Tiszta kód

Jeszenszky Péter
Debreceni Egyetem, Informatikai Kar
jeszenszky.peter@inf.unideb.hu

Utolsó módosítás: 2024. május 5.

Kódújraszervezés

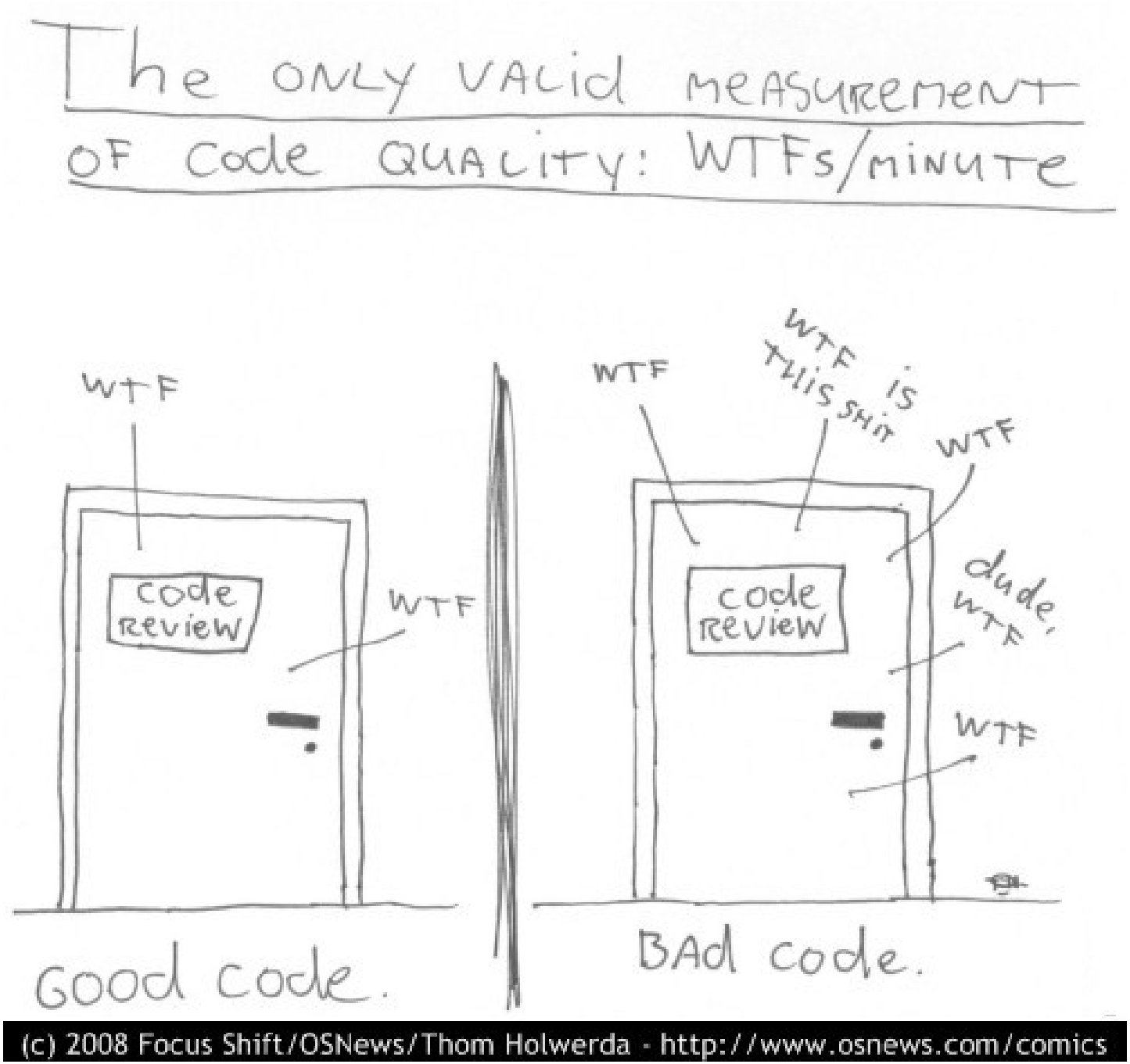
- *„Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”*
  - Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
  - Martin Fowler, Kent Beck. *Refactoring: Improving the Design of Existing Code*. Second Edition. Addison-Wesley, 2018. https://martinfowler.com/books/refactoring.html
  - Martin Fowler. *Refactoring: Kódjavítás újratervezéssel*. Kiskapu, 2006.
- Egy szoftverrendszer olyan módon történő megváltoztatásának folyamata, mely nem változtatja meg a kód külső viselkedését, de javítja a belső szerkezetét.

Kódszag

- *„A code smell is a surface indication that usually corresponds to a deeper problem in the system.”*
- Olyan tünet, mely általában a rendszer egy mélyebben gyökerező problémájának felel meg.
  - Martin Fowler. *CodeSmell*. 9 February 2006. https://martinfowler.com/bliki/CodeSmell.html

Irodalom

- Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.
- Robert C. Martin. *Tiszta kód: Az agilis szoftverfejlesztés kézikönyve*. Kiskapu Kft., 2010.
  - Tiszta kód írásának elvei, mintái és gyakorlatai
  - Esettanulmányok
  - Kódszagok és heurisztikák



Forrás: Thom Holwerda. *WTFs/m*. 2008. https://www.osnews.com/story/19266/wtfsm/

Mi a tiszta kód? (1)

- Bjarne Stroustrup:
  - *„I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.”*
  - Szeretem, ha a kódom elegáns és hatékony. A logikája egyszerű kell, hogy legyen, hogy nehezen bújjanak meg a hibák a kódban. A függőségek legyenek minimálisak a könnyű karbantarthatóság érdekében. A hibakezelés legyen teljes, a teljesítmény pedig közel optimális, hogy ne kísértsen a kódot piszkító optimalizálásra. A tiszta kód egy dolgot csinál, és azt jól.

Mi a tiszta kód? (2)

- Grady Booch (az UML egyik kifejlesztője):
  - *„Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.”*
  - A tiszta kód egyszerű és közvetlen. Olyan a tiszta kódot olvasni, mint a jól megírt prózát. A tiszta kód soha nem homályosítja el a tervezője szándékát, hanem inkább éles absztrakciókkal és egyértelmű sorokkal teli.

Mi a tiszta kód? (3)

- Dave Thomas (DRY elv, az Agilis kiáltvány egyik szerzője):
  - *„Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.”*
  - A tiszta kód egy olyan fejlesztő számára is olvasható, továbbfejleszthető, aki nem az eredeti szerző. Vannak hozzá egység- és elfogadási tesztek. Értelmes nevei vannak. Egy dolog elvégzéséhez egy módot biztosít több helyett. Minimális függőségei vannak, melyek világosan meghatározottak. Tiszta és minimális API-t biztosít. A kód *literate* kell, hogy legyen, mivel a nyelvtől függően nem minden szükséges információ fejezhető ki világosan pusztán a kódban.

Literate programming (1)

- Donald E. Knuth. *Literate Programming*. The Computer Journal, 27 (2): 97–111, 1984. http://www.literateprogramming.com/knuthweb.pdf
  - *„I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming.'*
  - *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
  - *The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reınforce each other.”*

Literate programming (2)

- Donald E. Knuth. *Literate Programming*. The Computer Journal, 27 (2): 97–111, 1984. http://www.literateprogramming.com/knuthweb.pdf
  - Úgy vélem, hogy megérett az idő a programok jelentősen jobb dokumentálására, melyet úgy érhetünk el a legjobban, ha a programokat irodalmi alkotásoknak tekintjük. A cím ezért „*Literate Programming*”.
  - Változtassuk meg hagyományos hozzáállásunkat a programok létrehozásához: ahelyett, hogy azt képzelnénk, hogy fő feladatunk az, hogy arra utasítsuk a számítógépet, hogy mit csináljon, inkább emberek számára próbáljuk elmagyarázni, hogy mit szeretnénk, hogy csináljon a számítógép.
  - A *literate* programozás művelőjére egy esszéistaként tekinthetünk, akinek fő gondja a magyarázás és a kiváló stílus. Egy ilyen szerző szótárral a keze ügyében körültekintően választja a változóneveket és minden egyes változó jelentését elmagyarázza. Olyan program írására törekszik, mely érthető, mivel a fogalmai az emberi felfogás számára legjobb sorrendben kerülnek bevezetésre, egymást erősítő formális és informális módszerek keverékének felhasználásával.

Értelmes nevek (1)

- Olyan neveket használjunk a kódban, melyekből kiderül a szándék.
  - Rossz gyakorlat:
    - `int d; // elapsed time in days`
  - Jó gyakorlat:
    - `int elapsedTimeInDays;`
    - `int daysSinceCreation;`
    - `int daysSinceModification;`
    - `int fileAgeInDays;`

Értelmes nevek (2)

- Egy nagyon rossz példa:

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Értelmes nevek (3)

- Kerüljük a félrevezető neveket.
  - Félrevezető például az `accountList` név, ha nem ténylegesen egy listáról van szó. Ha ez a helyzet, sokkal jobb például az `accountGroup` vagy az `accounts` név.

Értelmes nevek (4)

- Értelmesen megkülönböztethető neveket használjunk.
  - Példa nem informatív nevekre:

```
public static void copyChars(char a1[],
    char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

  - A fenti kódban használjunk inkább például a `source` és `target` neveket az `a1` és `a2` helyett.

Értelmes nevek (5)

- Ne használjunk olyan túl általános zajszavakat a nevekben, mint például `Data`, `Info` vagy `Object`.
  - A nevük alapján nem világos, hogy mi a különbség például a `Product`, `ProductData` és `ProductInfo` nevű osztályok között.
  - Soha ne használjuk a `variable` szót változó nevében.

Értelmes nevek (6)

- Használjunk kiejthető neveket.
- Használjunk kereshető neveket.
  - Például egybetűs nevek esetén problémás lehet a keresés.
  - Egybetűs neveket csak lokális változókhoz használjunk rövid metódusokban.
- Egy név hossza meg kell, hogy feleljen a hatásköre méretének.
  - Minél nagyobb a hatáskör mérete, célszerűbb annál hosszabb nevet választani.
  - Ha egy változót vagy konstanst a kódban több helyen is használunk, akkor érdemes neki kereshető nevet adni.

Értelmes nevek (7)

- Kerüljük a kódolásokat a nevekben, melyek a típusról vagy a hatáskörről szolgáltatnak információkat.
  - Például:
    - Magyar jelölés (*Hungarian notation*) https://en.wikipedia.org/wiki/Hungarian_notation
    - Tagok nevében `m_` előtag használata
    - `I` karakter az interfészek nevének elején

Értelmes nevek (8)

- Osztályok neveként használjunk főneveket vagy főnévi kifejezéseket.
  - Például: `Console`, `FileReader`, `RandomAccessFile`, ...
- Metódusok neveként használjunk igéket vagy igei kifejezéseket.
  - Például: `close`, `readByte`, `stripTrailingZeros`, ...
  - Lekérdező, beállító metódusoknál, predikátumoknál használjuk a névben a `get`, `set`, `is` előtagokat.

Értelmes nevek (9)

- Elnevezéseknél kerüljük a jópofáskodást.
- Következetesen alkossunk neveket.
  - Egy absztrakt fogalomra ugyanazt a szót használjuk.
    - Például ne adjuk különböző osztályok ekvivalens metódusainak a `fetch`, `retrieve` és `get` neveket.
- Használjunk olyan neveket, melyeket a többi programozó is megért.
  - Használjunk informatikai szakkifejezéseket, tervezési minták neveit, matematikai szakkifejezéseket, …
- Szükség esetén használjuk az adott problématerület neveit.

Értelmes nevek (10)

- Nevek kontextusba helyezése:
  - Vannak olyan nevek, melyek önmagukban értelmesek. A legtöbb azonban nem ilyen, ezeket az érthetőséghez az olvasó számára egy kontextusba kell helyezni, bezárva őket egy osztályba, függvénybe vagy névtérbe.
    - Végső megoldásként használjunk a nevekhez egy alkalmas előtagot.

Értelmes nevek (11)

- Nevek kontextusba helyezése (folytatás):
  - Nyilvánvaló például, hogy a `street`, `city`, `zipCode`, `state` nevű változók egy címet alkotnak.
  - Ha azonban csak a `state` változót látjuk egy metódusban, nem feltétlenül arra gondolunk, hogy ez egy cím része.
    - Kontextusba helyezhetjük a neveket egy előtag használatával (`addrStreet`, `addrCity`, `addrZip`, `addrState`), de jobb őket egy `Address` nevű osztályba bezárni.
  - Ne vezessünk be azonban feleslegesen kontextusokat.
    - Például ne használjunk osztályok nevének elején az alkalmazásra utaló előtagot.

Függvények (1)

- A függvények nagyon rövidek kell, hogy legyenek.
  - Nem szabad, hogy 100 sorosak legyenek. Nagyon ritkán legyenek 20 sorosak. Legyenek inkább 2–4 sorosak.
  - Utasításblokkok (`for`, `if`, `while`, …) egyetlen sornyi kódot kell, hogy tartalmazzanak, mely várhatólag egy függvényhívás.
    - Ez nem csupán röviden tartja a befoglaló függvényt, hanem dokumentációs értékkel is bír, mivel a meghívott függvény neve beszédes.
    - Ez azt is jelenti, hogy a függvények bekezdési szintjeinek (*indent level*) száma nem lehet több 2-nél. Ez a függvényeket könnyen olvashatóvá és érthetővé teszi.

Függvények (2)

- A függvények csak egy dolgot csináljanak, de azt jól.
- Függvényenként egy absztrakciós szint:
  - Annak biztosításához, hogy a függvények egy dolgot csináljanak, az utasítások azonos absztrakciós szintűek kell, hogy legyenek bennük.
    - Zavaró egy függvényben a különböző absztrakciós szintek keverése, mert az olvasó nem tudja eldönteni egy kifejezésről, hogy az lényeges-e vagy csupán egy technikai részlet.
  - *Stepdown* szabály: felülről lefelé haladva csökkenjen a függvények absztrakciós szintje.
    - A kód felülről lefelé haladva olvasható.

Függvények (3)

- Argumentumok:
  - A függvények megkülönböztetése az argumentumok száma szerint:
    - **Niladikus** (***niladic***): argumentum nélküli, ez az ideális
    - **Monadikus** (***monadic***): egyargumentumú
    - **Diadikus** (***diadic***): kétargumentumú
    - **Triadikus** (***triadic***): három argumentumú, lehetőleg kerülni kell
    - **Poliadikus** (***polyadic***): háromnál több argumentumú, soha ne használjuk

Függvények (4)

- Argumentumok (folytatás):
  - Az olvasó számára megnehezítik a kód megértését.
    - Más absztrakciós szinten vannak, mint a függvény neve, és egy olyan részlet ismeretére kényszerítenek, mely az adott ponton nem különösebben érdekes.
  - A tesztelést is bonyolítják.

Függvények (5)

- Argumentumok (folytatás):
  - Rossz gyakorlat jelző argumentumok használata.
    - Egy jelző argumentum egy olyan logikai típusú argumentum, melynek értékétől függ a függvény viselkedése.
      - Egy ilyen argumentummal rendelkező függvény egynél több dolgot csinál: egyet akkor, ha az argumentum értéke igaz, egy másikat akkor, ha hamis.
    - Lásd még: Martin Fowler. *FlagArgument*. 23 June 2011. https://martinfowler.com/bliki/FlagArgument.html

Függvények (6)

- Argumentumok (folytatás):
  - Például `Point p = new Point(0,0);` tökéletesen elfogadható, az argumentumok egyetlen érték komponensei, az argumentumok sorrendje pedig a komponensek sorrendjének felel meg.
    - Viszont például az `assertEquals(expected, actual)` JUnit függvénynél nagyon könnyű összekeverni az argumentumokat.
  - Példa elfogadható három argumentumú függvényre (JUnit):
    - `assertEquals(double expected, double actual, double epsilon)`

Függvények (7)

- Argumentumok (folytatás):
  - Kettőnél vagy háromnál több argumentum esetén bizonyosakat érdemes lehet becsomagolni egy osztályba.
    - Példa:
      - `Circle makeCircle(double x, double y, double radius);`
      - `Circle makeCircle(Point center, double radius);`
  - Változó argumentumszámú függvényekre lista argumentumú függvényekként tekinthetünk.
    - Lásd például: `java.lang.String.format(String format, Object... args)` https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#format(java.lang.String,java.lang.Object...)
    - *Varargs* https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

Függvények (8)

- Mellékhatásmentesség:
  - A függvények legyenek mellékhatásmentesek.
    - Csak azt csinálják, amit ígérnek.
  - Kerülni kell output argumentumok használatát.

Függvények (9)

- Parancs-lekérdezés szétválasztás:
  - Egy függvény vagy csináljon valamit, vagy válaszoljon valamit, de egyszerre mindkettőt ne.
    - Vagy változtassa meg egy objektum állapotát, vagy adjon vissza információt az objektumról.
  - Rossz gyakorlat:
    - Az alábbi függvény, mely beállítja egy adott attribútum értékét, sikeres beállítás esetén `true`-t ad vissza, nem létező attribútum esetén pedig `false`-t:
      - `public boolean set(String attribute, String value);`
    - Az olvasó számára nem világos, hogy mi a kódban például az alábbi utasítás jelentése:
      - `if (set("username", "unclebob")) { ... }`

Függvények (10)

- Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.
  - Így egyszerű további kivételek bevezetése, az új kivételek egy kivételosztály leszármazottai lesznek.

Függvények (11)

- A `try`/`catch` blokkokat emeljük ki önálló függvényekbe.
  - Összezavarják a kód szerkezetét, mivel keverik a szabályos feldolgozást és a hibakezelést.
  - A függvények egy dolgot kell hogy csináljanak, a hibakezelés is egy dolog.
    - Egy hibakezelő függvény kizárólag egy `try`/`catch` blokkot kell, hogy tartalmazzon.

Függvények (12)

- Példa `try`/`catch` blokkok kiemelésére:

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  } catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Függvények (13)

- Strukturált programozás:
  - Használható egy függvényben akár több `return` utasítás, ciklusokban `break` és `continue` utasítás.

Megjegyzések (1)

- A legjobb esetben is szükséges rosszak.
- A megjegyzések helyénvaló használata ellensúlyozza hiányosságainkat az önmagunk kóddal történő kifejezésében.
  - Ha elég kifejezőek lennének a programozási nyelvek és mesteri módon tudnánk velük bánni a szándékaink kifejezéséhez, akkor nem nagyon lenne szükség megjegyzésekre.

Megjegyzések (2)

- Azért nemkívánatosak a megjegyzések, mert nem mindig, és nem szándékosan, de túl gyakran közölnek pontatlan vagy valótlan információt.
- A kód változik, fejlődik, melyet nem minden esetben követnek a megjegyzések.
  - Nem életszerű a karbantartásuk.
  - Minél régebbi egy megjegyzés, annál valószínűbb, hogy rossz.
- A pontatlan megjegyzések még rosszabbak, mint a megjegyzések teljesen hiánya.

Megjegyzések (3)

- A megjegyzések írásának egyik gyakori oka a rossz kód. Érdemesebb inkább rendbe tenni a rossz kódot, mint megjegyzésekkel megtűzdelni.
- Törekedjünk arra, hogy ne legyen szükség megjegyzésekre, a kód beszéljen önmagáért.

Jó megjegyzések (1)

- Jogi megjegyzések
- Informatív megjegyzések
- Szándékot magyarázó megjegyzés
- Tisztázó megjegyzés
- Következményekre figyelmeztető megjegyzés
- TODO megjegyzés
- Megerősítő megjegyzés
- Javadoc megjegyzés nyilvános API-ban

Jó megjegyzések (2)

- **Informatív megjegyzés**:
  - Példa: `java.io.UnixFileSystem` (OpenJDK 21):

```
/* Check that the given pathname is normal. If not, invoke the
   real normalizer on the part of the pathname that requires
   normalization. This way we iterate through the whole pathname
   string only once. */
@Override
public String normalize(String pathname) {
  // ...
}
```

Jó megjegyzések (3)

- **Szándékot magyarázó megjegyzés**:
  - Példa: `java.util.Collections` (OpenJDK 21):

```
// Suppresses default constructor, ensuring non-instantiability.
private Collections() {
}

@SuppressWarnings({"rawtypes", "unchecked"})
public static void swap(List<?> list, int i, int j) {
  // instead of using a raw type here, it's possible to capture
  // the wildcard but it will require a call to a supplementary
  // private method
  final List l = list;
  l.set(i, l.set(j, l.get(i)));
}
```

Jó megjegyzések (4)

- **Szándékot magyarázó megjegyzés** (folytatás):
  - Példa: `org.apache.commons.lang3.StringUtils` (Commons Lang)

```
public static String normalizeSpace(final String str) {
  // LANG-1020: Improved performance significantly by normalizing
  // manually instead of using regex
  // See
  // https://github.com/librucha/commons-lang-normalizespaces-benchmark
  // for performance test
  // ...
}
```

Jó megjegyzések (5)

- **Tisztázó megjegyzés**:
  - Példa: `java.lang.Math` (OpenJDK 21):

```
public static double max(double a, double b) {
  if (a != a)
    return a;   // a is NaN
  // ...
}
```

Jó megjegyzések (6)

- **Következményekre figyelmeztető megjegyzés**:
  - Példa: `java.lang.Integer` (OpenJDK 21)

```
public static int parseInt(String s, int radix)
        throws NumberFormatException {
  /*
   * WARNING: This method may be invoked early during VM
   * initialization before IntegerCache is initialized.
   * Care must be taken to not use the valueOf method.
   */
  // ...
}
```

Jó megjegyzések (7)

- **TODO megjegyzés**:
  - Példa: `java.time.format.DateTimeFormatterBuilder` (OpenJDK 11)

```
@Override
public boolean format(DateTimePrintContext context,
    StringBuilder buf) {
  Long offsetSecs = context.getValue(OFFSET_SECONDS);
  if (offsetSecs == null) {
    return false;
  }
  String gmtText = "GMT"; // TODO: get localized version of 'GMT'
  if (gmtText != null) {
    buf.append(gmtText);
  }
  // ...
}
```

Jó megjegyzések (8)

- **TODO megjegyzés** (folytatás):
  - Példa: `jdk.internal.org.objectweb.asm.tree.ModuleNode` (OpenJDK 21):

```
// TODO(forax): why is there no 'mainClass' and 'packages' parameters
// in this constructor?
public ModuleNode(
        final int api,
        final String name,
        final int access,
        final String version,
        final List<ModuleRequireNode> requires,
        final List<ModuleExportNode> exports,
        final List<ModuleOpenNode> opens,
        final List<String> uses,
        final List<ModuleProvideNode> provides) {
    super(api);
    this.name = name;
    // ...
```

Rossz megjegyzések (1)

- Motyogás
- Fölösleges megjegyzés
- Félrevezető megjegyzés
- Kötelező megjegyzés
- Napló megjegyzés
- Zaj-megjegyzés
- Pozíciójelző/szalagcím megjegyzés
- Záró kapcsos zárójel megjegyzés
- Szerző neve megjegyzésben
- Megjegyzésbe tett kód
- HTML megjegyzés
- Nem lokális megjegyzés
- Túl sok információt tartalmazó megjegyzés
- A kódhoz nem nyilvánvalóan kapcsolódó megjegyzés
- Javadoc megjegyzés nem nyilvános kódban

Rossz megjegyzések (2)

- **Motyogás**: hanyagul odavetett megjegyzés, mely legfeljebb a szerzőnek jelent valamit, más számára nem érthető.
  - Példa: `java.util.Base64` (OpenJDK 21)

```
@Override
public int available() throws IOException {
  if (closed)
    throw new IOException("Stream is closed");
  return is.available(); // TBD:
}
```

Rossz megjegyzések (3)

- **Félrevezető megjegyzés**: nem eléggé pontos megjegyzés.
  - Példa: `org.apache.commons.io.FileUtils` (Commons IO)

```
// Private method, must be invoked will a directory parameter

/**
  * Writes a CharSequence to a file creating the file if it does
  * not exist.
  *
  * @param file the file to write
  ...
  */
public static void write(final File file, final CharSequence data,
  final String charsetName) throws IOException {
  // ...
}
```

Rossz megjegyzések (4)

- **Redundáns megjegyzés**: nem szolgál semmiféle plusz információval a kódról, nem könnyebb elolvasni sem, mint magát a kódot.
- **Kötelező megjegyzés**: a nyilvánvalót magyarázó megjegyzés, ami csak azért van ott, mert megkövetelik, hogy minden függvényhez, változóhoz tartozzon dokumentációs megjegyzés.
- **Napló megjegyzés**: egy modul elején a benne végzett minden egyes módosítást naplószerűen dokumentáló megjegyzés.
  - A verziókezelő rendszerek feleslegessé tették.

Rossz megjegyzések (5)

- **Zaj-megjegyzés**: új információval nem szolgáló, a magától értetődőt újra megfogalmazó megjegyzés.
  - Példa:

```
/** The day of the month. */
private int dayOfMonth;

/**
 * Returns the day of the month.
 *
 * @return the day of the month
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

Rossz megjegyzések (6)

- **Zaj-megjegyzés** (folytatás):
  - Példa: `org.apache.commons.lang3.builder.ToStringStyle` (Commons Lang)

```
/**
 * <p>Constructor.</p>
 */
protected ToStringStyle() {
  super();
}
```

Rossz megjegyzések (7)

- **Pozíciójelző/szalagcím megjegyzés**:
  - Példa: `java.time.LocalDate` (OpenJDK 21):

```
/**
 * The day-of-month.
 */
private final short day;

//-----------------------------------------------------------------------
```

Rossz megjegyzések (8)

- **Pozíciójelző/szalagcím megjegyzés** (folytatás):
  - Példa: `java.util.GregorianCalendar` (OpenJDK 21)

```
//////////////////
// Class Variables
//////////////////
// ...

//////////////////////
// Instance Variables
//////////////////////
// ...

///////////////
// Constructors
///////////////
// ...
```

Rossz megjegyzések (9)

- **Záró kapcsos zárójel megjegyzés**: hosszú függvényeknél, blokkoknál lehet létjogosultsága.
  - Példa: `java.text.SimpleDateFormat` (OpenJDK 21)

```
switch (patternCharIndex) {
case PATTERN_ERA: // ...
case PATTERN_WEEK_YEAR: // ...
case PATTERN_YEAR: // ...
case PATTERN_MONTH: // ...
// ...
default: // ...
} // switch (patternCharIndex)
```

Rossz megjegyzések (10)

- **Szerző neve megjegyzésben**: a verziókezelő rendszerek feleslegessé tették.
- **Megjegyzésbe tett kód**: a rossz megjegyzések legutálatosabbja, kerülendő. Mások azt fogják gondolni, hogy okkal van ott, fontos, és ezért nem lesz bátorságuk törölni.
  - Verziókezelés esetén nyugodtan törölhető bármi a kódból.

Rossz megjegyzések (11)

- **Megjegyzésbe tett kód** (folytatás):
  - Példa: `java.io.RandomAccessFile` (OpenJDK 17)

```
public final void writeByte(int v) throws IOException {
  write(v);
  //written++;
}
public final void writeShort(int v) throws IOException {
  write((v >>> 8) & 0xFF);
  write((v >>> 0) & 0xFF);
  //written += 2;
}
public final void writeInt(int v) throws IOException {
  write((v >>> 24) & 0xFF);
  write((v >>> 16) & 0xFF);
  write((v >>>  8) & 0xFF);
  write((v >>>  0) & 0xFF);
  //written += 4;
}
```

Rossz megjegyzések (12)

- **HTML megjegyzés**: rontja a forráskód olvashatóságát.
- **Nem lokális megjegyzés**: olyan megjegyzés, mely nem a közvetlen környezetében lévő kódra vonatkozik.
- **Túl sok információt tartalmazó megjegyzés**:
- **A kódhoz nem nyilvánvalóan kapcsolódó megjegyzés**:
- **Javadoc megjegyzés nem nyilvános kódban**:

Hibakezelés

- Hibakódok visszaadása helyett részesítsük előnyben a kivételeket.
- Használjunk nem ellenőrzött kivételeket.
- Ne adjunk át/vissza `null`-t.

Ellenőrzött és nem ellenőrzött kivételek (1)

- Java-ban a kivételek ellenőrzöttek (*checked*) vagy nem ellenőrzöttek (*unchecked*).
  - Lásd: *The Java Language Specification, Java SE 21 Edition – The Kinds of Exceptions* https://docs.oracle.com/javase/specs/jls/se21/html/jls-11.html#jls-11.1.1
- Nyelvek, melyekben nincsenek ellenőrzött kivételek: C#, C++, Kotlin, Python, Scala, …

Ellenőrzött és nem ellenőrzött kivételek (2)

- Ellenőrzött kivételek:
  - A metódusok deklarálják az ellenőrzött kivételeket, melyek bekövetkezhetnek a végrehajtásuk során, mely lehetővé teszi a fordításidejű ellenőrzést annak biztosításához, hogy a kivételek kezelésre kerüljenek.
  - A `throws` záradék szolgál azoknak az ellenőrzött kivételeknek a jelzésére, melyeket egy metódus vagy konstruktor törzs utasításai dobhatnak.

Ellenőrzött és nem ellenőrzött kivételek (3)

- Ellenőrzött kivételek:
  - Az ellenőrzött kivételek arra kényszerítik a programozót, hogy foglalkozzon velük, mivel el kell kapni őket, így a kód megbízhatóságát növelik.
  - Kritikus programkönyvtárak számára ajánlott ellenőrzött kivételek dobása.

Ellenőrzött és nem ellenőrzött kivételek (4)

- Nem ellenőrzött kivételek:
  - Bárhol dobhatók egy metódus vagy konstruktor törzsében.
  - A nem ellenőrzött kivételosztályok a `java.lang.RuntimeException` vagy a `java.lang.Error` osztály alosztályai.

Ellenőrzött és nem ellenőrzött kivételek (5)

- Az ellenőrzött kivételek használata megsértheti a nyitva zárt elevet.
  - Ha egy alacsony szintű metódust úgy módosítunk, hogy benne egy ellenőrzött kivételen kerüljön dobásra, akkor a metódus szignatúráját is megfelelően kell módosítani.
    - A `throws` záradéknak tartalmaznia kell a kivételosztályt.
  - Ez azt jelenti, hogy a módosított metódust meghívó minden metódust úgy kell módosítani, hogy kapja el a kivételt vagy hogy a `throws` záradéka tartalmazza a kivételt.

Ellenőrzött és nem ellenőrzött kivételek (6)

- További ajánlott olvasnivaló:
  - Philipp Hauer. *Checked Exceptions are Evil*. https://phauer.com/2015/checked-exceptions-are-evil/

Ne adjunk át/vissza `null`-t (1)

- A `null` referenciák azért rosszak, mert `NullPointerException` kivételeket okoznak.

Ne adjunk át/vissza `null`-t (2)

- Amikor `null` referenciát adunk vissza, akkor lényegében magunknak csinálunk munkát, mivel a `null`-t speciális esetként kell kezelni a `NullPointerException` elkerüléséhez.
- Ha kísértést érzünk egy metódusból `null` visszaadására, vegyük helyette fontolóra inkább egy kivétel dobását vagy egy speciális eset objektum (például egy üres lista) visszaadását.
  - Lásd még: `java.util.Optional<T>` https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Optional.html

Ne adjunk át/vissza `null`-t (3)

- Példa:

```
// A getEmployees() metódus visszaadhat null-t:
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

```
// A getEmployees() metódus nem add vissza null-t,
// hanem  helyette egy üres listát ad vissza:
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Ne adjunk át/vissza `null`-t (4)

- `null` visszaadása metódusokból rossz, de `null` átadása metódusoknak még rosszabb.
- Amikor csak lehetséges, el kell kerülni.
- A legtöbb programozási nyelven nincs jó megoldás a hívó által véletlenül átadott `null` referencia kezelésére.
  - Mivel ez a helyzet, az észszerű megközelítés alapértelmezésben megtiltani `null` átadását.

Ne adjunk át/vissza `null`-t (5)

- Null-értékűséget jelző annotációk (például `@NotNull`, `@NonNull`, `@Nullable`)
  - Felhasználhatják őket:
    - Statikus kódelemző eszközök (például Checker Framework, IntelliJ IDEA)
    - Kódgenerátor eszközök (például IntelliJ IDEA, Lombok)

Eszközök (1)

- *The Checker Framework* (programozási nyelv: Java; licenc: GPLv2 + *Classpath Exception*)
  https://checkerframework.org/
  https://github.com/typetools/checker-framework
  - Lásd: *Nullness Checker*
    https://checkerframework.org/manual/#nullness-checker
    - Megakadályozza a `NullPointerException` kivételeket.
    - Ha a *Nullness Checker* nem jelez figyelmeztetéseket egy adott programra, akkor a program futás közben soha nem fog `NullPointerException`-t dobni.

Eszközök (2)

- Lombok (programozási nyelv: Java; licenc: *MIT License*) https://projectlombok.org/ https://github.com/projectlombok/lombok
  - Lásd: `@NonNull` https://projectlombok.org/features/NonNull
    - Rekord komponenseken, metódus és konstruktor paramétereken használható.
    - Arra utasítja a Lombok-ot, hogy generáljon egy `null`-ellenőrző utasítást.

Eszközök (3)

- IntelliJ IDEA:
  - Lásd: Annotations https://www.jetbrains.com/help/idea/annotating-source-code.html
    - Statikus kódelemzás: támogatja a népszerű `null`-érték jelző annotációkat (például Checker Framework).
    - Kódgenerálás: `if` utasításokat ad hozzá az annotált kódelemekhez, melyek kivételt váltanak ki `null` esetén.
      - Probléma: csak az IDE-ben működik, jelenleg nincs támogatás *build* eszközökhöz.
        - Lásd: *Official `@NotNull` instrumenter maven and gradle plugin* https://youtrack.jetbrains.com/issue/IDEA-226562

Eszközök (4)

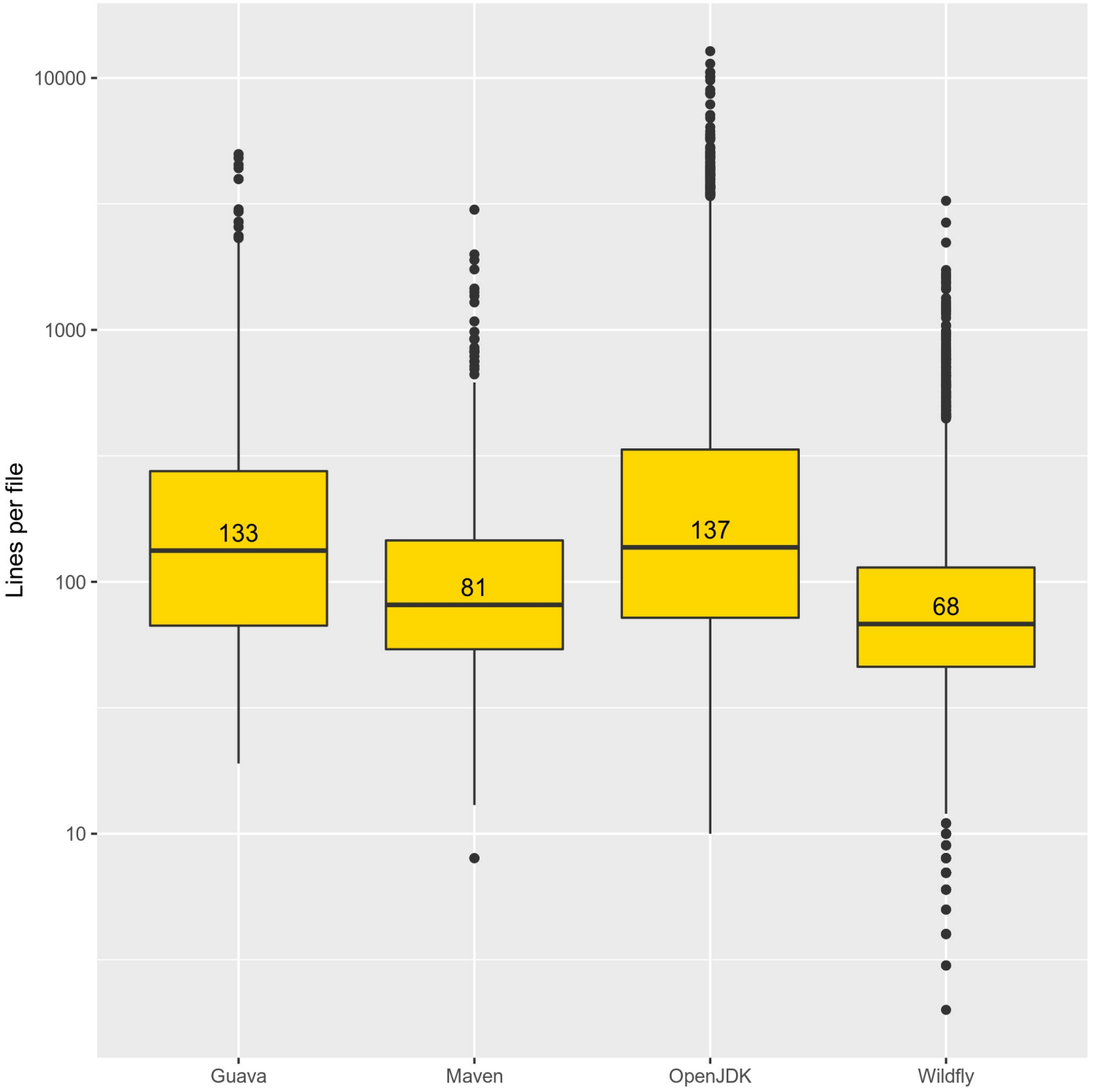
- IntelliJ IDEA:
  - *JetBrains annotations for JVM-based languages* (programozási nyelv: Java; licenc: *Apache License 2.0*) https://github.com/JetBrains/java-annotations
    - `@NotNull`
      https://javadoc.io/doc/org.jetbrains/annotations/latest/org/jetbrains/annotations/NotNull.html
    - `@Nullable`
      https://javadoc.io/doc/org.jetbrains/annotations/latest/org/jetbrains/annotations/Nullable.html

Formázás

- A forráskódot úgy kell formázni, hogy az jól olvasható legyen.
  - Az olvasó első benyomást a kódról a formázás alapján szerez.
- Egyszerű szabályokat kell választani a formázáshoz és azokat következetesen kell alkalmazni.
  - Csapatban történő fejlesztés esetén meg kell állapodni egy kódolási konvencióban és mindenkinek azt kell követni.
- Függőleges és vízszintes formázás.

Függőleges formázás (1)

- Mekkora legyen egy forrásállomány?
  - Empirikus vizsgálatként tekintsük a forrásállományok méretét az alábbi szoftverekben:
    - Apache Maven 3.8.4 https://maven.apache.org/
    - Guava 31.0.1 https://github.com/google/guava
    - OpenJDK 17.0.1 https://openjdk.java.net/
    - Wildfly 26.0.0.Final https://www.wildfly.org/
  - Az eredményt lásd a következő oldal doboz ábráján.


Lines per file
10000
1000
100
10
133
81
137
68
Guava
Maven
OpenJDK
Wildfly

Függőleges formázás (3)

- Újság metafora:
  - Egy jól megírt újságcikket felülről lefelé haladva olvasunk.
    - A tetején egy olyan cím van, mely alapján az olvasó eldöntheti, hogy érdekli-e egyáltalán. Az első bekezdés a teljes történet egy összefoglalását adja. Lefelé haladva a bekezdések egyre több és több részletet tartalmaznak.
  - Egy forrásállomány is legyen olyan, mint egy újságcikk.
    - A neve legyen egyszerű és beszédes. Az eleje magas szintű fogalmakat és algoritmusokat tartalmazzon. Lefelé haladva egyre nagyobb hangsúlyt kapnak a részletek. A végén legyenek a legalacsonyabb szintű függvények.

Függőleges formázás (4)

- Egy-egy üres sor jelöljön minden új fogalmat.
  - Válasszuk el egymástól a csomagdeklarációt, az import deklarációkat, a függvényeket, …

Függőleges formázás (5)

```
// Copyright (C) 2003-2009 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the CPL Common Public License version 1.0.
package fitnesse;

import java.io.IOException;
import java.net.Socket;
import java.util.concurrent.ExecutorService;

import fitnesse.socketservice.SocketServer;

public class FitNesseServer implements SocketServer {
  private final FitNesseContext context;
  private final ExecutorService executorService;

  public FitNesseServer(FitNesseContext context, ExecutorService executorService) {
    this.context = context;
    this.executorService = executorService;
  }

  @Override
  public void serve(Socket s) throws IOException {
    serve(s, 10000);
  }
  // ...
```

Függőleges formázás (6)

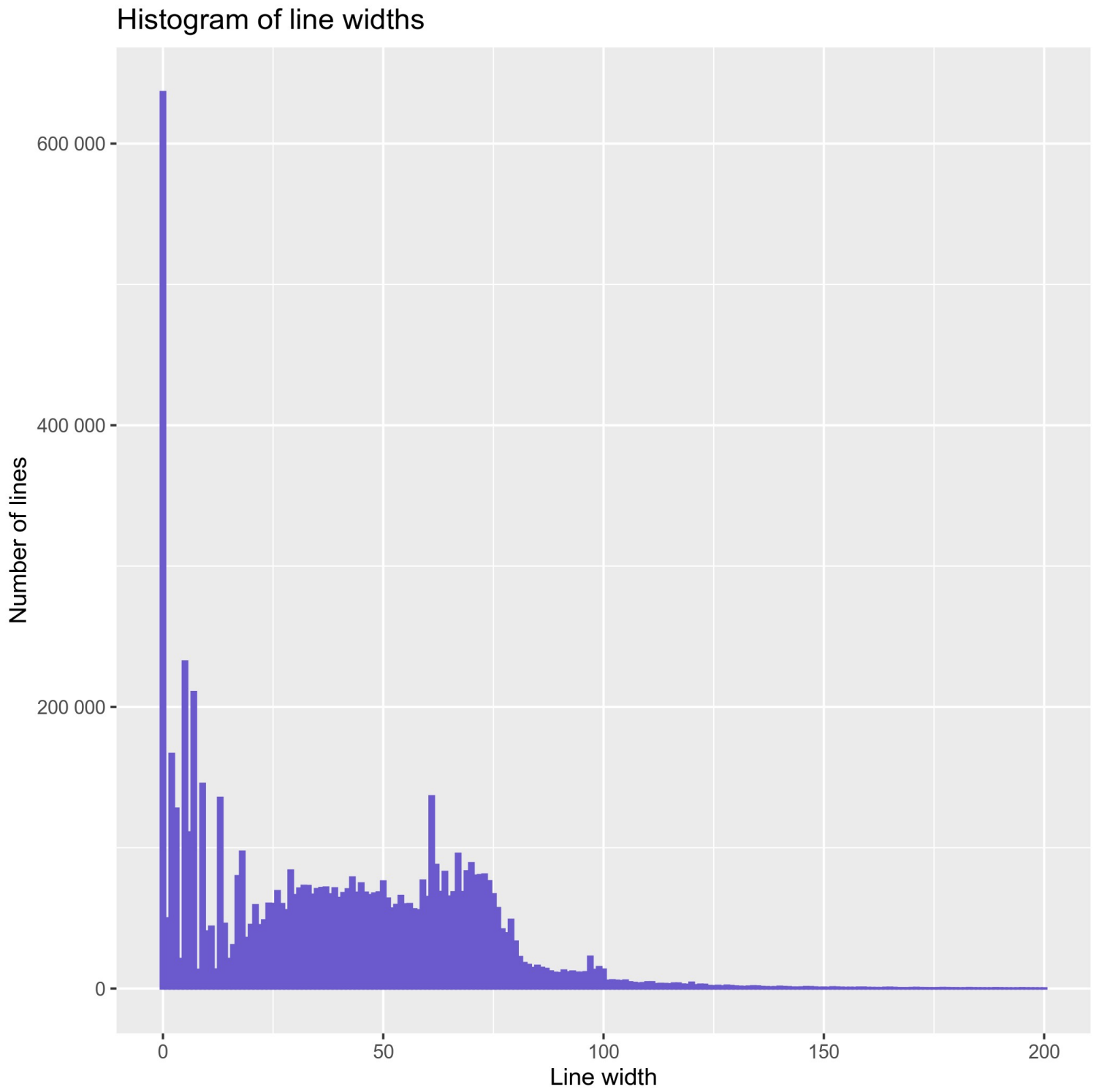
- A szorosan kapcsolódó programsorok sűrűn kell, hogy megjelenjenek.
  - Ne legyenek közöttük megjegyzések vagy üres sorok.
- A szorosan kapcsolódó fogalmakat tartsuk függőlegesen egymáshoz közel.
  - Csak nagyon indokolt esetben kerüljenek külön állományokba.
  - Függőleges távolságuk tükrözze azt, hogy mennyire fontos az egyik a másik megértéséhez.

Függőleges formázás (7)

- A változókat deklaráljuk a használatuk helyéhez a lehető legközelebb.
  - Mivel a függvények rövidek, a lokális változókat az elejükön deklaráljuk.
  - A ciklusváltozókat lehet a ciklusokban.
- A példányváltozókat az osztályok elején kell deklarálni.
- Ha egy függvény meghív egy másikat, akkor függőlegesen közel kell, hogy legyenek egymáshoz, és ha ez egyáltalán lehetséges, akkor a hívó előzze meg a hívottat.

Vízszintes formázás (1)

- Milyen hosszú legyen egy programsor?
  - Empirikus vizsgálatként tekintsük a kódsorok hosszának eloszlását az alábbi szoftverekben:
    - Apache Maven 3.8.4 https://maven.apache.org/
    - Guava 31.0.1 https://github.com/google/guava
    - OpenJDK 17.0.1 https://openjdk.java.net/
    - Wildfly 26.0.0.Final https://www.wildfly.org/
  - A vizsgálat eredményét a következő oldalon látható hisztogram mutatja.


Histogram of line widths
600 000
400 000
200 000
0
Number of lines
0
50
100
150
200
Line width

Vízszintes formázás (3)

- A sorok ne legyenek túl hosszúak.
  - Ne legyenek hosszabbak például 120 karakternél.

Vízszintes formázás (4)

- Használjunk szóközöket a gyengén összetartozó elemek között.
  - Például értékadó operátor két oldalán. Ne tegyünk viszont szóközt például egy függvény neve és az azt követő nyitó zárójel karakter közé.
  - Példa:

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Vízszintes formázás (5)

- Nem érdemes a deklarációkban és értékadásokban a neveket és a kifejezéseket igazítani.
  - Kerülendő például:

```
package fit;

public class Counts {
  public int right =      0;
  public int wrong =      0;
  public int ignores =    0;
  public int exceptions = 0;

  public Counts(int right, int wrong, int ignores, int exceptions) {
    this.right =      right;
    this.wrong =      wrong;
    this.ignores =    ignores;
    this.exceptions = exceptions;
  }
  // ...
```

Vízszintes formázás (6)

- Nagyon fontos a megfelelő behúzás.
  - Használható szóköz és tabulátor is.
  - Ne szegjük meg a behúzási szabályt rövid `if` utasítások, ciklusok vagy függvények kedvéért sem.
    - Kerülendő például:
      - `public int size() { return size; }`

Vízszintes formázás (7)

- Ha `for` vagy `while` ciklus törzseként üres utasítást kell használni, ezt célszerű egy új sorba elhelyezni.
  - Példa:

```
while ((c = read()) != '\n' && c != '\r' && c >= 0);
```

helyett inkább legyen

```
while ((c = read()) != '\n' && c != '\r' && c >= 0)
  ;
```

Kódolási konvenciók (1)

- Lásd még: kódolási stílus (*coding style*), kódolási szabvány (*coding standard*)

Kódolási konvenciók (2)

- **Google**: *Google Style Guides*
  https://google.github.io/styleguide/
  https://github.com/google/styleguide
  - Programozási nyelvek: C++, C#, HTML/CSS, Java, JavaScript, Python, R, Shell, …

Kódolási konvenciók (3)

- **C**:
  - *GNU Coding Standards* https://www.gnu.org/prep/standards/standards.html
  - *Linux kernel coding style – The Linux Kernel documentation* https://www.kernel.org/doc/html/latest/process/coding-style.html
  - *Kernighan and Ritchie* (K&R)
- **C++**:
  - Bjarne Stroustrup. *PPP Style Guide*. http://www.stroustrup.com/Programming/PPP-style.pdf
    - Ehhez a könyvhöz: Bjarne Stroustrup: *Programming: Principles and Practice using C++*. Second Edition. Addison-Wesley, 2014.
  - Bjarne Stroustrup et al*. C++ Core Guidelines.* http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines https://github.com/isocpp/CppCoreGuidelines
  - *Google C++ Style Guide* https://google.github.io/styleguide/cppguide.html

Kódolási konvenciók (4)

- **C#**:
  - *C# Coding Conventions (C# documentation)* https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions
  - *C# at Google Style Guide* https://google.github.io/styleguide/csharp-style.html
- **Java**:
  - *Code Conventions for the Java Programming Language* https://www.oracle.com/java/technologies/javase/codeconventions-contents.html
    - Elavult!
  - *Google Java Style Guide* https://google.github.io/styleguide/javaguide.html
    - Lásd: https://github.com/google/styleguide/blob/gh-pages/intellij-java-google-style.xml
  - Andreas Lundblad. *Java Style Guidelines*. http://cr.openjdk.java.net/~alundblad/styleguide/index-v6.html
  - Stuart W. Marks. *Local Variable Type Inference: Style Guidelines*. March 2018. https://openjdk.org/projects/amber/guides/lvti-style-guide

Kódolási konvenciók (5)

- **JavaScript**:
  - *JavaScript Standard Style*
    https://standardjs.com/rules.html
  - *Airbnb JavaScript Style Guide*
    https://github.com/airbnb/javascript
  - *Google JavaScript Style Guide*
    https://google.github.io/styleguide/jsguide.html
  - Douglas Crockford. *Code Conventions for the JavaScript Programming Language*.
    https://www.crockford.com/code.html

Kódolási konvenciók (6)

- **Python**:
  - *Google Python Style Guide*
    https://google.github.io/styleguide/pyguide.html
  - *PEP 8 – Style Guide for Python Code*
    https://www.python.org/dev/peps/pep-0008/
- **R**:
  - *Google's R Style Guide*
    https://google.github.io/styleguide/Rguide.html
  - Hadley Wickham. *The tidyverse style guide*.
    https://style.tidyverse.org/

Eszközök (1)

- *indent* (programozási nyelv: C; licenc: GPLv3) https://www.gnu.org/software/indent/
  - Parancssori program C nyelvű forráskód formázáshoz.
  - Beépített kódolási konvenciók: GNU (`-gnu`), Linux (`-linux`), K&R (`-kr`)

Eszközök (2)

- *cpplint* (programozási nyelv: Python; licenc: *New BSD License*) https://github.com/cpplint/cpplint
  - Parancssori eszköz C++ nyelvű forráskód a Google C++ stílusnak való megfelelésének ellenőrzéséhez.
- *google-java-format* (programozási nyelv: Java; licenc: *Apache License 2.0*) https://github.com/google/google-java-format
  - Java forráskód a Google Java stílusnak megfelelő formázása.
  - Könyvtár, parancssori eszköz, IDE bővítmény (IntelliJ IDEA, Eclipse).
- *JavaScript Standard Style* (programozási nyelv: JavaScript; licenc: *MIT License*) https://standardjs.com/ https://github.com/standard/standard
  - TODO: JavaScript style guide, linter, and formatter.

Eszközök (3)

- *Checkstyle* (programozási nyelv: Java; licenc: LGPLv2.1) https://checkstyle.org/ https://github.com/checkstyle/checkstyle/
  - Java kódolási konvencióknak való megfelelés ellenőrzése.
  - Beépített konfigurációs állományok az alábbi kódolási konvenciókhoz:
    - Oracle (`sun_checks.xml`)
    - Google Java Style (`google_checks.xml`)
  - Fejlesztőeszköz támogatás: https://checkstyle.org/#Related_Tools_Active_Tools

Eszközök (4)

- *EditorConfig* https://editorconfig.org/ https://github.com/editorconfig/
  - Állományformátum kódolási konvenciók definiálásához.
  - Számos szövegszerkesztő és integrált fejlesztői környezet támogatja (például Eclipse, IntelliJ IDEA, NetBeans).
    - Natív módon vagy bővítmény révén.

Eszközök (5)

- **IntelliJ IDEA**:
  - *Code style and formatting*
    https://www.jetbrains.com/help/idea/code-style.html

További ajánlott források

- Al Sweigart. *Beyond the Basic Stuff with Python: Best Practices for Writing Clean Code*. No Starch Press, 2020. https://inventwithpython.com/beyond/
- Mariano Anaya. *Clean Code in Python*. 2nd ed. Packt Publishing, 2021. https://github.com/PacktPublishing/Clean-Code-in-Python-Second-Edition
- Stephan Roth. *Clean C++20: Sustainable Software Development Patterns and Best Practices*. 2nd ed. Apress, 2021. https://github.com/Apress/clean-cpp20
- James Padolsey. *Clean Code in JavaScript*. Packt Publishing, 2020. https://github.com/PacktPublishing/Clean-Code-in-JavaScript
- Robert C. Martin. *The Clean Code Blog*. https://blog.cleancoder.com/