Objektum orientált programozás

# Fogalom

Programozási paradigma ami az objektumok koncepcióján alapul, amik adatot és kódot tartalmaznak. Az adat mezők (field) formájában van jelen (úgy is ismert, mint attribútum, property vagy adattag). A kód procedúrák formájában van jelen (úgy is ismert, mint metódus vagy tagfüggvény).

Az objektumok gyakori eleme, hogy hozzájuk kapcsolt metódusaik vannak, és ezek elérik és módosítani tudják az objektumok adattagjait. Általában van kulcsszó a saját magukra hivatkozásra is (this vagy self).   
Ebben a paradigmában a programot úgy tervezzük meg, mint egymással interakcióban lévő objektumokból álljon.  
Sokféle OOP (object oriented programming) nyelv létezik, a legnépszerűbbek osztály alapúak, ami azt jelenti, hogy az objektumok osztályok (class) példányai (instance), ami meghatározza a típusukat is (type).

Sok széles körben használt programozási nyelv (pl C++, Java vagy Python) több paradigmát is használ és ezek egyike az OOP valamilyen mélységben.

Az OOP lehetővé teszi a programozók számára, hogy az adatokat és a hozzájuk kapcsolódó függvényeket egy logikai egységbe zárják (osztály), amelyet példányosíthatnak, és így könnyen új objektumokat hozhatnak létre az adatok és függvények különböző kombinációival.

Az OOP segít a nagyobb programok fejlesztésében, mivel a moduláris kialakítás segít az összetett rendszerek átláthatóságának és karbantarthatóságának javításában. Az OOP paradigma fejlődésével együtt megjelentek a tervezési minták, amelyek az objektumorientált tervezési problémákra kidolgozott útmutatók és bevált gyakorlatok.

Az OOP paradigma továbbra is fontos szerepet játszik a modern szoftvertervezési és -fejlesztési folyamatokban, és az objektumorientált programozási módszerek számos területen alkalmazhatók, például a számítógépes grafikában, az adatbázis-kezelésben és az üzleti alkalmazások fejlesztésében.

Használatos a kliens kód (client code) fogalom, ami az objektum, osztály, vagy általában kód felhasználóit jelenti. Másik osztály, user vagy teszt kód például.

# Tulajdonságok

OOP programozás objektumokat használ, de egy OOP kompatibilis nyelv általában nem támogat minden OOP technikát és struktúrát.

### Nem OOP

* Változók: Ezek tudnak információt tárolni kisszámú beépített adattípusokban, mint integer vagy karakter. Ezek tartalmazhatnak adatszerkezeteket, mint string, lista, hash tábla, amik vagy beépítettek, vagy memória mutatókkal (pointer) vannak megvalósítva
* Procedúra: function, metódus, routine, subroutine – ezek fogadnak inputot, hoznak létre outputot és adatot manipulálnak. Használhatnak strukturált programozás elemeket, mint ciklusok és elágazások.

A moduláris programozási támogatás lehetővé teszi a függvények csoportosítását fájlokba és modulokba szervezett formában. A modulok névtérrel vannak ellátva, így az azonos nevű azonosítók egy modulban nem ütköznek egy másik fájlban vagy modulban lévő függvénnyel vagy változóval.

### Objektumok és osztályok

* Osztály: Az adatformátumnak és a rendelkezésre álló szolgáltatásoknak a definíciója. Opcionálisan tartalmazhat adattagokat és függvényeket (ezek az osztály függvények és adattagok).
* Objektum: A példányok. Ezek tartalmazzák az adattagokat és a függvényeket.

Az objektumok gyakran megfelelnek a valós világban található dolgoknak. Például egy grafikus programban lehet "kör", "négyzet", "menü" objektum. Egy online vásárlási rendszernek lehetnek "kosár", "vásárló" és "termék" objektumai. Néha az objektumok absztraktabb entitásokat képviselnek, például egy olyan objektumot, amely egy nyitott fájlt jelképez, vagy egy olyan objektumot, amely lehetővé teszi az amerikai hagyományos mértékegységekből a metrikus rendszerbe való átváltást.

Minden objektum egy adott osztály példánya. Az objektumorientált programozásban a függvényeket metódusoknak nevezik, míg a változókat mezőknek, tagoknak, attribútumoknak vagy tulajdonságoknak. Ebből a következő fogalmak származnak:

* Osztályváltozók (class variable) - az osztályhoz tartoznak; minden változóból csak egy példány létezik, amely az összes osztálypéldánnyal meg van osztva
* Példánymezők (instance variable) - az egyes objektumokhoz tartozó adatok; ezekből minden objektumnak van saját példánya
* Tagváltozók (member variable) – az osztály és objektum változóira is vonatkozik
* Osztálymetódusok (class method) - az osztályhoz tartoznak és csak az osztályváltozókhoz és a függvényhívások bemeneteihez férnek hozzá (példánymezőkhöz és példánymetódusokhoz nem)
* Példánymetódusok (instance method) - az egyes objektumokhoz tartoznak, és hozzáférnek a kérdéses objektum példánymezőihez, bemenetekhez és osztályváltozókhoz

Az objektumokat komplex belső szerkezetű változókhoz hasonlóan is el lehet érni. Sok programozási nyelvben effektíve mutatóként működnek az adott objektum, a heap vagy a stack memóriaterületén tárolt egyetlen példányára. Az objektumok absztrakciós réteget nyújtanak, amelyet használni lehet a belső és a külső kód elkülönítésére. Külső kód használhat egy objektumot úgy, hogy meghívhatja egy adott függvényét bizonyos bemeneti paraméterekkel, olvashatja egy példányváltozóját, vagy írhat bele. Az objektumokat létrehozhatjuk azzal, hogy meghívunk egy olyan konstruktor függvényt, amelyet az osztályban külön meg kell határozni. Egy program futása során sok példányt hozhatunk létre ugyanabban az osztályban, amelyek függetlenül működnek egymástól. Ez könnyű módja annak, hogy ugyanazokat az eljárásokat használjuk különböző adathalmazokon.

### Adat absztrakció

Az adatok absztrakciója egy tervezési minta, amely szerint az adatok csak a szemantikailag összefüggő funkciók számára láthatóak, hogy elkerüljék az esetleges visszaéléseket. Az adatok absztrakciójának sikeressége gyakran magával vonja az adatok rejtését mint tervezési elvet az objektumorientált és a tiszta funkcionális programozásban.

Ha egy osztály nem engedi a hívó kódnak, hogy hozzáférjen az objektum belső adataihoz, és csak a metódusokon keresztül lehet hozzáférni, ez az információ elrejtés egy formája, amelyet absztrakciónak neveznek. Néhány nyelv (például a Java) lehetővé teszi az osztályok számára, hogy kifejezetten korlátozzák a hozzáférést, például a belső adatokat a privát kulcsszóval jelölve, és a kódon kívüli osztályok által használt metódusokat a publikus kulcsszóval jelölve. A metódusok lehetnek publikusak, privátak vagy köztes szintek, mint például a védett (protected) (amely lehetővé teszi a hozzáférést ugyanabban az osztályban és annak leszármazottaiban, de nem más osztályok objektumaiban). Más nyelvekben (mint például a Python) ez csak konvenció szerint van érvényben (például a privát metódusoknak az aláhúzásjellel kezdődő nevük lehet). A C#, Swift és Kotlin nyelvekben az internal kulcsszó csak az adott osztályhoz tartozó fájlokhoz engedi a hozzáférést az assembly-ben, csomagban vagy modulban.

### Encapsulation

Az encapsulation megakadályozza, hogy az objektum belső működésére külső kód rálásson, vagy foglalkoznia kelljen vele. Ez megkönnyíti a kód átszervezését (refactoring), például lehetővé teszi az osztály készítőjének, hogy megváltoztassa, hogyan ábrázolják (data representation) belsőleg az osztály objektumai az adataikat, anélkül, hogy megváltoztatnák a kódot, ami az osztályt használja (ergo a "public" metódus hívások ugyanúgy működnek a változtatás után is). Ez arra is ösztönzi a programozókat, hogy az adott adathalmazzal kapcsolatos összes kódot ugyanabba az osztályba helyezzék, ami más programozók számára könnyen érthetővé teszi a kódot. Az encapsulation egy olyan technika, amely ösztönzi a decouplingot.

### Kompozíció és öröklés

Az objektumok tartalmazhatnak más objektumokat az instanciaváltozóikban; ezt objektumkompozíciónak nevezik. Például az Employee osztály egy Address osztály típusú objektumot tartalmazhat (vagy közvetlenül, vagy egy mutatón (pointer) keresztül), saját példányváltozói mellett, mint például a "first\_name" és a "position". Az objektumkompozíció arra szolgál, hogy "has-a" kapcsolatokat ábrázol: minden alkalmazottnak van egy címe, így minden Employee objektum módot kap arra, hogy tároljon egy Address objektumot (vagy közvetlenül beágyazva önmagába, vagy egy pointerrel hivatkozva egy külön helyre (memóriacím)).

Az osztályokat támogató nyelvek majdnem mindig támogatják az öröklődést (inheritance). Ez lehetővé teszi az osztályok hierarchiájának kialakítását, amely ábrázolja az "is-a(-type-of)" kapcsolatokat. Például az Employee osztály örökölhet a Person osztályból. Az összes adat és metódus, amelyek a szülőosztályban (parent class) elérhetők, ugyanazokkal a nevekkel megjelennek a gyermekosztályban (child class) is. Például a Person osztály meghatározhatja a "first\_name" és "last\_name" változókat és a "make\_full\_name()" metódust. Ezek is elérhetők lesznek az Employee osztályban, amely hozzáadhatja a "position" és a "salary" változókat. Ez a technika lehetővé teszi az azonos eljárások és adatdefiníciók egyszerű újrafelhasználását, és intuitív módon tükrözheti a valóságban jelen lévő kapcsolatokat. A fejlesztő így az alkalmazási területükből származó objektumokat használja fel, és nem az adatbázistáblákat és alprogramokat.

Az alosztályok (child class) felülírhatják (override) a szülőosztály által meghatározott metódusokat. Néhány nyelvben engedélyezik a többszörös öröklődést (multiple inheritence), bár ez megnehezítheti az override-ok feloldását.

Abstract osztályok nem hozhatnak létre példányokat, csak más "konkrét" osztályokba történő öröklés céljából léteznek, amelyek példányosíthatók. A Java-ban a "final" kulcsszó használható arra, hogy megakadályozzuk egy osztály származtatását.

### Többalakúság

A hívó kód független az alkalmazott öröklési hierarchiában szereplő osztálytól, legyen az az ősosztály vagy valamelyik leszármazottja. A lényege, hogy azonos nevű műveletek az öröklési hierarchiában szereplő különböző objektumok esetében eltérően működhetnek.

Például, az Alakzat osztályból származik a Kör és a Négyzet típusú objektum is. Mindkét típusú alakzatnak van egy rajzolás függvénye, amely implementálja a saját maga kirajzolásához szükséges műveleteket. Eközben a hívó kód számára mindegy, hogy éppen melyik típusú alakzat kerül rajzolásra.

Ez egy másik absztrakciós szint, amely egyszerűsíti az osztály hierarchián kívüli kódot, lehetővé téve a felelősségi körök erős szétválasztását.

### Dynamic dispatch

A futásidejű kiválasztási folyamatot jelenti, amely során az adott többalakú (polymorph) művelet (metódus) implementációjának kiválasztása történik. Ez gyakran alkalmazott és fontos jellemzője az objektumorientált programozási (OOP) nyelveknek és rendszereknek.

### Nyílt rekurzió

Azon nyelvekben, amelyek támogatják az open recursion-t, az objektum metódusai hívhatják a másik metódusokat ugyanazon az objektumon belül (beleértve önmagukat is), általában egy különleges változó vagy kulcsszó, a this vagy a self használatával. Ez a változó late-bound, ami lehetővé teszi egy osztályban definiált metódusnak, hogy meghívjon egy másikat, amely később, az egyik gyerek osztályban van definiálva.

# Solid elvek

### Single responsibility

Egyetlen felelősség elve

Egy osztály vagy modul csak egyetlen felelősséggel rendelkezzen (azaz: egy oka legyen a változásra).

Azaz minden elkülönítve is értelmezhető funkció illetve felelősségi kör kezelése kerüljön külön osztályba vagy modulba.

Az a lényege, hogy minden osztálynak egyetlen felelősséget kell lefednie, de azt teljes mértékig. Amennyiben egy osztály nem fedi le teljesen a saját felelősségi körét, akkor muszáj lesz implementációra programozni (GOF1 alapelv), hogy egy másik osztály megvalósítsa azokat a szolgáltatásokat, amik kimaradtak az osztályból. Amikor egy osztály több felelősségi kört is ellát, akkor sokkal jobban ki van téve a változásoknak, mintha csak egy felelősséget látna el.

### Open/closed

Nyílt/zárt elv

Egy osztály vagy modul legyen nyílt a kiterjesztésre, de zárt a módosításra.

Azaz a funkcionalitás bővítéséhez ne legyen szükség az osztály forráskódjának módosítására, ehelyett legyen mód leszármazással vagy kompozíción keresztül kiterjeszteni a meglévő implementációt.

Ez az jelenti, hogy új alosztályt vagy egy új metódust nyugodtan felvehetünk, de meglévőt nem írhatunk felül. Ennek azért van értelme, mert ha már van egy működő, letesztelt, kiforrott metódusunk, és azt megváltoztatjuk, akkor több hátrányos dolog is történhet: a változás miatt az eddig működő ágak hibássá válhatnak, illetve a változás miatt a tőle implementációs függőségben lévő kódrészek megváltoztatására is szükség lehet.

### Liskov substitution

Liskov helyettesítési elv

Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.

Azaz garantáljuk, hogy egy adott (akár absztrakt) típussal átvett objektum teljesíti, amit a típustól elvárunk. Azaz típusörökléskor a leszármazottnak továbbra is megegyező módon támogatnia kell a felmenői által biztosított funkciókat. Árnyalja a képet, hogy a Java-világban elterjedt az opcionálisan implementálható metódusok használata, alapvetően az API egyszerűségének megőrzése érdekében. Ilyenkor az interfész technikailag ugyan tartalmazza az adott metódust, ám az implementációknak nem kötelező támogatniuk.

### Interface segregation

Interfész elválasztási elv

Az interfészek szétválasztásának elve: egyetlen kliens se legyen rákényszerítve arra, hogy olyan eljárásoktól függjön, amelyeket nem is használ.

Ez a gyakorlatban általában azt jelenti, hogy egy átvett paraméter típusa lehetőleg arra legyen specializálva, amilyen művelet(ek)et ténylegesen elvégzünk rajta. Az ilyen célra kialakított legáltalánosabb interfészek gyakran tartalmazzák az „-able” végződést (pl. Comparable, Iterable stb.).

Az objektumorientált tervezésben az interfészek absztrakciós rétegeket biztosítanak, amelyek egyszerűsítik a kódot, és indirektté teszik a kapcsolatot az implementációkkal, függőségekkel.

A [kiáltványt](http://manifesto.softwarecraftsmanship.org/) aláíró számos szoftverszakértő szerint a jól felépített és érthetően megírt szoftverkód szinte ugyanolyan fontos, mint maga a megfelelően működő szoftver. Az interfészek általában jól használhatók arra, hogy érthetővé tegyék a szoftver működési elvét.

Ha egy rendszer nagyon sok belső összekapcsolódást tartalmaz, akkor egy kisebb módosítás is számos további változtatást tesz szükségessé a rendszer más helyein. Interfészek intenzív használatával ez elkerülhető.

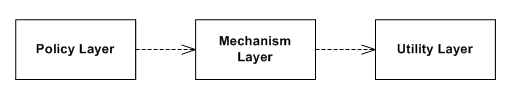
### Dependency inversion

Függőség megfordítási elv

A magas szintű modulok ne függjenek az alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.

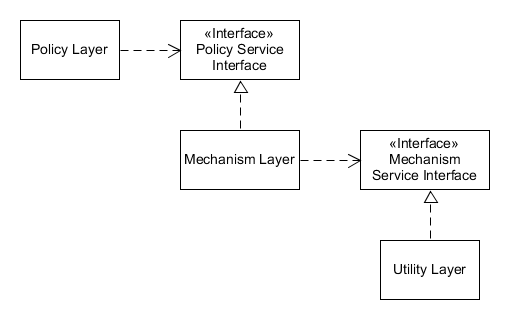
Tehát az alacsonyabb szint legyen egy API megvalósítása, a magasabb szint pedig ezen az API-n keresztül használja.

A hagyományos alkalmazásszerkezetben az alacsony szintű komponenseket arra tervezik, hogy magasabb szintű komponensek használják őket. A magas szintű komponensek közvetlenül függenek az alacsonyabb szintűektől. Ez a függés korlátozza a magasabb szintű komponensek újrafelhasználását.



A függőség megfordításának az a célja, hogy meglazítsa ezt a szoros csatolást egy absztrakt réteg közbeiktatásával, amivel javítja a magasabb szintű kód újrafelhasználhatóságát és tesztelhetőségét.

Az absztrakt réteg közbeiktatásával gyengül a felsőbb szint függése az alacsonyabb szintűtől. A megfordítás szó nem arra utal, hogy most az alacsony szint függ a magas szinttől. Mindkét réteg az absztrakciótól függ.



Közvetlen alkalmazás esetén az absztrakciókat a felsőbb rétegek birtokolják. Ebben a szerkezetben a felsőbb rétegek és az absztrakciók egy csomagban vannak. Az alacsonyabb szintű rétegek az absztrakciókból örökölnek, vagy azokat valósítják meg.

A függőség megfordítása és a tulajdonosi szerkezet támogatja az újrafelhasználhatóságot. A felsőbb rétegek így más megvalósításokat is tudnak használni. Ekkor néha adapterre (lásd Illesztő programtervezési minta) is szükség van.

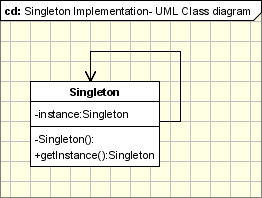
# OOP design patterns

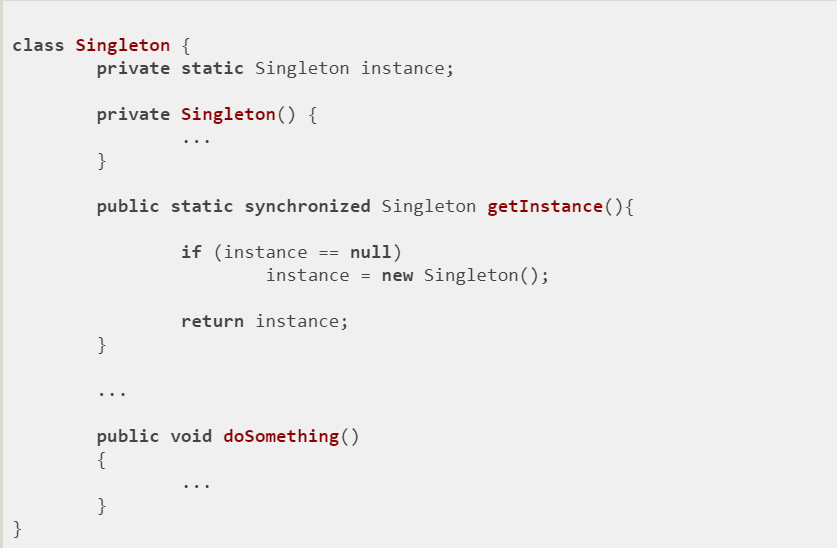
<https://www.oodesign.com/>

## Singleton

A Singleton tervezési minta egy olyan objektumorientált programozási minta, amely biztosítja, hogy egy adott osztályból csak egyetlen példány létezzen, és hogy az példányhoz egységes hozzáférést lehessen biztosítani az alkalmazás egész területén. A Singleton minta célja az, hogy gondoskodjon a globális adatok helyettesítéséről egy olyan megoldással, amely biztosítja az adatok összetettségének csökkentését, és megkönnyíti a kódban való használatukat.

Az osztály egyszeri példányának létrehozását általában a privát konstruktorral végzik, és az osztály által visszaadott objektum példányosítását az osztályon belüli statikus metódusok végzik. A Singleton osztály példánya globálisan elérhető, és az összes kódhoz hozzáférhet az alkalmazás területén. Ha az osztály példánya egyszer létrejött, az újabb példányokat már nem hozzák létre, és az eredeti példányt adják vissza minden alkalommal, amikor az osztály példányosítására van szükség. A Singleton osztály példánya gyakran adatkapcsolatok, naplózás vagy más erőforrások kezelésére használatos, amelyek csak egyetlen példánnyal rendelkezhetnek az alkalmazásban.





## Factory

A Factory tervezési minta olyan objektumorientált programozási technika, amely segít a kódban egy vagy több objektum létrehozásának elvégzésében, anélkül hogy pontosan meg kellene határoznunk, hogy melyik objektumra van szükségünk, vagy hogyan kellene azt létrehozni. A Factory tervezési minta gyakran használható olyan helyzetekben, amikor sok különböző objektumot kell létrehozni, amelyek mindegyike hasonló módon viselkedik, de különböző paraméterekkel rendelkezik.

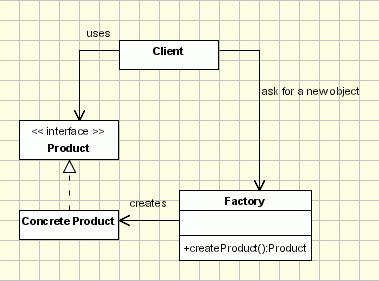
A Factory minta azt javasolja, hogy helyezzük át az objektum létrehozását a kliensoldalról az osztályokra (factory), így lehetővé téve, hogy ők kezeljék az objektum létrehozását. Ezzel csökkenthetjük a kódban az ismétlődő kódrészleteket, és lehetővé teszi a kód könnyebb karbantartását és bővítését. A Factory minta lehetővé teszi a kód könnyebb változtatását is, mivel az objektumok létrehozása kizárólag a factory osztályokban történik, és a klienseknek nem kell törődniük az objektumok konkrét típusával vagy létrehozásának módjával.

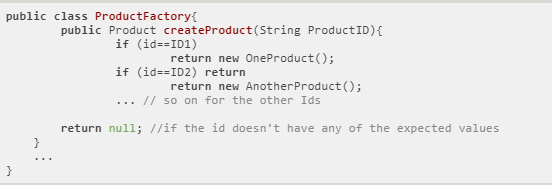
A Factory minta általában egy absztrakt osztályt vagy egy interfészt használ, amelynek konkrét implementációi az egyes objektumtípusokat képviselik. A Factory osztály feladata az, hogy a klienseknek egy közös interfész révén visszaadja az objektumokat. Az objektumok létrehozása történhet konkrét objektumok létrehozásával, vagy egy már meglévő objektum készletből való visszaadásával.

Például, ha egy játékban meg kell terveznünk egy karakter osztályt, amelynek különböző típusai lehetnek, mint például harcos, varázsló vagy íjász, akkor a Factory mintát használhatjuk annak érdekében, hogy egyszerűbbé és dinamikusabbá tegyük az osztályok létrehozását. A Factory osztály az általunk kiválasztott típusnak megfelelő karaktert hoz létre, amelyet a játék futása közben használhatunk.

Lényegében:

* Lérehozhatunk objektumokat anélkül, hogy a létrehozás logikáját megmutatnánk a kliensnek
* Az újonnan létrehozott objektumokat az interface-en keresztül lehet elérni







## Command

A Command tervezési minta segítségével egy parancsot lehet elkülöníteni a kódtól, amely azt használja, és lehetővé teszi annak kezelését, mint egy objektumot. Ez lehetővé teszi a parancsok dinamikus konfigurálását, paraméterezését, történetének nyilvántartását vagy visszavonását.

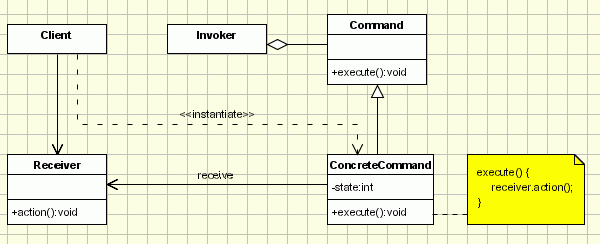
Az alapvető felépítése a mintának a következő:

* Command: Az interfész definiálja a parancsok általános viselkedését, amelynek megvalósítására a konkrét parancsok támaszkodnak. Tartalmazhat egy execute() metódust, amely végrehajtja a parancsot.
* ConcreteCommand: Az interfész implementálja a Command interfészt, és magában foglalja a végrehajtandó specifikus parancsot. Például egy SwitchOn parancs megvalósítása a fénykapcsoló bekapcsolására szolgáló metódus hívást tartalmazhat.
* Invoker: Ez az osztály kéri a parancsot a ConcreteCommand-tól a Command interfész segítségével. A kliens oldali kód ezt az osztályt hívja meg a parancsok végrehajtásához.
* Receiver: Az objektum, amelyen a ConcreteCommand a parancsot végrehajtja. Például a fénykapcsoló be- és kikapcsoló objektum lehet a Receiver.

A folyamat a következő:

1. A kliens oldali kód létrehoz egy parancsot, például egy SwitchOn parancsot.
2. Az Invoker osztály a Command interfész segítségével elkéri a parancsot a ConcreteCommand-tól.
3. A ConcreteCommand a Receiver objektumon keresztül végrehajtja a parancsot.

Ez a minta lehetővé teszi, hogy a parancsokat dinamikusan konfigurálják és cseréljék a kliens oldali kód változtatása nélkül. Továbbá lehetővé teszi a parancsok paraméterezését, módosítását, történetének nyilvántartását és visszavonását.



## Iterator

Az Iterator oop minta azt a célt szolgálja, hogy lehetővé tegye a kollekciók elemzését (pl. listák, halmazok, fájlok), anélkül, hogy a kódban feltétlenül ismernünk kellene a kollekció belső szerkezetét.

Az Iterator minta három fő elemből áll: az Iterator interfészből, amely meghatározza a használandó metódusokat; az Iterator osztályból, amely az interfészt implementálja és a kollekció elemeit adja vissza; és a Client osztályból, amely a kollekció elemeit használja.

Az Iterator interfész tartalmazza a következő metódusokat:

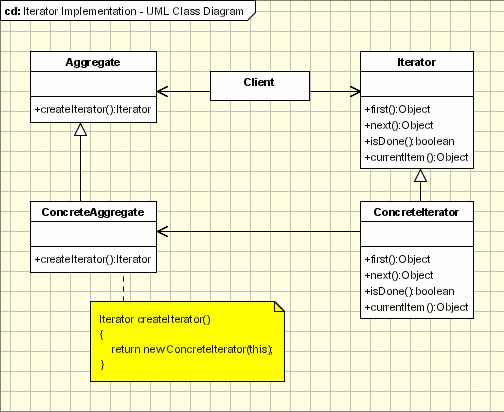
* next(): visszaadja a következő elemet a kollekcióból.
* hasNext(): megadja, hogy van-e még következő elem a kollekcióban. (vagy isDone())
* remove(): eltávolítja az aktuális elemet a kollekcióból.
* currentItem(): visszaadja az altuális elemet a kollekcióból.

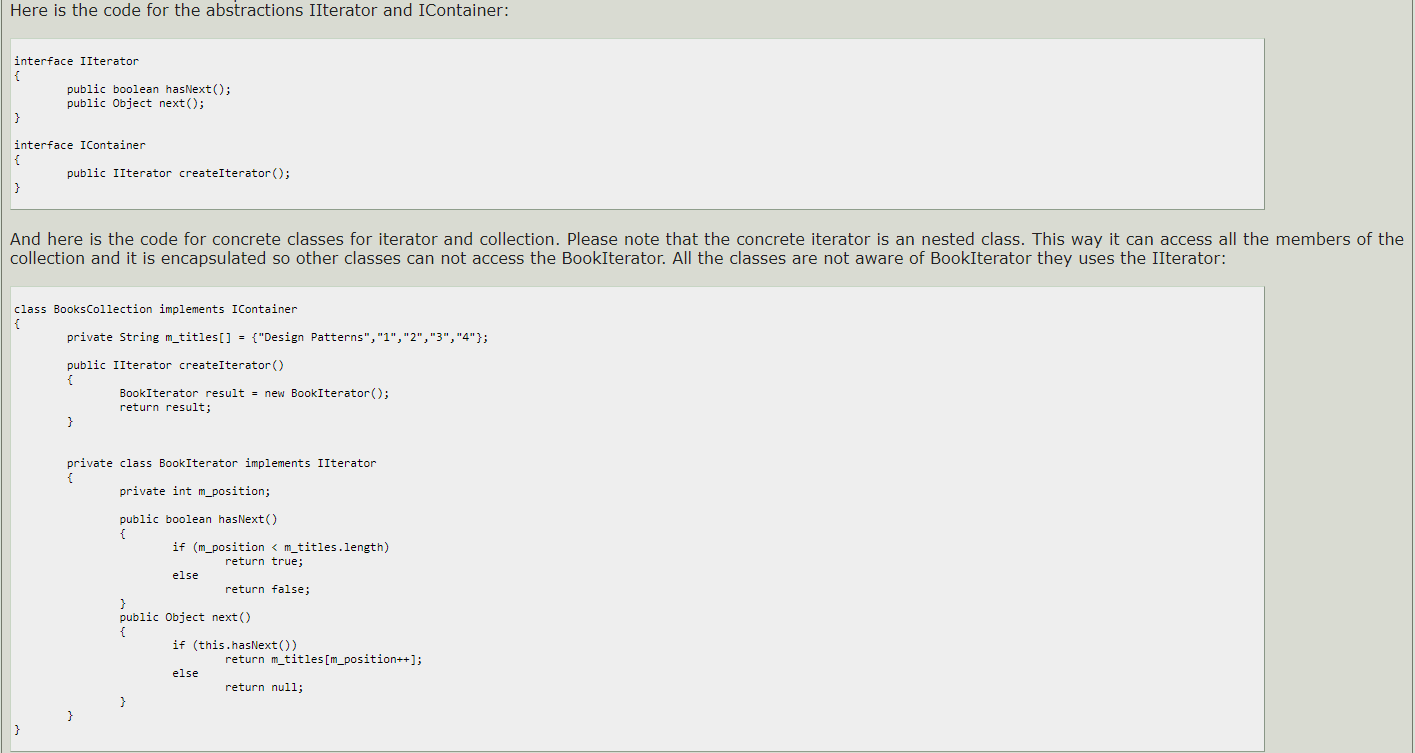
Az Iterator osztály implementálja az Iterator interfészt, és felelős az elemek ciklusonként történő visszaadásáért a kollekcióból. Az Iterator osztályban eltároljuk a hivatkozást a kollekcióra, és az aktuális pozíciót, amelyen belül az Iterator éppen tartózkodik.

A Client osztály használja az Iterator osztályt a kollekció eleminek lekérdezésére. A Client-osztály egy iterátort hív meg a kollekció elemeinek lekérdezéséhez, és iteratívan lekérdezi az összes elemet, amíg azok léteznek.

Az Iterator minta előnye, hogy lehetővé teszi a kollekciók elemzését anélkül, hogy a kódban feltétlenül ismernünk kellene a kollekció belső szerkezetét. Ezenkívül a kód áttekinthetőbbé és könnyebben karbantarthatóvá válik általa, mivel az elemzésre vonatkozó összes kód egy helyen található.

Az iterátor tervezési minta lényege, hogy fogja a kollekció elemeinek elérésével és végighaladásával járó felelősséget és azt az iterátor objektumba helyezi. Az iterátor objektum fenntartja az iteráció állapotát, követi a jelenlegi elemet, és lehetőséget biztosít azon elemeinek azonosítására, amelyeket még végig kell járni.





## Template

A Template minta egy olyan tervezési minta, amely lehetővé teszi az egyes részletek későbbi definiálását. Ez egy olyan sablon vagy keretrendszer, amely előre definiált struktúrák és függvények kombinációját tartalmazza. A minta lényege az, hogy előre definiáljuk a keretrendszert, majd ezt az örökléssel felülírhatjuk, hogy a részletekben eltérjünk attól, amit előre definiáltunk.

A minta általában absztrakt osztályokat vagy interfészeket használ, amelyek meghatározzák a keretrendszer vázát. Ezután a származtatott osztályok a keretrendszer konkrét megvalósítását nyújtják, amely eltérhet az absztrakt osztályokban vagy interfészekben definiáltaktól. Az így kapott osztályok azonban nem változtatják meg a keretrendszer vázát.

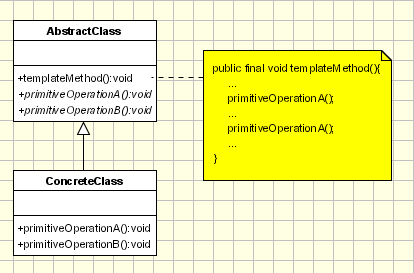
Például gondoljunk egy olyan alkalmazásra, amelyben különböző fájlokra kell keresni. A Template minta szerint megalkothatjuk az absztrakt SearchClass osztályt, amely tartalmazza a keresés alapvető folyamatát, beleértve a fájlok átnézését, a találatok számának meghatározását stb. Azonban az egyes fájltípusokban eltérő keresési algoritmusok lehetnek. Például egy szövegfájlban a kulcsszavakat kell keresni, míg egy adatbázisban a rekordokat kell átnézni.

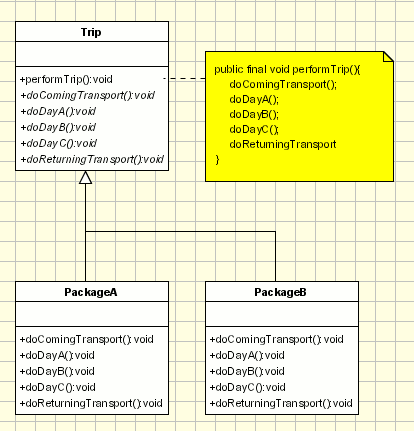
Ezután létrehozhatjuk a származtatott osztályokat, például a TextSearchClass-t vagy az SQLSearchClass-t, amelyek eltérő keresési algoritmusokat tartalmaznak, de az absztrakt SearchClass vázát követik. A kliens csak a megfelelő származtatott osztályt hívja meg a fájltípus alapján, amelynek keresését végre kell hajtani.

A Template minta előnye, hogy lehetővé teszi a kód újrafelhasználását és a konzisztens kódolást az összes származtatott osztályban. Emellett rugalmasságot biztosít az alkalmazásban, mivel a keresési algoritmusok könnyen cserélhetők vagy bővíthetők, anélkül hogy az alkalmazás többi részében változtatni kellene a kódot.

Ha megnézzük a sablon meghatározását a szótárban, láthatjuk, hogy egy sablon egy előre beállított formátum, amelyet egy adott alkalmazás kiindulópontjaként használnak, így a formátumot nem kell minden alkalommal újra létrehozni, amikor használják.

Ezen az ötleten alapul a sablon módszer is. Egy sablon módszer egy algoritmust definiál egy alap osztályban, absztrakt műveleteket használva, amelyeket az leszármazott osztályok felülírnak, hogy konkrét viselkedést biztosítsanak.





## Proxy

A Proxy tervezési minta arról szól, hogy egy osztály álljon közvetítőként egy másik osztály és a kliensek között, hogy megóvja az eredeti osztályt a kliensek direkt hívásaitól.

A Proxy osztály interfésze ugyan az, mint az eredeti osztályé, így a kliensek nem tudják megkülönböztetni az eredeti osztályt a Proxy osztálytól. A Proxy osztály egyfajta csomagolóként működik, amely átirányítja a kliens kéréseit az eredeti osztály felé, de lehetőséget biztosít a Proxy osztálynak (és ezzel a kód szerzőjének) arra, hogy módosítsa az eredeti osztály kliens felé látszó viselkedését anélkül, hogy közvetlenül megváltoztatná az eredeti osztály működését.

A Proxy osztály általában akkor hasznos, ha az eredeti osztály nagy méretű, bonyolult, lassú vagy távoli helyen található, és a kliensnek szüksége van arra, hogy gyorsan és hatékonyan hozzáférjen az erőforrásokhoz. A Proxy osztály lehetőséget biztosít arra, hogy késleltessen vagy csökkentse az eredeti osztály hívásait, vagy lehetőséget biztosítson a cache-elésre és az adatok előzetes betöltésére, hogy az eredeti osztályhoz való hozzáférés még hatékonyabb legyen.

A mintának az a lényege, hogy nyúlt egy placeholdert az objektumhoz, hogy a felé érkező kommunikációt irányítani lehessen.

