_1_1CXXMemberCallExpr.html
2012.
- [9] QualType,
http://clang.llvm.org/doxygen/classclang_1_1QualType.html
2012.