Objektumorientált tervezési alapelvek

Jeszenszky Péter
jeszenszky.peter@inf.unideb.hu

Utolsó módosítás: 2023. április 16.

Statikus kódelemzés

- A **statikus kódelemzés** (***static code analysis***) a programkód elemzésének folyamata, mely a kód végrehajtása nélkül történik.
  - Az elemezés irányulhat: hibák észlelésére; annak ellenőrzésére, hogy a kód megfelel-e egy kódolási szabványnak, …
- **Statikus kódelemző (eszköz)** (***static code analyzer***, ***static code analysis tool***): statikus kódelemzést végző automatikus eszköz.

Statikus kódelemző eszközök (1)

- **C#**:
  - *InferSharp* (programozási nyelv: C#; licenc: *MIT License*) https://github.com/microsoft/infersharp
  - *Roslyn Analyzers* (programozási nyelv: C#; licenc: *MIT License*) https://github.com/dotnet/roslyn-analyzers
  - *Roslynator* (programozási nyelv: C#; licenc: *Apache License 2.0*) https://github.com/JosefPihrt/Roslynator
- **C++**:
  - *Cppcheck* (programozási nyelv: C++; licenc: GPLv3) https://cppcheck.sourceforge.io/ https://github.com/danmar/cppcheck

Statikus kódelemző eszközök (2)

- **ECMAScript/JavaScript**:
  - *ESLint* (programozási nyelv: JavaScript; licenc: *MIT License*) https://eslint.org/ https://github.com/eslint/eslint
  - *JSHint* (programozási nyelv: JavaScript; licenc: *MIT License*) https://jshint.com/ https://github.com/jshint/jshint
  - *JSLint* (programozási nyelv: JavaScript; licenc: *Unlicense*) https://www.jslint.com/ https://github.com/jslint-org/jslint
  - *RSLint* (programozási nyelv: Rust; licenc: *MIT License*) https://rslint.org/ https://github.com/rslint/rslint

Statikus kódelemző eszközök (3)

- **Java**:
  - *Checkstyle* (programozási nyelv: Java; licenc: LGPLv2.1) https://checkstyle.org/ https://github.com/checkstyle/checkstyle
  - *Error Prone* (programozási nyelv: Java; licenc: *Apache License 2.0*) https://errorprone.info/ https://github.com/google/error-prone
  - *NullAway* (programozási nyelv: Java; licenc: *MIT License*) https://github.com/uber/NullAway
  - *SpotBugs* (programozási nyelv: Java; licenc: LGPLv2.1) https://spotbugs.github.io/ https://github.com/spotbugs/spotbugs

Statikus kódelemző eszközök (4)

- **Python**:
  - *Prospector* (programozási nyelv: Python; licenc: GPLv2) http://prospector.landscape.io/ https://github.com/PyCQA/prospector/
  - *Pylint* (programozási nyelv: Python; licenc: GPLv2) https://pylint.org/ https://github.com/PyCQA/pylint

Statikus kódelemző eszközök (5)

- **Több nyelvet támogató eszközök**:
  - *Coala* (programozási nyelv: Python; licenc: AGPLv3) https://coala.io/ https://github.com/coala/coala
  - *Infer* (programozási nyelv: OCaml; licenc: *MIT License*) https://fbinfer.com/ https://github.com/facebook/infer
  - *PMD* (programozási nyelv: Java; licenc: *BSD License*) https://pmd.github.io/ https://github.com/pmd/pmd
  - *Semgrep* (programozási nyelv: OCaml; licenc: LGPLv2.1) https://semgrep.dev/ https://github.com/returntocorp/semgrep

Statikus kódelemző eszközök (6)

- További eszközökért lásd: https://github.com/analysis-tools-dev/static-analysis

PMD (1)

- Statikus kódelemző (programozási nyelv: Java, licenc: BSD-stílusú) https://pmd.github.io/ https://github.com/pmd/pmd
  - Támogatott programozási nyelvek: ECMAScript (JavaScript), Java, Scala, …

PMD (2)

- Eszköz integráció:
  - **Apache Maven**:
    - *Apache Maven PMD Plugin* (licenc: *Apache License 2.0*) https://maven.apache.org/plugins/maven-pmd-plugin/ https://github.com/apache/maven-pmd-plugin
  - **Gradle**:
    - *The PMD Plugin* (licenc: *Apache License 2.0*) https://docs.gradle.org/current/userguide/pmd_plugin.html
  - **Eclipse IDE**:
    - *pmd-eclipse-plugin* (licenc: BSD-stílusú) https://marketplace.eclipse.org/content/pmd-eclipse-plugin https://github.com/pmd/pmd-eclipse-plugin
    - *PMD Plug-in* (licenc: *Eclipse Public License 2.0*) https://eclipse-pmd.github.io/ https://github.com/eclipse-pmd/eclipse-pmd
  - **IntelliJ IDEA**:
    - *PMDPlugin* (licenc: *MIT License*) https://plugins.jetbrains.com/plugin/1137-pmd https://github.com/amitdev/PMD-Intellij
    - *QAPlug* (licenc: nem szabad) https://qaplug.com/

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.
- Az ellenkezője a WET.
  - *„We enjoy typing”*, *„write everything twice”*, *„waste everyone's time”*, …

DRY (2)

- Forrás:
  - Andrew Hunt, David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
  - David Thomas, Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition*. Addison Wesley, 2019. https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/
    - Ingyenes fejezet: *DRY – The Evils of Duplication* https://media.pragprog.com/titles/tpp20/dry.pdf

DRY (3)

- 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*, *duplicate code*), *copy-and-paste programming*

DRY (4)

- A **kódismétlés** (***duplicate code***) azonos (vagy nagyon hasonló) forráskódrész, mely egynél többször fordul elő egy programban.
- Nem minden kódismétlés információ ismétlés!

DRY (5)

- PMD támogatás: *Copy/Paste Detector* (CPD)
  - *Finding duplicated code with CPD*
    https://pmd.github.io/latest/pmd_userdocs_cpd.html
  - Támogatott programozási nyelvek: C/C++, C#, ECMAScript (JavaScript), Java, Kotlin, Python, Scala, …
    - Lásd:
      https://pmd.github.io/latest/pmd_userdocs_cpd.html#supported-languages
- IntelliJ IDEA:
  - *Analyze duplicates*
    https://www.jetbrains.com/help/idea/analyzing-duplicates.html

DRY (6)

- A DRY elv megsértései nem mindig kódismétlés formájában jelennek meg.
  - A DRY elv az információk megismétléséről szól. A tudás egy darabkája két teljesen eltérő módon is kifejezhető két különböző helyen.
  - Példa (Thomas & Hunt, 2019):

```
class Line {
  Point start;
  Point end;
  double length; // DRY violation
}
```

DRY (7)

- Példa (folytatás):
  - Az elv megsértése kiküszöbölhető a `length` adattag egy metódusra való kicserélésével:

```
class Line {
  Point start;
  Point end;

  double length() {
    return start.distanceTo(end);
  }
}
```

DRY (8)

- Példa (folytatás):
  - A jobb teljesítmény érdekében választható a DRY elv megsértése.
    - Ilyenkor az elv megszegését ajánlott a külvilág elől elrejteni.

```
class Line {

  private Point start;
  private Point end;
  private double length;

  public Line(Point start, Point end) {
    this.start = start;
    this.end = end;
    calculateLength();
  }

  public void setStart(Point p) {
    this.start = p;
    calculateLength();
  }

  public void setEnd(Point p) {
    this.end = p;
    calculateLength();
  }

  public Point getStart() { return start; }

  public Point getEnd() { return end; }

  public double getLength() { return length; }

  private void calculateLength() { this.length = start.distanceTo(end); }
}
```

DRY (10)

- Reprezentációs ismétlés (Thomas & Hunt, 2019):
  - A kód gyakran függ a külvilágtól: például API-kon keresztül más programkönyvtáraktól, külső adatforrások adataitól, mely mindig a DRY elv valamiféle megsértését vonja maga után: a kódnak olyan tudással kell rendelkeznie, mely a külső dologban is ott van.
    - Ismernie kell az API-t, a sémát, vagy a hibakódok jelentését.

DRY (11)

- Reprezentációs ismétlés (Thomas & Hunt, 2019):
  - Ez az ismétlés elkerülhetetlen.
  - Eszközök, melyek segítenek megbirkózni az ilyen fajta ismétlésekkel:
    - Sémákból kódot generáló eszközök (például JAXB, JPA)
    - OpenAPI
    - …

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.”*

YAGNI (1)

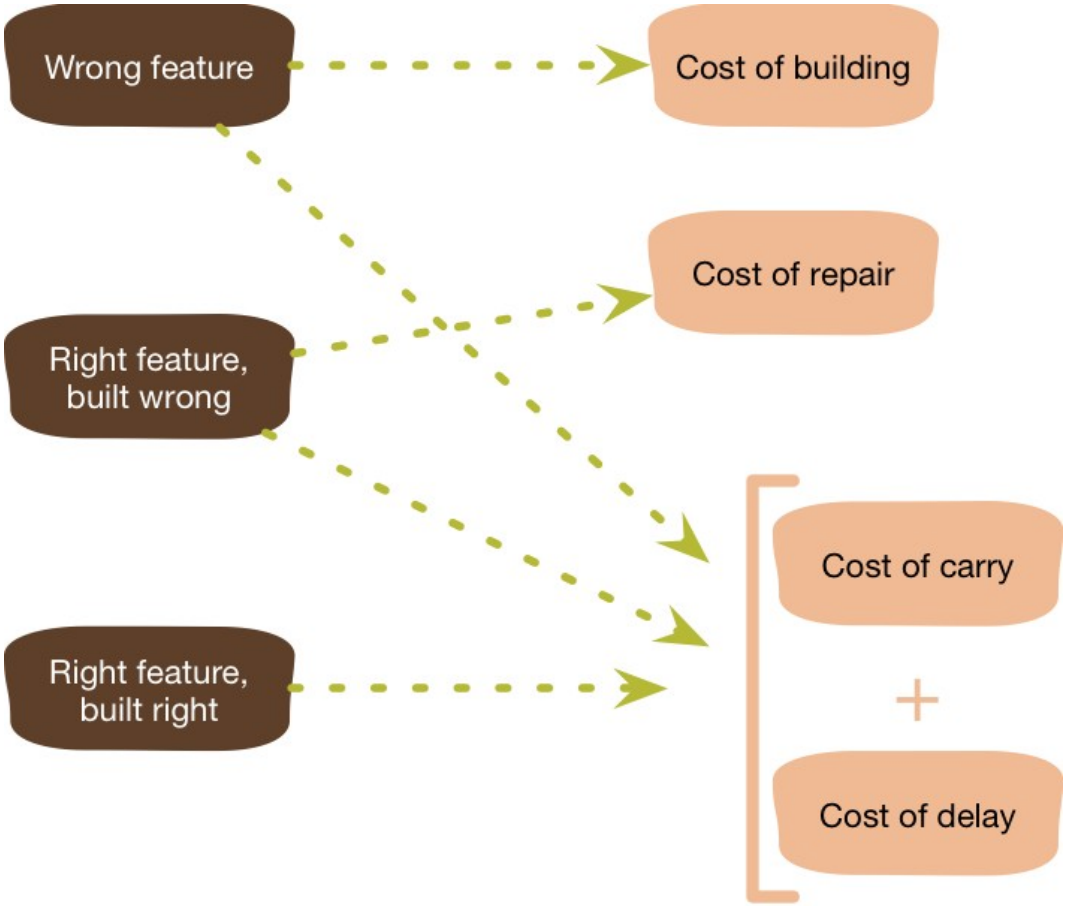
- A „*You Aren't Gonna Need It*” („nem lesz rá szükséged”) rövidítése.
- Az extrém programozás (XP) egy alapelve.

YAGNI (2)

- *„Mindig akkor implementálj valamit, amikor tényleg szükséged van rá, soha ne akkor, amikor csak sejted, hogy kell.”*
  - Lásd: Ronald E. Jeffries. *You’re NOT gonna need it!* Apr 4, 1998. https://ronjeffries.com/xprog/articles/practices/pracnotneed/
- Lásd még:
  - Martin Fowler. *Yagni*. 26 May 2015. https://martinfowler.com/bliki/Yagni.html

YAGNI (3)

- Egy olyan lehetőség kifejlesztésének költségei, mely jelenleg nem szükséges (Martin Fowler):



YAGNI (4)

- A YAGNI alapelv csak azon képességekre vonatkozik, melyek egy feltételezett lehetőség támogatásához kerülnek beépítésre a szoftverbe, nem vonatkozik a szoftver módosítását könnyítő törekvésekre.
- A YAGNI csak akkor járható stratégia, ha a kód könnyen változtatható.

Csatoltság (1)

- **Csatoltság** (***coupling***): egy szoftvermodul függésének mértéke egy másik szoftvermodultól.
  - Más szóval, a szoftvermodulok közötti csatoltság annak mértéke, hogy mennyire szoros a kapcsolatuk.
  - A csatoltság laza vagy szoros lehet.
- Hivatkozás:
  - Joseph Ingeno. *Software Architect's Handbook*. Packt Publishing, 2018. https://packtpub.com/product/software-architects-handbook/9781788624060

Csatoltság (2)

- Szoros csatoltság:
  - A bonyolultságot növeli, mely megnehezíti a kód módosítását, tehát a karbantarthatóságot csökkenti.
  - Az újrafelhasználhatóságot is csökkenti.

Csatoltság (3)

- Laza csatoltság:
  - Lehetővé teszi a fejlesztők számára a nyitva zárt elvnek megfelelő kód írását, azaz a kódot kiterjeszthetővé teszi.
  - Kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.
  - Lehetővé teszi a párhuzamos fejlesztést.

Demeter törvénye (1)

- Demeter törvényét (*Law of Demeter*) Ian Holland javasolta 1987-ben.
- A törvény a Demeter projektről kapta a nevét, melyben Holland dolgozott, amikor felfedezte.
- 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)

- Hivatkozások:
  - 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
  - Karl Lieberherr. *Law of Demeter: Principle of Least Knowledge*. http://www.ccs.neu.edu/home/lieber/LoD.html

Demeter törvénye (3)

- 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 (4)

- Osztályokra vonatkozó változat (*class form*):
  - Egy *C* osztály egy *M* metódusa csak az alábbi osztályok és ősosztályaik tagjait (metódusait, adattagjait) 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 (5)

- Objektumokra vonatkozó változat (*object form*):
  - Egy *O* objektum egy *M* metódusa csak az alábbi objektumok tagjait (metódusait, adattagjait) használhatja:
    - *O*
    - *O* adattagjai
    - *M* argumentum objektumai
    - Közvetlenül *M*-ben létrehozott/példányosított objektumok
    - Globális változókban lévő objektumok
  - Csak futásidőben ellenőrizhető.

Demeter törvénye (6)

- 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 (7)

- Példa a törvény megsértésére a PMD dokumentációból:

```
public class Foo {

  public void example(Bar b) {
    // Ez a metódushívás rendben van mivel b a metódus paramétere.
    C c = b.getC();

    // Ez a metódushívás megszegi a törvényt, mivel a B-től kapott
    // c-t használjuk.
    c.doIt();

    // Ez is megszegi a törvényt (valójában ugyanaz, mint az előző,
    // csak metódusláncolást használ ideiglenes változó helyet).
    b.getC().doIt();

    // Ez a metódushívás rendben van, mivel D egy új példányát
    // lokálisan hozzuk létre.
    new D().doSomethingElse();
  }

}
```

Demeter törvénye (8)

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

Demeter törvénye (9)

- Megengedett-e a metódusláncolás?
  - Például:
    - Építő tervezési minta
    - Folyékony interfész (*fluent interface*)
      - Lásd: Martin Fowler. *FluentInterface*. 20 December 2005. https://martinfowler.com/bliki/FluentInterface.html

Demeter törvénye (10)

- Példák metódusláncolásra:

```
// Építő tervezési minta:
import java.util.StringJoiner;

String s = new StringJoiner(",", "[", "]")
  .add("George")
  .add("John")
  .add("Paul")
  .add("Ringo")
  .toString();
```

```
// Folyékony interfész:
import java.time.LocalTime;

LocalTime time = LocalTime.now().plusHours(1).plusMinutes(45);
```

GoF alapelvek (1)

- Felhasznált irodalom:
  - 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)

- 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.

GoF alapelvek (6)

- Az öröklődés hátrányai (folytatás):
  - 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)

- Objektum-összetétel:
  - 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.
  - 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 alapelvek (9)

- Az objektum összetétel hátrányai:
  - 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.

GoF alapelvek (10)

- Példa:
  - Joshua Bloch. *You should favor composition over inheritance in Java. Here's why.* July 14, 2022. https://blogs.oracle.com/javamagazine/post/java-inheritance-composition

SOLID (1)

- Robert C. Martin („Bob bácsi”) által megfogalmazott/rendszerezett/népszerűsített objektumorientált programozási és tervezési alapelvek.
  - Blog: https://blog.cleancoder.com/
  - https://github.com/unclebob
  - Uncle Bob. *Getting a SOLID start*. 2009. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start
- Felhasznált 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)

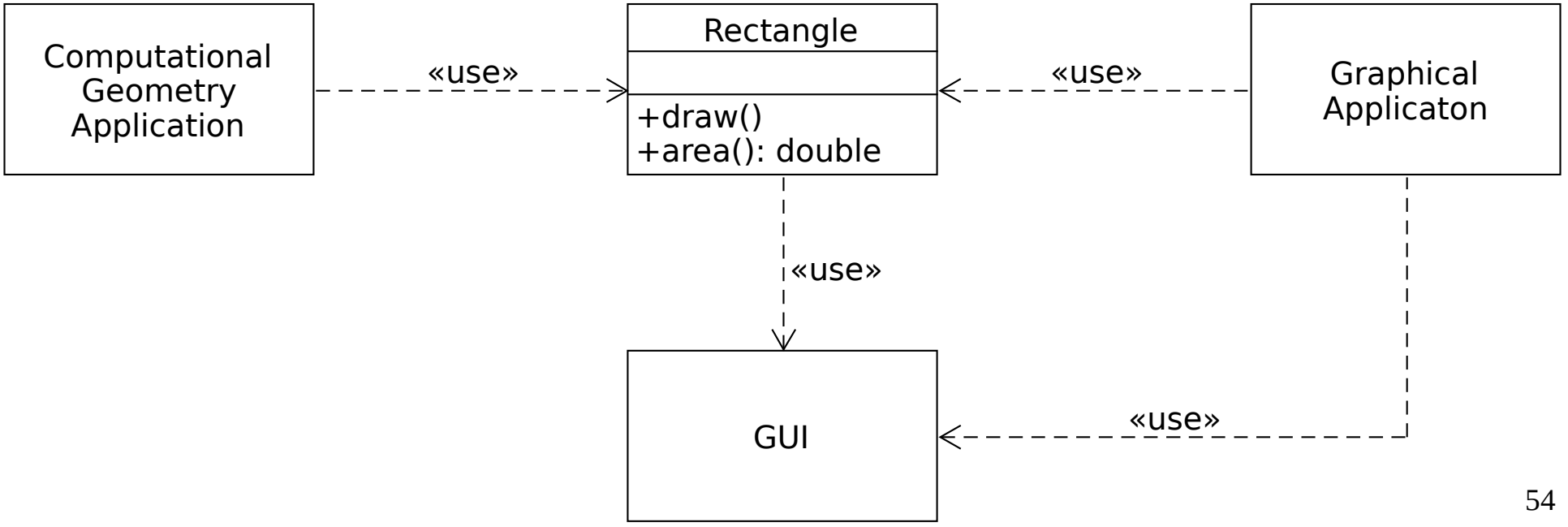
- Robert C. Martin által megfogalmazott elv:
  - *„A class should have only one reason to change.”*
  - Egy osztálynak csak egy oka legyen a változásra.
- Kapcsolódó tervezési minták: díszítő, felelősséglánc

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.

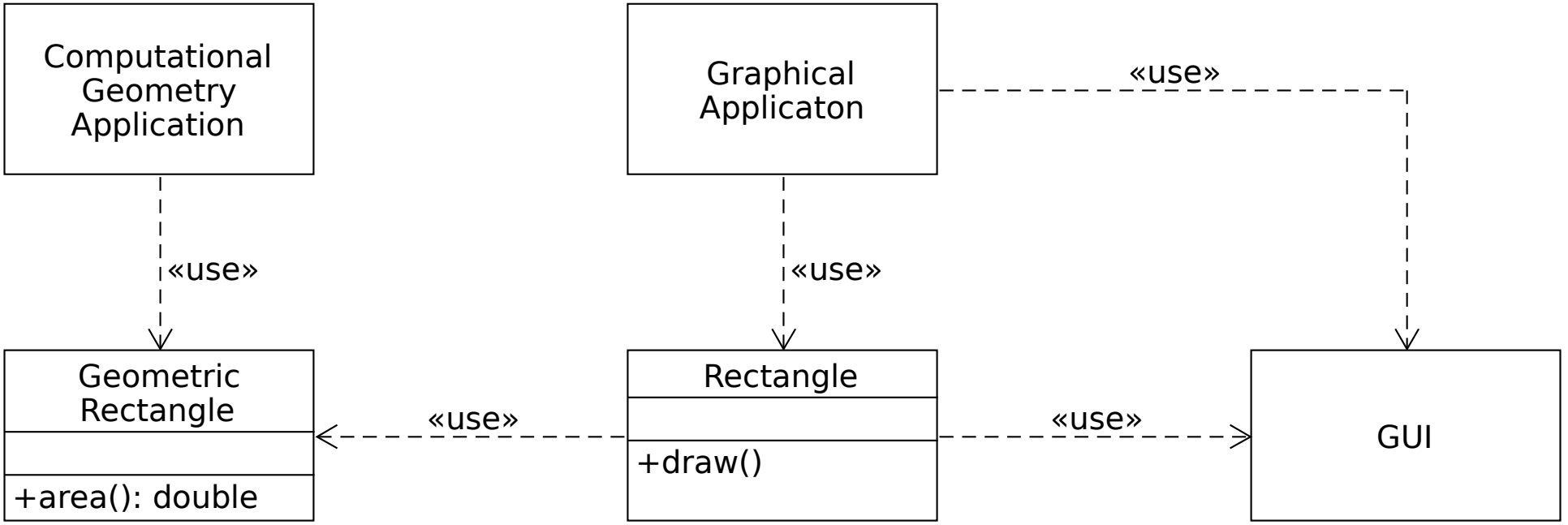


SOLID – egyszeres felelősség elve (4)

- Példa az elv megsértésére: (folytatás)
  - Problémák:
    - 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)

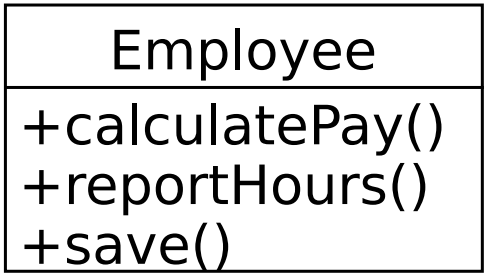
- Az elv megfogalmazásának finomodása:
  - *„A **class** should have only one reason to change.”*
    - Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2002. p. 95.
  - *„… a **class or module** should have one, and only one, reason to change.”*
    - Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. p. 138.
  - *„A module should be responsible to one, and only one, actor.”*
    - Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017. p. 62.

SOLID – egyszeres felelősség elve (7)

- A szoftverek aktorok kielégítése céljából változnak.
  - Az „aktor” kifejezést itt emberek (például felhasználók) egy olyan csoportjára használjuk, akik azt akarják, hogy a szoftver ugyanúgy változzon.
- Az elv tehet így fogalmazható újra:
  - Egy modul egy, és csak egyetlen aktornak van alárendelve.

SOLID – egyszeres felelősség elve (8)

- Példa (Robert C. Martin):
  - Az alábbi `Employee` osztály megszegi az egyszeres felelősség elvét, mivel a három metódus nagyon különböző aktoroknak van alávetve:
    - `calculatePay()`: a bérosztály határozza meg
    - `reportHours()`: a munkaügyi osztály határozza meg
    - `save()`: az adatbázis adminisztrátorok határozzák meg



SOLID – nyitva zárt elv (1)

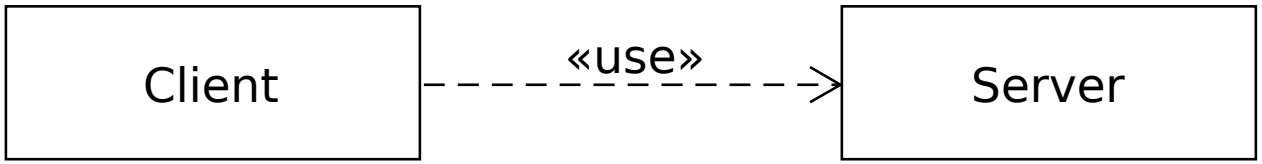
- Bertrand Meyer által megfogalmazott alapelv.
  - Lásd: Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- A szoftver entitások (osztályok, modulok, függvények, …) legyenek nyitottak a bővítésre, de zártak a módosításra.
- Kapcsolódó tervezési minták: gyártó metódus, helyettes,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

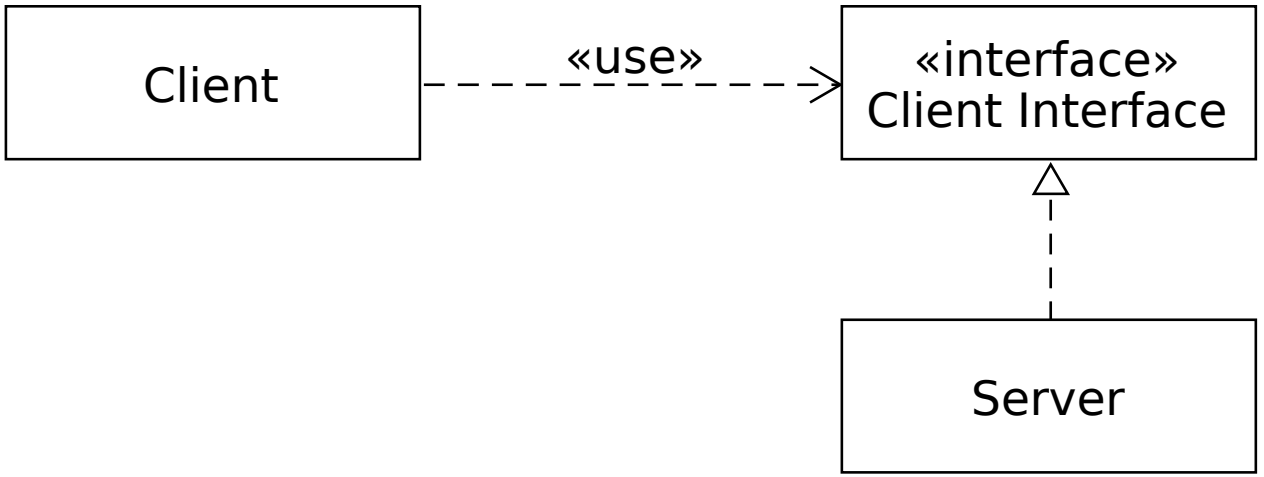
SOLID – nyitva zárt elv (3)

- 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) https://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)

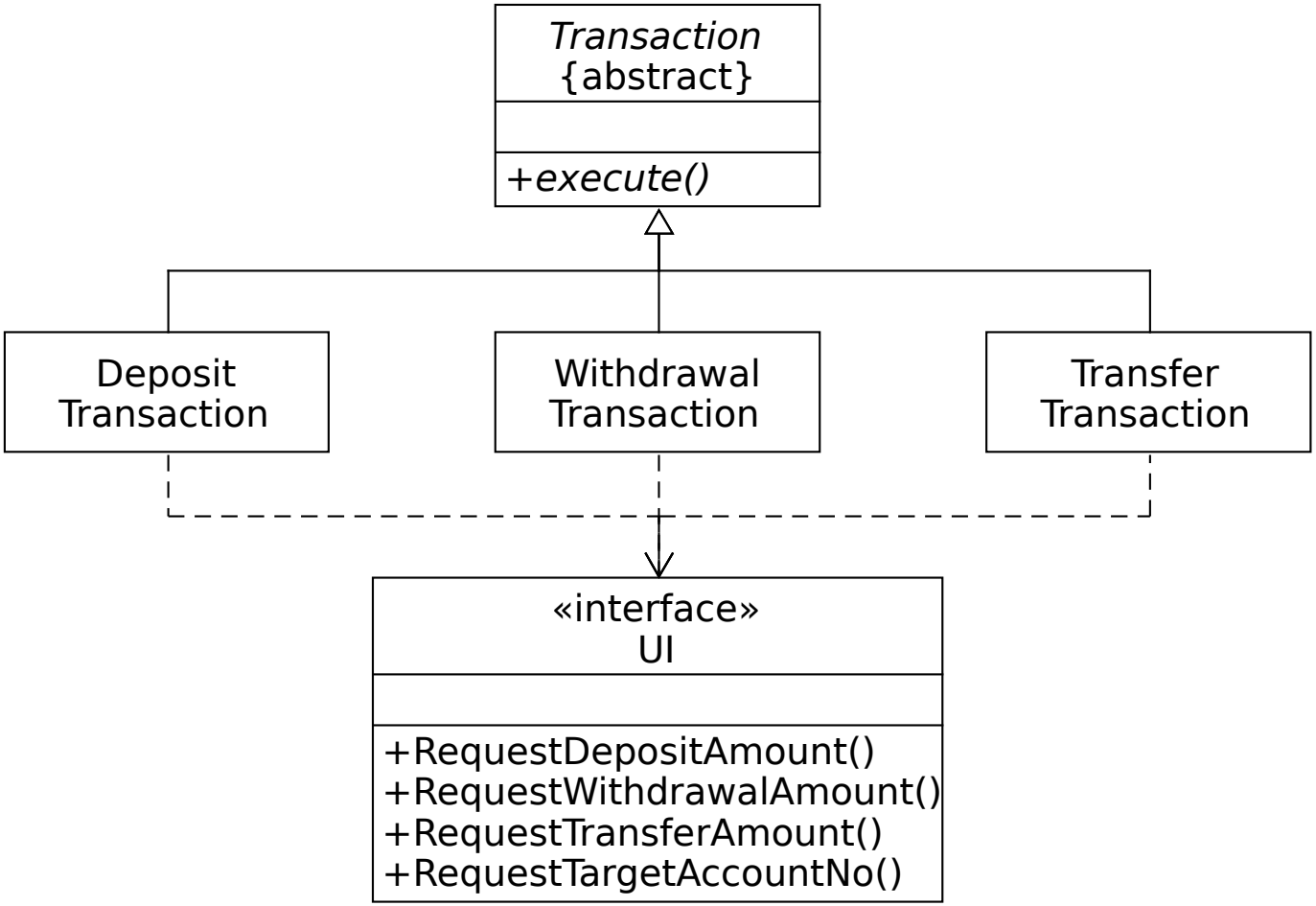
- **Interfész szennyezés** (***interface pollution***):
  - Egy interfész szennyezése szükségtelen metódusokkal.

SOLID – interfész szétválasztási elv (5)

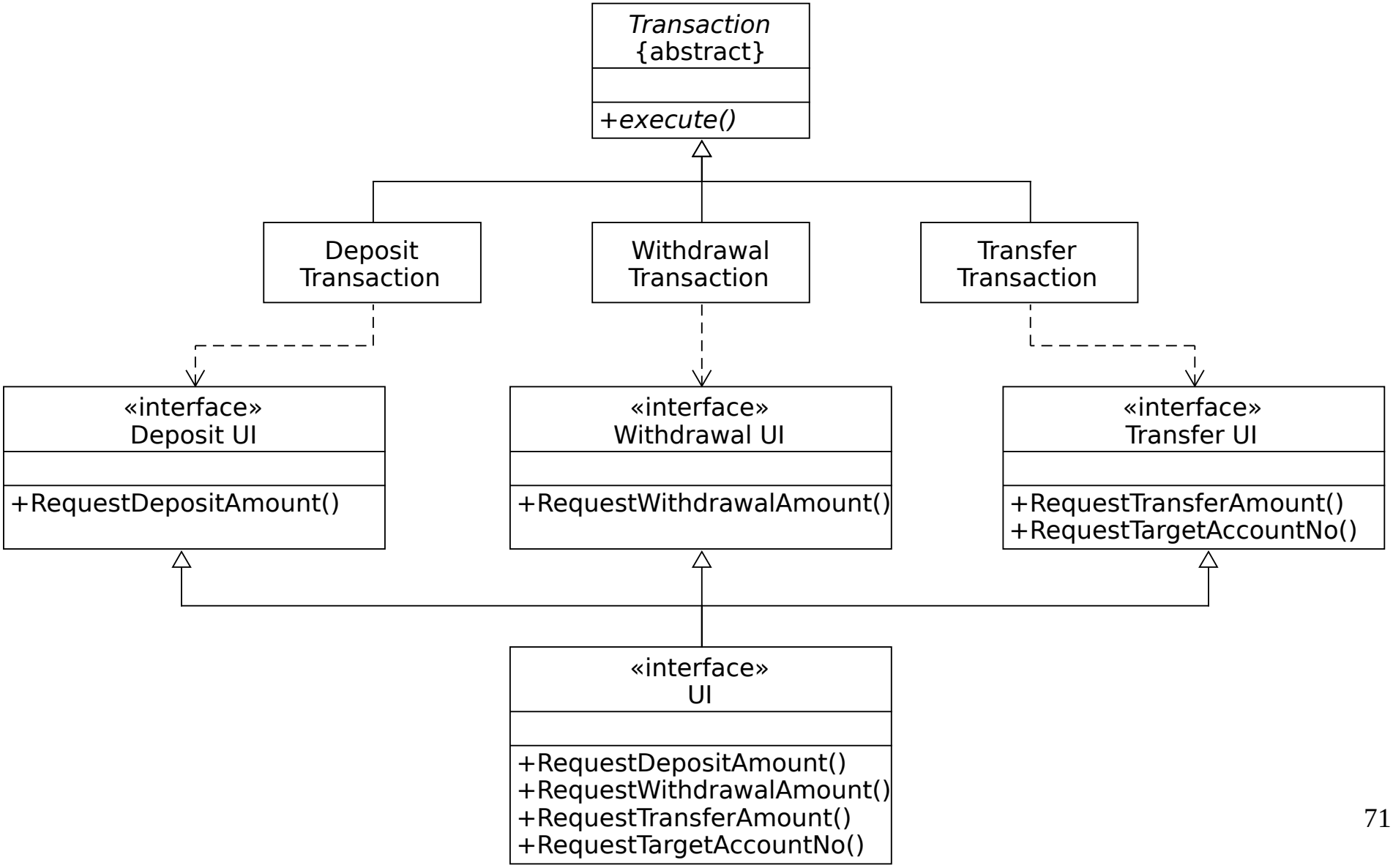
- Amikor egy kliens egy olyan osztálytól függ, melynek vannak olyan metódusai, melyeket a kliens nem használ, más kliensek azonban igen, akkor a többi kliens által az osztályra kényszerített változások hatással lesznek arra a kliense is.
- Ez a kliensek közötti nem szándékos csatoltságot eredményez.

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)

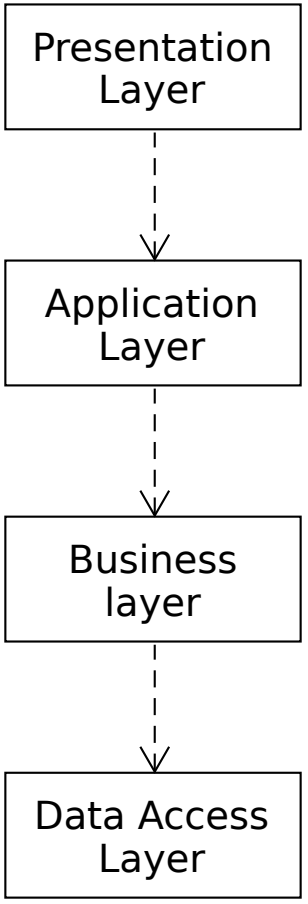
- A magas szintű modulok tartalmazzák az alkalmazás üzleti logikáját, ők adják az alkalmazás identitását. Ha ezek a modulok alacsony szintű moduloktól függenek, akkor az alacsony szintű modulokban történő változásoknak közvetlen hatása lehet a magas szintű modulokra, szükségessé tehetik azok változását 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.

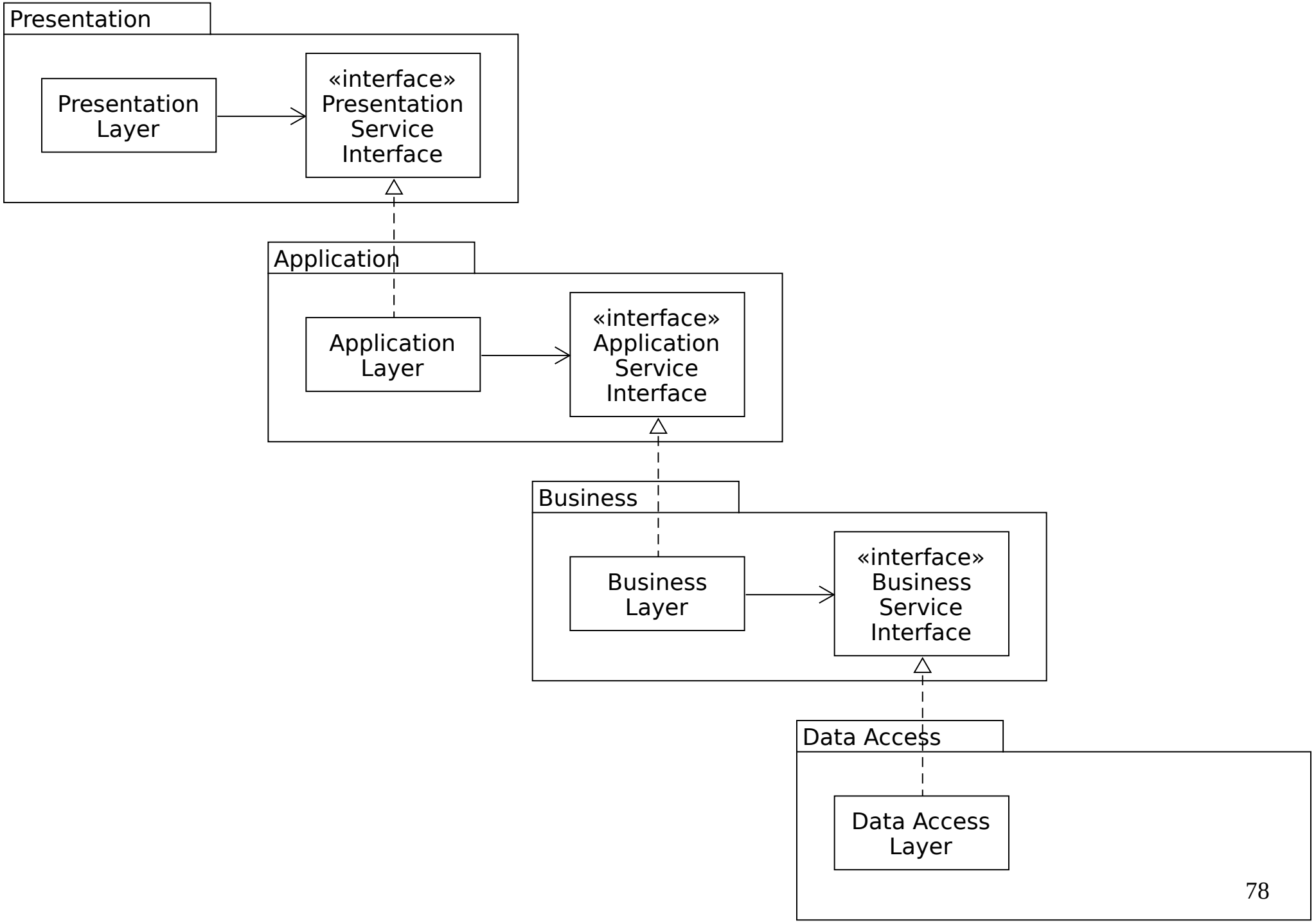
SOLID – függőség megfordítási elv (5)

- 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)

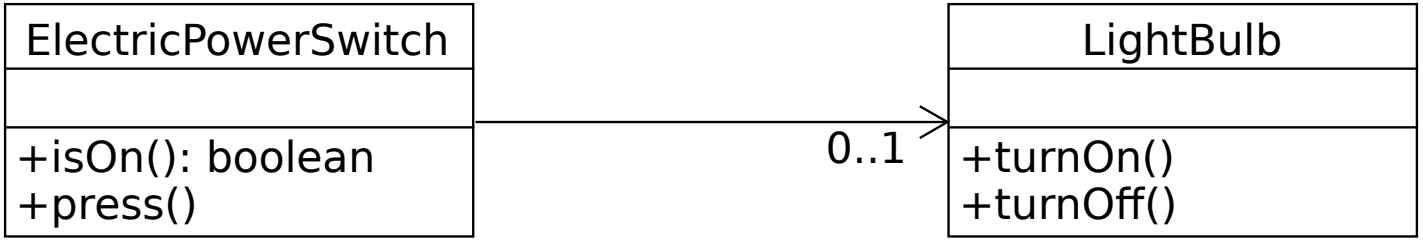
- Az előbbi példa az elvnek megfelelő változata: (folytatás)
  - Nem csupán a függőségek kerültek megfordításra, hanem az interfész tulajdonlás is (*inversion of ownership*).
    - Hollywood elv: Ne hívj, majd mi hívunk. (*Don't call us, we'll call you.*)

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.

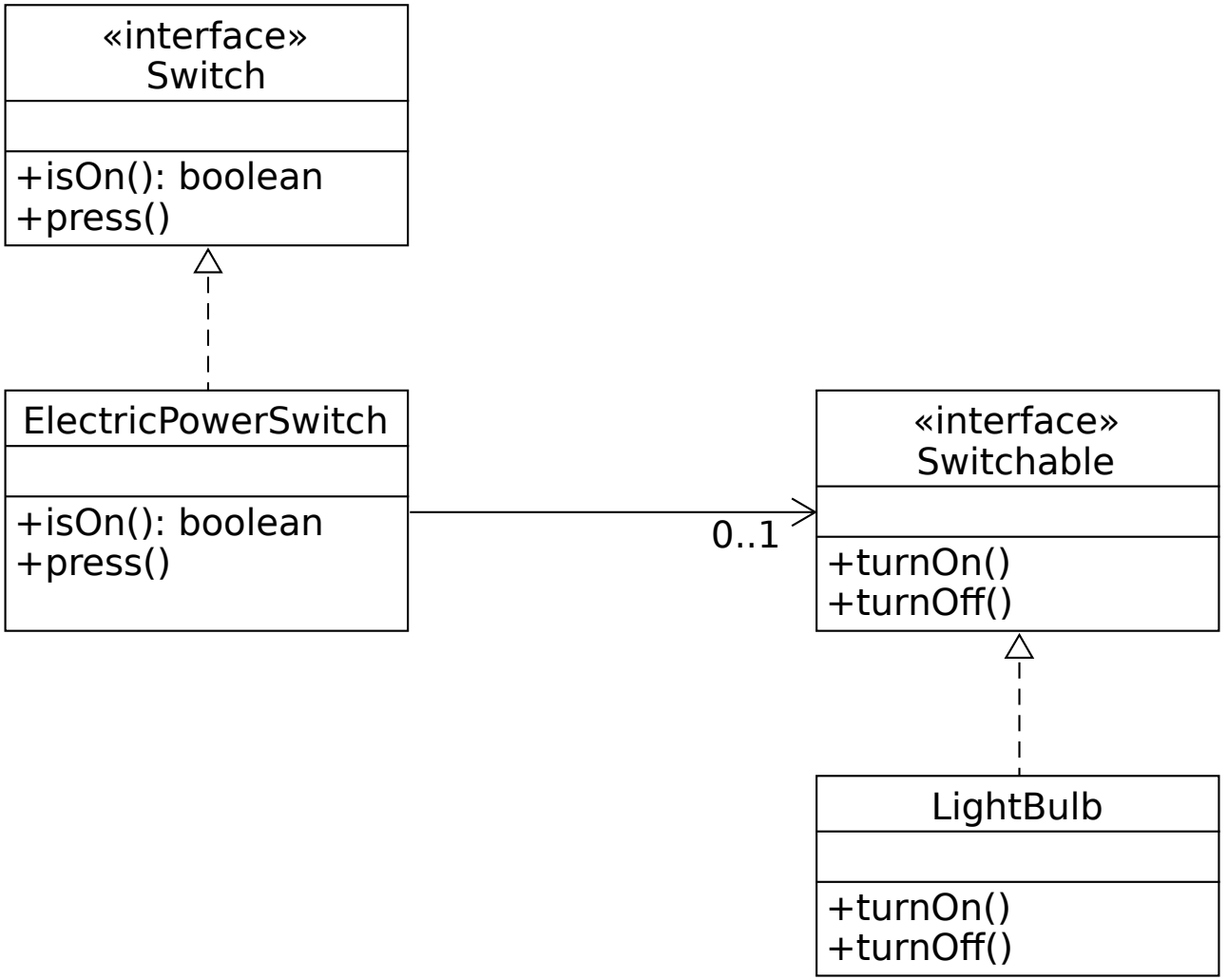
SOLID – függőség megfordítási elv (10)

- 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:



SRP és ISP (1)

- Néhányan úgy gondolják, hogy az SRP és az ISP olyan szorosan kapcsolódnak egymáshoz, hogy egyikük redundáns.
  - Lásd például:
    - Mark Seemann. *SOLID or COLDS?* September 29, 2009. https://blog.ploeh.dk/2009/09/29/SOLIDorCOLDS/

SRP és ISP (2)

- Robert C. Martin a Twitter-en (2018. május 16.): https://twitter.com/unclebobmartin/status/996739060348653568
  - *ISP can be seen as similar to SRP for interfaces; but it is more than that.*
    - *ISP generalizes into: “Don't depend on more than you need.”*
    - *SRP generalizes to "Gather together things that change for the same reasons and at the same times.”*
  - *Imagine a stack class with both push and pop. Imagine a client that only pushes. If that client depends upon the stack interface, it depends upon pop, which it does not need. SRP would not separate push from pop; ISP would.*

SRP és ISP (3)

- Robert C. Martin a Twitter-en (2018. május 16.): https://twitter.com/unclebobmartin/status/996739060348653568
  - Az ISP hasonlóan tűnhet az SRP-hez interfészekre, de annál több.
    - Az ISP így általánosítható: „Ne függj annál többtől, mint amire szükséged van.”
    - Az SRP pedig így általánosítható: „Gyűjts össze azokat a dolgokat, melyek ugyanabból az okból és ugyanakkor változnak.”
  - Képzeljünk el egy verem osztályt *push* és *pop* műveletekkel. Képzeljünk egy klienst, mely csak a *push* művelet használja. Ha a kliens a verem interfésztől függ, akkor a *pop*-tól is függ, amire nincs szüksége. Az SRP nem választaná el a *push*-t a *pop*-tól, az ISP viszont igen.

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
  - Krunal Patel, Nilang Patel. *Java 9 Dependency Injection*. Packt Publishing, 2018. https://www.packtpub.com/application-development/java-9-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 Principles, Practices, and Patterns*. Manning, 2019. https://www.manning.com/books/dependency-injection-principles-practices-patterns

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 kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.

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.

Függőség befecskendezés (6)

- **DI konténer** (**DI container**): függőség befecskendezési funkcionalitást nyújtó programkönyvtár.
  - Az ***Inversion of Control (IoC) container*** kifejezést is használják rájuk.
- A függőség befecskendezés alkalmazható DI konténer nélkül.
- **Tiszta DI**: függőség befecskendezés alkalmazásának gyakorlata DI konténer nélkül.

Függőség befecskendezés (7)

- 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 (8)

- 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 (9)

- Példa: függőség befecskendezés konstruktorral (*constructor injection*):

```
public class TextEditor {

  private SpellChecker spellChecker;

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

  // ...

}
```

Függőség befecskendezés (10)

- Példa: függőség befecskendezés beállító metódussal (*setter injection*):

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor() {}

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

  // ...

}
```

Függőség befecskendezés (11)

- Példa: függőség befecskendezés 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 (12)

- A függőség befecskendezés előnyei:
  - **Kiterjeszthetőség**
  - **Karbantarthatóság**
  - **Tesztelhetőség**: a függőség befecskendezés támogatja az egységtesztelést.
    - Valós függőségek helyet a tesztelt rendszerbe befecskendezhetők „teszt dublőrök” (*test doubles*).

Függőség befecskendezés C++-ban

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

Függőség befecskendezés Java-ban (1)

- *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 és a `javax.inject` csomag tartalmazza.
    - Lásd: https://javaee.github.io/javaee-spec/javadocs/javax/inject/package-summary.html
  - Implementációk: *Dagger*, *Guice*, *HK2*, *Spring Framework*, …
  - Utód: *Jakarta Dependency Injection* (Jakarta EE) https://jakarta.ee/specifications/dependency-injection/

Függőség befecskendezés Java-ban (2)

- *JSR 365: Contexts and Dependency Injection for Java 2.0* https://www.jcp.org/en/jsr/detail?id=365
  - Haladóbb lehetőségekkel bővíti ki a JSR-330-at, mint például a modulok.
  - A Java EE 8-ban jelent meg és a `javax.enterprise.inject` csomag és alcsomagjai tartalmazzák.
  - Implementációk: *Apache OpenWebBeans*, *Weld*
  - További információk: https://www.cdi-spec.org/
  - Utód: *Jakarta Contexts and Dependency Injection* (Jakarta EE) https://jakarta.ee/specifications/cdi/ https://projects.eclipse.org/projects/ee4j.cdi

Függőség befecskendezés Java-ban (3)

- Keretrendszerek:
  - *Apache OpenWebBeans* (licenc: *Apache License 2.0*) https://openwebbeans.apache.org/ https://github.com/apache/openwebbeans
    - A JSR-365 egy Java SE implementációja.
  - *Dagger* (licenc: *Apache License 2.0*) https://dagger.dev/ https://github.com/google/dagger
    - A JSR-330-on alapuló fordításidejű keretrendszer függőség befecskendezéshez.
  - *Guice* (licenc: *Apache License 2.0*) https://github.com/google/guice
    - Pehelysúlyú függőség befecskendezési keretrendszer. Eredetileg a Guice volt a JSR-330 referencia implementációja.
    - Lásd: Natasha Mathur. *Implementing Dependency Injection in Google Guice*. September 9, 2018. https://hub.packtpub.com/implementing-dependency-injection-google-guice/

Függőség befecskendezés Java-ban (4)

- Keretrendszerek (folytatás):
  - *GlassFish HK2* (licenc: *Eclipse Public License 2.0*, GPLv2 + *Classpath Exception*) https://eclipse-ee4j.github.io/glassfish-hk2/ https://github.com/eclipse-ee4j/glassfish-hk2
    - A JSR-330 egy Java SE implementációja a GlassFish alkalmazásszerverhez és más termékekhez.
  - *Spring Framework* (licenc: *Apache License 2.0*) https://spring.io/projects/spring-framework https://github.com/spring-projects/spring-framework
    - A Spring Framework egy keretrendszer és IoC (*inversion of control*) konténer vállalati Java alkalmazások fejlesztéséhez. Támogatja a JSR-330-at.
  - *Weld* (licenc: *Apache License 2.0*) https://weld.cdi-spec.org/ https://github.com/weld/core
    - JSR-365 referencia implementáció.

Függőség befecskendezés .NET-ben

- Keretrendszerek:
  - *Castle Windsor* (licenc: *Apache License 2.0*) http://www.castleproject.org/ https://github.com/castleproject/Windsor
  - *Lamar* (licenc: *MIT License*) https://jasperfx.github.io/lamar/ https://github.com/jasperfx/lamar
  - *Ninject* (licenc: *Apache License 2.0*) http://www.ninject.org/ https://github.com/ninject/ninject
  - *Simple Injector* (licenc: *MIT License*) https://simpleinjector.org/ https://github.com/simpleinjector/SimpleInjector
  - *Spring.NET* (licenc: *Apache License 2.0*) https://springframework.net/ https://github.com/spring-projects/spring-net