Objektumorientált szoftvertervezés - 2. előadás
Objektumorientált tervezési heurisztikák

Osztályok tervezése

- **Az attribútumok mindig legyenek privátok**, a hozzáférést getter/setter-rel biztosítjuk, így megmarad a konzisztencia
    - protected, public, package láthatóság sérti az információrejtés elvét
    - kötődés alakul ki → erősebb csatoltság lesz :(
    - year, month, day értékkészleteire figyelnünk kéne, ha publikusak akkor nem tudjuk őket felügyelni
- **Ne használjuk másik osztály nempublikus tagjait**
    - ha nagyon egybefüggenek egyen belőlük egy osztály → erős csatolás helyett erős kohézió lesz, ami jó
- **Minimalizáljuk a publikus metódusok számát**
    - feleslegesen semmi ne legyen publikus, ISP egy megvalósítása
- **Implementáljuk a sztenderd metódusokat**
    - string-é alakítás, összehasonlítás, hash kód generálás
- **Egy osztály ne függjön az őt használó osztályoktól**, azaz saját leszármazottjától se
- **Egy osztály csak egy absztrakcióval rendelkezzen**
    - ha keverednek az absztrakciós szintek, daraboljuk több részre
    - rossz helyen allokált felelősségek is az absztrakciós szintek keveredéséhez vezethetnek
        - kivétel: DRY elv sérülését kerüljük el → Visitor, Strategy
- **Az összetartozó adatot és viselkedést tartsuk egy helyen**, azaz egyetlen közös osztályban
    - körkörös viselkedést is érdemes egy osztályba vonni
- **A metódusok használjanak minél több attribútumot és metódust a saját osztályukból**
    - kerüljük az "isten osztályokat", daraboljuk többfele
    - kivétel: adatbáziskezelés, és hálózati lekérés osztályok
- **A viselkedést modellezzük, ne a szerepeket**

Felelősségek

- **A felelősségeket egyenletesen osszuk szét**, ne legyenek se túl sok, se túl kevés felelősséggel rendelkező osztály
- **Kerüljük az isten-osztályokat**
- **Kerüljük a csak adattárolásra használt osztályokat**
- **Kerüljük azokat az osztályokat, amelyeknek függvényeknek kellene lenniük**
    - legyen gyanús, ha egy osztálynak egy függvénye van, vagy az egész osztály neve valamilyen igét tartalmaz
- **Modellezzük a valódi világ működését**

    - mint a valóságban a mozdony húzza az első vagont, ami a másodikat, stb
- **Modellezzünk a megfelelő absztrakciós szinten**
    - azaz a való világot egyszerűsítsük
- **Modellezésnél maradjunk a rendszer határain belül**
    - pl pénzkiadó automatának nem része az ember, csak interfésze van
- **Mindig a nézet függjön a modelltől, sosem a modell a nézettől**
    - pull modell - grafika folyamatosan lekérdezi a modell állapotát, újrarajzoljuk x időnként a UI-t
    - push modell - a modell értesíti a grafikát, ha valami változott, csak ténylegesen szükséges esetben frissítjük
        - modellnek kell a grafikát ismernie (ez egy interface-el leválasztható)

Asszociációk

- cél: spagetti kód elkerülése
- **Minimalizáljuk az együttműködő osztályok számát**
    - ISP, DIP segítségül hívható
- **Minimalizáljuk az együttműködő osztályok között használt metódusok számát**
- **Osszuk szét a felelősségeket a tartalmazás mentén**
    - tartalmazott komponensekből jöjjön létre a tartalmazó osztály
- **Asszociáció helyett preferáljuk a tartalmazást**
    - a tartalmazás sokkal zártabb (fekete doboz szerű)
    - de vigyázzunk a tartalmazásból keletkező körre, illetve egy dolgot többen nem tartalmazhatnak
- **A tartalmazó objektum használja a tartalmazott objektumokat**
    - ha kiadja akkor megsértjük a zártságot, ha meg csak tartalmazza, akkor attribútum is elég lett volna
- **A tartalmazott objektum ne használja a tartalmazó objektumot**
    - mivel a konténer már eleve függ a tartalmazott objektumtól
- **A tartalmazott objektumok ne beszélgessenek egymással közvetlenül**
    - túl sok keresztfüggőséghez vezetne ez, inkább konténeren keresztül beszélgessenek

Öröklődés

- **Az öröklődés célja mindig a viselkedés újrahasznosítása**
    - arra a tartalmazás és delegáció való
- **Az öröklődés helyett preferáljuk a tartalmazást**
    - erre jó a dekorátor minta
- **A közös viselkedéssel nem rendelkező közös adat, tartalmazás relációban legyen**
    - azaz ne a Polyline, Line és Circle rendelkezzen x, y attribútummal az ősüktől származtatva
    - hanem legyen külön egy Point osztály, és abból tartalmazzanak elemeket
- **A közös viselkedéssel rendelkező közös adat ősosztályban legyen**
    - kivétel, ha többszörös öröklődésre nincs lehetőség a nyelvben → strategy osztály használata ajánlott
- **A közös viselkedés és közös adat minél magasabban legyen az öröklési hierarchiában**
    - azaz amilyen magasan csak lehet

- **Közös interfészt csak akkor valósítsunk meg, ha a viselkedés is közös**
    - duck typing → ha valami úgy mozog és hápog mint egy kacsa, akkor az egy kacsa, de nem annak kéne lennie!
- **Egy osztály ne függjön a saját leszármazottaitól**
- **Protected láthatóságot csak metódusoknál használjunk, az attribútumok mindig privátok legyenek**
- **Az öröklési hierarchia legyen mély, de legfeljebb hét szintű**
- **Absztrakt osztályok az öröklési hierarchia gyökerében legyenek**
- **Az öröklési hierarchia gyökerében interfészek vagy absztrakt osztályok legyenek**
- **Soha ne vizsgáljuk egy objektum típusát, használjunk helyette polimorfizmust**
    - TDA elv
    - Acyclic Visitor nem szegi ezt meg, pedig lekér típust (ez valami súlyos dolog :|)
- **Soha ne kódoljuk a típust enum vagy int értékekbe, használjunk helyette polimorfizmust**
- **Ne készítsünk függvényeket a típusok illetve a képességek megkülönböztetésére, használjunk helyettük polimorfizmust**
- **Ne keverjük össze a leszármazottakat az objektumokkal! Vigyázzunk azokkal a leszármazottakkal, amelyekből csak egyetlen példányt hozunk létre!**
    - tényleg adnak e hozzá viselkedést, vagy módosítanak e az ős viselkedésén?
    - ha nagyon hasonlít egymásra a leszármazottak viselkedése, érdemes lehet új őst csinálni
- **A statikus szemantikát és kényszereket a modell struktúrájába építsük be!**
    - ha kevés a helyes kombináció (pl pajzsos torony, repülő csapda, lövő torony van csak)
- **A statikus szemantikát és kényszereket a konstruktorba építsük be!**
    - ha túl sok a helyes kombináció
- **A dinamikus szemantikát és kényszereket viselkedésként implementáljuk**
- **A gyakran változó dinamikus szemantikát és kényszereket külső viselkedésként implementáljuk**
- **Az opcionális elemeket tartalmazásként implementáljuk, ne öröklődéssel**
    - hasznos lehet a nullobject minta
- **Ne keverjük össze a statikus és dinamikus kényszereket**
- **Ha reflection-re van szükségünk, modellezzünk osztályokat, ne objektumokat**
- **Ha az ős működését üres implementációval írjuk felül, akkor hibás az öröklési hierarchia**
- **Törekedjünk újrahasznosítható API írására, ne csak újrahasznosítható osztályokéra**
- **Ha többszörös öröklődésre van szükségünk, gondoljuk át még egyszer a terveket**
- Heterogén kollekció problémáira sokszor a Visitor minta jelent megoldást