ELTE IK - Programtervező Informatikus BSc

Záróvizsga tételek

9. Objektumelvű tervezés

Objektumelvű tervezés

Nagy rendszerek fejlesztési fázisai, fejlesztési módszerek. SOLID tervezési elvek. Architekturális minták (MV, MVC stb.). Tervezési minták szerepe, osztályozása (létrehozási, szerkezeti, viselkedési), és kategóriánként 2-2 nevezetes tervezési minta bemutatása.

1 Nagy rendszerek fejlesztési fázisai, kapcsolataik

1.1 Fejlesztési fázisok

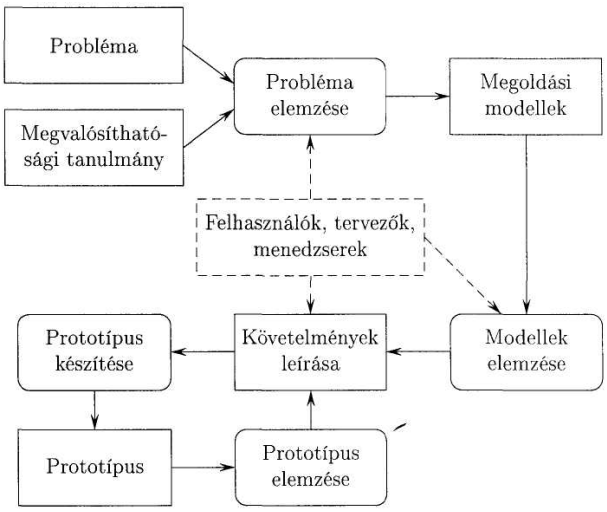
1. A probléma megoldásának előzménye

   Egy probléma megoldása előtt meg kell vizsgálni a megvalósíthatóságát, és annak mikéntjét. Eredmény: *Megvalósíthatósági tanulmány*, mely a következőkre válaszol:

   - Erőforrások (hardver, szoftver, szakember)
   - Költségek
   - Határidő
   - Üzemeltetés

2. Követelmények leírása

   Rendszerint iteratív módon állítjuk elő, és a prototípust használjuk a finomításra (ábra 1).



ábra 1: Követelményleírás elkészítésének folyamata

Követelmények leírásának tartalma:

- Probléma
- Korlátozó tényezők (hardver, szoftver, stb.)
- Elfogadható megoldás

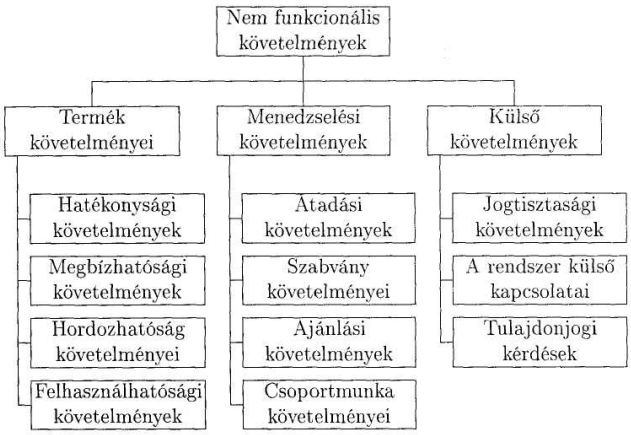
Követelmények leírásának fajtái:

- Funkcionális követelmények

  A rendszer szolgáltatásainak, leképzéseinek leírása:

  - Elindítás formája
  - Bemenő adatok (és azok megadásának formája)
  - Igénybevétel előfeltétele, korlátozások
  - Szolgáltatás kezdeményezésére a válasz, eredmények
  - Válasz megjelenési formája
  - Bemenő adatok és válasz közti reláció
- Nem funkcionális követelmények

  A nem funkcionális követelményeket rendszerint három osztályba soroljuk: a termék követelményei, menedzselési követelmények, külső követelmények. Az osztályokat tovább lehet bontani (ábra 2).



ábra 2: Nem funkcionális követelmények osztályozása

3. Követelmények elemzése és prototípus

A következőket kell megvizsgálni:

- Önmagában jó-e a követelmények leírása?
  - Konzisztens (nincs ellentmondás)
  - Komplett (teljes)
- Validáció vizsgálat

  (Megfelel-e a felhasználó által elképzelt problémának?)
- Megvalósíthatósági vizsgálat

  (A követelményeknek megfelelő megoldás megvalósítható-e?)
- Tesztelhetőségi vizsgálat

  (A követelmények úgy vannak-e megfogalmazva, hogy azok tesztelhetők?)
- Nyíltság kritériumainak vizsgálata.

  (A követelmények nem mondanak-e ellent a módosíthatóság, a továbbfejleszthetőség követelményének?)

A követelmények elemzésének egyik eszköze a prototípus-készítés. A prototípus magas szintű programozási környezetben létrehozott, a külső viselkedés szempontjából helyes megoldása a problémának.

4. Programspecifikáció

   A programspecifikáció a következő kérdésekre kell, hogy válaszoljon a követelmények leírása alapján:

   - Mik a bemenő adatok? (Forma, jelentés, megjelenés.)
   - Mik az eredmények? (Forma, jelentés, megjelenés.)
   - Mi a reláció a bemenő adatok és az eredmény adatok között?

5. Tervezés

   A tervezés során a következő kérdésekre adjuk meg a választ:

   (a) Statikus modell
   - Rendszer szerkezete
   - Programegységek, azok feladata és kapcsolata

   (b) Dinamikus modell
   - Hogyan oldja meg a rendszer a problémát?
   - Milyen egységek működnek együtt?
   - Milyen üzenetek játszódnak le?
   - Rendszer és egységek állapotai
   - Események (melyek hatására állapotváltás történik)

   (c) Funkcionális modell
   - Milyen adatáramlások révén valósulnak meg a szolgáltatások?
   - Milyen leképezések játszanak szerepet az adatáramlásokban?
   - Mik az ajánlások az implementáció számára?
     – Implementációs stratégiára vonatkozó ajánlás.
     – Programozási nyelvre vonatkozó előírás, ajánlás.
     – Tesztelési stratégiára vonatkozó ajánlás.

   A gyakorlatban két tervezési módszer terjedt el: *procedurális* és a *objektumelvű*

   (*procedurális*: megvalósítandó funkciókból, műveletekből indulunk ki, és ezek alapján bontjuk fel a rendszert kisebb összetevőkre, modulokra
   objektumelvű: a rendszer funkciói helyett az adatokat állítjuk a tervezés középpontjába. A rendszer által használt adatok felelnek meg majd bizonyos értelemben az objektumoknak.)

6. Implementáció

   Fontos szempontok:

   - Reprezentáció (Adatok ábrázolása)
   - Események leképezések megvalósítása
     Algoritmusok és optimalizálások

   Az implementáció egyik alapvető kérdése az implementációs stílus. A jó programozási stílus néhány fontos eleme:

   - absztrakció különböző szintjeinek alkalmazása
   - öröklődési technika használata, absztrakciós szintek hierarchikus rendszere
   - absztrakciós szintekre bontás osztályon belül (deklaráció + megvalósítás)

- korlátolt láthatóság;
- információ elrejtés (information hiding);
- információ beburkolás (encapsulation).

7. Verifikáció, validáció

   A rendszer eleget tesz-e a vele szemben támasztott elvárásoknak?

   Verifikáció: a specifikációszerinti helyesség igazolása

   Validáció: Minőségi előírások teljesítése (robosztusság hatékonyság, erőforrásigény)

   Ennek folyamata: tesztelés, melynek szakaszai:

   - Egységteszt
   - Rendszerteszt

   A tesztelésnek két módja lehet:

   - fekete doboz - Csak a maguknak a hibáknak a felderítése
   - fehér doboz - Hibák helyének felderítése

8. Rendszerkövetés és karbantartás (maintenance)

   Karbantartás: Üzemebe helyezés után szükségessé váló szoftver jellegű munkák [pl.: rejtett hibák kijavítása, adaptációs munkák (új harver-, szoftverkörnyezet), továbbfejlesztési munkák]

   Rendszerkövetés: a felhasználókkal való kapcsolattartás menedzsment jellegű, dokumentációs feladatai [pl.: konfigurációk nyilvántartása, verziók menedzselése, dokumentáció menedzselése]

9. Dokumentáció

   Egy nagy méretű program önmagában nem tekinthető szoftverterméknek dokumentáció nélkül. Egy jó dokumentáció a következőképp épül fel.

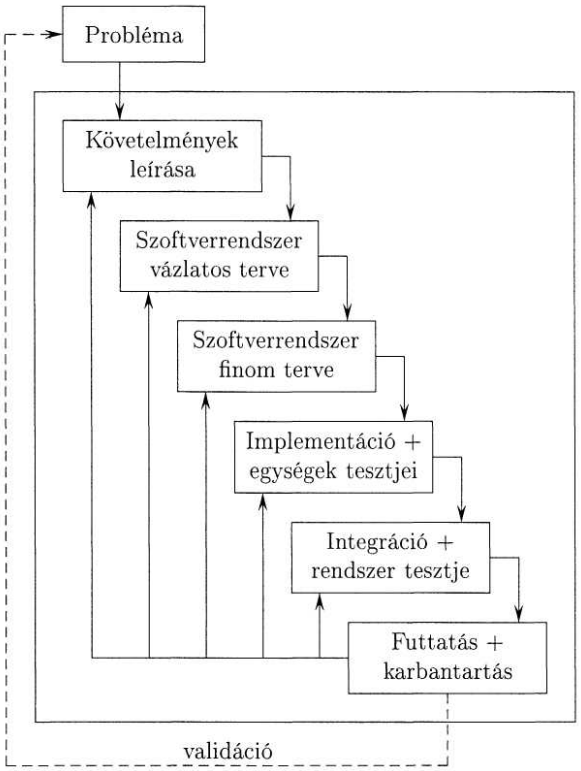
   - Felhasználói leírás
     - Feladatleírás
     - Futtató környezet
     - Fejlesztések, verziók
     - Installálás
     - Használat
     - Készítők
   - Fejlesztői leírás
     - Modulok (és azok szerkezete)
     - Osztályok (és azok kapcsolata)
     - Rendszer dinamikus viselkedése
     - Osztályok implementálása (adatszerkezetek, sablon osztályok)
     - Tesztelés

1.2 Fejlesztési fázisok kapcsolatai

A fejlesztési fázisok leírására többféle modellt használhatunk

1. Vízesés modell

   Az egyes fázisok egymást követik, a módosítások a futtatási eredmények ismeretében történnek. Egy bizonyos fázisban elvégzett módosítás az összes rákövetkező fázist is érinti.



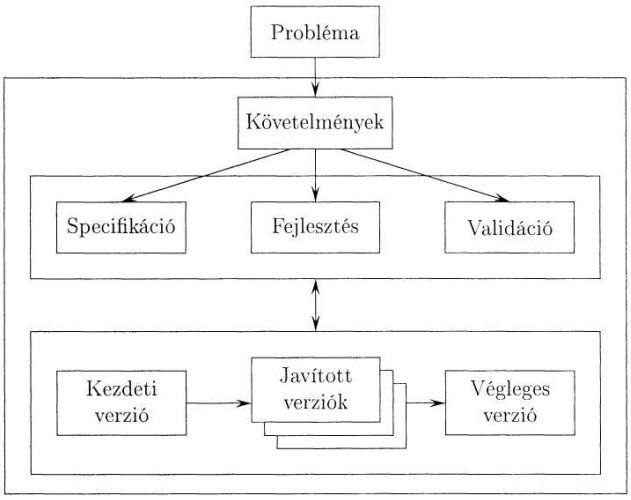
ábra 3: Vízesés modell

   Hátrányai:

   - Új szolgáltatás minden fázison módosítást igényel
   - Validáció az egész életciklus megismétlését követelheti meg

2. Evolúciós modell

   A megoldást közelítő verzióinak, prototípusainak sorozatát állítjuk egymás után elő, és így haladunk lépésenként egészen a végleges megoldásig. Ennek során egy verzió elkészítésekor a specifikáció, a fejlesztés és a validáció párhuzamosan történik.



ábra 4: Evolúciós modell

Hátrányai:

- Nehéz a projekt áttekintése
- A gyors fejlesztés rendszerint a dokumentáltság rovására megy.

3. Boehm-féle spirális modell

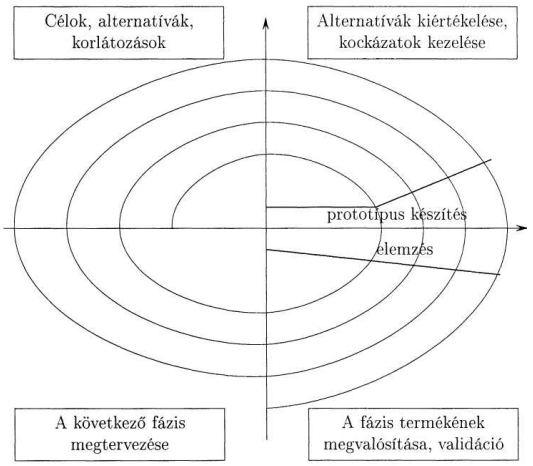
   Ez a modell egy iterációs modell. Az iteráció a spirális egy fázisával modellezhető, amely négy szakaszra bontható:

   (a) Célok, utak, alternatívák, korlátozások definiálása

   (b) Kockázatelemzés, stratégia kidolgozás

   (c) Feladat megoldása, validáció

   (d) Következő iteráció megtervezése



ábra 5: Evolúciós modell

Hátrányai:

- A modell alkalmazása általában munkaigényes, bonyolult feladat.
- A projekt kidolgozásához szükséges szakembereket nem könnyű gazdaságosan foglalkoztatni.

2 SOLID tervezési elvek

Az objektumorientált programozásban a SOLID egy mozaikszó, amely az öt tervezési alapelv (Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle) kezdőbetűjéből áll, és célja, hogy a szoftvertervezést még érthetőbbé, rugalmasabbá és karbantarthatóbbá tegye. A SOLID-nak semmi köze a felelősségek hozzárendelésének általános mintáihoz (General Responsibility Assignment Software Patterns, GRASP). A S.O.L.I.D. alapelvek szülőatyja Robert Cecil Martin amerikai mérnök informatikus és tanácsadó, aki nem csak a "tiszta kód mozgalom" vezérszónoka, hanem többek között az Agile Manifesto egyik eredeti megfogalmazója is. Ezek az alapelvek adják a magját az agilis szoftverfejlesztésnek vagy adaptív szoftverfejlesztésnek. Az elméletét Martin Design Principles and Design Patterns című könyvében mutatta be, de mint mozaikszó csak később Michael Feathers révén terjedt el.

- **Egyetlen felelősség elve - Single Responsibility Principle** Egy osztály vagy modul egy, és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra).
- **Nyílt/zárt elv - Open/Closed Principle** Egy osztály vagy modul legyen nyílt a kiterjesztésre, de zárt a módosításra.

- **Liskov helyettesítési elv - Liskov substitution principle** Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.
- **Interfész elválasztási elv - Interface segregation principle** Az interfészek (kapcsolódási felületek) szétválasztásának elve: egyetlen kliens se legyen rákényszerítve arra, hogy olyan eljárásoktól függjön, amelyeket nem is használ.
- **Függőség megfordítási elv - Dependency inversion principle** A magas szintű modulok ne függjenek az alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.

3 Architekturális minták (MV, MVC stb.)

3.1 MVC

A modell-nézet-vezérlő (MNV) (angolul model-view-controller) a szoftvertervezésben használatos programtervezési minta. Összetett, sok adatot a felhasználó elé táró számítógépes alkalmazásokban gyakori fejlesztői kívánalom az adathoz (modell) és a felhasználói felülethez (nézet) tartozó dolgok szétválasztása, hogy a felhasználói felület ne befolyásolja az adatkezelést, és az adatok átszervezhetők legyenek a felhasználói felület változtatása nélkül. A modell-nézet-vezérlő ezt úgy éri el, hogy elkülöníti az adatok elérését és az üzleti logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.

Hagyományosan asztali felhasználói felületekhez használt, de manapság már webalkalmazásokhoz is népszerűvé vált. Népszerű programozási nyelvek mint a JavaScript, Python, Ruby, PHP, Java már külön telepítés szükségessége nélkül rendelkeznek MNV keretrendszerekkel web- és mobilalkalmazások fejlesztésére.

Gyakori egy alkalmazás több rétegre való felbontása: megjelenítés (felhasználói felület), tartománylogika és adatelérés. Az MNV-ben a megjelenítés tovább bomlik nézetre és vezérlőre. Az MNV sokkal inkább meghatározza egy alkalmazás szerkezetét, mint az egy programtervezési mintára jellemző.

Modell (Model)

Az alkalmazás által kezelt információk tartomány-specifikus ábrázolása. A tartománylogika jelentést ad a puszta adatnak (pl. kiszámolja, hogy a mai nap a felhasználó születésnapja-e, vagy az összeget, adókat és szállítási költségeket egy vásárlói kosár elemeihez).

Sok alkalmazás használ állandó tároló eljárásokat (mint mondjuk egy adatbázis) adatok tárolásához. Az MNV nem említi külön az adatelérési réteget, mert ezt beleérti a modellbe.

Nézet (View)

Megjeleníti a modellt egy megfelelő alakban, mely alkalmas a felhasználói interakcióra, jellemzően egy felhasználói felületi elem képében. Különböző célokra különböző nézetek létezhetnek ugyanahhoz a modellhez.

Vezérlő (Controller)

Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat.[8]

Az MNV gyakran látható webalkalmazásokban, ahol a nézet az aktuális HTML oldal, a vezérlő pedig a kód, ami összegyűjti a dinamikus adatokat és létrehozza a HTML-ben a tartalmat. Végül a modellt a tartalom képviseli, ami általában adatbázisban vagy XML állományokban van tárolva.

Habár az MNV-nek sok értelmezése létezik, a vezérlés menete általánosságban a következőképp működik:[9]

- A felhasználó valamilyen hatást gyakorol a felhasználói felületre (pl. megnyom egy gombot).
- A vezérlő átveszi a bejövő eseményt a felhasználói felülettől, gyakran egy bejegyzett eseménykezelő vagy visszahívás útján.
- A vezérlő kapcsolatot teremt a modellel, esetleg frissíti azt a felhasználó tevékenységének megfelelő módon (pl. a vezérlő frissíti a felhasználó kosarát). Az összetett vezérlőket gyakran alakítják ki az utasítás mintának megfelelően, a műveletek egységbezárásáért és a bővítés egyszerűsítéséért.

- A nézet (közvetve) a modell alapján megfelelő felhasználói felületet hoz létre (pl. a nézet hozza létre a kosár tartalmát felsoroló képernyőt). A nézet a modellből nyeri az adatait. A modellnek nincs közvetlen tudomása a nézetről.
- A felhasználói felület újabb eseményre vár, mely az elejéről kezdi a kört.

A modell és a nézet kettéválasztásával az MNV csökkenti a szerkezeti bonyolultságot, és megnöveli a rugalmasságot és a felhasználhatóságot.

Szolgáltatás (Service)

A vezérlő és a modell közötti réteg. A modelltől kér le adatokat és a vezérlőnek adja azt. Ennek a rétegnek a segítségével az adat tárolás (modell), adat lekérés (szolgáltatás) és a adat kezelés (vezérlő) elkülöníthetőek egymástól. Mivel ez a réteg nem része az eredeti MNV mintának, ezért használata nem kötelező.

Előnyök:

- Egyidejű fejlesztés – Több fejlesztő tud egyszerre külön a modellen, vezérlőn és a nézeteken dolgozni.
- Magas szintű összetartás – MNV segítségével az összetartozó funkciók egy vezérlőben csoportosíthatóak. Egy bizonyos modell nézetei is csoportosíthatóak.
- Függetlenség – MNV mintában az elemek alapvetően nagy részben függetlenek egymástól
- Könnyen változtatható – Mivel a felelősségek szét vannak választva a jövőbeli fejlesztések könnyebbek lesznek
- Több nézet egy modellhez – Modelleknek több nézetük is lehet
- Tesztelhetőség - mivel a felelősségek tisztán szét vannak választva, a külön elemek könnyebben tesztelhetőek egymástól függetlenül

Hátrányok:

A MNV hátrányait általában a szükséges extra kódból adódnak.

- Kód olvashatósága – A keretrendszer új rétegeket add a kódhoz ami megnöveli a bonyolultságát
- Sok boilerplate kód – Mivel a programkód 3 részre bomlik a ebből az egyik fogja a legtöbb munkát végezni a másik kettő pedig az MNV minta kielégítése miatt létezik.'
- Nehezebben tanulható – A fejlesztőnek több különböző technológiát is ismernie kell a MNV használatához.

4 Tervezési minták szerepe, osztályozása, és kategóriánként 2-2 nevezetes tervezési minta bemutatása.

Az informatikában a programtervezési mintának (angolul Software Design Patterns) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.

A tervminták nem nyújtanak kész tervet, amit közvetlenül le lehet kódolni, habár vannak hozzájuk példakódok, amiket azonban meg kell tölteni az adott helyzetre alkalmas kóddal. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

A minták rendszerint osztályok és objektumok közötti kapcsolatokat mutatnak, de nem specifikálják konkrétan a végleges osztályokat vagy objektumokat. A modellek absztrakt osztályai helyett egyes esetekben interfészek is használhatók, habár azokat maga a tervminta nem mutatja. Egyes nyelvek beépítetten tartalmaznak tervmintákat. A tervminták tekinthetők a strukturált programozás egyik szintjének a paradigma és az algoritmus között.

A legtöbb tervminta objektumorientált környezetre van kidolgozva. Mivel a funkcionális programozás kevéssé ismert és használt, arra a környezetre még csak kevés tervminta ismert, például a monád. Az objektumorientált minták közül nem mindegyiket lehet, és nem mindegyiket érdemes itt használni. Van, amit módosítani kell.

A programtervezési minták a 90-es évek elején (1994) tettek szert népszerűségre, amikor a négyek bandájaként vagy gammáékként (angolul "Gang of Four" vagy GoF)-ként emlegetett Erich Gamma, Richard Helm, Ralph Johnson és John Vlissides programozó négyes kiadta a Programtervezési minták című könyvüket, amely ma is alapjául szolgál az objektumorientált programozási minták kutatásának. Magát a tervmintát nem definiálták. A fogalom maga évekig formalizálatlan maradt.

Ez a könyv összesen 23 mintát mutat be, és a következő kategóriákba sorolja őket:

- létrehozási minta
- szerkezeti minta
- viselkedési minta

A programtervezési minták a GoF definíciója szerint: egymással együttműködő objektumok és osztályok leírásai, amelyek testre szabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben. A szoftvertervezésben egy-egy problémára végtelen sok különböző megoldás adható, azonban ezek között kevés optimális van. Tapasztalt szoftvertervezők, akik már sok hasonló problémával találkoztak, könnyen előhúzhatnak egy olyan megoldást, amely már kipróbált és bizonyított. Kezdő fejlesztőknek viszont jól jön, ha mindazt a tudást és tapasztalatot, amit csak évek munkájával érhetnek el, precízen dokumentálva kézbe vehetik, tanulhatnak belőle és az általa bevezetett kifejezésekkel könnyebben beszélhetik meg egymás között az ötleteiket. A programtervezési minták ilyen összegyűjtött tapasztalatok, amelyek mindegyike egy-egy gyakran előforduló problémára ad általánosított választ. Azonban mindezek mellett még számos előnyük van:

- lerövidítik a tapasztalatszerzési időt. A programtervezési mintákat nem feltalálták, hanem a gyakran előforduló problémákra adott optimális válaszokat gyűjtötték össze, ezáltal olyan megoldásokat adtak, amelyekre előbb-utóbb a legtöbb fejlesztő magától is rájönne – csak esetleg jóval később. Természetesen nem kizárt, hogy léteznek jobb, hatékonyabb megoldások, és ritka amikor egy-egy mintát pontosan úgy lehet alkalmazni egy problémára, ahogy az a könyvekben le van írva, de mindenképpen érdemes megismerni őket, ha másért nem is, hogy elsajátíthassunk valamennyit a szerzők látásmódjából.
- lerövidítik a tervezési időt. Az összes minta jól dokumentált, könnyen újrafelhasználható, így ha egyszer alkalmazzuk őket, jó eséllyel egy hasonló problémánál újra eszünkbe fognak jutni az összes előnyükkel és hátrányukkal együtt. Így azonnal hatékony, rugalmas megoldást adhatunk és megkímélhetjük magunkat sok tervezéstől és esetleges újratervezéstől. Ráadásul a minták után található következmények rész elősegíti, hogy teljesebb képet kapjunk az alkalmazás hatásairól is.
- közös szótárat ad a fejlesztők kezébe. Ez megkönnyíti az egymás közti kommunikációt és a program dokumentálását is, hiszen könnyebb úgy beszélni egy probléma megoldásáról, ha van egy közös alap, ahonnan indulunk vagy amihez lehet hasonlítani az új terveket.
- magasabb szintű programozást tesz lehetővé. Mivel ezek a minták elterjedtségük miatt már kiállták nagyon sok programozó próbáját, feltehetőleg az optimális megoldást tartalmazzák a problémára.

A tervezési minták a modulokat és kapcsolataikat szervezik. Alacsonyabb szintűek, mint az architekturális minták, amelyek a teljes rendszer általános felépítését jellemzik.

Több különböző tervminta létezik, például:

- Algoritmus stratégia minták
- Magas szintű stratégiák, amelyek leírják, hogyan kell algoritmust szervezni egy adott architektúrára.
- Számítástervezési minták
- A kulcsfontosságú számítások megkeresését célozzák.
- Végrehajtási minták

- A végrehajtás alacsonyabb szintjén címzésekkel, feladatok végrehajtásának szervezésével, optimalizálásával, szinkronizálásával foglalkozó minták.
- Implementációs stratégia minták
- A forráskód implementációjában támogatják a program szervezését, és a párhuzamos programozás számára fontos adatszerkezetek felépítését.
- Szerkezeti tervezési minták
- Az alkalmazás globális struktúrájával foglalkoznak.

A tervmintákat Gammáék eredetileg három kategóriába csoportosították: létrehozási minták, szerkezeti minták, és viselkedési minták, amihez használták a delegálás, az aggregálás, és a konzultáció fogalmát. Objektumorientált környezetben az öröklődés, polimorfizmus és az absztrakt ősosztály fogalmait is felhasználják, habár az absztrakt osztályok többnyire interfészek is lehetnek. Egyes szerzők elkülönítik az architekturális tervezési mintákat is, mint a modell-nézet-vezérlő.

4.1 Létrehozási minták (Egyke, Prototípus)

4.1.1 Egyke (Singleton)

Biztosítja, hogy az osztályból csak egy példány készül, és biztosít egy publikus hozzáférést ehhez a példányhoz.

Az egyke programtervezési minta olyan programtervezési minta, amely egy objektumra korlátozza egy osztály létrehozható példányainak számát. Gyakori, hogy egy osztályt úgy kell megírni, hogy csak egy példány lehet belőle. Ehhez jól kell ismerni az objektumorientált programozás alapelveit. Az osztályból példányt a konstruktorával lehet készíteni. Ha van publikus konstruktor az osztályban, akkor akárhány példány készíthető belőle, tehát publikus konstruktora nem lehet az egykének. De ha nincs konstruktor, akkor nem hozható létre a példány, amin keresztül hívhatnánk a metódusait. A megoldást az osztályszintű (statikus) metódusok jelentik. Ezeket akkor is lehet hívni, ha nincs példány. Az egykének tehát van egy osztályszintű metódusa (szerezPéldány, angolul getInstance), ami minden hívójának ugyanazt a példányt adja vissza. Természetesen ezt a példányt is létre kell hozni, ehhez privát konstruktort kell készíteni, amit a szerezPéldány az egyke osztály tagjaként meghívhat.

4.1.2 Prototípus (Prototype)

Meghatároz egy előzetes mintát az objektumok létrehozásához, és később ez kerül másolásra. Gyakran használjuk, ha az objektum pontos típusa csak futásidőben derül ki.

A prototípus tervezési minta fő technikája a klónozás. A klónozás feladata, hogy az eredeti objektummal megegyező objektumot hozzon létre. Erre az egyszerű értékadás nem alkalmas, mert azok csak az objektum referenciáját másolják, így a két referencia ugyanoda mutat. A klónozásnak két fajtája van: sekély klónozás (angolul: shallow copy) és mély klónozás (angolul: deep copy).

4.2 Szerkezeti minták (Díszítő, Helyettes)

4.2.1 Díszítő (Decorator)

Lehetővé teszi az absztrakció változtatása nélkül további funkciók, felelősségi körök dinamikus hozzáadását.

A díszítő minta az átlátszó csomagolás klasszikus példája. Klasszikus példája a karácsonyfa. Attól, hogy a karácsonyfára felteszek egy gömböt, az még karácsonyfa marad, azaz a díszítés átlátszó. Ezt úgy érjük el, hogy az objektum összetételben szereplő mindkét osztály ugyanazon őstől származik, azaz ugyanolyan típusúak. Ez azért hasznos, mert a díszítő elemek gyakran változnak, könnyen elképzelhető, hogy új díszt kell felvenni. Ha díszítő egy külön típus lenne, akkor a karácsonyfa feldolgozó algoritmusok esetleg bonyolultak lehetnek.

A díszítő mintánál egy absztrakt ősből indulunk ki. Ennek kétfajta gyermeke van, alap osztályok, amiket díszíteni lehet és díszítő osztályok. A karácsonyfa példa esetén az alap osztályok a különböző fenyőfák. A díszítő osztályokat általában egy absztrakt díszítő osztály alá szervezzük, de ez nem kötelező.

A díszítés során az ős minden metódusát implementálni kell, úgy hogy, a becsomagolt példány metódusát meghívjuk, illetve ahol ez szükséges, ott hozzáadjuk a plusz funkcionalitást. Kétféle díszítésről beszélhetünk:

- Amikor a meglévő metódusok felelősségkörét bővítjük. Ilyen a karácsonyfás példa.
- Amikor új metódusokat is hozzáadunk a meglévőkhöz. Ilyen a Java adatfolyam (angolul: stream) kezelése, illetve a lenti kölcsönözhető jármű példa.

Mindkét esetben a példányosítás tipikusan így történik:
ŐsOsztály példány = new DíszítőN(...new Díszítő1(new AlapOsztály())...);

Mivel a csomagolás átlátszó, ezért akárhányszor becsomagolhatjuk a példányunkat, akár egy díszítővel kétszer is. Ez rendkívül dinamikus, könnyen bővíthető szerkezetet eredményez, amit öröklődéssel csak nagyon sok osztállyal lehetne megvalósítani.

Érdekes megfigyelni a minta UML ábráján, hogy a díszítő osztályból visszafelé mutat egy aggregáció az ős osztályra. Ez az adatbázis kezelés Alkalmazott - Főnök reláció megoldásához hasonlít, amikor az Alkalmazott tábla önmagával áll egy-több kapcsolatban, ahol a külső kulcs a főnök alkalmazott_ID értékét tartalmazza.

4.2.2 Helyettes (Proxy)

Egy másik objektum elfedésére, helyettesítésére alkalmazott tervezési minta.

A helyettes (angolul: proxy) tervezési minta egy nagyon egyszerű kompozícióra ad példát, ami ráadásul átlátszó becsomagolás. Egy valamilyen szempontból érdekes (drága, távoli, biztonsági szempontból érzékeny, ...) példányt birtokol a helyettese. Ez az érdekes objektum nem érhető el kívülről, csak a helyettesén keresztül érhetők el a szolgáltatásai. Ugyanakkor a külvilág azt hiszi, hogy az érdekes objektumot közvetlenül éri el, mert a helyettes átlátszó módon csomagolja be az érdekes objektumot. Az átlátszóság miatt a helyettesnek és az érdekes objektumnak közös őse van.

Sokféle helyettes létezik aszerint, hogy milyen szempontból érdekes a helyettesített objektum, pl.:

- Virtuális proxy: Nagy erőforrás igényű objektumok (pl. kép) helyettesítése a példányosítás (vagy más drága művelet) elhalasztásával, amíg ez lehetséges. A szövegszerkesztők ezt használják a képek betöltésére. Ha csak gyorsan átlapozom a dokumentumot, akkor a kép nem töltődik be (elhalasztódik a betöltés), csak helye látszik.
- Távoli proxy: Távoli objektumok lokális megjelenítése átlátszó módon. A kliens nem is érzékeli, hogy a tényleges objektum egy másik gépen van, amíg van hálózati kapcsolat. Ezt alkalmazza a távoli metódus hívás (remote method invocation – RMI).
- Védelmi proxy: A hozzáférést szabályozza különböző jogok esetén.
- Okos referencia: Az egyszerű referenciát helyettesíti olyan esetekben, amikor az objektum elérésekor további műveletek szükségesek.
- Gyorsító tár (cache): Ha van olyan számítás (ide sorolva a letöltéseket is), ami drága, akkor a számítás eredményét érdemes letárolni egy gyorsító tárban, ami szintén egyfajta proxy.

4.3 Viselkedési minták (Állapot, Megfigyelő)

4.3.1 Állapot (State)

Az objektum viselkedése megváltoztatható a belső állapottól függően.

Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota. Példa: TCP-Connection osztály egy hálózati kapcsolatot reprezentál; Három állapota lehet: Listening, Established, Closed; a kéréseket az állapotától függően kezeli.

Használjuk, ha

- az objektum viselkedése függ az állapotától, és a viselkedését az aktuális állapotnak megfelelően futás közben meg kell változtatnia, illetve
- a műveleteknek nagy feltételes ágai vannak, melyek az objektum állapotától függenek.

Előnyök:

- Egységbe zárja az állapotfüggő viselkedést, így könnyű új állapotok bevezetése.
- Áttekinthetőbb kód (nincs nagy switch-case szerkezet).
- A State objektumokat meg lehet osztani.

Hátrányok: Nő az osztályok száma (csak indokolt esetben használjuk).

4.3.2 Megfigyelő (Observer)

Meghatároz ez egy-a-többhöz függőséget objektumok között. Egy adott objektum módosulásáról automatikus értesítő információt küld a tőle függő objektumoknak, amik ezek alapján frissülnek.

ehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk. Részei:

- Alany: Tárolja a beregisztrált megfigyelőket, interfészt kínál a megfigyelők be- és kiregisztrálására valamint értesítésére.
- Megfigyelő: Interfészt definiál azon objektumok számára, amelyek értesülni szeretnének az alanyban bekövetkezett változásról. Erre a frissít (update) metódus szolgál.

Két fajta megfigyelő megvalósítást ismerünk:

- “Pull-os” megfigyelő: Ebben az esetben a megfigyelő lehúzza a változásokat az alanytól.
- “Push-os” megfigyelő: Ebben az esetben az alany odanyomja a változásokat a megfigyelőnek.

A kettő között ott van a különbség, hogy a frissít metódus milyen paramétert kap. Ha az alany átadja önmagát (egy frissít(this) hívás segítségével) a megfigyelőnek, akkor ezen a referencián keresztül a megfigyelő képes lekérdezni a változásokat. Azaz ez a “pull-os” megoldás.

Ha a frissít metódusnak az alany azokat a mezőit adja át, amik megváltoztak és amiket a megfigyelő figyel, akkor “push-os” megoldásról beszélünk. A következő példában épp egy ilyen megvalósítást láthatunk.