

Szoftvertchnológia és -technikák

6. Előadás – Benedek Zoltán
Tervezési minták 1



Automatizálási és
Alkalmazott
Informatikai Tanszék

Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű. Kifejezett felhasználási engedély nélküli felhasználása szerzői jogi jogsértésnek minősül.

Tartalom

Tervezési minták 1

- > Definíció
- > Áttekintés
- > Szervezésük
- > Miben segítenek a tervezési minták
- > Template Method
- > Strategy

Definíció

- Programtervezési minta v. tervezési minta v. design pattern
- Tervezési minta leír egy gyakran előforduló programtervezési problémát, annak környezetét és a megoldás magját, amit alkalmazva számos gyakorlati eset hatékonyan megoldható.

Irodalom

- Az interneten bármely minta nevére rákeresve számos leírást találunk, a legtöbb minta a wiki-n is megtalálható
 - > (de kezeljük némi fenntartással, ezek a wiki leírások lehetnek hiányosak, vagy nem feltétlen a leggyakorlatiasabb megközelítést mutatják be)
 - [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: DESIGN PATTERNS, Elements of Reusable Object-oriented Software
 - > A nagy klasszikus, de ma már kicsit „régies”, ebben C++ és más, ma már ritkábban használt nyelven vannak a példák
 - https://sourcemaking.com/design_patterns
-
- Megjegyzés: előadáson és gyakorlatokon C# példák lesznek

Példakódok

- A legtöbb mintához tartozik példakód, kódjegyzésekkel ellátva
- GitHub-on érhető el
 - > A kód webböngészőben is nézhető/böngészhető a kód <https://github.com/bzolka/AUT-SZTT>
 - Ez alatt navigáljuk le a Előadás/Demo/DesPattCode mappába, az egyes minták demo kódja általában a mintának megfelelő nevű almappában található
 - > De célszerűbb az egészet a <https://github.com/bzolka/AUT-SZTT> alatt „Clone and download” gomb segítségével letölteni (vagy a git clone paranccsal klónozni), így Visual Studio-ban a solution megnyitható
 - > A legtöbb mintához „futtatható” kód is tartozik: a mintához tartozó mappában levő Program.cs fájlban a Main2 függvényt előbb nevezzük át Main-re
 - > Ha más mintához tartozó kódot szeretnénk „futtatni”, nevezzük vissza a Main-t Main2-re, mert a VS projektünkben csak egy Main nevű statikus függvény lehet: ha több is van, fordítási hibát kapunk.

Bevezető példa

Példa

- Egy alkalmazást fejlesztünk, melyben adatokat kell tömöríteni
 - > Több tömörítési algoritmust szeretnénk támogatni: zip, rar, 7zip
 - > Futás közben is lecserélhető legyen az algoritmus
 - > Könnyen bővíthető legyen új algoritmusokkal

Példa – kiindulás

Csak zip algoritmust támogat (nézzük VS alatt,

```
class DataProcessor
{
    // Ciklusban beolvassa, tömöríti, majd feldolgozza a tömörített adatokat
    public void Run()
    {
        byte[] inputData;
        while ( (inputData = readData()) != null) {
            byte[] compressedData = zipData(inputData);
            processCompressedData(compressedData);
        }
    }

    byte[] readData() { /* adat olvasása hálózatról */ }

    byte[] zipData(byte[] data) { /* adattömörítés zip algoritmussal*/ }

    void processCompressedData(byte[] data) { /* tömörített adat feld.*/*; }
}
```


Példa –előző továbbfejlesztve

Több algoritmust támogat – 1. rész

```
class DataProcessor2
{
    // ...

    CompressionAlg compressionAlg;

    public DataProcessor(CcompressionAlg compressionAlg)
    {
        this.compressionAlg = compressionAlg;
    }

    public void Run() // változatlan
    {
        byte[] inputData;
        while ((inputData = readData()) != null)
        {
            byte[] compressedData = compressData(inputData);
            processCompressedData(compressedData);
        }
    }

    byte[] readData() { /* adat olvasása hálózatról */ }
    void processCompressedData(byte[] data) { /* töm. adat. feld. */; }
```

Példa –továbbfejlesztve

Több algoritmust támogat – 2. rész

```
byte[] compressData(byte[] data)
{
    switch (compressionAlg)
    {
        case CompressionAlg.Zip:
            return compressUsingZip(data);
        case CompressionAlg.Zip7:
            return compressUsingZip7(data);
        case CompressionAlg.Rar:
            return compressUsingRar(data);
    }
}

byte[] compressUsingZip(byte[] data) { /* adattömörítés zip alg. */ }
byte[] compressUsingZip7(byte[] data) { /* adattömörítés 7zip alg.*/ }
byte[] compressUsingRar(byte[] data) { /* adattömörítés rar alg. */ }

}
```

Példa –továbbfejlesztve

- Megoldás elve
 - > Konstruktorban átadjuk paraméterként (enum), milyen tömörítő algoritmust szeretnénk, ezt eltároljuk egy tagváltozóban (compressionAlg)
 - > Ezen tagváltozó alapján választjuk ki később a tömörítő algoritmust
- Problémák
 - > A megoldás nem bővíthető könnyen új algoritmussal
 - A DataProcessor2 osztályba be van építve, milyen algoritmusokat ismer.
 - Ha újat szeretnénk bevezetni, a DataProcessor2 kódját módosítani kell. Ezt követően pedig újra kell tesztelni az osztályt.
 - Olyan bővíthetőségi megoldást keresünk, mely NEM igényli a DataProcessor2 módosítását új algoritmus bevezetésekor.
 - A megoldás majd a **STRATEGY** tervezési minta alkalmazása lesz, rövidesen visszatérünk rá
- Sok tervezési minta létezik, pl. Strategy, Singleton, Composite, Observer, stb.
 - > Mind más programtervezési problémára nyújt megoldást

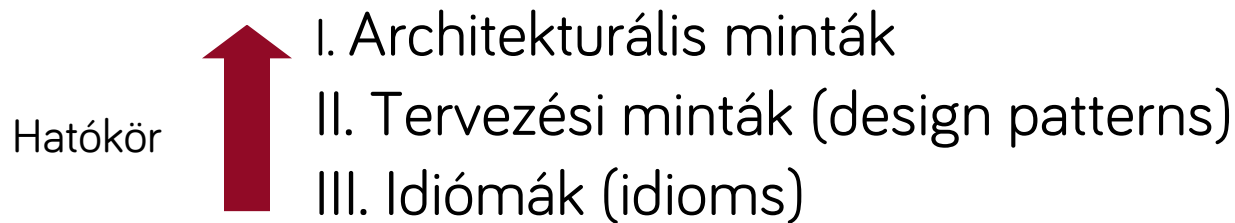
Tervezési minták jellemzői

Tervezési minták leírása

- Minden tervezési minta leírásához négy alapelem tartozik
 - > Pattern név (pattern name)
 - A minta neve, hatékonyan azonosítja a mintát.
 - **Fontos a kommunikáció miatt:** a fejlesztőknek elég a minta nevére hivatkozniuk, ebből már értik is, hogy milyen objektumok milyen szerepkörben szerepelnek az adott tervben/kódban (persze csak ha ismerik az adott mintát)
 - > Probléma (problem)
 - A **probléma** és a **környezet (context)** bemutatása, gyakran konkrét példán keresztül
 - > Megoldás (solution)
 - Leírja a megoldásban szereplő elemeket, kapcsolatukat, az egyes elemek felelősségét és együttműködését. Nem konkrét eset megoldása, inkább a megoldás absztrakt leírása.
 - > Következmények (consequences)
 - A minta alkalmazásának következményeit írja le. Fontos tapasztalatokat tartalmaz, és segít a minta kiválasztásban.

Minták csoportosítása hatókör szerint

- A szoftverrendszer milyen szintjén használhatók (architektúra → alrendszerek → ... → kódolás egy adott programnyelven)
- Három csoport



Minták csoportosítása hatókör szerint

- **Architektúra szint**

- > Alapvetően meghatározza az adott alkalmazás/alrendszer felépítését
- > Egy későbbi előadáson megnézünk párat, pl. MVVM, Document-View, Layers

- **Tervezési minta szint**

- > Egy alkalmazásban/alrendszerben több, sok is használható
- > Függetlenek a programozási nyelvektől

- **Idióma**

- > Egész alacsony szintűek, sokszor csak adott programozási nyelv kontextusában használhatók

- Mostantól a középső, „**Tervezési minta**” szintre besorolható tervezési mintákról lesz szó

- > Ezt nem kell túl szigorúan venni, több design pattern jól alkalmazható architektúra vagy idióma szinten is

„Klasszikus”/GoF tervezési minták

- Alapirodalom
 - > [Erich Gamma](#), Richard Helm, Ralph Johnson, John Vlissides: DESIGN PATTERNS, Elements of Reusable Object-oriented Software
 - > Először nevesítettek és katalogizáltak tervezési mintákat
 - > Négyen írták: köznyelvben „[Gang of Four](#)” (röviden GoF), vagyis a „négyek bandája”ként szokás rájuk hivatkozni.
 - Nagyon elterjedt ez a hivatkozás, illik tudni!

GoF tervezési minták kategóriák

A kiemelteket nézzük majd (ezeket kell tudni)

Létrehozási (creational)	Strukturális (structural)	Viselkedési (behavioral)
Factory Method	Adapter	Template Method
Abstract Factory	Composite	Strategy
Singleton	Façade	Observer
Builder	Proxy	Command
Prototype	Bridge	Memento
	Decorator	Interpreter
		Chain of Responsibility
		Iterator
		Mediator
		Flyweight
		State
		Visitor

Hogyan segítenek a tervezési minták a probléma megoldásában

- A **tervezési** minták a fejlesztés **tervezés** fázisában segítenek (az analízis minták az analízis fázisában)
 - > Tervezés: amikor kódra „képezzük le” az analízis során kidolgozott fogalmi modellt
- Az alábbiak elérése nehéz, ezekben a magasszintű célokban segítenek a tervezési minták:
 - > I. Könnyen változtatható, kiterjeszthető kódot készíteni (design for change)
 - > II. Újrafelhasználható kódot készíteni (design for reuse)
 - > III. Könnyen unit (egység) tesztelhető kódot készíteni

Részletesebben



I. Változtathatóság, bővíthetőség

- Nem adódik magától, szem előtt kell tartani a tervezéskor. A tervezés egyik célja!
- Miért?
 - > Rendszert tervezni nehéz
 - > Sokszor nem is tudjuk az elején pontosan definiálni a feladatot, a megrendelő sem lát nagyon előre, nem is lehet elvárni tőle
 - Az utóbbi években előtérbe kerülő **agilis szemléletmód** esetén ez hatványozottan igaz
 - A szoftver termékek az első kiadás után általában hosszú éveken át, számos iterációban fejlődnek tovább
 - > A szoftverfejlesztésben csak egy dolog (biztos) és változatlan: maga a változás
 - Változnak a felhasználói követelmények
 - Változnak a technológiák

I. Változtathatóság, bővíthetőség

- Tervezési alapelv: tervezzünk úgy, hogy a rendszer nem végleges (design for change)!

Válasszuk külön az alkalmazás azon részeit, melyek változatlanok és melyek változnak (arra számítunk, hogy változni/bővülni fognak). Zárjuk egységbe ezen változó kódrészeket, hogy később úgy tudjuk bővíteni/változtatni az alkalmazást, hogy a nem változó funkciók/részek kódjához ne kelljen hozzányúlni.

- > Nem mindent tervezünk bővíhetőre, mert költséges!
 - Csak ahol számítunk arra, hogy bővíteni/módosítani kell majd

I. Változtathatóság, bővíthetőség

- Negatív példa: készítünk egy alkalmazást, melyben minden mindennel összefügg, az SRP elvekre sem ügyelünk, valamint hiányzik a kritikus pontokban a módosításra/bővíthetőségre való felkészítés
 - > A kollégák nem fognak szeretni minket, ha a kódunkhoz kell nyúlniuk
 - > Nekünk sem lesz kedvünk a kódhoz többet hozzányúlni
 - > Egy bug javításával/pici módosítással
 - Sok kódot kell átnézni, újra tesztelni
 - Több új bugot viszünk be a rendszerbe
- Pozitív példa
 - > A meglévő kód bővítése, módosítása is pont annyira élvezhető feladat, mintha teljesen új kódot kellene írni

II. Újrafelhasználhatóság (reuse)

- Nem adódik magától, szem előtt kell tartani a tervezéskor. A szoftvertervezés egyik célja!
- A költségcsökkentés leghatékonyabb módja a cégek számára
- Akár egy projekt különböző moduljaiban, akár projektek között is!
- Ha az alkalmazásban minden kódrészlet mindennel összefügg, akkor nem tudunk belőle részeket újra felhasználni, mert minden rész újrafelhasználásához szükség van az összes függőségre
 - > Lazán csatolt „részekre” van szükségünk
- Hosszú távon kifizetődő lehet: saját **keretrendszer** kifejlesztése

II. Újrafelhasználhatóság (reuse) *

- **Kitérő: Keretrendszer (framework) definíció***

- > Minden keretrendszer adott probléma területet céloz meg (pl. ablakozós rendszerek, dokumentumkezelő rendszerek, matematikai rendszerek, grafikus rendszerek, CAD rendszerek, játékok, stb.)
- > Meghatározza az alkalmazás architektúráját, ehhez alapvető építőköveket (pl. osztályok) biztosít.
- > OO környezetben: előre definiált, együttműködő osztályok vannak, pontosan meghatározott felelősséggel. Az alkalmazás fejlesztésekor gyakran:
 - le kell belőlük származtatni,
 - Virtuális/absztrakt függvényeket kell felüldefiniálni
- > **Cél: Adott szakterületen egy alkalmazás fejlesztésekor a lehető legkevesebbet kelljen kódolni (mert a keretrendszer már szolgáltatásként nyújtja)**
- > Konzisztens struktúrája lesz az alkalmazásainknak
- > Magas szinten már “tervezni sem kell”, hiszen az architektúra meghatározott - terv újrafelhasználás (design reuse)

III. Unit (egység) tesztelhetőség

- Egy hosszú életű projekt esetén fontos, hogy a kód minél nagyobb részéhez automata módon futtatható **unit (egység) tesztek** készítsünk (legalábbis a logikai részekhez, felhasználói felületekhez kevésbé)
- Unit tesztek esetén kóddal tesztelünk kódot
- Egy unit teszt az osztályt önmagában, a függőségei (más logikai osztályok, melyeket használ) nélkül tesztel
 - > A unit tesztek futtatásához az osztály függőségeit le szeretnénk cserélni olyan implementációkra, melyek a tesztet szolgálják
 - > Ha egy osztályba be vannak építve a függőségei, akkor ez nem tehető meg, az osztály gyakorlatilag nem lesz unit tesztelhető
 - > A megoldás: az osztály a függőségeit lazán csatoltan kell kezelje (pl. interfészeken keresztül)
- Bizonyos tervezési minták abban is segítenek, hogy a kódunk könnyen unit tesztelhető legyen

System of Patterns *

- Mi a system of patterns?
 - > A rendszer tervezésekor több mintát is felhasználunk
- Általában egy (illetve alrendszerenként egy) architektúra szintűt
 - > Ez megadja az alapstruktúrát
- Az egyes részek megtervezésekor (detailed design) több tervezési mintát is felhasználunk, akár egymással kombinálva
- A kombinált megoldásba minden minta beleviszi a maga pozitív tulajdonságait, vagyis egy megoldást egy részproblémára

Néhány fontosabb, gyakrabban használt tervezési minta

- Létrehozási
 - > Factory method
 - > Abstract factory
 - > Singleton
 - > Dependency Injection*
- Strukturális és viselkedési
 - > Template method
 - > Strategy
 - > Observer
 - > Command
 - > Command Processor
 - > Memento
 - > Adapter
 - > Facade
 - > Proxy
 - > Composite

Mit kell tudni?

A minta neve alapján vagy példán, vagy általánosságában ismertetni a mintát (milyen környezetben, milyen problémát, hogyan old meg) . A bemutatáshoz a legtöbb mintához osztály, illetve néhány esetben szekvencia diagramot is kell rajzolni).

Bővíthetőséghez, kiterjeszthetőséghez kapcsolódó alap tervezési minták

Template method (sablon metódus) minta

Strategy (stratégia) minta

Egyéb

Célok, elvek

- Azon osztályoknál, melyeknél fontos a bővíthetőség, építsük be ennek lehetőségét
 - > Absztrakt/virtuális függvények hívásával -> Template Method minta
 - > Delegáljuk más osztályoknak -> Strategy minta
- A cél, hogy ha bővíteni kell a funkcionalitást, az osztály kódját NE KELLJEN MÓDOSÍTANI

Template method (Sablonmetódus)

Korábbi példa némi kiegészítéssel

- Egy alkalmazást fejlesztünk, melyben adatokat kell tömöríteni
 - > Több tömörítési algoritmust szeretnénk támogatni: zip, rar, 7zip
 - > Könnyen legyen bővíthető új tömörítési algoritmusokkal
 - > Futás közben is legyen lecserélhető az algoritmus
 - > Legyen megszakítható (cancel) a művelet, de ne legyen beégetve a megszakítás módja, pl.:
 - Billentyű lenyomás
 - Gomb kattintás

Kiindulási példa áttekintése

- Lásd: DesPattCode\
StrategyAndTemplateMethod_Common\
DataProcessor_Initial mappa DataProcessor osztály
feletti megjegyzések !

https://github.com/bzolka/AUT-SZTT/blob/master/EI%C5%91ad%C3%A1s/Demo/DesPattCode/DesPattCode/StrategyAndTemplateMethod_Common/DataProcessor_Initial/DataProcessor.cs

DataProcessor

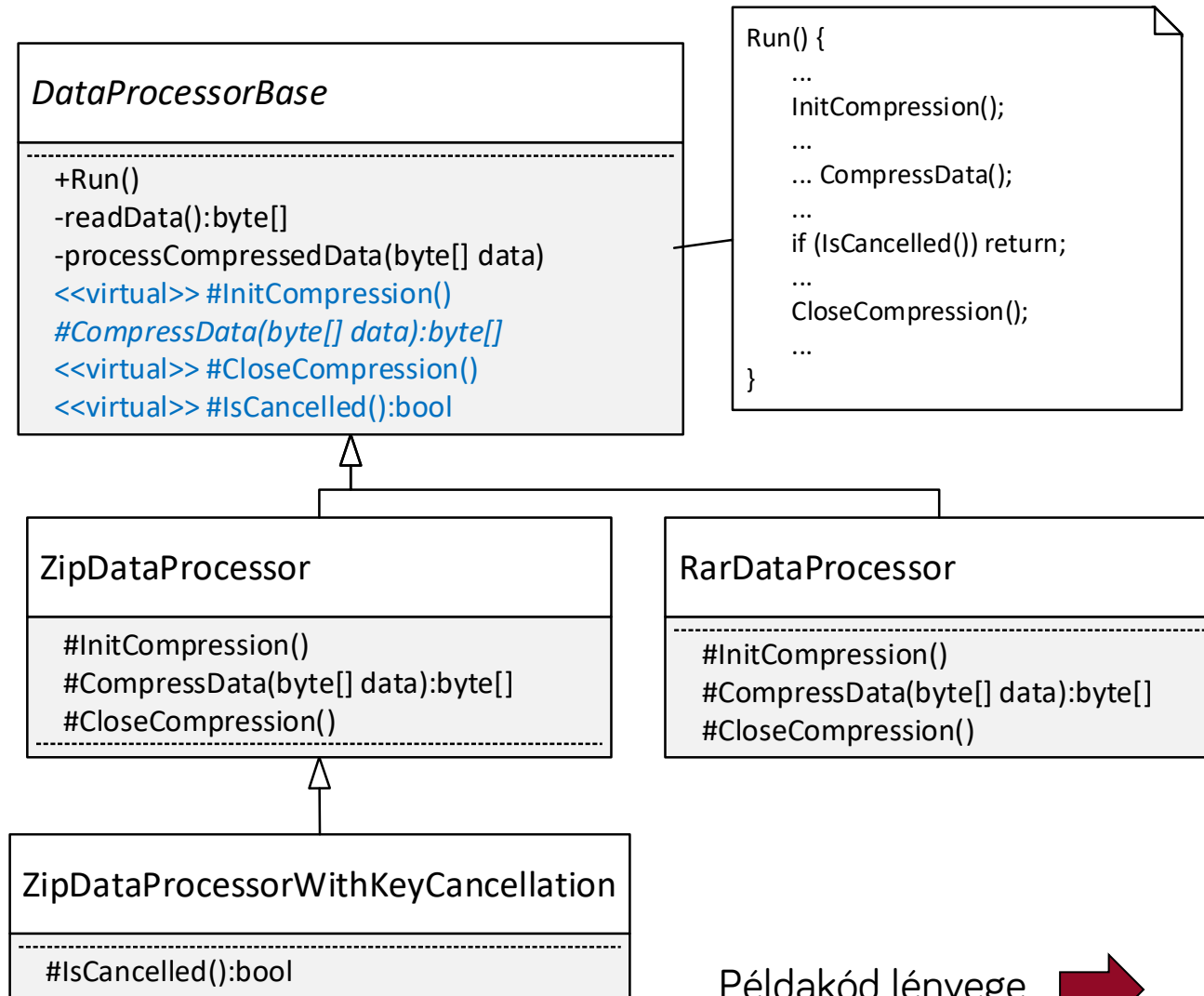
```
+Run()  
-readData():byte[]  
-processCompressedData(byte[] data)  
-initZip()  
-compressData(byte[] data):byte[]  
-closeZip  
-isCancelled
```

Template method minta alkalmazása

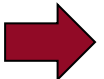
- A változatlan kódrészeket tegyük egy őss osztályba
- A változó/kiterjeszthető részeket nem drótozzuk az ős műveleteibe, hanem virtuális és/vagy absztrakt függvényekre bízunk
- A leszármazott osztályban ezen függvények felülírásával adjuk meg a specifikus viselkedést
 - > A példában:
 - Tömörítés módja
 - Cancel módja

Template Method példa áttekintése

- Lásd forráskód:
DesPattCode\
TemplateMethod
mappában fájlok
- **DataProcessorBase**
őssztály a lényeg, a
különböző
implementációkra
közös kódot
tartalmazza. Az
implementációfüggő
részeket
virtuális/absztrakt
függvényekre bízva
(InitCompression,
CompressData,
CloseCompression,
IsCancelled)



Példakód lényege



Template Method példa

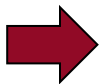
- DataProcessorBase őssosztály Run művelete a lényeg, a különböző implementációkra közös kódot tartalmazza. Az implementációfüggő részeket virtuális/absztrakt függvényekre bízva (InitCompression, CompressData, CloseCompression, IsCancelled)

```
...
public void Run()
{
    byte[] inputData;
    InitCompression(); // Tömörítés inicializás, impl. függő
    try
    {
        while ((inputData = readData()) != null)
        {
            // Tömörítés, impl. függő
            byte[] compressedData = CompressData(inputData);
            processCompressedData(compressedData);
            if (IsCancelled()) // Cancel vizsgálat, impl. függő
                return;
        }
    }
    finally
    {
        CloseCompression(); // Tömörítés lezárás, impl. függő
    }
}
...
```

Template Method célja

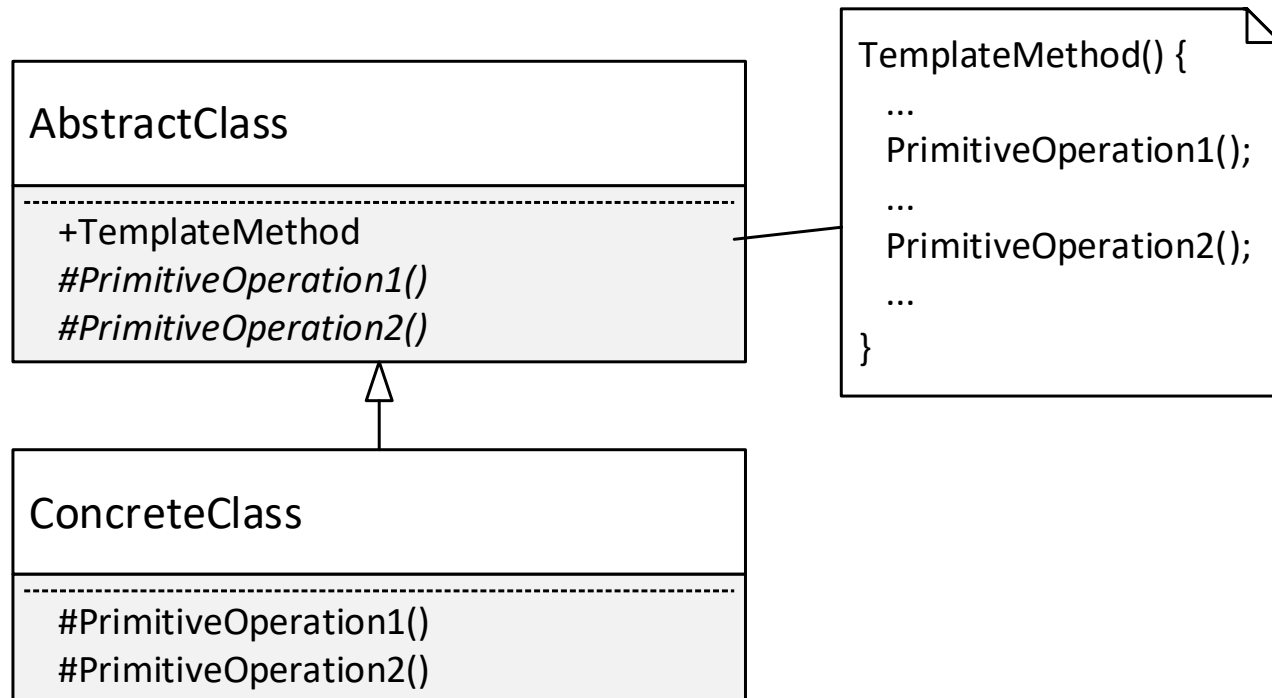
- Egy műveleten belül algoritmus vázat definiál, és az algoritmus bizonyos lépéseinek implementálását a leszármazott osztályra bízta.
 - > A példánkban az algoritmus váz a Run függvény volt, a leszármazottra bízott lépések az InitCompression, CompressData, CloseCompression, IsCancelled

Általánosítsuk



Template Method általánosan

- Amit eddig néztünk (DataProcessor), az csak egy **példa** volt a Template Method mintára. Maga a minta teljesen általános:



Template Method következmények

- Következmények

- > Lehetővé teszi, hogy az algoritmus/folyamat invariáns részeit egy helyen definiáljuk és a változó részeket a leszármazott osztályban adjuk meg.
- > Így megoldható a kód duplikálás elkerülése (DRY elv!): a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method-ban), mely a különböző viselkedést megvalósító egyéb műveleteket hívja meg. Ezeket a “különböző viselkedést megvalósító egyéb műveleteket” a leszármazott osztályban felül kell/lehet definiálni.
- > Lehetővé teszi kiterjesztési pontok definiálását a kódban (ún. hook függvényeknek is szokás nevezni)

- Megjegyzés

- > Keretrendszerek esetében gyakori

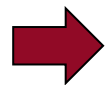
Kiértékelés

- Pozitívum
 - > Kódduplikáció elkerülése (lásd előző dia)
 - > **Új viselkedés könnyen bevezethető**, nem kell a meglévő kódot (lényegi helyen) változtatni
 - A példában DataProcessorBase kódjához nem kell hozzányúlni, csak le kell belőle származtatni egy új osztályt
- Negatívum
 - > Futás közben nem tudjuk egyszerűen cserélni viselkedést, rugalmatlan
 - A származtatási hierarchia fordításkor eldől
 - > Sok keresztkombinációra lehet szükség, lásd következő diák

Ha a viselkedésnek több aspektusa/dimenziója van

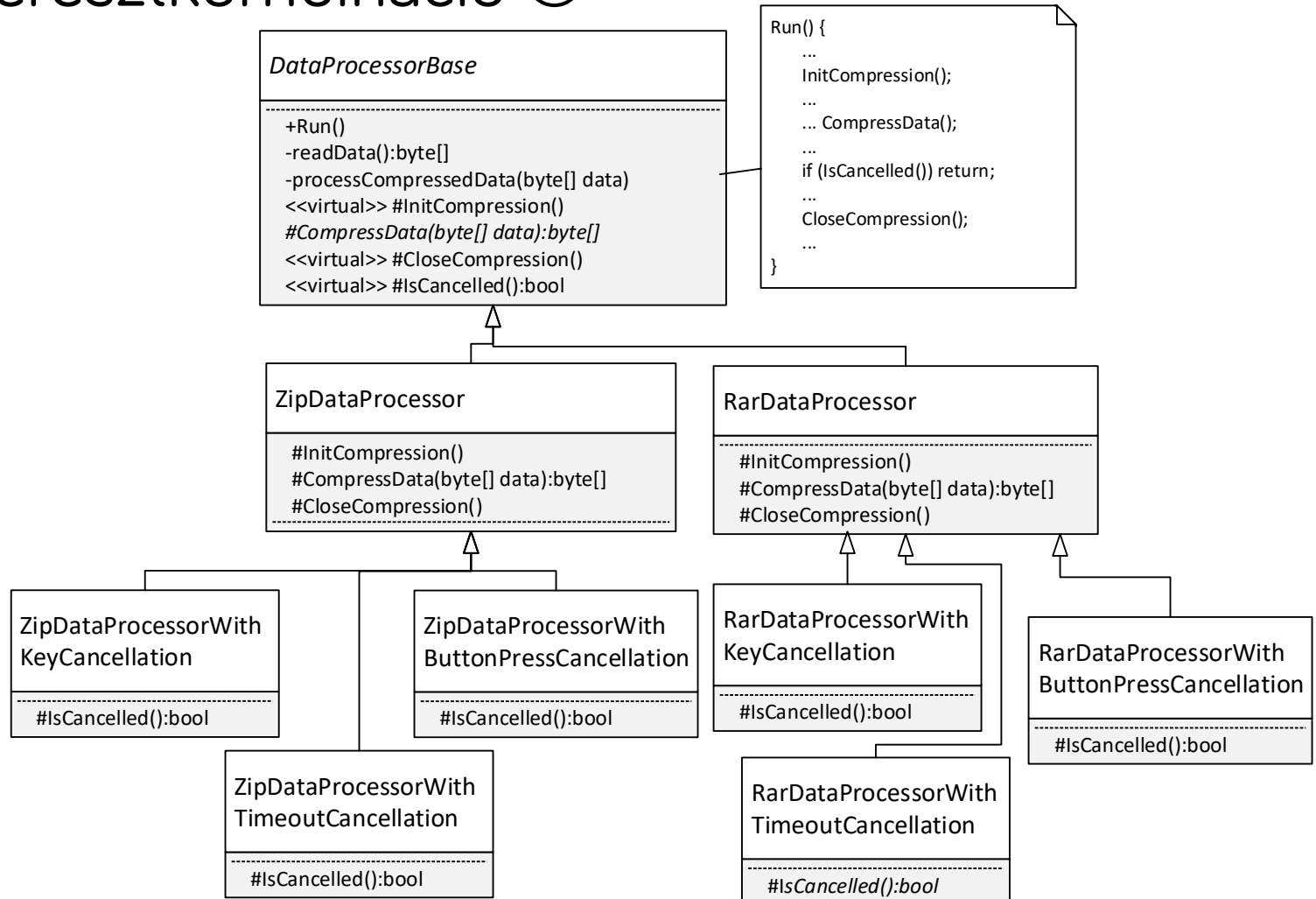
- DataProcessor példa aspektusok (kettő van)
 - > Tömörítés mód: Zip, Rar, 7Zip
 - > Cancel mód: billentyű lenyomás, timeout, gombkattintás (pl. UWP Cancel gomb)
- Minden használt kombinációnak egy külön leszármazott osztály kell
 - > Áttekinthetetlen és karbantarthatatlan hierarchiát eredményez, ha sok a kombináció

Példa



DataProcessor példa

Sok keresztkombináció ☹️



Sok keresztkombináció

A megoldást a Strategy minta alkalmazása jelenti
Rövidesen ...

Előfordulása

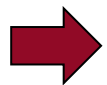
- Egyszerűbb esetekben gyakran használjuk
- Keretrendszeres esetén gyakori, pl. le kell származtatni a keretrendszer beépített Window, Page, Application, Control, stb. osztályából és virtuális függvényeket lehet/kell felüldefiniálni

Strategy (Stratégia)

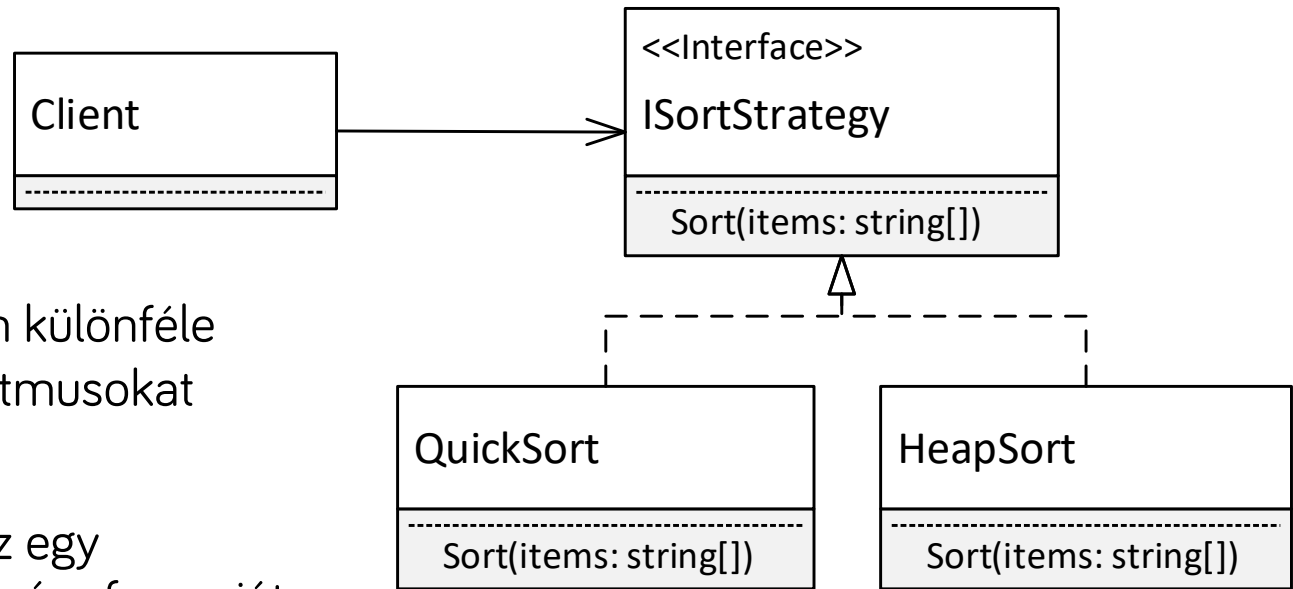
Strategy minta célja

- Célja az algoritmusok/viselkedések egy csoportján belül az egyes algoritmusok/viselkedések egységbe zárása és egymással kicserélhetővé tétele.
 - > A kliens szemszögéből az általa használt algoritmusok/viselkedések szabadon kicserélhetők.

Ez így nehezen érthető, nézzünk egy egyszerű példát



Strategy egyszerű példa - sorrendezés



- A kliens objektum különféle sorrendező algoritmusokat használhat
- A kliens tartalmaz egy ISortStrategy típusú referenciát egy konkrét (QuickSort v. HeapSort) objektumra.
- A referencia által hivatkozott implementációs objektumot könnyű kicserélni.

Példakód

- Kód, lásd „Strategy – Sort” mappa

```
// Strategy interfész
interface ISortStrategy
{
    void Sort(string[] items);
}

// QuickSort strategy implementáció
class QuickSort : ISortStrategy
{
    public void Sort(string[] items)
    {
        Console.WriteLine("Sorting items using quick sort algoritm ...");
    }
}

// HeapSort strategy implementáció
class HeapSort : ISortStrategy
{
    public void Sort(string[] items)
    {
        Console.WriteLine("Sorting items using heap sort algoritm ...");
    }
}
```

Példa folytatása

```
// Kliens, a Sort metódusa az aktuálisan beállított stratégiával rendez
class Client
{
    ISortStrategy sort; // Hivatkozás az aktuálisan beállított stratégia implementációra

    // Lehetőséget biztosít a stratégia beállítására
    public void SetSortStrategy(ISortStrategy sort)
    {
        this.sort = sort;
    }

    public void Sort(string[] items)
    {
        sort.Sort(items); // Sorrendezés az aktuális stratégiával
    }
}

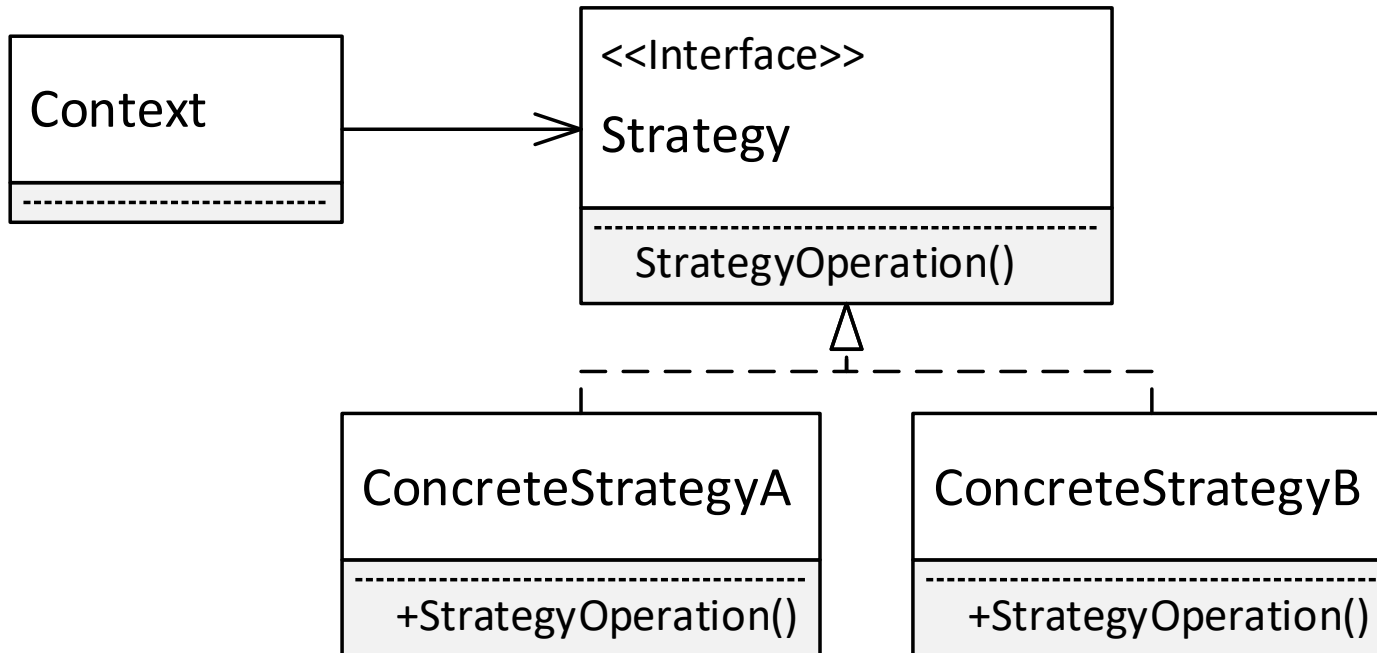
class Program
{
    static void Main(string[] args)
    {
        var items = new string[] { "körte", "alma", "szilva" };

        Client client = new Client();

        // Kezdetben használjuk a quick sort algoritmust
        client.SetSortStrategy(new QuickSort());
        client.Sort(items);

        // Mostantól a heap sort algoritmust használjuk
        client.SetSortStrategy(new HeapSort());
        client.Sort(items);
    }
}
```

Strategy - általános osztálydiagram



- **Strategy** interfész név mellett szokás a **Behavior** illetve **Policy** nevek használata is.
- Lehet több összetartozó művelet is (az ábrán csak egy szerepel)

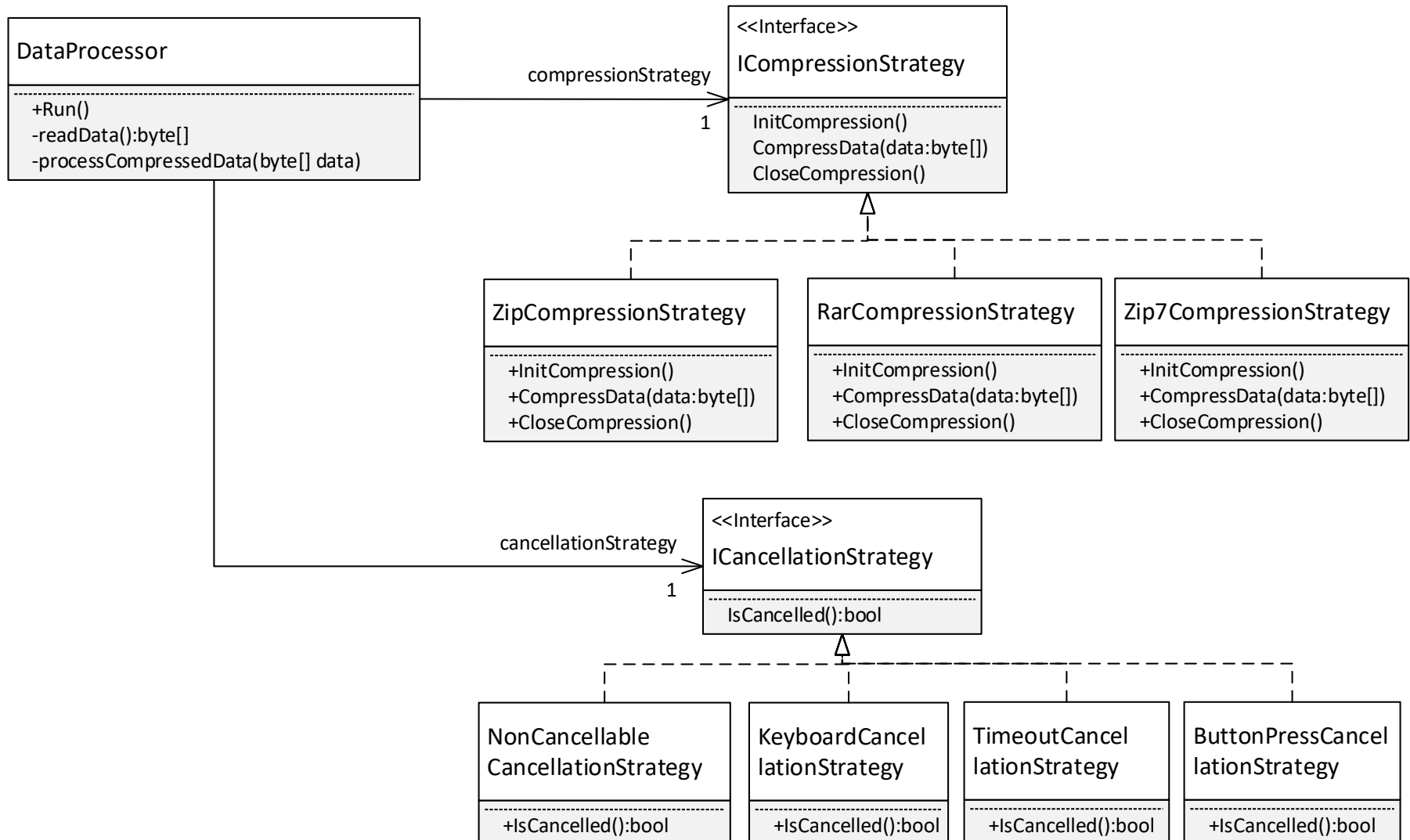
Strategy, DataProcessor példa

- Az osztály viselkedésének minden olyan aspektusára/dimenziójára, melyet lecserélhetővé/bővíthetővé szeretnénk tenni, külön strategy hierarchiát vezetünk be
 - > **Tömörítés mód variációk:** Zip, Rar, 7Zip
 - Interfész: ICompressionStrategy
 - Implementációk: ZipCompressionStrategy, RarCompressionStrategy, Zip7CompressionStrategy
 - > **Cancel mód variációk:** nem megszakítható, billentyű lenyomás, timeout, gombkattintás (pl. UWP Cancel gomb)
 - Interfész: ICancellationStrategy
 - Implementációk: NonCancellableCancellationStrategy, KeyboardCancellationStrategy, TimeoutCancellationStrategy, ButtonPressCancellationStrategy
- Az osztály külön (interfész típusú) hivatkozást tartalmaz minden viselkedés aspektusra/dimenzióra
 - > ICompressionStrategy tag
 - > ICancellationStrategy tag

Lássuk a gyakorlatban



DataProcessor osztálydiagram



DataProcessor példa, kód

- Kód, lásd DesPattCode\Strategy mappa
 - > Nincs minden osztály megvalósítva
 - > A DataProcessor konstruktorban kapja meg a függőségeit, egy ICompressionStrategy és ICancellationStrategy objektumot
 - Ha menet közben is meg akarjuk változtatni valamelyiket, egy megfelelő Setter függvényt vezessünk be (lásd korábbi sorrendező példa)

DataProcessor példa, kód

```
class DataProcessor
{
    ICompressionStrategy compressionStrategy; // Aktuális cancel stratégia/viselkedés
    ICancellationStrategy cancellationStrategy; // Aktuális tömörítési stratégia/viselkedés

    public DataProcessor(ICompressionStrategy compressionStrategy,
        ICancellationStrategy cancellationStrategy)
    {
        ...
        // Eltároljuk a paraméterként kapott stratégiákat
        this.compressionStrategy = compressionStrategy;
        this.cancellationStrategy = cancellationStrategy;
    }

    // Ciklusban beolvassa, tömöríti, majd feldolgozza a tömörített adatokat
    public void Run()
    {
        byte[] inputData;
        compressionStrategy.InitCompression(); // Tömörítés inicializás, impl. függő
        try
        {
            while ((inputData = readData()) != null)
            {
                // Tömörítés, impl. függő
                byte[] compressedData = compressionStrategy.CompressData(inputData);
                processCompressedData(compressedData);
                if (cancellationStrategy.IsCancelled()) // Cancel vizsgálat, impl. függő
                    return;
            }
        }
        finally
        {
            compressionStrategy.CloseCompression(); // Tömörítés lezárás, impl. függő
        }
    }
}
```

DataProcessor példa, kód

- A stratégiák bármilyen kombinációban egyszerűen használhatók!

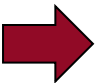
```
static void Main(string[] args)
{
    // A stratégiák bármilyen kombinációban egyszerűen használhatók. Nézzünk párat!
    // Jelen esetben a Zip algoritmust választjuk, és billentyű alapú Cancel lehetőséget
    var processor = new DataProcessor(
        new ZipCompressionStrategy(),
        new KeyboardCancellationStrategy(ConsoleKey.X)
    );
    processor.Run();

    // Második esetben a Rar algoritmust választjuk, és nem akarunk Cancel lehetőséget
    // biztosítani
    var processor2 = new DataProcessor(
        new RarCompressionStrategy(),
        new NonCancellableCancellationStrategy()
    );
    processor2.Run();
}
```

Kiértékelés - pozitívumok

- Bővíthetőség
 - > Új viselkedés könnyen bevezethető, nem kell a meglévő kódot (lényegi helyen) változtatni
 - A példában a meglévő DataProcessor kódjához nem kell hozzányúlni, csak új ICompressionStrategy vagy ICancellationStrategy implementációt kell bevezetni
 - > Több aspektus/dimenzió esetén nincs kombinatorikus robbanás az osztályhierarchiában
- Unit tesztelhetőséget segíti

Példa



Kiértékelés – unit tesztelhetőség

- Unit tesztek során az osztályt önmagában, a függőségei nélkül teszteljük (ezért is hívják UNIT tesztnek).
- Példa: a DataProcessor tesztelése során nem kívánjuk tesztelni azt, hogy az adatok tömörítése, vagy a cancel során a billentyű lenyomás hogyan történik.
 - > Pl.
 - ha hiba van a zip algoritmusban, és az elesik, akkor az **ne akassza meg** a DataProcessor unit tesztjeit
 - A tömörítés lassú, **ne lassítsa** ez feleslegesen a DataProcessor unit tesztjeit
 - > A tömörítés és cancel unit tesztelését más, ezekre dedikált unit tesztek kell végezzék.
 - > Ha ezek implementációja be van építve a DataProcessor osztályba, akkor a DataProcessor nem unit tesztelhető

Kiértékelés – unit tesztelhetőség

- A Strategy minta segíti, hogy a DataProcessor unit tesztelhető legyen. Hogyan? →
- Amikor a DataProcessor osztályt unit teszteljük, olyan „dummy” ICompressionStrategy és ICancellationStrategy implementációkat adunk át, melyek nem csinálnak semmit, vagy magát a tesztet segítik
 - Pl. DummyCompressionStrategy, mely nem tömörít, csak visszaadja az eredeti adatot
 - Pl. NonCancellableCancellationStrategy, mely nem Cancel-el, illetve egy AlwaysCancelCancellationStrategy, mely azonnal automatikusan cancel-el (attól függően, hogy mit tesztelünk éppen).
 - Konkrét példa a félév végén tesztelésről szóló gyakorlaton (?)

DataProcessor példa, SRP elv

- A DataProcessor legutolsó megvalósítása valóban teljesíti az SRP elvet, az osztálynak valóban egyetlen felelőssége van?
- Nem, még mindig több felelőssége van!
 - > A Run-ban a folyamat vezérlése
 - > Adatbeolvasás logikája (readData művelet)
 - > Adatfeldolgozás logikája (processCompressedData művelet)
- Az adatbeolvasás és adatfeldolgozás logikáit is ki lehetne faktorálni az osztályból, a compression és a cancel mintájára interfészek mögé lehet tenni
 - > Ez segíteni a unit tesztelhetőséget, és a bővíthetőséget
 - > Gyakorlásképpen megcsinálható

Strategy

- **Program to an interface** (and not to an implementation) elv egyik megtestesülése

Kiterjeszthetőség összefoglaló

Áttekintés

- SOLID Open/Closed elv
- Lehetőségek
 - > Template method
 - Leszármazottra bízva a kiterjesztést
 - Egyszerűbb
 - Kevésbé rugalmas
 - A leszármaztatás miatt (a leszármazási hierarchia fordításkor dől el, futás közben nem változtatható)
 - Több aspektus: áttekinthetetlen, sok keresztkombináció
 - > Strategy
 - Más osztályokra bízva (delegál), ez a legrugalmasabb
 - > Van más lehetőség is → metódusreferencia, lambda használata

Egyéb kiterjeszthetőségi technikák

- Metódusreferenciák (delegate-ek), lambda kifejezések használata
 - > Egyre inkább terjed
 - > C#, C++, JavaScript nyelvek is támogatják
 - > Java-ban „csak” lambda kifejezések
- Mostantól C#-ot nézünk
 - > Azt a kódot, amely a kiterjeszthetőséget szolgálja, egy függvény/delegate vagy lambda formájában adjuk át
 - > A részletek "Eseményvezérelt és vizuális programozás,, tárgyból

Metódusreferencia, lambda használata

- Lásd DesPattCode\Delegate példa

```
class DataProcessor
{
    // ...

    Func<bool> onCancelled; // Metódusreferencia, konstruktorban kapja meg, Run-ban hívjuk

    public DataProcessor(Func<bool> onCancelled)
    {
        this.onCancelled = onCancelled;
    }

    // Ciklusban beolvassa, tömöríti, majd feldolgozza a tömörített adatokat
    public void Run()
    {
        ...
        if (onCancelled != null && onCancelled()) // Meghívjuk a kódot
            return;
        ...
    }
}

...
// Konstruktorban adjuk át a DataProcessor által hívandó kódot, pl. lambda formájában
var processor1 = new DataProcessor(() => false);
```

Kiterjeszthetőség, módosíthatóság

- Számos más minta is (gyakorlatilag valamennyi) a könnyű kiterjeszthetőséget, módosíthatóságot szolgálja, csak kevésbé direkt módon!