# Objektumorientált tervezési alapelvek

Jeszenszky Péter
jeszenszky.peter@inf.unideb.hu

Utolsó módosítás: 2018. december 13.

# PMD

- Statikus kódelemző (programozási nyelv: Java, licenc: BSD-stílusú) https://pmd.github.io/
  - Támogatott programozási nyelvek: Apex, Java, JavaScript, PLSQL, Visualforce, Apache Velocity, XML, XSL
- Bővítmények:
  - *Apache Maven PMD Plugin* https://maven.apache.org/plugins/maven-pmd-plugin/
  - *Gradle: The PMD Plugin* https://docs.gradle.org/current/userguide/pmd_plugin.html
  - *Eclipse PMD Plug-in* http://acanda.github.io/eclipse-pmd/
  - *PMDPlugin* (*IntelliJ IDEA*) https://plugins.jetbrains.com/plugin/1137-pmdplugin
  - *SQE (NetBeans)* https://github.com/sqe-team/sqe

# DRY (1)

- Ne ismételd magad (*Don't Repeat Yourself*)
  - *„Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”*
  - A tudás minden darabkájának egyetlen, egyértelmű, hiteles reprezentációja kell, hogy legyen egy rendszerben.
- Forrás:
  - Andrew Hunt, David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- Az ellenkezője a WET.
  - *We enjoy typing*, *write everything twice*, *we edit terribly*, …

# DRY (2)

- Az ismétlések fajtái:
  - **Kényszerített ismétlés** (***imposed duplication***): a fejlesztők úgy érzik, hogy nincs választásuk, a környezet láthatólag megköveteli az ismétlést.
  - **Nem szándékos ismétlés** (***inadvertent duplication***): a fejlesztők nem veszik észre, hogy információkat duplikálnak.
  - **Türelmetlen ismétlés** (***impatient duplication***): a fejlesztők lustaságából fakad, az ismétlés látszik a könnyebb útnak.
  - **Fejlesztők közötti ismétlés** (***interdeveloper duplication***): egy csapatban vagy különböző csapatokban többen duplikálnak egy információt.
- Kapcsolódó fogalom: kódismétlés (*code duplication*)

# DRY (3)

- PMD támogatás: *Copy/Paste Detector* (CPD)
  - *Finding duplicated code*
    https://pmd.github.io/pmd-6.10.0/pmd_userdocs_cpd.html
  - Támogatott programozási nyelvek: Apex, C/C++, C#, ECMAScript (JavaScript), Fortran, Go, Groovy, Java, Groovy, JSP, Matlab, Objective-C, Perl, PHP, PL/SQL, Python, Ruby, Scala, Swift, Visualforce
    - Lásd:
      https://pmd.github.io/pmd-6.10.0/pmd_userdocs_cpd.html#supported-languages

# KISS

- *Keep it simple, stupid*
  - 1960-as évek, amerikai haditengerészet.
  - Kelly Johnson (1910–1990) repülőmérnöknek tulajdonítják a kifejezést.
- Az egyszerűségre való törekvés:
  - Leonardo da Vinci (1452–1519): „Az egyszerűség a kifinomultság csúcsa.”
  - Ludwig Mies van der Rohe (1886–1969): „A kevesebb több.”
  - Albert Einstein (1879–1955):
    - *„Everything should be made as simple as possible, but not simpler.”*
    - *„Mindent olyan egyszerűen kell csinálni, amennyire csak lehetséges, de semmivel sem egyszerűbben.”*

# Demeter törvénye (1)

- Demeter törvénye (*Law of Demeter*):
  - Karl J. Lieberherr, Ian M. Holland, Arthur Joseph Riel. Object-Oriented Programming: An Objective Sense of Style. Proceedings on Object-oriented programming systems, languages and applications (OOPSLA), pp. 323– 334, 1988. https://doi.org/10.1145/62084.62113
  - Ian M. Holland, Karl J. Lieberherr. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, vol. 6, no 5, pp. 38– 48, 1989. https://doi.org/10.1109/52.35588
- Más néven: ne beszélgess idegenekkel (*Don't Talk to Strangers*)
  - Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Prentice Hall, 2005.

# Demeter törvénye (2)

- A metódusok üzenetküldési szerkezetét korlátozza.
  - Azt mondja, hogy minden metódusnál korlátozott azon objektumok köre, melyeknek üzeneteket küldhet.
- Célja az osztályok közötti függőségek szervezése és csökkentése.

# Demeter törvénye (3)

- Osztályokra vonatkozó változat (*class form*):
  - Egy *C* osztály egy *M* metódusa csak az alábbi osztályok és ősosztályaik tagjait (adattagjait, metódusait) használhatja:
    - *C*
    - *C* adattagjainak osztályai
    - *M* paramétereinek osztályai
    - Osztályok, melyek konstruktorai *M*-ben meghívásra kerülnek
    - *M*-ben használt globális változók osztályai
- Fordítási időben ellenőrizhető.

# Demeter törvénye (4)

- Alkalmazása növeli a karbantarthatóságot és az érthetőséget.
  - Ténylegesen szűkíti a metódusokban meghívható metódusok körét, ilyen módon korlátozza a metódusok csatoltságát.
  - Információ elrejtés (szerkezet elrejtés) kikényszerítése: egy objektum belső felépítését kizárólag saját maga ismeri.

# Demeter törvénye (5)

- Példa:
  - Mind a négy `hello()` metódushívás megengedett az alábbi programkódban:

```
class C {
  private B b;
  void m(A a) {
    b.hello();
    a.hello();
    Singleton.INSTANCE.hello();
    new Z().hello();
  }
}
```

  - Nem megengedett például az `a.x.hello()` metódushívás.
  - Forrás: Yegor Bugayenko. *The Law of Demeter Doesn't Mean One Dot*. 2016. http://www.yegor256.com/2016/07/18/law-of-demeter.html

# Demeter törvénye (6)

- Kapcsolódó PMD szabályhalmaz:
  - *Design (Java)*
    https://pmd.github.io/pmd-6.10.0/pmd_rules_java_design.html
    - Lásd a `LawOfDemeter` szabályt:
      https://pmd.github.io/pmd-6.10.0/pmd_rules_java_design.html#lawofdemeter

# Vonatkozások szétválasztása (1)

- Vonatkozások szétválasztása (SoC – *Separation of Concerns*):
  - Egy szoftverrendszert oly módon érdemes tervezni, hogy minden egyes komponensének pontosan meghatározott szerepe legyen, ideális esetben ezek a szerepek ne fedjék át egymást.
    - Példák: modell-nézet-vezérlő (MVC) architekturális minta, TCP/IP protokollkészlet, HTML + CSS + JavaScript, …
  - Felhasznált irodalom:
    - Ian Sommerville. *Software Engineering*. 10th ed. Pearson Education, 2015. http://iansommerville.com/software-engineering-book/

# Vonatkozások szétválasztása (2)

- A programok elemei (például osztályok, metódusok) pontosan egy dolgot csináljanak.
  - Az egyes elemekkel úgy lehet foglalkozni, hogy nem kell tekintettel lenni a program többi elemére.
    - Például a program egy része a vonatkozása ismeretében úgy érthető meg, hogy ahhoz nem szükséges a program többi elemének megértése.
    - Amikor változtatások szükséges, azok csak kevés számú elemet érintenek.
    - A program egyes részei egymástól függetlenül fejleszthetők, újrafelhasználhatók.

# Vonatkozások szétválasztása (3)

- A vonatkozás valami olyan, ami érdekes vagy fontos egy érintett vagy érintettek egy csoportja számára.
  - Például: teljesítmény, adott funkció biztosítása, karbantarthatóság, …
- A rendszerkövetelményeket tükrözik.

# Vonatkozások szétválasztása (4)

- Vonatkozások fajtái:
  - **Alapvető vonatkozások** (***core concerns***): a rendszer elsődleges céljához kötődő funkcionális vonatkozások.
  - **Másodlagos vonatkozások** (***secondary concerns***): például a rendszer nem funkcionális követelményeinek kielégítéséhez szükséges funkcionális vonatkozások.
  - **Átszövő vonatkozások** (***cross-cutting concerns***): alapvető rendszerkövetelményeket tükröző rendszerszintű vonatkozások.
    - A másodlagos vonatkozások lehetnek átszövőek is, bár nem minden esetben szövik át az egész rendszert.
    - Például: biztonság, naplózás.

# Vonatkozások szétválasztása (5)

- Kapcsolódó programozási paradigma: aspektus-orientált programozás (AOP – *aspect-oriented programming*)
  - Például: *AspectJ* https://eclipse.org/aspectj/
    - A Java programozási nyelv aspektus-orientált kiterjesztése.

# GRASP (1)

- **A felelősségek hozzárendelésének általános mintái** (**GRASP** – ***General Responsibility Assignment Software Patterns***)
  - Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Prentice Hall, 2005.
- A felelősségek hozzárendelésének alapelveit kifejező tervezési minták.
  - Tanulási segédeszközök, melyek az objektumorientált tervezés megértését és alkalmazását segítik.

# GRASP (2)

- Felelősség: egy osztályozó egy szerződése vagy kötelezettsége (UML).
- A felelősségek fajtái:
  - Valaminek a tevése (*doing*)
  - Valaminek a tudása (*knowing*)
- A felelősségek hozzárendelése az osztályokhoz a tervezés során történik.

# GRASP (3)

- 9 tervezési minta és alapelv:
  - **Információs szakértő** (***Information Expert***)
  - **Létrehozó** (***Creator***)
  - **Laza csatolás** (***Low Coupling***)
  - **Magas kohézió** (***High Cohesion***)
  - **Vezérlő** (***Controller***)
  - **Polimorfizmus** (***Polymorphism***)
  - **Tiszta kitaláció** (***Pure Fabrication***)
  - **Indirekció** (***Indirection***)
  - **Védett változatok** (***Protected Variations***)

# GRASP (5)

- Mintasablon:
  - **Név**:
  - **Probléma**:
  - **Példa**:
  - **Tárgyalás**:
  - **Ellenjavallatok**:
  - **Előnyök**:
  - **Háttér**:
  - **Kapcsolódó minták**:

# GRASP minták (1)

- **Információs szakértő**:
  - A felelősséget az információs szakértőhöz rendeljük, vagyis ahhoz az osztályhoz, mely rendelkezik a felelősség megvalósításához szükséges információkkal.
    - Példa: melyik osztály felelőssége tudni egy rendelés összértékét egy értékesítési rendszerben?

# GRASP minták (2)

- **Létrehozó**:
  - Az *A* osztály egy példánya létrehozásának felelősségét ruházzuk a *B* osztályra, ha az alábbiak valamelyike igaz:
    - A *B* osztály A objektumokat aggregál.
    - A *B* osztály *A* objektumokat tartalmaz.
    - *B* nyilvántartja *A* példányait.
    - *B* szorosan használja *A*-t.
    - *B* rendelkezik azokkal az inicializáló adatokkal, melyek *A*-nak átadásra kerülnek a létrehozásakor (azaz *B* információs szakértő *A* létrehozása szempontjából).
  - *B* az *A* objektumok létrehozója.
  - Ha fenti pontok közül egynél több teljesül, akkor egy *A*-t aggregáló vagy tartalmazó *B* osztályt részesítsünk előnyben.

# GRASP minták (3)

- **Laza csatolás**:
  - A csatolás annak mértéke, hogy egy elem mennyire kapcsolódik más elemekhez, mennyi információja van róluk, vagy mennyire függ tőlük.
  - A felelősségeket úgy rendeljük hozzá az elemekhez, hogy azok lazán csatoltak maradjanak.
    - Az elemek közé tartoznak az osztályok, alrendszerek, rendszerek, …
    - Egy lazán (vagy gyengén) csatolt elem nem függ túl sok elemtől, ahol a „túl sok” környezetfüggő.
    - Egy erősen csatolt osztály sok más osztálytól függ. Az ilyen osztályok nemkívánatosak, az alábbi problémákkal küzdenek:
      - A kapcsolódó osztályokban történő változások lokális változásokat kényszerítenek ki.
      - Nehezebb őket önmagukban megérteni.
      - Nehezebb őket újrafelhasználni, mivel szükségesek hozzájuk azok az osztályok, melyektől függenek.

# GRASP minták (4)

- **Magas kohézió**:
  - A kohézió annak mértéke, hogy egy elem felelősségei milyen szorosan kapcsolódnak egymáshoz.
  - A felelősségeket úgy rendeljük hozzá az elemekhez, hogy tartsuk fenn a magas kohéziót.
    - Az elemek közé tartoznak az osztályok, alrendszerek, rendszerek, …
    - Magas a kohéziója egy olyan elemnek, melynek felelősségei szorosan kapcsolódnak és nem végez túl sok munkát.
    - Egy alacsony kohéziójú elem sok egymáshoz nem kapcsolódó dolgot csinál. Az ilyen osztályok nemkívánatosak, az alábbi problémákkal küzdenek:
      - Nehézen érthetőek.
      - Nehéz az újrafelhasználhatóságuk.
      - Nehéz a karbantartásuk.

# GRASP minták (5)

- **Vezérlő**:
  - A rendszer eseményei fogadásának vagy kezelésének felelősségét egy olyan osztályhoz rendeljük hozzá mely
    - a teljes rendszert, alrendszert vagy eszközt ábrázolja,
    - vagy egy olyan forgatókönyvet ábrázol, melyben az esemény bekövetkezik.
  - A vezérlő egy nem felhasználói interfész objektum.

# GRASP minták (6)

- **Polimorfizmus**:
  - Amikor összetartozó viselkedések változnak típustól (osztálytól) függően, akkor a viselkedést leíró felelősséget polimorf műveletek segítségével rendeljük hozzá azon típusokhoz, melyeknél a viselkedés változik.
    - Ne vizsgáljuk egy objektum típusát és ne használjunk feltételes programlogikát olyan változó alternatívák végrehajtásához, melyek a típustól függenek.
- **Tiszta kitaláció**:
  - Szorosan összetartozó felelősségek egy halmazát rendeljük hozzá egy mesterséges osztályhoz, mely nem a probléma szakterületének egy fogalmát ábrázolja, hanem a magas kohézió, a laza kapcsolódás és az újrafelhasználhatóság támogatása érdekében találtuk ki.
    - Példa: DAO osztályok

# GRASP minták (7)

- **Indirekció**:
  - Objektumok túl szoros kapcsolódásán lazíthatunk egy köztes objektummal, mely közöttük közvetítő szerepet tölt be. A közvetítő az objektumok közötti indirekciót hoz létre, mivel azok nem közvetlenül kapcsolódnak.
- **Védett változatok**:
  - Azonosítsuk az előre látható változások és bizonytalanságok által érintett elemeket és hozzunk létre köréjük egy stabil interfészt. A polimorfizmus révén az interfészhez különféle implementációkat hozhatunk létre.

# GoF alapelvek (1)

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Programtervezési minták: Újrahasznosítható elemek objektumközpontú programokhoz*. Kiskapu, 2004.

# GoF alapelvek (2)

- **Interfészre programozzunk, ne implementációra!**
  - *„Program to an interface, not an implementation.”*
- Lásd a létrehozási mintákat!

# GoF alapelvek (3)

- **Részesítsük előnyben az objektum-összetételt az öröklődéssel szemben!**
  - *„Favor object composition over class inheritance.”*
- A két leggyakoribb módszer az újrafelhasználásra az objektumorientált rendszerekben:
  - Öröklődés (fehér dobozos újrafelhasználás)
  - Objektum-összetétel (fekete dobozos újrafelhasználás)
- A fehér/fekete dobozos jelző a láthatóságra utal.

# GoF alapelvek (4)

- Az öröklődés előnyei:
  - Statikusan, fordítási időben történik, és használata egyszerű, mivel a programozási nyelv közvetlenül támogatja.
  - Az öröklődés továbbá könnyebbé teszi az újrafelhasznált megvalósítás módosítását is. Ha egy alosztály felülírja a műveletek némelyikét, de nem mindet, akkor a leszármazottak műveleteit is megváltoztathatja, feltételezve, hogy azok a felülírt műveleteket hívják.

# GoF alapelvek (5)

- Objektum összetételt használó tervezési minták:
  - Szerkezeti objektumminták: (objektum) illesztő, híd, összetétel, díszítő, homlokzat, pehelysúlyú, helyettes.
  - Viselkedési objektumminták: felelősséglánc, parancs, bejáró, közvetítő, emlékeztető, megfigyelő, állapot, stratégia, látogató.

# GoF alapelvek (6)

- Az öröklődés hátrányai:
  - Először is, a szülőosztályoktól örökölt megvalósításokat futásidőben nem változtathatjuk meg, mivel az öröklődés már fordításkor eldől.
  - Másodszor – és ez általában rosszabb –, a szülőosztályok gyakran alosztályaik fizikai ábrázolását is meghatározzák, legalább részben. Mivel az öröklődés betekintést enged egy alosztálynak a szülője megvalósításába, gyakran mondják, hogy az öröklődés megszegi az egységbe zárás szabályát. Az alosztály megvalósítása annyira kötődik a szülőosztály megvalósításához, hogy a szülő megvalósításában a legkisebb változtatás is az alosztály változását vonja maga után.
  - Az implementációs függőségek gondot okozhatnak az alosztályok újrafelhasználásánál. Ha az örökölt megvalósítás bármely szempontból nem felel meg az új feladatnak, arra kényszerülünk, hogy újraírjuk, vagy valami megfelelőbbel helyettesítsük a szülőosztályt. Ez a függőség korlátozza a rugalmasságot, és végül az újrafelhasználhatóságot.

# GoF alapelvek (7)

- Az objektum-összetétel dinamikusan, futásidőben történik, olyan objektumokon keresztül, amelyek hivatkozásokat szereznek más objektumokra.
- Az összetételhez szükséges, hogy az objektumok figyelembe vegyék egymás interfészét, amihez gondosan megtervezett interfészek kellenek, amelyek lehetővé teszik, hogy az objektumokat sok másikkal együtt használjuk.

# GoF alapelvek (8)

- Az objektum összetétel előnyei:
  - Mivel az objektumokat csak az interfészükön keresztül érhetjük el, nem szegjük meg az egységbe zárás elvét.
  - Bármely objektumot lecserélhetünk egy másikra futásidőben, amíg a típusaik egyeznek.
  - Továbbá, mivel az objektumok megvalósítása interfészek segítségével épül fel, sokkal kevesebb lesz a megvalósítási függőség.
  - Az öröklődéssel szemben segít az osztályok egységbe zárásában és abban, hogy azok egy feladatra összpontosíthassanak.
  - Az osztályok és osztályhierarchiák kicsik maradnak, és kevésbé valószínű, hogy kezelhetetlen szörnyekké duzzadnak.

# GoF (9)

- Másrészről az objektum-összetételen alapuló tervezés alkalmazása során több objektumunk lesz (még ha osztályunk kevesebb is), és a rendszer viselkedése ezek kapcsolataitól függ majd, nem pedig egyetlen osztály határozza meg.

# SOLID (1)

- Robert C. Martin („Bob bácsi”) által megfogalmazott/rendszerezett/népszerűsített objektumorientált programozási és tervezési alapelvek.
  - Bob bácsi honlapja: http://cleancoder.com/
  - Uncle Bob. *Getting a SOLID start*. 2009. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start
- Irodalom:
  - Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2002.
    - C++ és Java nyelvű programkódok.
  - Robert C. Martin, Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

# SOLID (2)

- ***S**ingle responsibility principle* (SRP) – Egyszeres felelősség elve
- ***O**pen/closed principle* (OCP) – Nyitva zárt elv
- ***L**iskov substitution principle* (LSP) – Liskov-féle helyettesítési elv
- ***I**nterface segregation principle* (ISP) – Interfész szétválasztási elv
- ***D**ependency inversion principle* (DIP) – Függőség megfordítási elv

# SOLID – egyszeres felelősség elve (1)

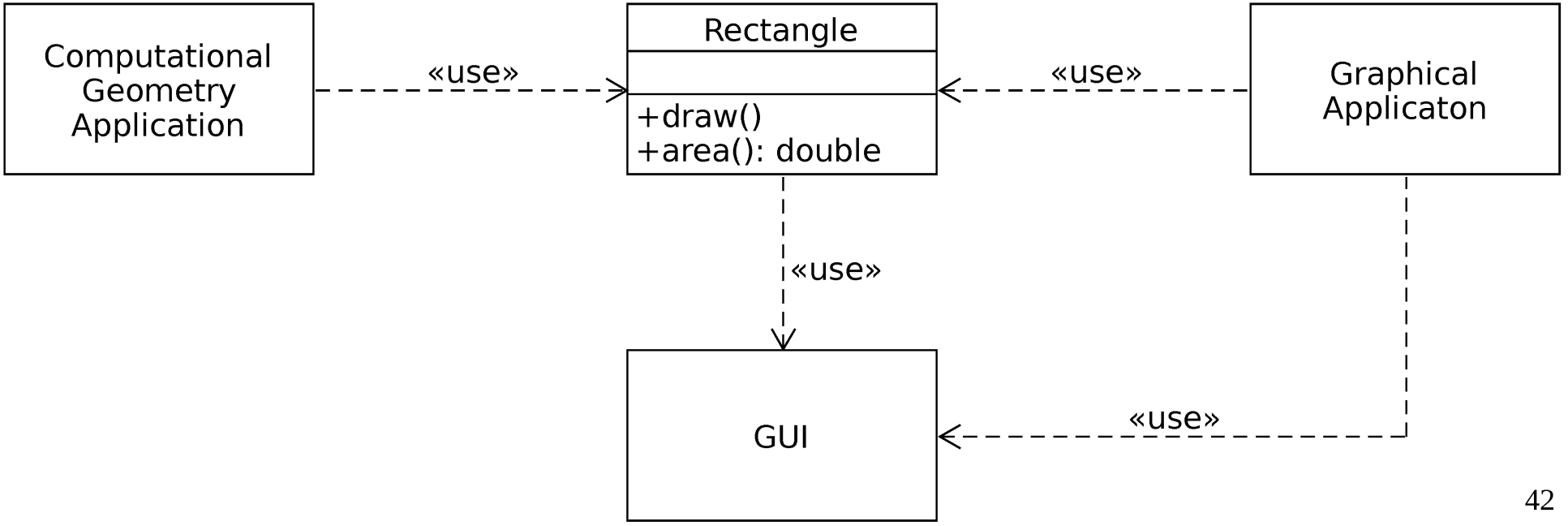
- Robert C. Martin által megfogalmazott elv:
  - *„A class should have only one reason to change.”*
  - Egy osztálynak csak egy oka legyen a változásra.
- Kapcsolódó tervezési minták: díszítő, felelősséglánc

# SOLID – egyszeres felelősség elve (2)

- Egy felelősség egy ok a változásra.
- Minden felelősség a változás egy tengelye. Amikor a követelmények változnak, a változás a felelősségben történő változásként nyilvánul meg.
- Ha egy osztálynak egynél több felelőssége van, akkor egynél több oka van a változásra.
- Egynél több felelősség esetén a felelősségek csatolttá válnak. Egy felelősségben történő változások gyengíthetik vagy gátolhatják az osztály azon képességét, hogy eleget tegyen a többi felelősségének.

# SOLID – egyszeres felelősség elve (3)

- Példa az elv megsértésére:
  - A Rectangle osztály két felelőssége:
    - Egy téglalap geometriájának matematikai modellezése.
    - Téglalap megjelenítése a grafikus felhasználói felületen.

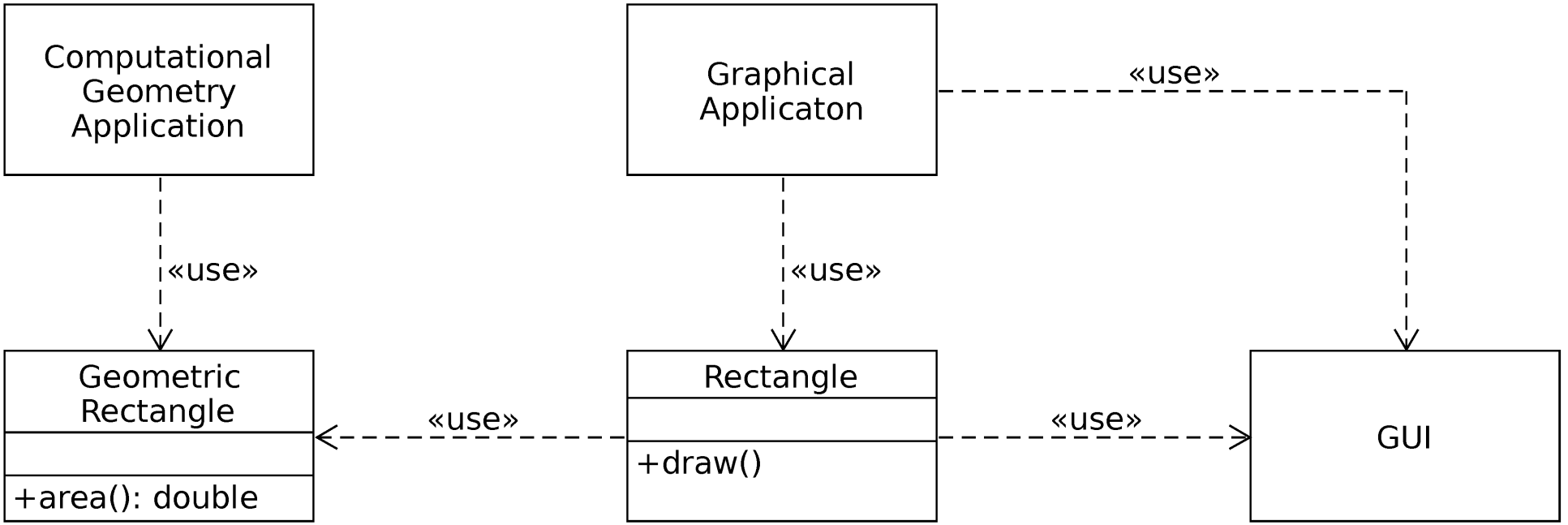


# SOLID – egyszeres felelősség elve (4)

- Példa az elv megsértésére: (folytatás)
  - A számítógépes geometriai alkalmazásnak tartalmaznia kell a grafikus felhasználói felületet.
  - Ha a grafikus alkalmazás miatt változik a `Rectangle` osztály, az szükségessé teheti a számítógépes geometriai alkalmazás összeállításának, tesztelésének és telepítésének megismétlését (*rebuild*, *retest*, *redeploy*).

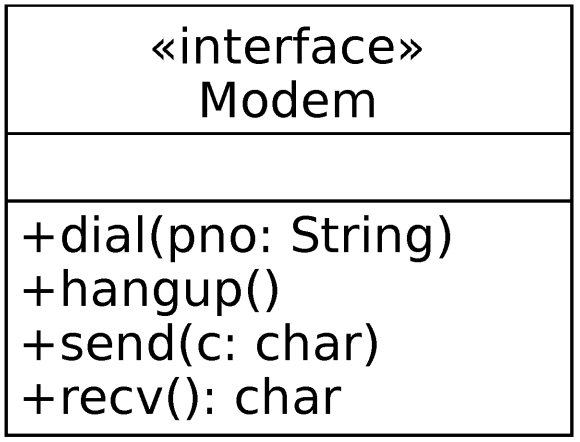
# SOLID – egyszeres felelősség elve (5)

- Az előbbi példa az elvnek megfelelő változata:



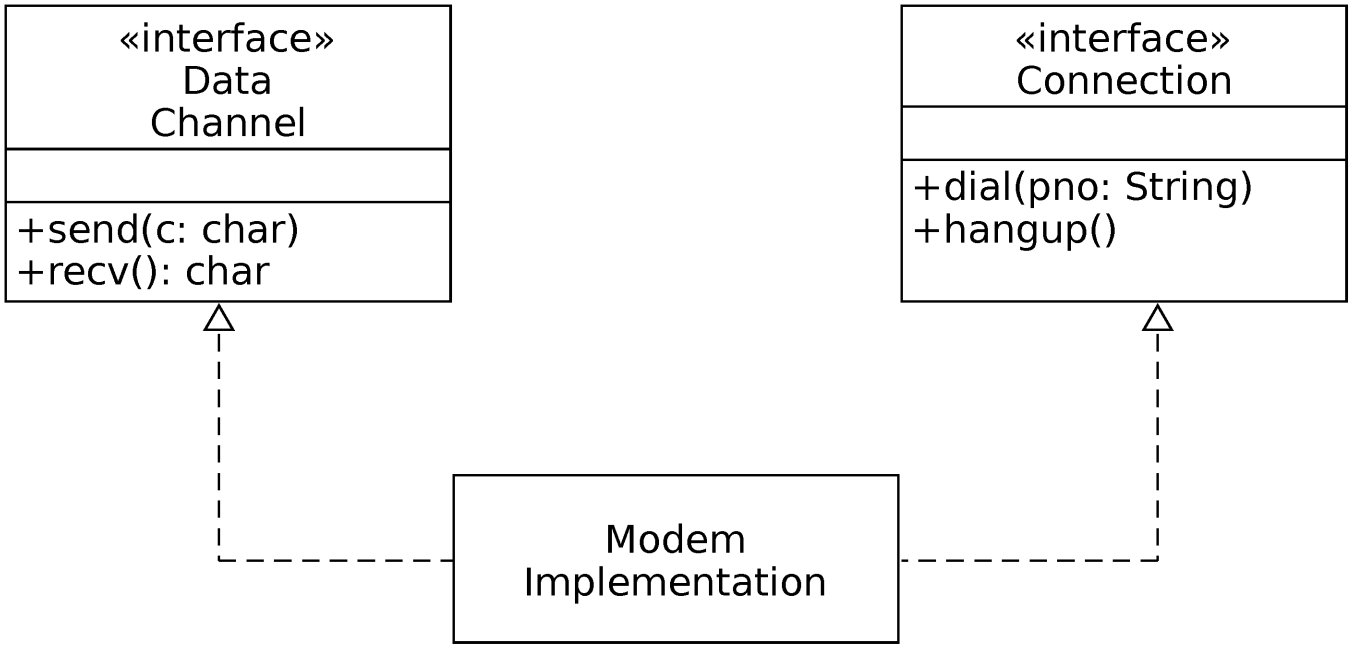
# SOLID – egyszeres felelősség elve (6)

- Példa: Mi a felelősség?
  - Az alábbi `Modem` interfésznél két felelősség állapítható meg: a kapcsolatkezelés és az adatkommunikáció.
  - Hogy érdemes-e őket szétválasztani, az attól függ, hogyan változik az alkalmazás.



# SOLID – egyszeres felelősség elve (7)

- Példa: Mi a felelősség? (folytatás)
  - Ha például úgy változik az alkalmazás, hogy az hatással van a kapcsolatkezelő függvények szignatúrájára, akkor a két felelősséget szét kell választani.



# SOLID – egyszeres felelősség elve (8)

- Az elv megfogalmazásának finomodása:
  - *„A **class** should have only one reason to change.”*
    - Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2002. p. 95.
  - *„… a **class or module** should have one, and only one, reason to change.”*
    - Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. p. 138.

# SOLID – egyszeres felelősség elve (9)

- A vonatkozások szétválasztásának elve és az egyszeres felelősség elve szorosan összefügg. Így a felelősségek befoglaló halmazát alkotják a vonatkozások.
- Ideális esetben minden vonatkozás egy felelősségből áll, mégpedig a fő funkció felelősségéből. Azonban egy felelősségben gyakran több vonatkozás is keveredik.
- A vonatkozások szétválasztásának elve azt nem mondja ki, hogy egy felelősség csak egy vonatkozásból állhat, hanem csak annyit követel meg, hogy a vonatkozásokat el kell különíteni egymástól, vagyis tisztán felismerhetőnek kell lennie, ha több vonatkozás is jelen van.

# SOLID – egyszeres felelősség elve (10)

- Példa az egyszeres felelősség elvének megfelelő, de vonatkozások szétválasztásának elvét megsértő kódra:
  - Artur Trosin. *Separation of Concern vs Single Responsibility Principle ( SoC vs SRP )*. 2009. https://weblogs.asp.net/arturtrosin/separation-of-concern-vs-single-responsibility-principle-soc-vs-srp

# SOLID – nyitva zárt elv (1)

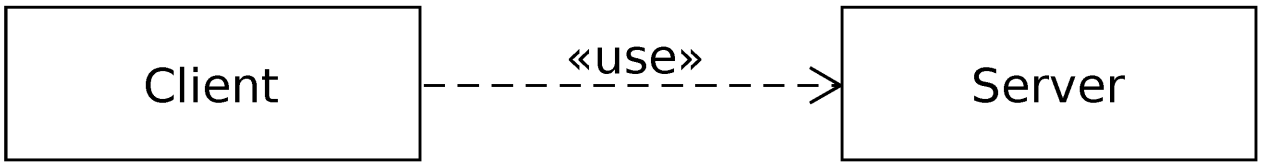
- Bertrand Meyer által megfogalmazott alapelv.
  - Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- A szoftver entitások (osztályok, modulok, függvények, …) legyenek nyitottak a bővítésre, de zártak a módosításra.
- Kapcsolódó tervezési minták: gyártó metódus, helyettes, stratégia, sablonfüggvény, látogató

# SOLID – nyitva zárt elv (2)

- Az elvnek megfelelő modulnak két fő jellemzője van:
  - Nyitott a bővítésre: azt jelenti, hogy a modul viselkedése kiterjeszthető.
  - Zárt a módosításra: azt jelenti, hogy a modul viselkedésének kiterjesztése nem eredményezi a modul forrás- vagy bináris kódjának változását.

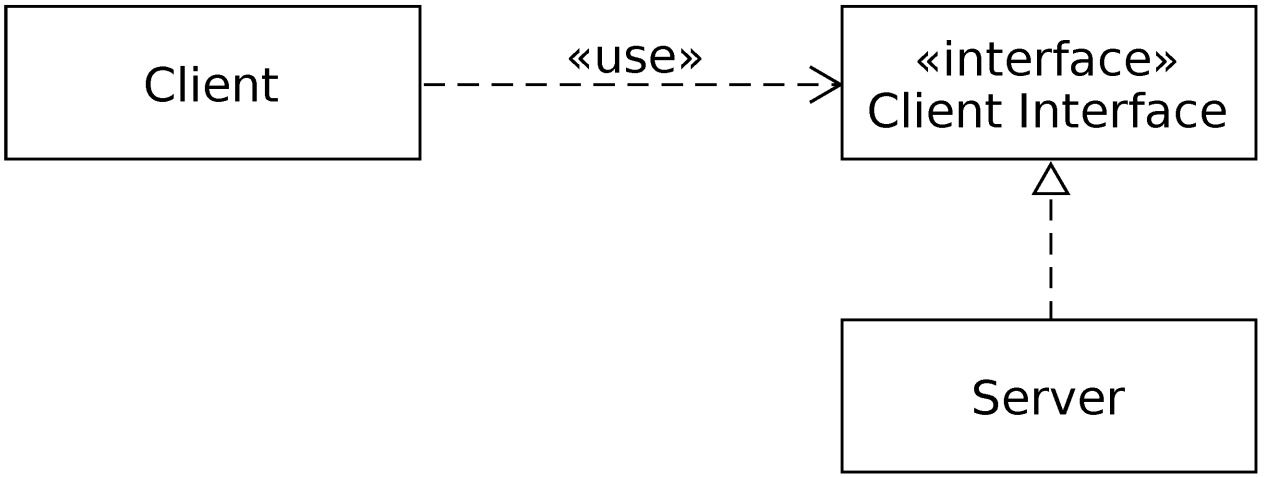
# SOLID – nyitva zárt elv (3)

- Példa az elv megsértésére:
  - A `Client` és a `Server` konkrét osztályok. A `Client` osztály a `Server` osztályt használja. Ha azt szeretnénk, hogy egy `Client` objektum egy különböző szerver objektumot használjon, a `Client` osztályban meg kell változtatni a szerver osztály nevét.



# SOLID – nyitva zárt elv (4)

- Az előbbi példa az elvnek megfelelő változata:



# SOLID – Liskov-féle helyettesítési elv

- Barbara Liskov által megfogalmazott elv.
  - Barbara Liskov. *Keynote Address – Data Abstraction and Hierarchy*. 1987.
- Ha az *S* típus a *T* típus altípusa, nem változhat meg egy program működése, ha benne a *T* típusú objektumokat *S* típusú objektumokkal helyettesítjük.

# SOLID – interfész szétválasztási elv (1)

- Robert C. Martin által megfogalmazott elv:
  - *„Classes should not be forced to depend on methods they do not use.”*
  - Nem szabad arra kényszeríteni az osztályokat, hogy olyan metódusoktól függjenek, melyeket nem használnak.

# SOLID – interfész szétválasztási elv (2)

- **Vastag interfész** (***fat interface***) (Bjarne Stroustrup)
  http://www.stroustrup.com/glossary.html#Gfat-interface
  - *„An interface with more member functions and friends than are logically necessary.”*
  - Az ésszerűen szükségesnél több tagfüggvénnyel és baráttal rendelkező interfész.

# SOLID – interfész szétválasztási elv (3)

- Az interfész szétválasztási elv a vastag interfészekkel foglalkozik.
- A vastag interfészekkel rendelkező osztályok interfészei nem koherensek, melyekben a metódusokat olyan csoportokra lehet felosztani, melyek különböző klienseket szolgálnak ki.
- Az ISP elismeri azt, hogy vannak olyan objektumok, melyekhez nem koherens interfészek szükségesek, de azt javasolja, hogy a kliensek ne egyetlen osztályként ismerjék őket.

# SOLID – interfész szétválasztási elv (4)

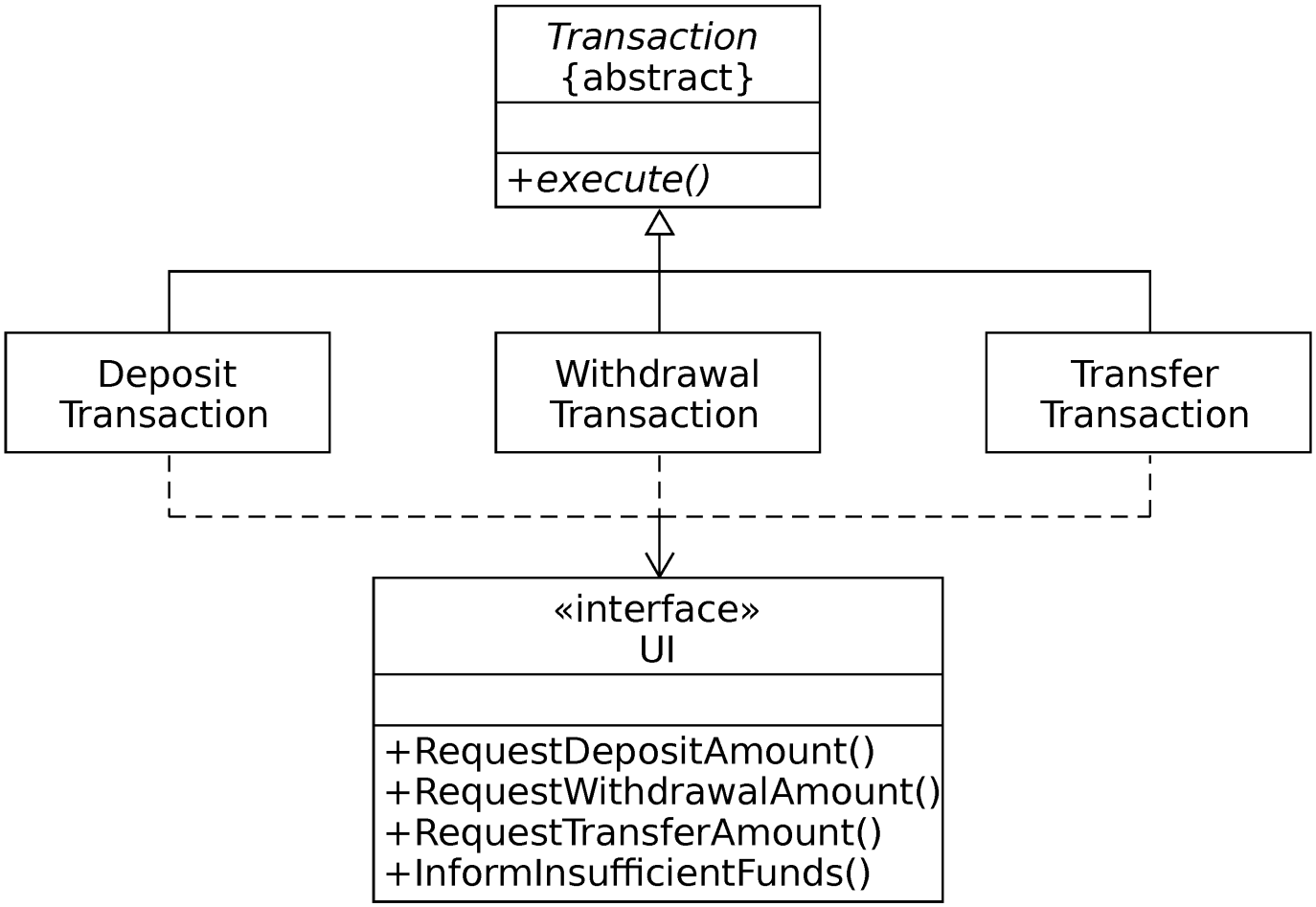
- **Interfész szennyezés** (***interface pollution***):
  - Egy interfész szennyezése szükségtelen metódusokkal.

# SOLID – interfész szétválasztási elv (5)

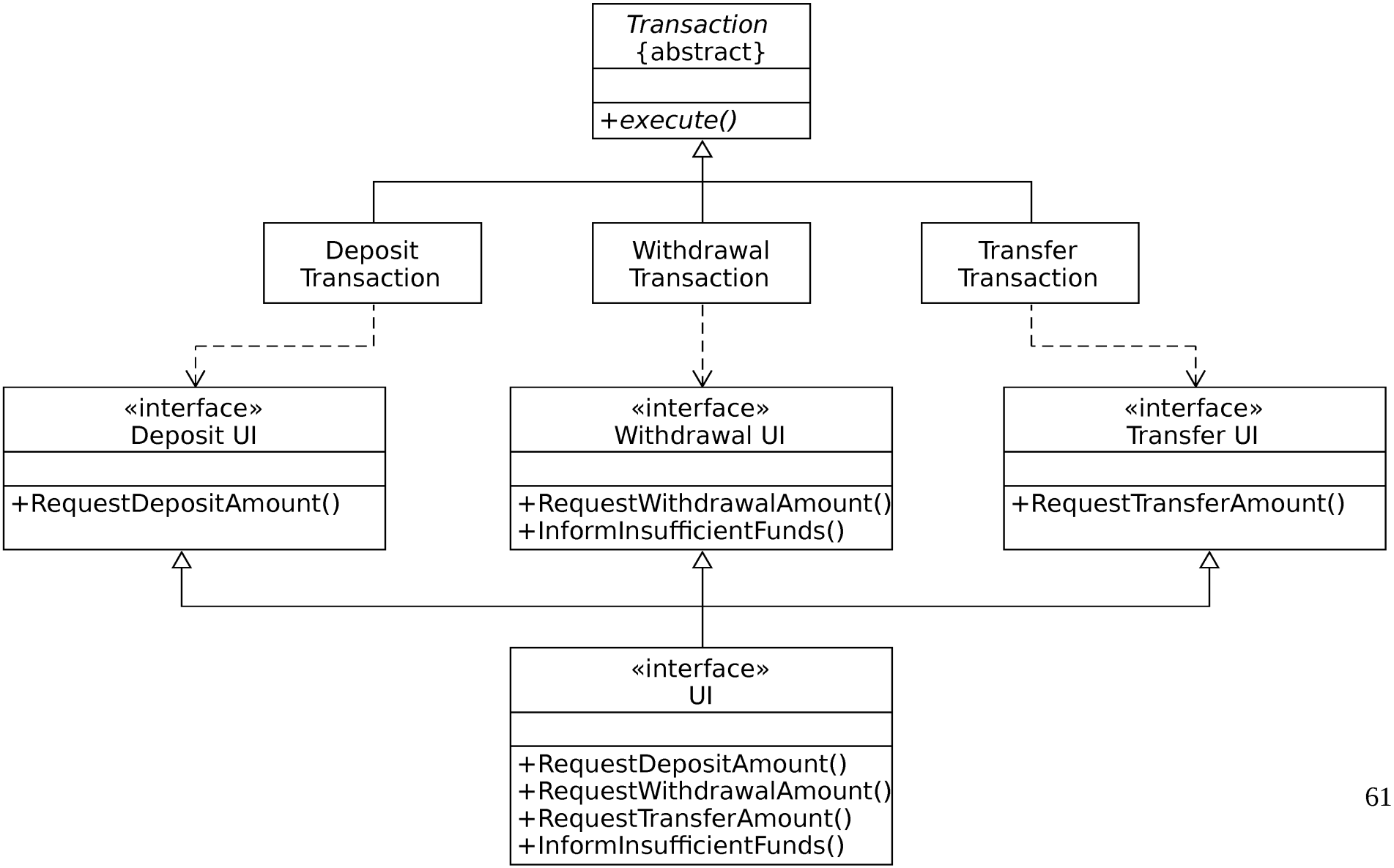
- Amikor egy kliens egy olyan osztálytól függ, melynek vannak olyan metódusai, melyeket a kliens nem használ, más kliensek azonban igen, akkor a többi kliens által az osztályra kényszerített változások hatással lesznek arra a kliense is.
- Ez a kliensek közötti nem szándékos csatoltságot eredményez.

# SOLID – interfész szétválasztási elv (6)

- Példa: ATM (Robert C. Martin)



# SOLID – interfész szétválasztási elv (7)



# SOLID – függőség megfordítási elv (1)

- Robert C. Martin által megfogalmazott elv:
  - Magas szintű modulok ne függjenek alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön.
  - Az absztrakciók ne függjenek a részletektől. A részletek függjenek az absztrakcióktól.

# SOLID – függőség megfordítási elv (2)

- Az elnevezés onnan jön, hogy a hagyományos szoftverfejlesztési módszerek hajlamosak olyan felépítésű szoftvereket létrehozni, melyekben a magas szintű modulok függenek az alacsony szintű moduloktól.
- Kapcsolódó tervezési minta: illesztő

# SOLID – függőség megfordítási elv (3)

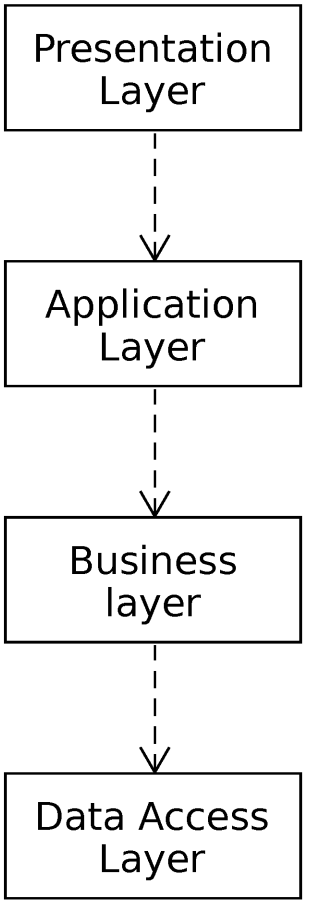
- A magas szintű modulok tartalmazzák az alkalmazás üzleti logikáját, ők adják az alkalmazás identitását. Ha ezek a modulok alacsony szintű moduloktól függenek, akkor az alacsony szintű modulokban történő változásoknak közvetlen hatása lehet a magas szintű modulokra, szükségessé tehetik azok változását is.
- Ez abszurd! A magas szintű modulok azok, melyek meg kellene, hogy határozzák az alacsony szintű modulokat.

# SOLID – függőség megfordítási elv (4)

- A magas szintű modulokat szeretnénk újrafelhasználni. Az alacsony szintű modulok újrafelhasználására elég jó megoldást jelentenek a programkönyvtárak.
- Ha magas szintű modulok alacsony szintű moduloktól függenek, akkor nagyon nehéz az újrafelhasználásuk különféle helyzetekben.
- Ha azonban a magas szintű modulok függetlenek az alacsony szintű moduloktól, akkor elég egyszerűen újrafelhasználhatók.

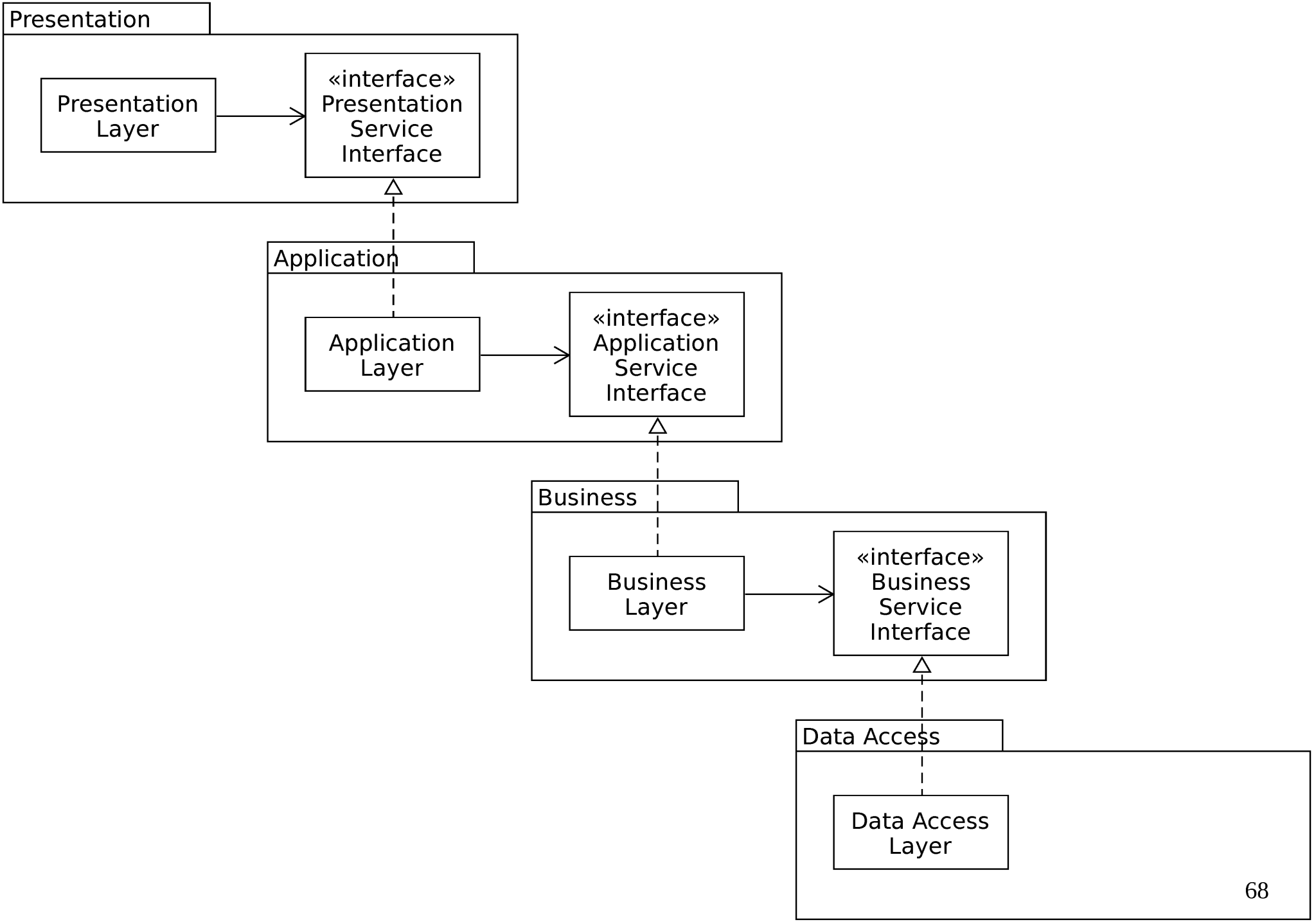
# SOLID – függőség megfordítási elv (5)

- Példa a rétegek architekturális minta hagyományos alkalmazására:



# SOLID – függőség megfordítási elv (6)

- Az előbbi példa az elvnek megfelelő változata:
  - Minden egyes magasabb szintű interfész deklarál az általa igényelt szolgáltatásokhoz egy interfészt.
  - Az alacsonyabb szintű rétegek realizálása ezekből az interfészekből történik.
  - Ilyen módon a felsőbb rétegek nem függenek az alsóbb rétegektől, hanem pont fordítva.


Presentation
Presentation Layer
«interface» Presentation Service Interface
Application
Application Layer
«interface» Application Service Interface
Business
Business Layer
«interface» Business Service Interface
Data Access
Data Access Layer

# SOLID – függőség megfordítási elv (8)

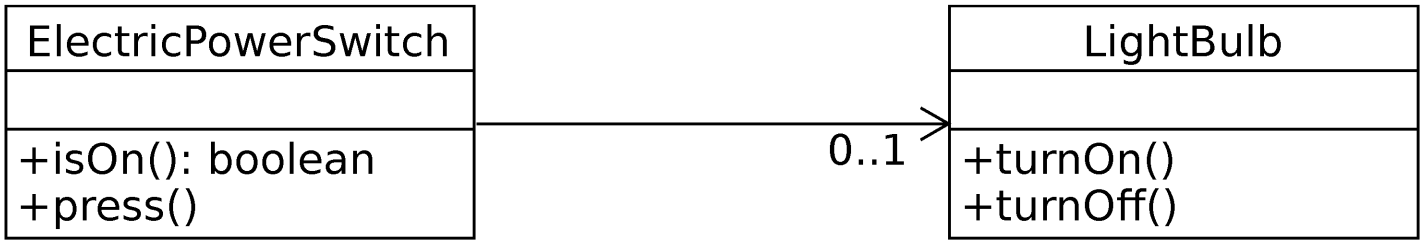
- Az előbbi példa az elvnek megfelelő változata: (folytatás)
  - Nem csupán a függőségek kerültek megfordításra, hanem az interfész tulajdonlás is (*inversion of ownership*).
    - Hollywood elv: Ne hívj, majd mi hívunk. (*Don't call us, we'll call you.*)

# SOLID – függőség megfordítási elv (9)

- Függés absztrakcióktól:
  - Ne függjön a program konkrét osztályoktól, hanem inkább csak absztrakt osztályoktól és interfészektől.
    - Egyetlen változó se hivatkozzon konkrét osztályra.
    - Egyetlen osztály se származzon konkrét osztályból.
    - Egyetlen metódus se írjon felül valamely ősosztályában implementált metódust.
  - A fenti heurisztikát a legtöbb program legalább egyszer megsérti.
  - Nem túl gyakran változó konkrét osztályok esetén (például `String`) megengedhető a függés.

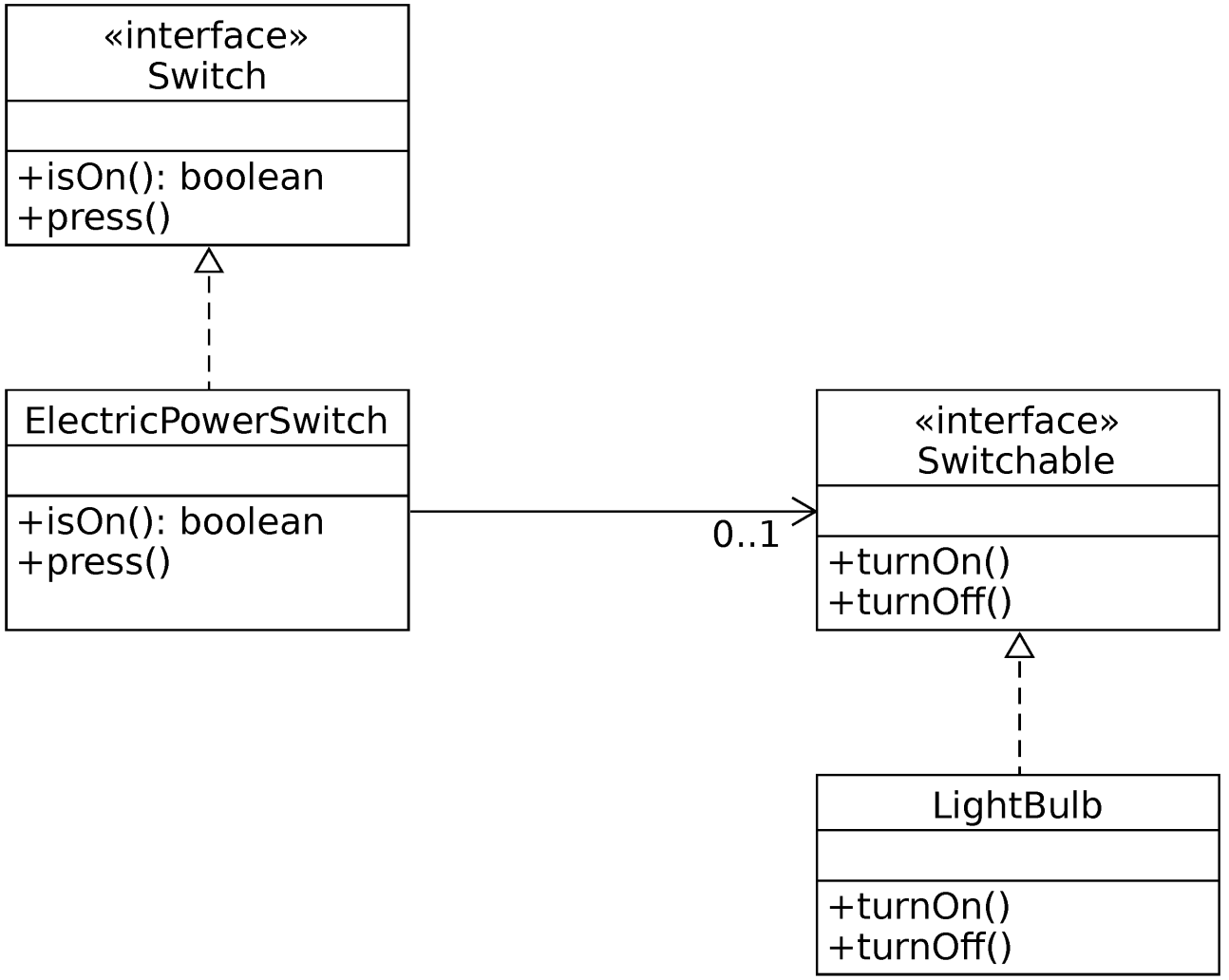
# SOLID – függőség megfordítási elv (10)

- Példa az elv megsértésére:
  - Forrás: https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/



# SOLID – függőség megfordítási elv (11)

- Az előbbi példa az elvnek megfelelő változata:



# Függőség befecskendezés (1)

- Felhasznált irodalom:
  - Dhanji R. Prasanna. *Dependency Injection Design patterns using Spring and Guice*. Manning, 2009. https://www.manning.com/books/dependency-injection
  - Mark Seemann. *Dependency Injection in .NET*. Manning, 2011. https://www.manning.com/books/dependency-injection-in-dot-net
  - Steven van Deursen, Mark Seemann. *Dependency Injection*. Second Edition. Manning, 2019. https://www.manning.com/books/dependency-injection-in-dot-net-second-edition

# Függőség befecskendezés (2)

- A függőség befecskendezés (DI – *dependency injection*) kifejezés Martin Fowlertől származik.
  - Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. https://martinfowler.com/articles/injection.html
- A vezérlés megfordítása (IoC – *inversion of control*) nevű architekturális minta alkalmazásának egy speciális esete.
  - Martin Fowler. *InversionOfControl*. 2005. https://martinfowler.com/bliki/InversionOfControl.html

# Függőség befecskendezés (3)

- Definíció (Seemann):
  - *„Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.”*
  - A függőség befecskendezés olyan szoftvertervezési elvek és minták összessége, melyek lazán csatolt kód fejlesztését teszik lehetővé.
- A lazán csatoltság a kód karbantarthatóságát javítja.

# Függőség befecskendezés (4)

- Egy objektumra egy olyan szolgáltatásként tekintünk, melyet más objektumok kliensként használnak.
- Az objektumok közötti kliens-szolgáltató kapcsolatot függésnek nevezzük. Ez a kapcsolat tranzitív.

# Függőség befecskendezés (5)

- **Függőség** (***dependency***): egy kliens által igényelt szolgáltatást jelent, mely a feladatának ellátásához szükséges.
- **Függő** (***dependent***): egy kliens objektum, melynek egy függőségre vagy függőségekre van szüksége a feladatának ellátásához.
- **Objektum gráf** (***object graph***): függő objektumok és függőségeik egy összessége.
- **Befecskendezés** (***injection***): egy kliens függőségeinek megadását jelenti.
- **DI konténer** (***DI container***): függőség befecskendezési funkcionalitást nyújtó programkönyvtár.
  - Az ***Inversion of Control (IoC) container*** kifejezést is használják rájuk.

# Függőség befecskendezés (6)

- A függőség befecskendezés objektum gráfok hatékony létrehozásával, ennek mintáival és legjobb gyakorlataival foglalkozik.
- A DI keretrendszerek lehetővé teszik, hogy a kliensek a függőségeik létrehozását és azok befecskendezését külső kódra bízzák.

# Függőség befecskendezés (7)

- Példa: nincs függőség befecskendezés

```
public interface SpellChecker {

  public boolean check(String text);

}

public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor() {
    spellChecker = new HungarianSpellChecker();
  }

  // ...

}
```

# Függőség befecskendezés (8)

- Függőség befecskendezés konstruktorral (*constructor injection*):

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // ...

}
```

# Függőség befecskendezés (9)

- Függőség befecskendezés beállító metódussal (*setter injection*):

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor() {}

  public void setSpellChecker(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // ...

}
```

# Függőség befecskendezés (10)

- Függőség befecskendezés beállító interfésszel (*interface injection*):

```
public interface SpellCheckerSetter {

  void setSpellChecker(SpellChecker spellChecker);

}

public class TextEditor implements SpellCheckerSetter {

  private SpellChecker spellChecker;

  public TextEditor() {}

  @Override
  public void setSpellChecker(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // ...

}
```

# Függőség befecskendezés (11)

- C++ keretrendszerek:
  - *[Boost].DI* (licenc: *Boost Software License*)
    http://boost-experimental.github.io/di/
  - *Fruit* (licenc: *Apache License 2.0*)
    https://github.com/google/fruit
  - *Hypodermic* (licenc: *MIT License*)
    https://github.com/ybainier/Hypodermic
  - ...

# Függőség befecskendezés (12)

- Java:
  - *JSR 330: Dependency Injection for Java*
    https://www.jcp.org/en/jsr/detail?id=330
    - Szabványos annotációk biztosítása függőség befecskendezéshez.
    - A Java EE 6-ban jelent meg.
    - `javax.inject` csomag.
      - Lásd:
        https://javaee.github.io/javaee-spec/javadocs/javax/inject/package-summary.html
    - A specifikációt implementáló DI keretrendszer szükséges használatához!
      - Például: *Dagger*, *Guice*, *HK2*, *Spring Framework*, …

# Függőség befecskendezés (13)

- Java keretrendszerek:
  - *Dagger* (licenc: *Apache License 2.0*) https://google.github.io/dagger/
  - *Guice* (licenc: *Apache License 2.0*) https://github.com/google/guice
  - *HK2* (licenc: CDDL + GPLv2) https://hk2.java.net/
  - *Java EE CDI* https://javaee.github.io/tutorial/cdi-basic.html
  - *Spring Framework* (licenc: *Apache License 2.0*) https://projects.spring.io/spring-framework/ http://www.vogella.com/tutorials/SpringDependencyInjection/article.html
  - …

# Függőség befecskendezés (14)

- .NET keretrendszerek:
  - *Castle Windsor* (licenc: *Apache License 2.0*)
    https://github.com/castleproject/Windsor
  - *Ninject* (licenc: *Apache License 2.0*)
    http://www.ninject.org/
  - *StructureMap* (license: *Apache License 2.0*)
    http://structuremap.github.io/
  - …