

UINF/PAZ1c

Programovanie, algoritmy, zložitosť

public class Java void trieda new private String DAO pre prvok kód meniť metóda dátá OOB CitatDao static vieme prvkov biznis test JFrame interfejs Elipsa GUI objekt metód MainForm nie alebo každý tovar return továreň premenné správanie autor ÚINF/PAZ1c implementácia IdbcTemplate automat vložíMincu interfejsy používateľ boolean kontrakt List Citat Návrh testy zoznam metódy stav ktorá len pri int throws Jave máme Riešenie Kružnica add(new môže Automat výnimky aplikácie schopnosti komponentov inštanciu problém závislosti ActionListener interface

Motívy predmetu



- pokročilé OOP
 - dizajn a udržiavateľnosť programov
 - príklady z praxe
 - okienkové aplikácie (JavaFX)
 - databázy
 - Webové aplikácie (React + SpringBoot)
- 

Hodnotenie

- <https://paz1c.ics.upjs.sk>
- Zápočty: 28.10, 9.12
 - 2x30b, min. 20b
- Záverečné projekty: 16.12 + február
- 1. projekt (povinný)
 - Okienková appka
 - 34b, min 10b
 - Body aj pre DBS1a (4.11.)
- 2. projekt (nepovinný)
 - Webová appka
 - 10b

Výsledná známka:

A: 90+

B: 70

C: 55

D: 45

E: 37

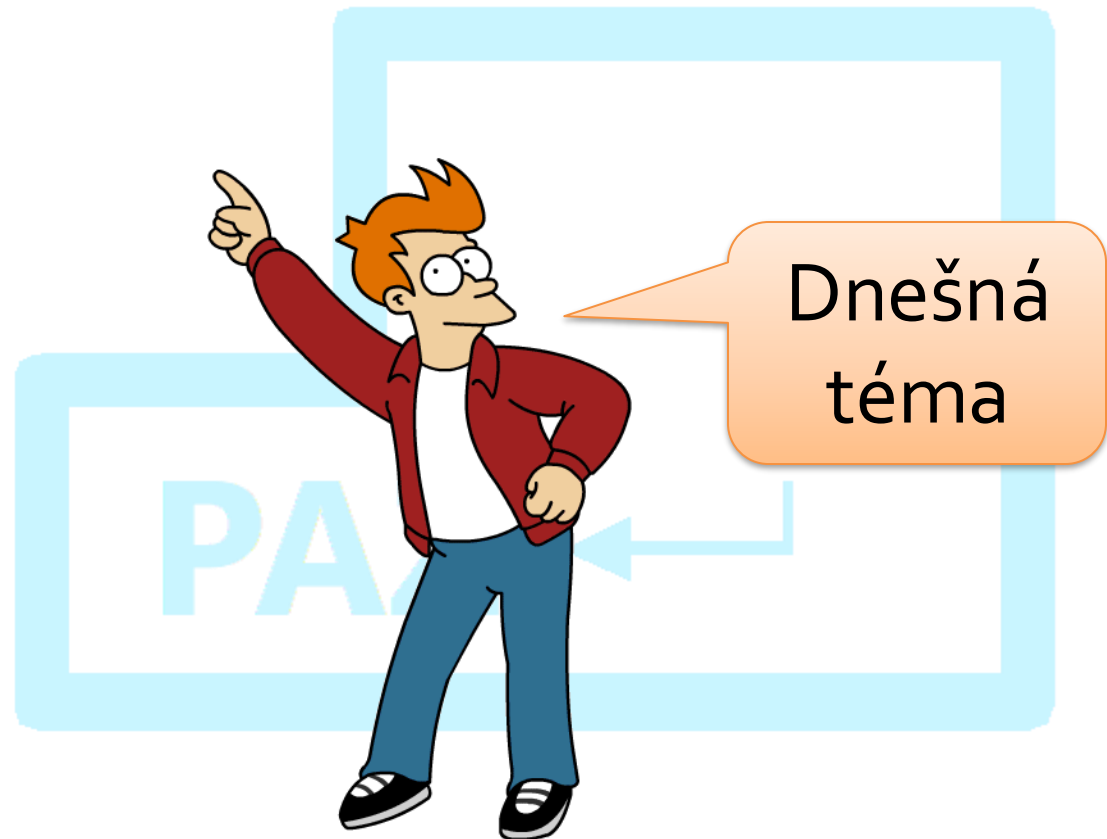
Hodnotenie

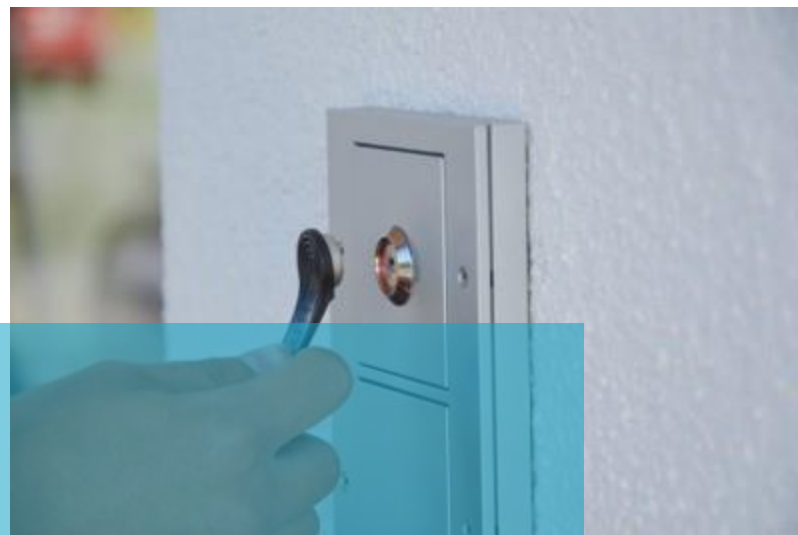
Skúša sa funkčnosť projektu a schopnosť študenta orientovať sa vo vlastnom zdrojovom kóde.

Nie je zakázané používať knižnice ani zdroje 3. strán, ani AI asistentov.

Pozor na prílišné spoliehanie sa na AI - neznalosť "vlastného" kódu sa hodnotí FX – projekt je zjavne plagiát.

Dizajn a implementácia tried a unit testy





projekt:
Prístupový systém



pozbierajme používateľské požiadavky

- po prečítaní čipu zapípa
- keď priložím správny čip, otvorí dvere
- čítačka musí byť čierna
- admin pridáva nové čipy do prístupového systému
- chcem vedieť, kto má ktorý čip
- ak niekto stratí čip alebo odíde z firmy/školy/bytovky, chcem ho vedieť zo systému odstrániť
- chcem mať záznamy o tom, kto kedy otvoril dvere
- ak vypadne elektrina, systém musí fungovať



Louisiana under
the French Company
of the Indies, 1717-1731

softvérová analytčka
formalizuje používateľské
požiadavky

THE HISTORY OF THE CITY OF NEW ORLEANS
Free and Open to the Public
533 Royal Street
www.hnuc.org • (504) 823-4662

**Tarot
& Palm
Readings**



čo zákazník
vysvetlil



čo naozaj
chcel



čo pochopil
analytik



čo bolo
nakódené

Use-case

zoznam krokov, ktoré vykonáva
aktor so systémom
v úmysle dosiahnuť svoj cieľ.

ivar jacobson





Úmysel: chcem otvoriť dvere

Aktorom je vlastník čipu

Úmysel: chcem pridať čip

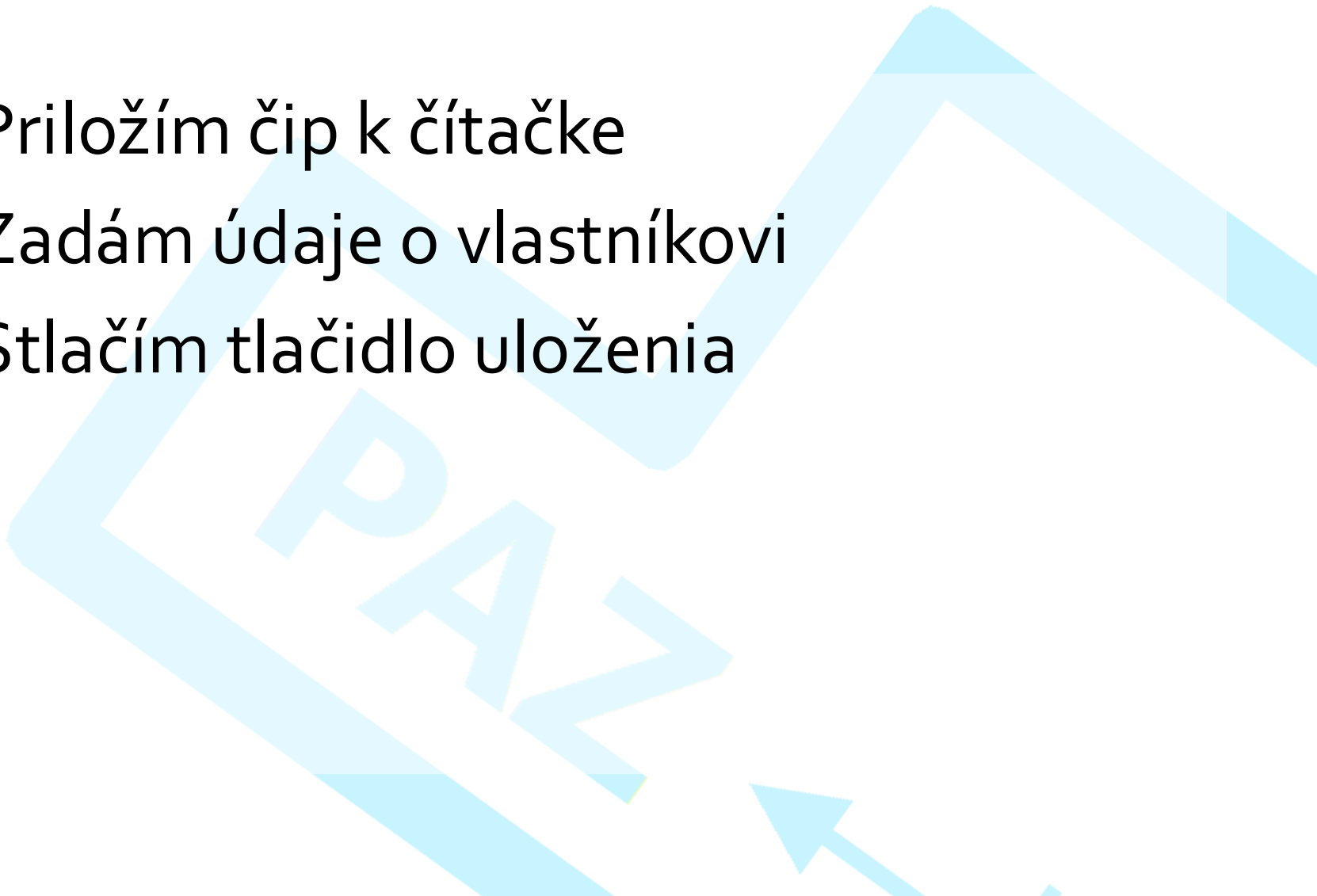
Aktorom je správca systému

Úmysel: chcem vidieť príchody

Aktorom je správca systému...

use-case: chcem pridať čip



1. Priložím čip k čítačke
 2. Zadám údaje o vlastníkovi
 3. Stlačím tlačidlo uloženia
- 

use-case: chcem pridať čip

1. Priložím čip k čítačke
2. Zadám údaje o vlastníkovi
3. Stlačím tlačidlo uloženia

Optimistický
scenár !

Je možné donekonečna vylepšovať a pokrývať
záchranné, alebo neúspešné scenáre

use-case: chcem pridať čip



1. Priložím čip k čítačke

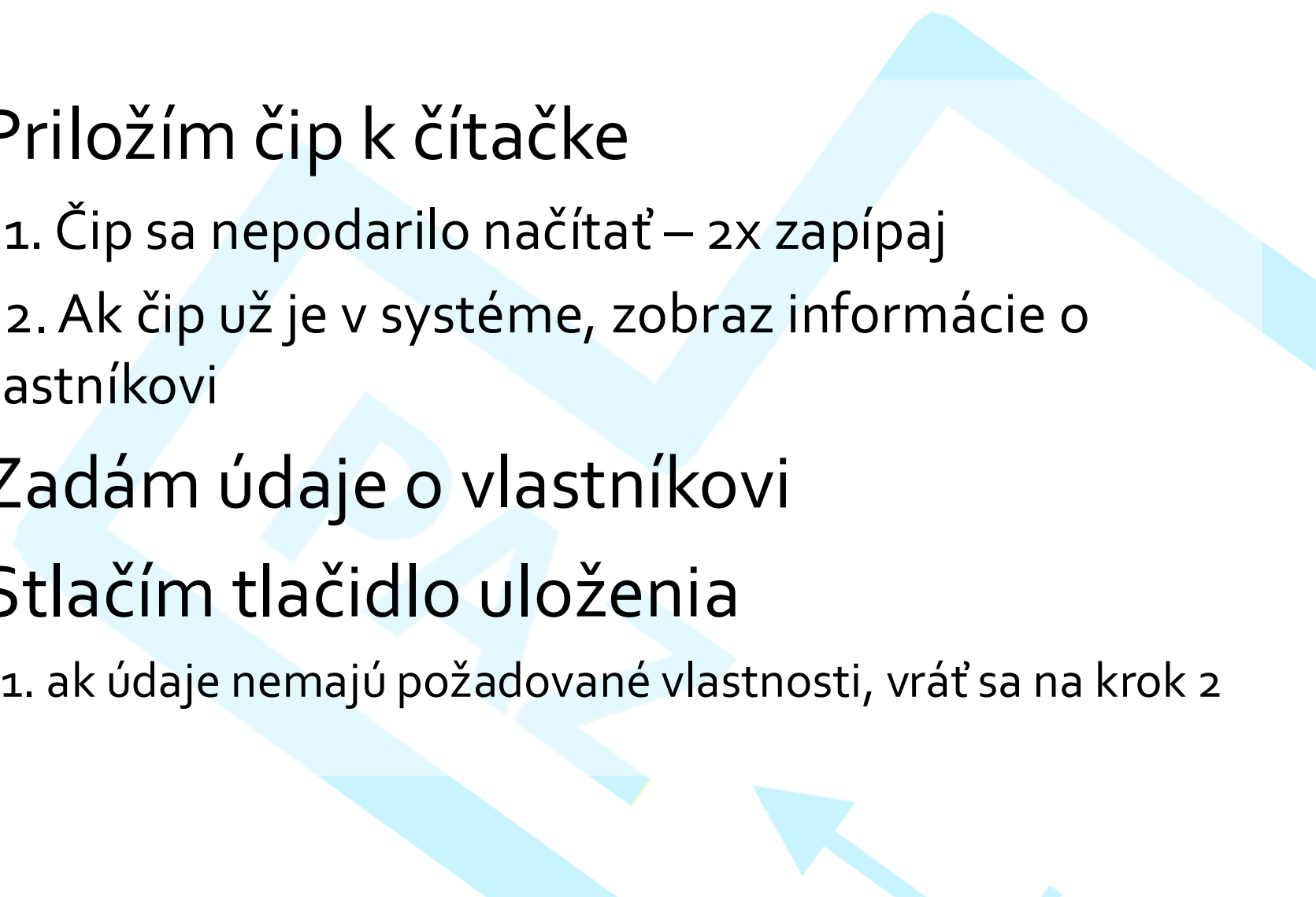
1.1. Čip sa nepodarilo načítať – 2x zapípaj

1.2. Ak čip už je v systéme, zobraz informácie o vlastníkovi

2. Zadám údaje o vlastníkovi

3. Stlačím tlačidlo uloženia

2.1. ak údaje nemajú požadované vlastnosti, vráť sa na krok 2



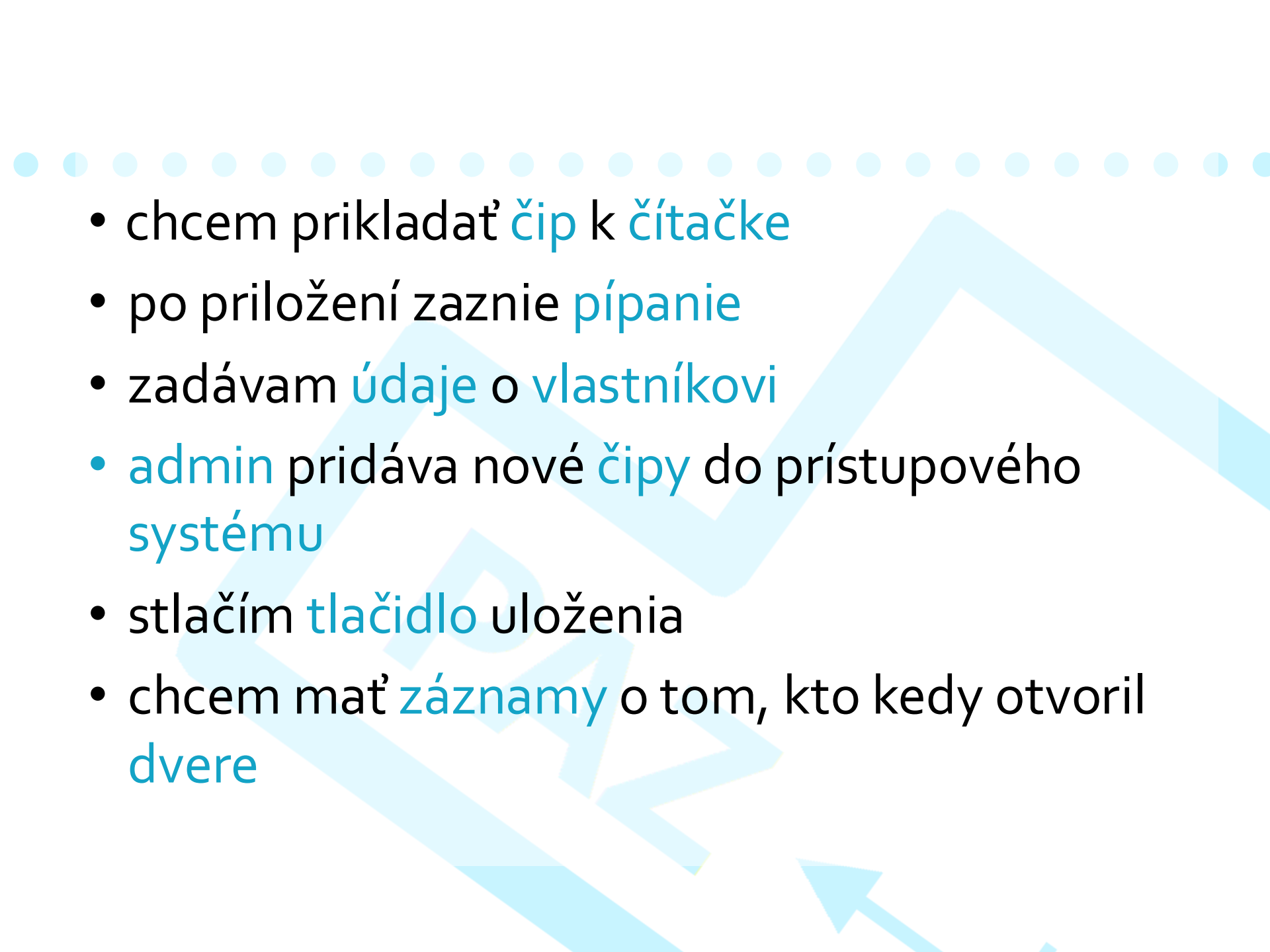
identifikujme triedy



identifikujme triedy



podstatné mená sú
kandidáti pre triedy

- 
- chceme prikladať **čip** k čítačke
 - po priložení zaznie **pípánie**
 - zadávam **údaje** o **vlastníkovi**
 - **admin** pridáva nové **čipy** do prístupového systému
 - stlačím **tlačidlo** uloženia
 - chceme mať **záznamy** o tom, kto kedy otvoril dvere



Trieda by mala mať schopnosti a stav!

najprv schopnosti!

schopnost' = rozkaz = metoda



skáč!

zlez!

potom stav

stav = inštančná premenná



druh: lev

farba: #ADADAD

meno: Simba

Stav v Jave...

```
public class User {  
    private String name;  
    private boolean active;  
    ...  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (o == null || getClass() != o.getClass()) return false;  
        User user = (User) o;  
        return active == user.active && Objects.equals(name, user.name);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(name, active);  
    }  
}
```

Stav v Jave...

```
public class User {  
    private String name;  
    private boolean  
    ...  
    public String get  
        return name;  
    }  
  
    public void setName  
        this.name = n  
    }  
  
    @Override  
    public boolean e  
        if (o == null |  
            User user = (U  
        return active  
    }  
  
    @Override  
    public int hashC  
        return Object  
    }  
}
```



Stav v Java...

```
public class User {  
    private String name;  
    private boolean active;  
    ...  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (o == null || getClass() != o.getClass()) return false;  
        User user = (User) o;  
        return active == user.active && Objects.equals(name, user.name);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(name, active);  
    }  
}
```

equals pre porovnanie objektov podľa hodnôt inštančných premenných a nie podľa referencií

Vyskúšajte, čo vypíše keď User **má/nemá** equals

```
var jano1 = new User();  
jano1.setName("Jano");
```

```
var jano2 = new User();  
jano2.setName("Jano");
```

```
System.out.println(jano1.equals(jano2));
```

hashCode pre "pred"-porovnanie objektov v HashMap/HashSet podľa hashe inštančných premenných a nie podľa referencií

Fakt to musíme písať?

- Gettre, settre, equals, hashCode vieme generovať
 - Pravý klik/Generate...

Fakt to musíme písať?



Človek ľahko zabudne

Record triedy (Java 14+),

@Data (Lombok)

- Bez písania do triedy vložia gettre (**@Data** aj settre)
- Vložia korektné **equals** a **hashCode**, konštruktor
- Records sú nemenné (**immutable**)
 - Nemáme settre, nevieme ani priamo priradiť do premennej
 - Používanie settra na objekt v HashMap/HashSet ticho rozbije dátovú štruktúru (aka memleaky v GC jazyku...)
 - Treba robiť kópie ak chceme niečo zmeniť
 - Odporúčam použiť **@With** z Lomboku pre efektívne robenie kópií so zmenenou inšt. premennou
 - Dobré pre konkurentný kód a celkovo efektivitu CPU
 - Prekladače vedia lepšie optimalizovať inštrukcie, ak majú garanciu, že existujúce dáta sa nemôžu zmeniť

Record - Záznam

```
public record User(  
    String name,  
    boolean active  
) {}
```

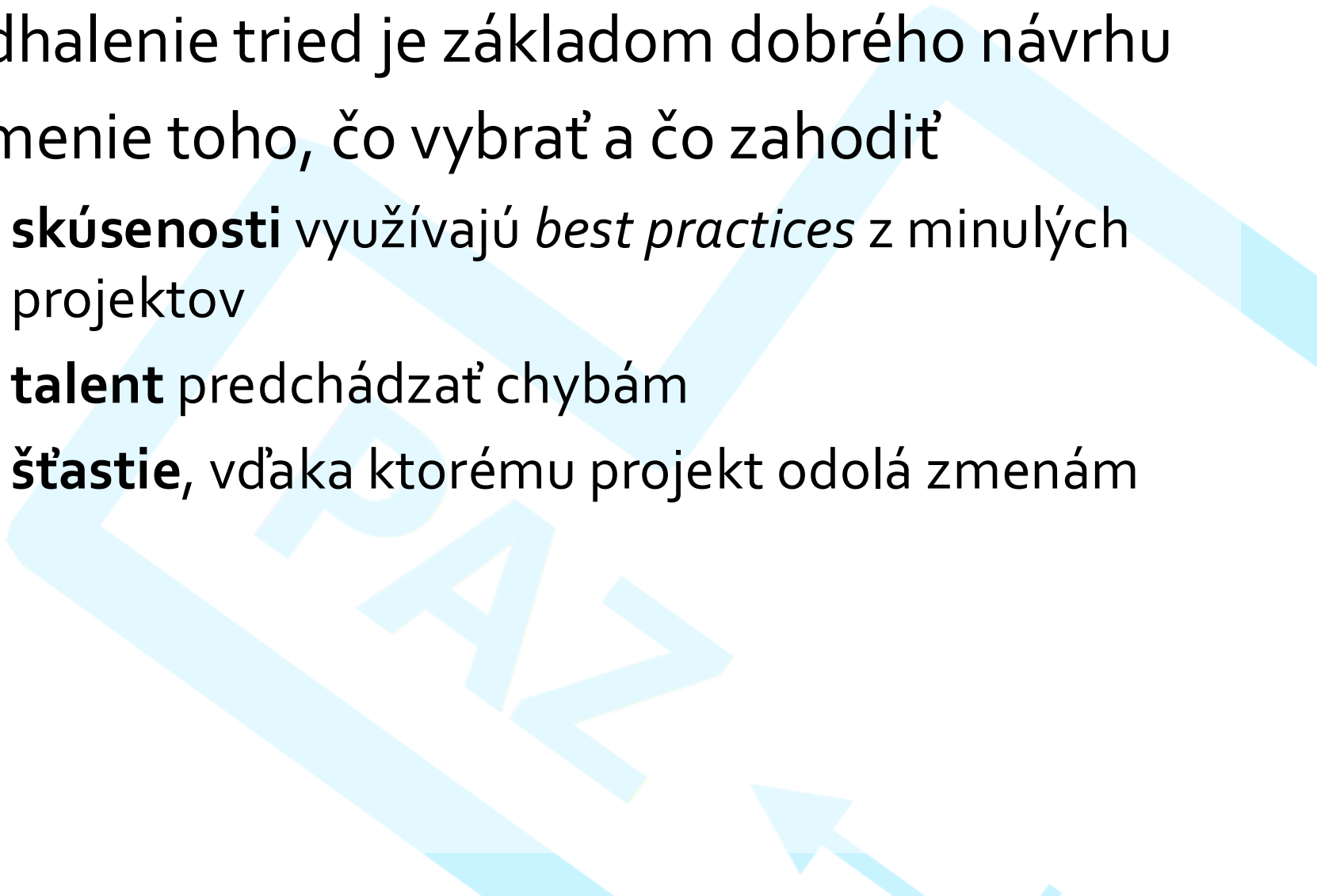
- Record má **hlavný konštruktor**
- Parametre konštruktora == privátne inštančné premenné (navyše **final**), ktoré už nepíšeme
- Verejné gettre **name()** a **active()**, ktoré nepíšeme
- Ináč je to normálna trieda

iba rozumné triedy

- triedy bez schopností a stavu sú zbytočné
 - údaj, pípanie
- triedy, ktoré môžu presunúť zodpovednosti na iné triedy sú zbytočné
 - „čip s číslom a vlastník s menom“ -> „vlastník s menom a číslom čipu“

dobrý návrh nad zlato



- odhalenie tried je základom dobrého návrhu
 - umenie toho, čo vybrať a čo zahodiť
 - **skúsenosti** využívajú *best practices* z minulých projektov
 - **talent** predchádzať chybám
 - **šťastie**, vďaka ktorému projekt odolá zmenám
- 



**And So
You Code**

dôraz na udržiavateľnosť

- s narastajúcou veľkosťou projektu sa zvyšujú nároky na **udržiavateľnosť**
 - udržiavateľný (objektový) návrh je nutnosť
- programátor vyvíja softvér **pre ostatných**, nie pre seba
 - snaha predísť write-only kódu
- väčší projekt je skladačka, ktorá funguje, iba **ak jej časti robia to, čo robiť majú**
 - ako vieme, čo funguje a čo nie?



Ako otestovať našu triedu?

- Java začiatočník: vytvorí metódu `main()`
- výstupy riešime cez `System.out.println()`
- kontrolu urobíme od oka
- ak vidíme výstup, je to OK

Lesk a bieda main()

- Čo ak máme viac variantov použitia?
- V triede môže byť len jedna metóda main()!
- **Hlúpe riešenie** č. 1:
 - Podľa potreby odkomentovávame a zakomentovávame výseky kódu.
- **Hlúpe riešenie** č. 2:
 - vytvoriť viac testovacích tried
 - každá má svoj **main()**
 - spúšťame podľa potreby



Meditácie nad main()om

- Čo ak potrebujeme testovať **viacero vstupov**?
 - Budeme čítať z klávesnice dokola?
 - Kedy nás to prestane baviť?
- Kedy nás prestane baviť **okom kontrolovať** výstupy vo veľkom softvéri?
- ~~Prečo má byť **testovací kód na kope** s reálnym kódom?~~
 - V prípade, že máme **main()** rovno v testovanej triede...



Chyby, chyby, chyby

- každý softvér obsahuje chyby!
 - *Code Complete*: 15-50 chýb na 1000 riadkov dodaného kódu
- testovaním predchádzame chybám
- viacero druhov testov a viacero klasifikácií

~~Kto neprogramuje, nech ani neje!~~
Kto netestuje, nech ani neprogramuje!

Rozličné druhy testov

Účel

- korektnosť
- výkonnosť
- spoľahlivosť
- zabezpečenie

Zámer

- **jednotkové testy**
- **komponentové testy**
- **integračné testy**
- **koncové testy (e-2-e)**
- **výkonnostné testy**
- **testy zabezpečenia**
- **testy nasadenia**
- **akceptačné testy**
- **systémové testy**



Unit Testy!



Unit testy / Jednotkové testy

Robí kód naozaj to, čo chceme aby robil?

- testovanie jednotiek kódu na korektnosť použitia
- **unit** = najmenšia testovateľná časť aplikácie
- v OOP = **test triedy** / rozhrania
 - v procedurálnom programovaní: test procedúry
- testujeme malé, izolované časti funkcionality
- ak fungujú malé časti, (možno) budú fungovať aj tie väčšie

Základná idea unit testov

- pre testované metódy definujeme
 - testovací vstup
 - očakávaný výstup
- **test:** je **vypočítaný** výstup zhodný s **očakávaným** výstupom?
 - ak áno, test uspel
- definujeme toľko testovacích vstupov a očakávaných hodnôt, koľko treba.



Unit testy a JUnit

- JUnit – de facto štandardný framework pre unit testy v Java
- zabudovaná podpora v každom schopnom IDE: Eclipse / IntelliJ / NetBeans



org.junit

Aktuálne Junit 5 (jupiter)

- milión metód pre porovnanie očakávaného a vypočítaného vstupu

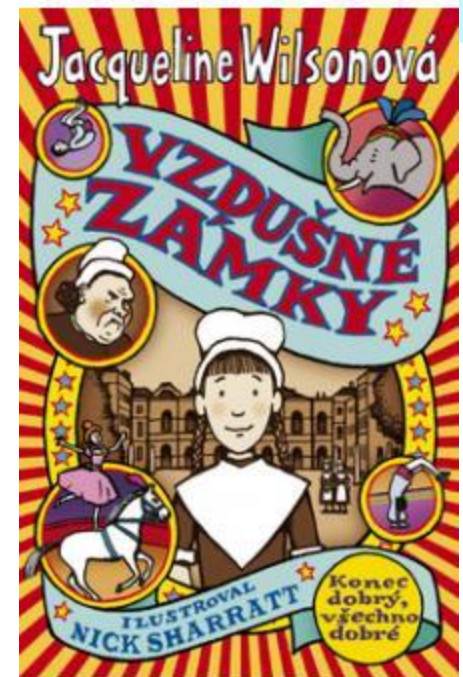
```
assertEquals() assertNull() assertNotNull() assertFalse() assertTrue() fail()
```

- test na vyhodnenú výnimku

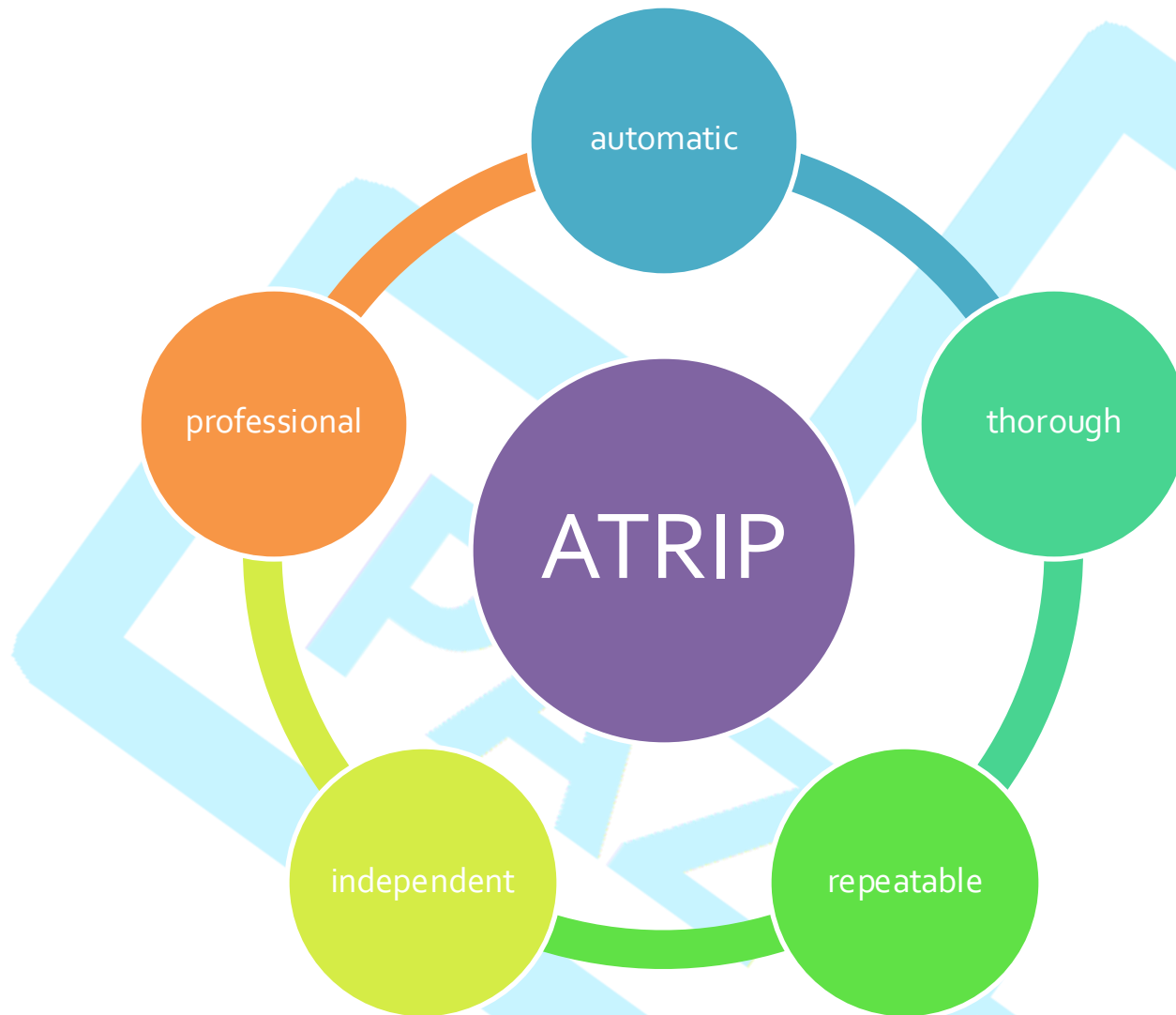
```
assertThrows(IllegalArgumentException.class, f())
```

Poznámky k unit testom

- každá verejná metóda triedy má mať test/y
- v **ideálnom svete** otestujeme správanie pre:
 - každý riadok kódu
 - každú vetvu kódu
 - každú výnimku, ktorá môže nastať
- obvyklá **realita**:
 - zopár bežných vstupov
 - typické hlúpe a hraničné vstupy
 - chýbajúce či nesprávne dáta
 - ...



Vlastnosti dobrého testu - ATRIP



Vlastnosti dobrého testu - ATRIP

- automatic(ký)
 - máme mať možnosť spustiť ich pred vydaním verzie automaticky
- thorough (podrobný)
 - pokrytie kódu (code coverage)
 - cieľ: typicky 70-80% pokrytie programu
 - nad 90% už väčšinou nepraktické
- repeatable (opakovateľný)
 - pri každom behu rovnaké výsledky
 - nezávislý od vonkajšieho prostredia

Vlastnosti dobrého testu - ATRIP

- independent (**nezávislý**)
 - testy sú navzájom nezávislé
 - v jednom teste overujeme len jednu konkrétnu verejnú metódu
 - paralelizovateľné na viac CPU jadier
- professional
 - unit testy sú rovnako dôležité ako zvyšok kódu
 - pomer riadkov kódu k riadkom testu: 1:1 – 1:3
 - netestujeme zbytočnosti
 - testy jednoriadkových getterov, setterov, prázdnych konštruktorov

Testovanie
zabíja čas!

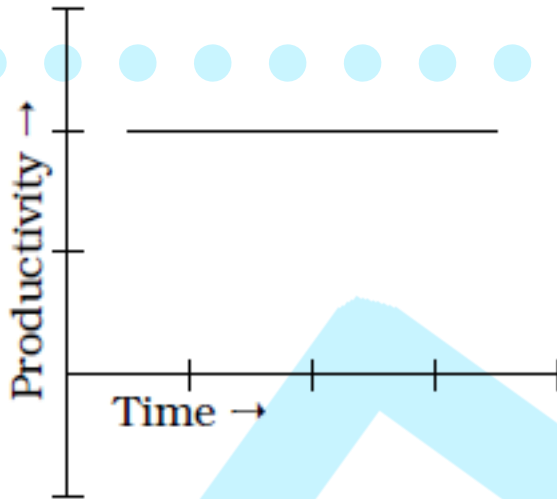
- ~~čas, ktorý zabijem písaním
testu môžem venovať
ďalším vlastnostiam!~~

Mýtus od počiatku IT!

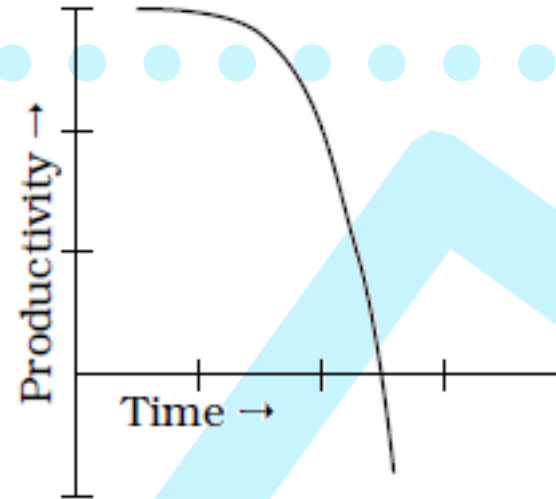


- pretože čas investovaný do testov sa vráti v dlhodobom horizonte

PAY-AS-YOU-GO

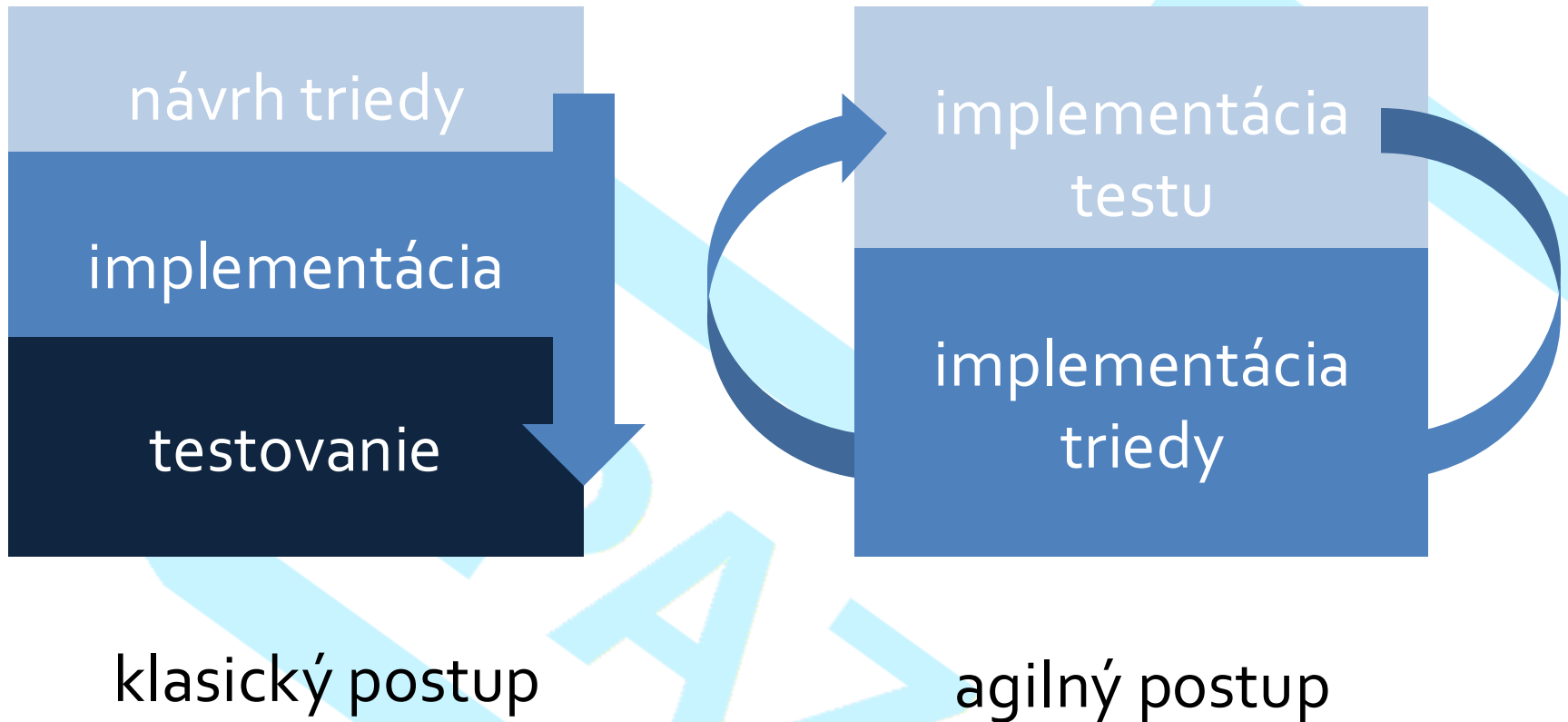


SINGLE TEST PHASE



- klasický spôsob: testujeme na záver projektu
- pri použití testov:
 - vyššia priebežná investícia
 - tá je však konštantná
 - ale menej priebežných chýb: vždy pracujeme s otestovaným kódom!

Test Driven Development - Napíšte najprv testy!



**I DON'T ALWAYS TEST
MY CODE**

**BUT WHEN I DO I DO IT
IN PRODUCTION**

MEMECREATOR.NET

- Ručne kompletne otestovať appku trvá hodiny-dni-mesiace.
- Zákazník nebude ochotný platiť za zabugovaný program
 - **Reklamácie programu vás oberajú o čas a o reputáciu**
- Doslova nik na to nemá čas to robiť pred vydaním novej verzie programu (1x mesiac – 2x do roka)
- **Ani AI na to nestačí**
- **Treba čo najviac pokryť naskriptovanými testovacími scenármi.**

**YOU DAWG, I HEARD YOU LIKE
UNIT TESTS**

**SO I WROTE A UNIT TEST FOR
YOUR UNIT TEST**

memegenerator.net

"Testy by sa mali čítať ako kniha"

Zhora nadol, zľava doprava

- Nepoužívajte návrhové vzory
- Nepoužívajte "chytrú" architektúru kódu
- Kludne copy-pastujte celé telá testov a upravujte len zopár riadkov
- Test by mal prežiť zmenu kódu.
 - Zabezpečuje **spätnú kompatibilitu**
 - Najlepšia appka je tá, ktorá neodstraňuje zaužívanú funkcionality, iba ju vylepšuje, alebo pridáva novú.
- Ideálne testovací kód je starší ako reálny kód.

odnesme si domov

- zozbierajme prípady použitia
- identifikujme triedy podstatnými menami
- identifikujme
 - 1. správanie
 - 2. stav metód
- najprv píšme test, potom kód



Otázky?