UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

BORUT LESJAK

UGOTAVLJANJE ISTOVETNOSTI RAČUNALNIŠKIH SISTEMOV

UNIVERZITETNO DIPLOMSKO DELO

LJUBLJANA, 2008

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

BORUT LESJAK

UGOTAVLJANJE ISTOVETNOSTI RAČUNALNIŠKIH SISTEMOV

UNIVERZITETNO DIPLOMSKO DELO

MENTOR DOC.DR.TOMAŽ DOBRAVEC

LJUBLJANA, 2008

original izdane teme

ZAHVALA

Na prvem mestu se iskreno zahvaljujem bratrancu Tinetu Lesjaku za vso njegovo spodbudo in nesebično pomoč ter dobro voljo pri pripravah na izpite. Brez njega mi ne bi bilo tako enostavno priti do sem. Burim!

Iz srca se za podporo zahvaljujem tudi družini in prijateljem, ki niso dvomili vame.

Posebna zahvala pa gre tudi mentorju Tomažu Dobravcu za idejno in tehnično pomoč ter za potrpežljivost in pozornost, predvsem pa, ker mi je omogočil, da sem to delo opravljal sproščeno in z veseljem.

POSVETILO

Svojima staršema, Olgi in Leopoldu, ki sta dolgo čakala na to, v znak otroške ljubezni.

KAZALO

1. POVZETEK

Ugotavljanje istovetnosti računalniških sistemov

Istovetnost računalniških sistemov je v računalništvu prav tako pomemben koncept, kot je pomembna istovetnost posameznika v človeški družbi. Vendar pa je istovetnost računalniškega sistema neprimerno teže ugotoviti, ker pri tem še nimamo nobenega standardnega sistema označevanja, poleg tega pa je računalniški sistem v svoji "življenjski dobi" precej bolj nedoločljiv, saj ga redne in potrebne nadgraditve strojne in programske opreme kmalu naredijo precej neprepoznavnega: po zamenjavi večjega števila sestavnih delov že kar težko govorimo, da gre sploh še za isti računalnik.
V tem delu smo se problematike lotili analitično. Najprej smo dobro definirali, kaj resnično določa istovetnost računalniškega sistema. Spoznali smo osnovne *identifikatorje*. Prišli smo do zaključka, da posamezni identifikatorji nikakor ne morejo zadovoljivo določiti istovetnosti računalniškega sistema, pač pa potrebujemo sodelovanje med njimi. Uvedli smo koncept *sestavljenega identifikatorja*. Pri tem smo upoštevali enostavno premiso, da "več glav, več ve". V tem primeru več identifikatorjev, ki so sami za sebe lahko le slabo določljivi, sestavljeni skupaj dosežejo višjo stopnjo zaupanja in se bolj približajo idealnemu *univerzalno unikatnemu identifikatorju*.
Rezultat našega dela je *programska knjižnica*, ki je povsem nastavljiva in nadgradljiva. Naše raziskave so namreč jasno pokazale, da rešitev danega problema ne leži v nekem dokončnem in statičnem "popolnem" izdelku, ki bi znal odgovoriti na vsa zastavljena vprašanja, temveč v dinamični in prilagodljivi strukturi, ki jo lahko vsak uporabnik prikroji svojim potrebam glede na domeno delovanja.
Glavni poudarek našega dela je jedro knjižnice, ki preudarno implementira vso potrebno infrastrukturo za izdelavo posameznih identifikatorjev ter za zlaganje le-teh v hierarhično sestavljene celote. Poleg tega pa je knjižnica že nared za takojšnjo uporabo z vnaprej pripravljenimi identifikatorji strojne in programske opreme, ki lahko s precejšnjo zanesljivostjo ugotovijo istovetnost računalniškega sistema.

Ključne besede : *računalniški sistem, istovetnost*, *identifikator*, *sestavljeni identifikator*, univerzalno *unikatni*, *programska knjižnica*, *Java*.

1. ABSTRACT

Computer system identification

The concept of *computer system identity* in computer science bears just as much importance as does the identity of an individual in a human society. Nevertheless, the identity of a computer system is incomparably harder to determine, because there is no standard system of identification we could use and, moreover, a computer system during its life-time is quite indefinite, since all of its regular and necessary hardware and software upgrades soon make it almost unrecognizable: after a number of components have been replaced, it becomes increasingly more difficult to state that we are dealing with the same computer at all.
In this work we took an analytical approach. First we properly defined what is it that really determines the identity of a computer system. We presented basic *identifiers*. A conclusion has been reached that no individual identifier by itself can successfully determine the identity of a computer system, but that we need a cooperation among them. We introduced the concept of a *compound identifier*. With that we followed a simple premise: two heads are better than one. In this case, a number of identifiers, that are not very determining each by itself, together combine to a greater level of dependability and come closer to the ideal of *universally unique identifier*.
The result of our work is a *program library*, which is completely adaptable and upgradeable. Our research has, in fact, shown that a solution to this problem lies not in a final and static "perfect" product, but instead in a dynamic and adaptable structure that can by modified by the users with respect to the demands of their domain.
The essential point of our work is the core of the library, which thoughtfully implements the necessary infrastructure for creating the individual identifiers and for combining them into the hierarchical compound entities. At the same time, the library is fully operational and ready to be used with prepared hardware and software identifiers that are able to determine the identity of a computer system with great reliability.

Keywords : *computer system*, *identity*, *identifier*, *compound identifier*, *universally unique*, *program library*, *Java*.

2. UVOD

Ker je "etimologija pogosto opora natančni rabi izrazov" [1] in ker smo študentje naravoslovja pogosto odvisni od natančnosti definicij in njihove uporabe, se nam zdi več kot vredno najprej predstaviti rabo pojma istovetnost v zgodovinski perspektivi ter prek pogleda v preteklost določiti sedanjost.
Izraz i*stovetnost* je slovenska beseda za *identiteto,* pojem identiteta pa izhaja iz izrazoslovja srednjeveške latinščine. Identiteta je bila v tistem času rabljena za dokazovanje narave osebnosti Jezusa Kristusa, za kristološki spor, ali je Jezus Bogu enak po bistvu ali mu je po bistvu zgolj podoben. Danes vemo, da je koncil v Niceji (325 n.št.) določil, da sta Bog in Jezus po biti enaka in ne zgolj podobna: v srednjeveški latinščini *identicus*, istovetna, ena in ista.
V novolatinski verziji je identiteta *identitas*, kar je ujeto v vse romanske jezike. Kot istovetnost se deli na absolutno (popolno) identiteto (a=a), ki nas bo v nadaljevanju še posebej zanimala, ter na relativno (pogojno) identiteto, kjer je poudarjena zgolj bližina enakosti oziroma odtenek, niansa.

V svetu, ki je tako raznovrsten in kompleksen, kot je svet, v katerem živimo, lahko učinkovito in elegantno delujemo le tedaj, ko imamo možnost razlikovati in nedvoumno poimenovati objekte, ki nas obdajajo, pa naj bodo to ljudje in živali ali pa, kar zanima nas, računalniki. Pojavi se vprašanje, kako istovetnost ugotoviti. Pri ljudeh imamo kar nekaj neodvisnih sistemov določanja identitete, na katere smo že bolj ali manj navajeni: nekateri so zgolj osebni, drugi pa so povsem formalni, kot na primer EMŠO ali številka potnega lista. Računalniki pa dandanes še ne premorejo nobene take preproste, razumljive in določujoče identifikacije, sploh pa ne na globalnem nivoju.

Računalnik nima "potnega lista", po možnosti celo s prepoznavno fotografijo, s katerim bi se lahko nedvoumno izkazal, kadar že pride do te potrebe. In kdaj lahko do take potrebe pride? Preprost primer je, če skušamo zaščititi programsko opremo pred nepooblaščenim, piratskim kopiranjem. Ko se neka aplikacija zažene, moramo znati ugotoviti, ali se je zagnala na istem računalniku kot prvič, ko jo je uporabnik namestil s pomočjo licenčnega ključa. Vprašanje je enostavno: ali gre za isti računalnik kot prej? Kako lahko ugotovimo istovetnost računalniškega sistema?

Najprej definirajmo, kaj "računalniški sistem" sploh je [2]. Najbolj grobo lahko rečemo, da se računalniški sistem sestoji iz strojne (hardware) in programske (software) opreme. Strojna oprema nadalje vključuje centralno procesno enoto (CPU) ali procesor, matično ploščo (motherboard), glavni pomnilnik (RAM), zunanji pomnilnik, kamor sodi trdi disk (hard disk), ter vhodno/izhodne naprave ali enote (I/O devices): to so mrežna kartica (network adapter, NIC), grafična (video ali graphics accelerator card) in zvočna (audio card, soundblaster) kartica, monitor (CRT, LCD display), tiskalnik, tipkovnica, miška... Programska oprema pa je operacijski sistem in vse aplikacije, ki tečejo na le-tem, pa naj bodo to sistemska orodja ali programski jeziki in razvojna okolja, grafična orodja, urejevalniki, predvajalniki in tako naprej.

Nekateri od teh osnovnih gradnikov računalniškega sistema že sami za sebe premorejo tako

ali drugačno identifikacijsko številko, vendar še zdaleč ne vsi. Prej lahko govorimo o izjemah, kot o pravilu. Poleg tega je velika razlika, ali je ta identifikacija unikatna in globalna ali ne, kar pomeni naslednje: ali se lahko zanesemo, da se identifikacijska številka, na katero naletimo pri nekem gradniku, na primer grafični kartici, ne bo ponovila pri kaki drugi grafični kartici?

Izkaže se, da povsem globalnih in unikatnih identifikacijskih številk pravzaprav na tem nivoju ni. Ravno nasprotno: pri pridobivanju ali ustvarjanju globalnih ali univerzalno-unikatnih identifikatorjev (UUID) se uporabljajo ali so se uporabljali kot sestavni deli identifikatorji posameznih gradnikov, na primer MAC naslov mrežne kartice [3]. MAC naslov mrežne kartice je v praksi eden najbolj nedvoumno določujočih sistemskih identifikatorjev, kar jih lahko srečamo.

Druga oteževalna okoliščina, s katero se srečamo pri naši problematiki, je odvisnost večine sistemskih parametrov od operacijskega sistema, ki teče na danem računalniškem sistemu. Drugače povedano, nekateri identifikatorji so preprosto dosegljivi ali "berljivi" na enem operacijskem sistem ter praktično neobstoječi ali vsaj neuporabni na drugem. S tem povezana težavnost je tudi ta, da implementacija nekega identifikatorja posledično kaj lahko zahteva znanje in uporabo tako imenovanih nižjenivojskih (low-level) programskih jezikov, kot sta na primer C ali celo zbirni jezik, čemur bi se radi izognili. V vsakem primeru pa nas to oddalji od tega, da bi bila naša rešitev popolnoma prenosljiva brez posebnega truda.

V nadaljevanju bomo pokazali, da najbolj obetavna smer proti našemu cilju, določanju istovetnosti računalniških sistemov, leži v preudarnem izdelovanju hierarhično zgrajenega, sestavljenega identifikatorja. Ker se ne moremo zanesti na en sam identifikator nekega gradnika, si moramo pomagati s sodelovanjem večjega števila identifikatorjev množice gradnikov, ki tvorijo računalniški sistem. Pri tem pa je treba zelo natanko upoštevati vse bistvene značilnosti posameznih komponent tega sestavljenega identifikatorja in jih pri zlaganju skupaj pravilno utežiti in uravnovesiti med seboj. Ime računalnika, kot ena komponenta sestavljenega identifikatorja, na primer, ne sme imeti enake teže pri določanju istovetnosti celotnega sistema, kot jo ima MAC naslov, saj se da ime računalnika precej enostavno spremeniti in tudi ni tako nenavadno, če po naključju naletimo na dva računalnika z istim imenom. Eden od glavnih rezultatov pričujočega dela je definicija in vpeljava sistema za preudarno gradnjo sestavljenih identifikatorjev.

Ta sistem je zasnovan in implementiran v obliki programske knjižnice (imenovane "syd") v programskem jeziku Java [4,5]. Temelji na osnovni definiciji identifikatorja, izražene v obliki Java vmesnika (interface), ki natanko določa obvezne atribute in operacije, ki jih mora vsebovati vsak identifikator, tako primitiven (atomičen) kot sestavljen. Najbolj pomembne operacije vsakega identifikatorja so: *inspect* (preglej sistem in pridobi/preberi sistemski parameter), *compareTo* (primerjaj dva identifikatorja istega tipa) in *serialize* (pretvori v obliko, primerno za shranjevanje ali pošiljanje).

Knjižnica syd je pripravljena tako, da jo lahko uporabnik (razvijalec/programer) popolnoma prilagodi svoji rabi ali po mili volji razširi z novimi identifikatorji (ki se imenujejo kar "sydi"). Precej sydov knjižnica že vnaprej implementira. Vendar pa se zavedamo, da je računalništvo eno od najhitreje razvijajočih se področij, zato predvidevamo, da se bo kmalu lahko pokazala potreba po novih identifikatorjih, torej novih implementacijah sydov, ki jih bo

lahko uporabnik sam izdelal. V veliko pomoč pri tem mu bodo nekateri abstraktni razredi, ki jih knjižnica nudi. Implementacija novega syda s pomočjo teh abstraktnih razredov je res enostavna - pravzaprav je potrebno le nastaviti določene uteži in syd pravilno umestiti v hierarhijo sestavljenih sydov.

Jedro naše knjižnice pa je preudaren mehanizem zlaganja posameznih sydov v sestavljen syd. V ta namen je pripravljen spet drug abstrakten razred, ki že implementira vse potrebne metode za sestavljanje sydov, za izračunavanje skupne uteži ali zanesljivosti sestavljenega syda in za hierarhično, kaskadno izvajanje osnovnih operacij pregledovanja, primerjanja in shranjevanja.

Primer "podpisa" istovetnosti računalniškega sistema, kot je bil določen z uporabo naše knjižnice:

```
HW:
  BIOS Data: null
  MAC Address:
    00:1D:E0:6F:B7:25
    00:1E:37:88:EF:8D
  Hard Disk ID: null
  CPU Data: null
SW:
  OS:
    OS Name: Linux
    OS Version: 2.6.24-19-generic
    OS Architecture: amd64
  Java:
    Java Specification Version: 1.6
    Java VM Version: 10.0-b22
    Java VM Specification Version: 1.0
  Computer Name: null
```

V nadaljevanju tega dela si bomo najprej pod drobnogledom ogledali knjižnico syd z vidika njene zasnove in uporabnosti kot abstraktnega sistema za gradnjo sestavljenih identifikatorjev, nato pa še konkretne implementacije posameznih sydov kot identifikatorjev gradnikov računalniških sistemov.

3. KNJIŽNICA SYD

Knjižnica syd je množica razredov, napisanih v programskem jeziku Java, ki omogočajo:

- določiti in ugotoviti istovetnost danega računalniškega sistema: *podpis*;
- shraniti ta ugotovljeni podpis ali identifikacijo v poseben zapis, ki omogoča enostavno shranjevanje v podatkovni bazi ali kakem drugem sistemu pomnjenja in enostavno prenašanje prek običajnih komunikacijskih linij;
- primerjati podpise med seboj in ugotavljati stopnjo podobnosti med njimi.

Knjižnica že vključuje vse potrebno za takojšnjo uporabo, se pravi vse potrebne implementacije posameznih identifikatorjev (*sydov*) za zelo zanesljivo določanje istovetnosti računalniških sistemov. Po drugi strani pa ponuja tudi možnost neomejene in izredno enostavne širitve z dodajanjem novih implementacij sydov za povsem nove računalniške gradnike.

V tem poglavju si bomo najprej ogledali sestavo knjižnice in našteli ter pojasnili vse njene sestavne dele, Javine vmesnike in razrede. Nato se bomo osredotočili na glavni vmesnik, ki definira bistvo syda, in ga podrobno opisali ter skupaj z njim tudi definirali, kaj tukaj pojmujemo pod izrazom identifikator. V naslednjem razdelku pa se bomo posvetili osnovnemu abstraktnemu razredu knjižnice in ob tem povedali, kako lahko identifikatorje uporabljamo za določanje istovetnosti, pa tudi, kako lahko ustvarimo svoj lastni syd. V zadnjem delu tega poglavja pa bomo skrbno pokazali, kako delujejo sestavljeni sydi in kako lahko implementiramo nove sestavljene syde ali popravljamo že obstoječe.

3.1. Sestava knjižnice

Knjižnica syd sestoji iz naslednjih sklopov:

- vmesnik *Sydable*, ki definira osnovne značilnosti syda;
- abstraktni razred *AbstractSyd*, ki implementira glavne operacije syda;
- abstraktni razred *AbstractContainerSyd*, ki implementira vse potrebno za izdelavo sestavljenega syda;
- implementacije posameznih atomičnih sydov in sestavljenih sydov, razvrščene v hierarhično strukturo paketov in razredov;
- testni program *TestSyd*.

3.2. Definicija syda

Syd, kot formalni izraz identifikatorja oziroma nosilca identitete nekega določujočega parametra, je definiran z vmesnikom *Sydable*. Ta predpisuje, da mora vsak syd vsebovati naslednje metode:

- `String getName()`
- `Set<OS> getOsSupport()`
- `boolean inspect()`
- `Boolean isReady()`
- `String getValueAsString()`
- `double getReliability()`
- `double compareTo(Sydable syd)`
- `Similarity discretizeSimilarity(double continuousSimilarity)`
- `byte[] serialize() throws IOException`

3.2.1. getName

Ime syda je zgolj informativne narave in služi za izpise. Za samo delovanje ta podatek ni resnično potreben.
Primer imena syda je `"MAC Address"`.

Metoda *getName* vrne ime syda:

```
String getName();
```

3.2.2. getOsSupport

Ta parameter pove, katere operacijske sisteme implementacija syda podpira. S tem podatkom si lahko pomagamo, ko gradimo sestavljene syde tako, da so posebej prirejeni za uporabo na ciljnem operacijskem sistemu, če to želimo.

Metoda *getOsSupport* vrne podporo syda kot množico elementov naštevnega tipa *OS*:

```
Set<OS> getOsSupport();
```

Pri tem je tip *OS* definiran takole:

```
enum OS {
  WINDOWS, LINUX, MAC
}
```

3.2.3. inspect

Ta metoda je temeljna operacija syda. Pregleda sistem in pridobi sydovo konkretno vrednost, vrednost identifikatorja. Na primer, za *MacAddressSyd*, ki implementira syd mrežne kartice, operacija *inspect* pridobi vrednost MAC naslova mrežne kartice: `"01:23:45:67:89:ab"`.

Vsaka implementacija syda **mora obvezno** implementirati to metodo, in sicer tako, da v njej kliče tudi metodo *AbstractSyd.setReady*. (To bo v nadaljevanju še bolj nadrobno pojasnjeno.)

Pri uporabi mora biti operacija *inspect* poklicana vsaj enkrat, drugače bosta operaciji *compareTo* in *serialize* javili napako `IllegalArgumentException`.

Inspect običajno kliče potem naprej neko metodo, ki je implementirana v nižjem programskem jeziku in pripravljena v obliki *native* knjižnice.

Metoda *inspect*:

```
boolean inspect();
```

Metoda vrne vrednost *true*, kadar je pregled uspel, sicer pa *false*.

3.2.4. isReady

Ta operacija pove, ali je primerek syda pripravljen za uporabo ali ne. Primerek syda je pripravljen za uporabo, kadar ga lahko primerjamo z drugimi primerki istega (tipa) syda.

Semantika pripravljenosti primerka syda je bolj nadrobno povedano taka:
če je bila operacija *inspect* že poklicana nad tem primerkom syda, ima *pripravljenost* vrednost *true* ali *false*, če pa pregled sploh še ni bil opravljen, ima *pripravljenost* vrednost *null*.

Operaciji *compareTo* in *serialize* naj bi bili tako implementirani, da javita napako, kadar sta poklicani nad primerkom syda, ki ima parameter *pripravljenost* vrednosti *null*. Če ima primerek syda vrednost parametra *pripravljenost false*, se pri operaciji *compareTo* zgolj ignorira, brez javljene napake.

Metoda *isReady*:

```
Boolean isReady();
```

Metoda lahko vrne vrednosti *true*, *false* ali *null*.

3.2.5. getValueAsString

Metoda *getValueAsString* vrne vrednost syda, kot je bila pridobljena ob klicu operacije *inspect*, v tekstovni obliki, se pravi, kot String (niz znakov jezika Java).

Primer vrednosti MAC naslova: `"01:23:45:67:89:ab"`.

Metoda *getValueAsString*:

```
String getValueAsString();
```

3.2.6. getReliability

Ta parameter podaja faktor zanesljivosti tega primerka syda kot realno število na zaprtem intervalu [0,1]. Vrednost 0 pomeni, da je syd popolnoma nezanesljiv, vrednost 1 pa, da je popolnoma določljiv. V praksi seveda ne pričakujemo, da bi ti dve vrednosti kdaj nastopili.

Pozor: kadar syd ni pripravljen, se pravi, da metoda inspect sploh še ni bila poklicana ali pa le ni bila uspešno izvedena, mora vrniti metoda *getReliability* vrednost 0. Njena implementacija v razredu *AbstractSyd* to upošteva, paziti pa je potrebno, če uporabnik prepiše to metodo s svojo.

Za primer si oglejmo *ComputerNameSyd*, ki naj ne bi imel zanesljivosti večje od 0.05, saj je res preprosto spremeniti ime računalniku, pa tudi dokaj hitro se lahko zgodi, da imata dva računalnika isto ime. Ta syd torej ni ravno določujoč. Po drugi strani pa je *MacAddressSyd* zelo določujoč, saj je MAC naslov skoraj nemogoče spremeniti, poleg tega naj bi bili MAC naslovi univerzalno unikatni, zato ima lahko tak syd zanesljivost kar 0.95.

Faktor zanesljivosti se uporablja pri ugotavljanju podobnosti dveh primerkov sestavljenega syda ob klicu operacije *compareTo* nad vsemi pod-sydi. Pri tem se posamezne podobnosti primerkov pod-sydov upoštevajo uteženo glede na posamezne faktorje zanesljivosti le-teh. To je bolj nadrobno razloženo kasneje.

Metoda *getReliability*:

```
double getReliability();
```

Vrne faktor zanesljivosti.

3.2.7. compareTo

Ta operacija primerja primerek syda z drugim primerkom syda istega tipa. Kot rezultat vrne *podobnost*, realno število na zaprtem intervalu [0,1], pri čemer vrednost 0 pomeni, da sta syda

popolnoma različna, vrednost 1 pa, da sta syda povsem identična.

Vsaka implementacija te operacije mora obvezno klicati metodo *AbstractSyd. compareTo*, ki preveri vse osnovne zahteve, ki jih morata izpolnjevati oba primerka sydov, ki ju primerjamo. Te osnovne zahteve so:

- parameter ne sme biti *null*, sicer se javi napaka `NullPointerException`;
- primerka morata biti istega tipa, sicer se javi `ClassCastException`;
- primerka morata biti pripravljena, sicer se javi `IllegalArgumentException`.

Ta operacija je v tesni zvezi s spodaj opisano operacijo *podobnost*.

Metoda *compareTo*:

```
double compareTo(Sydable syd);
```

Parameter `syd` predstavlja primerek, s katerim želimo primerjati ta syd.
Vrne podobnost.

3.2.8. discretizeSimilarity

Ta operacija pretvori *podobnost*, podano v obliki realnega števila, kar je precej neberljivo, v diskretno vrednost, predstavljeno z naštevnim tipom `Similarity`:

```
enum Similarity {
  IDENTICAL, SAME, SIMILAR, DIFFERENT
}
```

Pri tem se upoštevajo diskretizacijski intervali, ki so lahko pri vsaki implementaciji syda posebej podani, lahko pa implementacija syda uporablja naslednje privzete vrednosti, kot so pripravljene v abstraktnem razredu *AbstractSyd*:

- `IDENTICAL: x = 1`
- `SAME: 0.9 <= x < 1`
- `SIMILAR: 0.5 <= x < 0.9`
- `DIFFERENT: x < 0.5`

Kadar podani parameter ne leži na zaprtem intervalu [0,1], se javi napaka `IllegalArgumentException`.

Primer: dva primerka operacijskega sistema sta lahko *enaka* (*SAME*), za razliko od *identična* (*IDENTICAL*), kadar se razlikujeta zgolj v številki revizije ali "builda", ali pa sta *podobna* (*SIMILAR*), za razliko od *različna* (*DIFFERENT*), kadar se razlikujeta v glavni različici, gre pa še vedno za isto vrsto operacijskega sistema. MAC naslova pa sta, po drugi strani, lahko kvečjemu različna ali pa sta identična - vmesne vrednosti nimajo smisla.

V vednost: abstraktni razred *AbstractSyd* vsebuje metodo *discreteCompareTo*, ki nam lahko

olajša delo, saj že sama kliče metodi *compareTo* in *discretizeSimilarity*.

Metoda *discretizeSimilarity*:

```
Similarity discretizeSimilarity(double continuousSimilarity);
```

Parameter `continuousSimilarity` predstavlja podobnost v obliki realnega števila.
Vrne podobnost kot vrednost naštevnega tipa `Similarity`.

3.2.9. serialize

Ta operacija pretvori syd v obliko, ki je pripravna za shranjevanje podpisa (se pravi vrednosti syda) v podatkovno bazo ali kak drug sistem pomnjenja. Ta oblika je preprosto zaporedje bajtov (angl. byte stream), definirano v Javi kot *byte[]*. Tako pripravljen (shranjen) podatek je tudi moč preprosto posredovati prek vseh običajnih komunikacijskih kanalov (elektronska pošta, TCP socket...).

Če operacija *inspect* še ni bila klicana, se javi napaka `IllegalArgumentException`.

Kadar pride do težav pri procesu serializacije, to je pretvorbe syda v zaporedje bajtov, se javi napaka `IOException`.

Metoda *serialize*:

```
byte[] serialize() throws IOException;
```

Vrne syd, preoblikovan v zaporedje bajtov.

3.3. Implementacija syda

Vsak identifikator, s pomočjo katerega bi želeli ugotavljati istovetnost računalniškega sistema, mora torej znotraj našega sistema oziroma knjižnice syd verno slediti definiciji, kot jo podaja vmesnik *Sydable*. Implementacija syda mora torej obvezno implementirati vse metode vmesnika *Sydable*. Ker pa imajo očitno vsi sydi precej skupnega, knjižnica ponuja abstraktni razred *AbstractSyd*, ki že vnaprej implementira večino metod vmesnika *Sydable* in uporabniku pripravi vse potrebno za kar najbolj enostavno delo.

Najprej si bomo pobližje ogledali *AbstractSyd*, nato pa še primer implementacije syda, ki temelji na uporabi razreda *AbstractSyd*.

3.3.1. Abstraktni razred *AbstractSyd*

AbstractSyd vsebuje naslednje spremenljivke:

- name (ime)

  ```
  private String name;
  ```

  Ta spremenljivka hrani ime syda, npr. `"Mac Address"`.

  Vrednost te spremenljivke vrne metoda *Sydable.getName*.

- ready (pripravljen)

  ```
  private Boolean ready=null;
  ```

  Ta spremenljivka pove, ali je syd pripravljen na uporabo ali ne, torej, ali ga lahko primerjamo z ostalimi primerki sydov.

  Če je bila operacija *inspect* že poklicana nad tem primerkom syda, ima ta spremenljivka vrednost *true* ali *false*, če pa pregled sploh še ni bil opravljen, ima vrednost *null*.

  Vrednost te spremenljivke vrne metoda *Sydable.isReady*.

- reliability (zanesljivost)

  ```
  private double reliability;
  ```

  Ta spremenljivka hrani faktor sydove zanesljivosti.
  Vrednost te spremenljivke vrne metoda *Sydable.getReliability*.

- diskretizacijski pragovi (discretization thresholds)

  ```
  private double[] discretizationThresholds={1, 0.9f, 0.5f};
  ```

  Ta spremenljivka definira pragove intervalov, ki se uporabljajo pri diskretizaciji vrednosti podobnosti pri primerjavi dveh sydov, se pravi ob klicu metode *Sydable.discretizeSimilarity*.
  Privzete vrednosti pragov so 1, 0.9 in 0.5.
  Te vrednosti pa je mogoče spremeniti prek konstruktorja AbstractSyd.

AbstractSyd ponuja dva konstruktorja:

```
public AbstractSyd(String name, double reliability);

public AbstractSyd(String name, double reliability, double
  thresholdIdentical, double thresholdSame, double thresholdSimilar);
```

S prvim lahko nastavimo vrednosti spremenljivk *ime* in *zanesljivost*, z drugim pa še vrednost spremenljivke *diskretizacijski pragovi*.

Privzetega konstruktorja ni, torej je vedno potrebno nastaviti vsaj ime in zanesljivost syda.

AbstractSyd vsebuje naslednje javne (public) metode:

- `public String getName()`
- `public Boolean isReady()`
- `public double getThresholdIdentical()`
- `public double getThresholdSame()`
- `public double getThresholdSimilar()`

  Te metode preprosto vrnejo vrednosti svojih pripadajočih spremenljivk.

- `public Set<OS> getOsSupport()`

  Ta metoda vrne privzeto vrednost za podporo operacijskim sistemom, kar je prazna množica: `new TreeSet<OS>()`.
  Vsaka implementacija syda, ki želi uporabljati to vnaprej pripravljeno infrastrukturo za vodenje evidence podprtih operacijskih sistemov, naj bi to metodo *prepisala* (*override*).

- `public double getReliability()`

  Ta metoda vrne vrednost spremenljivke reliability, kadar je syd pripravljen.
  Če pa syd ni pripravljen, se pravi, da metoda inspect sploh še ni bila poklicana ali pa le ni bila uspešno izvedena, vrne vrednost 0.
  Paziti je potrebno na to, kadar uporabnik prepiše to metodo s svojo.

- `public String toString()`

  Ta metoda prepiše metodo *Object.toString*.
  Vrne tekstovno predstavitev syda, pri čemer upošteva njegovo ime, vrednost, zanesljivost, pripravljenost in podporo. Vrednost syda se pri tem posredno izpiše v tekstovno obliko s klicem metode *Sydable.getValueAsString*, ki jo mora obvezno implementirati vsak syd. Implementaciji syda torej ni nujno potrebno prepisati metode *toString*, marveč zgolj metodo *getValueAsString*.

- `public Similarity discretizeSimilarity(double continuousSimilarity)`

  Ta metoda vrne *podobnost*, se pravi rezultat operacije *compareTo*, kot vrednost

naštevnega tipa z upoštevanjem zgoraj opisanih pragov diskretizacije.

Kadar želi neka implementacija syda preprosto uporabiti drugačne intervale pri določanju diskretne vrednosti za podobnost, mora v konstruktorju *AbstractSyd* podati druge pragove diskretizacije.
Če pa naj bo diskretizacija pri kakem sydu izvedena povsem svojstveno, naj implementacija prepiše to metodo in uporabi poljuben algoritem določanja diskretnih vrednosti.

- `public double compareTo(Sydable syd)`

  Ta metoda, kot je implementirana v tem razredu, služi kot dodatna pomoč pri realizaciji prave, dokončne operacije *compareTo*, ki mora običajno biti povsem lastna implementaciji vsakega syda, saj je semantika vrednosti syda poljubna.

  Metoda *AbstractSyd.compareTo* naj bi se obvezno klicala že na začetku klica *compareTo* vsake implementacije syda.

  Kot je nadrobno opisano v razdelku 3.2.7., ta metoda preveri vse osnovne zahteve, ki jih morata izpolnjevati oba primerka sydov, ki ju primerjamo.
  Poleg tega pa tudi preveri pripravljenost obeh sydov in vrne 1, če sta oba syda pripravljena, ter 0, če vsaj eden od njiju ni pripravljen.

- `public Similarity discreteCompareTo(Sydable syd)`

  Ta metoda preprosto pokliče metodo *compareTo* in zatem še metodo *discretizeSimilarity*.

- `public byte[] serialize() throws IOException`

  Ta metoda implementira metodo *Sydable.serialize*, kot je to nadrobno opisano v razdelku 3.2.9.

  Pozor: uporabnik mora poskrbeti, da bo njegova implementacija syda, ki nadgrajuje *AbstractSyd*, vključevala zgolj take elemente, ki implementirajo vmesnik *java.io.Serializable*. Sam vmesnik *Sydable* sicer že nadgrajuje vmesnik *java.io.Serializable*, vendar je to potrebno zagotoviti tudi za vse ostale elemente, ki jih implementacija syda uporablja, kot na primer vrednost, shranjeno v nekem ločenem razredu.

- ```
  public static <T extends Sydable> T deserialize(byte[] byteStream)
    throws IOException, ClassNotFoundException
  ```

  Ta metoda je obratna metodi *serialize* in ponovno zgradi primerek syda iz podanega zaporedja bajtov. Metoda je seveda statična (*static*), saj pred klicem objekt še ne obstaja.

  Primer uporabe:
  ```

```
Syd oldSyd=AbstractSyd.deserialize(syd.serialize());
```

AbstractSyd definira tudi zaščiteni (protected) metodi *setReady* in *setReliability*:

- `protected void setReady(boolean ready)`

  To metodo mora obvezno poklicati vsaka implementacija operacije *inspect* in s tem nastaviti status pripravljenosti na *true* ali *false*, namesto na nedefinirano stanje *null*, ki ponazarja, da metoda *inspect* še ni bila poklicana.

- `protected void setReliability(double reliability)`

  To metodo kliče izključno metoda *AbstractContainerSyd.inspect*. V vseh ostalih primerih se *zanesljivost* že nastavi ob klicu konstruktorja, četudi postane dostopna prek metode *getReliability* šele po uspešno opravljeni operaciji *inspect*.
  Ta metoda javi napako `IllegalArgumentException`, kadar je dana vrednost parametra izven meja [0,1].

AbstractSyd ponuja tudi javno spremenljivko *fullOsSupport*:

- ```
  public static final Set<OS> fullOsSupport=new TreeSet<OS>(Arrays.asList(
    OS.WINDOWS, OS.LINUX, OS.MAC))
  ```

3.3.2. Primer implementacije syda

Implementacija syda, ki uporablja *AbstractSyd*, je izjemno preprosta.
Tak syd mora običajno definirati neko privatno (*private*) spremenljivko, kamor bo shranjena njegova vrednost ob klicu operacije *inspect*, poleg tega pa mora obvezno implementirati konstruktor, kjer poda svoje ime in svojo zanesljivost, ter metode *getValueAsString*, *inspect* in *compareTo*.

Poglejmo si to pobližje na primeru syda *ComputerNameSyd*:

```
public class ComputerNameSyd extends AbstractSyd {
    ...
}
```

Ta najprej definira privatno spremenljivko *computerName*, kamor se bo shranila vrednost, torej ime računalnika, ob klicu metode *inspect*.

```
private String computerName;
```
```

Definira tudi privatni metodi za branje in nastavljanje (*getter* in *setter*) te spremenljivke, čeprav to ni nujno, marveč zgolj stvar programerskega stila:

```
  private String getComputerName() {
    return computerName;
  }

 private void setComputerName(String computerName) {
  this.computerName=computerName;
  }
```

Potreben je konstruktor, ki kliče super konstruktor, torej *AbstractSyd*, kjer se nastavita ime in zanesljivost syda:

```
  public ComputerNameSyd() {
    super("\n  Computer Name", 0.05);
  }
```

Ime v tem primeru vsebuje tudi nekoliko "oblikovanja", da je izpis sestavljenega syda, ki vključuje ta syd, bolj berljiv.
Zanesljivost 0.05 je precej nizka, kar je tudi pravilno za tovrstni syd.

Metoda *getOsSupport* definira polno podporo vseh operacijskih sistemov:

```
  @Override
  public Set<OS> getOsSupport() {
    return AbstractSyd.fullOsSupport;
  }
```

Sledi metoda *getValueAsString*, ki je trivialna, saj preprosto le pokliče getter spremenljivke *computerName*:

```
  public String getValueAsString() {
    return getComputerName();
  }
```

Metoda inspect je nekoliko resnejša:

```
  public boolean inspect() {
    setReady(false);
  try {
      setComputerName(InetAddress.getLocalHost().getHostName());
  } catch (UnknownHostException e) {
   return false;
    }
  setReady(true);
    return true;
  }
```

V jedru metode se poskusi prebrati ime računalnika prek imena gostitelja (*host*) njegovega lokalnega mrežnega naslova. Če to ne uspe, se ujame napaka in vrne se *false*.

Pri tem je treba biti pozoren na dejstvo, da se v vsakem primeru pokliče metoda *setReady*, saj le-ta inicializira parameter *ready*, da ni več *null*. Brez tega klica metode *setReady* bi kasnejši klic metod *compareTo* in *serialize* javil napako, kar pomeni, da celotna knjižnica ne bi bila uporabna. Ta zahteva je premišljeno umeščena, saj je vitalnega pomena za uspešno delovanje knjižnice, da se zagotovi pravočasen klic metode *inspect* na vseh hierarhičnih nivojih, se pravi, pred klicem metode *compareTo*. Torej: če uporabnik ne bo dodal klica metode *setReady* v implementacijo metode *inspect*, se bo že takoj ob prvem klicu metode *compareTo* javila napaka in opozorila na to.

Na koncu še metoda *compareTo*:

```
@Override
public double compareTo(Sydable syd) {
if (super.compareTo(syd)==0) return 0; // check if ready

  if (getComputerName().equalsIgnoreCase(
    ((ComputerNameSyd)syd).getComputerName())) return 1;
return 0;
}
```

Prva vrstica te metode naj bi bila vedno enaka: klic *super.compareTo*, ki preveri, če so vse osnovne zahteve za primerjavo dveh sydov izpolnjene, kot je to opisano v razdelku 3.2.7.

Sicer pa naj metoda *compareTo* vrne realno vrednost z zaprtega intervala [0,1]. V danem primeru pa imamo samo dve možnosti: ali sta imeni identični in vrnemo 1, ali pa sta različni in vrnemo 0.

Celotna izvorna koda implementacije syda *ComputerNameSyd* je podana v prilogi.

3.4. Sestavljeni syd

V prejšnjem podpoglavju smo si od blizu ogledali, kako preprosto lahko implementiramo nov osnovni, atomični syd s pomočjo razreda *AbstractSyd*. Prava moč naše knjižnice pa se pokaže šele takrat, ko začnemo kombinirati večje število atomičnih sydov v hierarhijo sestavljenih sydov! Pri tem pa se pojavi kar nekaj vprašanj:

- Kako naj se opišejo in implementirajo sestavljeni sydi?
- Kdo in kdaj naj pokliče metode *inspect* posameznih pod-sydov?
- Kakšna je skupna, kombinirana podpora operacijskim sistemom?
- Kako naj izgleda izpis sestavljenega syda?
- Kakšna je zanesljivost sestavljenega syda?
- Kako naj se izvrši primerjava dveh sestavljenih sydov?

Očitno sestavljanje sydov ni povsem trivialno.

Vendar pa je ena najmočnejših točk te knjižnice razred *AbstractContainerSyd*. Ta ne samo odgovori na vsa zgornja vprašanja, pač pa celo ponudi popolno implementacijo, ki je potrebna za izpeljavo sestavljenih sydov. Tako je implementacija sestavljenega syda s pomočjo razreda *AbstractContainerSyd* še bolj enostavna kot implementacija atomičnega syda. Pa si poglejmo, zakaj.

3.4.1. AbstractContainerSyd

AbstractContainerSyd tudi sam nadgrajuje *AbstractSyd* in poleg tega izpolnjuje vse zahteve vmesnika *Sydable*, torej je prav vsak sestavljeni syd, ki nadgradi *AbstractContainerSyd*, tudi sam syd.

AbstractContainerSyd vsebuje seznam vseh svojih elementov, pod-sydov:

```
List<Sydable> subSyds=new ArrayList<Sydable>();
```

Konstruktor je zelo enostaven, zahteva le ime sestavljenega syda ter poljubno število (lahko tudi nič) pod-sydov, ki postanejo njegovi elementi. Uporaba odprtega števila parametrov (*varargs*), kar omogoča Java od verzije 1.5 naprej, res prinese eleganco:

```
public AbstractContainerSyd(String name, Sydable... subSyds) {
  super(name, 0);
  this.subSyds.addAll(Arrays.asList(subSyds));
}
```

Kot vidimo, se najprej pokliče *AbstractSyd* konstruktor, nato pa se pod-sydi zložijo v seznam. Pri klicu *AbstractSyd* konstruktorja vidimo, da je podana skupna zanesljivost nastavljena na 0. Šele klic metode *inspect* bo pravilno izračunal skupno zanesljivost tega sestavljenega syda.

Implementacija metode *inspect* poskrbi, da se rekurzivno pokličejo vse metode *inspect* vseh pod-sydov, nato se izračuna skupna zanesljivost z metodo *calculateReliability*, nazadnje pa se pokliče tudi "obvezna" metoda *setReady*:

```
public boolean inspect() {
  boolean ready=false;
for (Sydable subSyd : getSubSyds()) {
 ready|=subSyd.inspect();
  }
  setReliability(calculateReliability(getSubSyds()));
  setReady(ready);
return ready;
}
```

Pripravljenost celotnega sestavljenega syda (spremenljivka *ready*) se določi tako, da je celotni syd pripravljen za uporabo, če rekurzivno vsebuje vsaj en pod-syd, ki je pripravljen za

uporabo. Tak pristop je najbolj realističen, saj ne moremo z gotovostjo vedeti in pričakovati, da bodo vsi pod-sydi zmogli uspešno opraviti pregled (se pravi uspešen klic metode *inspect*), glede na to, da je to v večini primerov odvisno od podpore operacijskega sistema, prave verzije programske opreme in morda celo ustrezne strojne opreme. Pristop je torej tak, da se tiste pod-syde, ki ne uspejo polnopravno sodelovati pri ugotavljanju istovetnosti, kar ignorira.

Metoda *getOsSupport* vrne presek vseh posameznih podpor pod-sydov:

```
@Override
public Set<OS> getOsSupport() {
Set<OS> result=new TreeSet<OS>(AbstractSyd.fullOsSupport);
for (Sydable subSyd : getSubSyds())
 result.retainAll(subSyd.getOsSupport()); // set intersection
return result;
}
```

Metodi *getValueAsString* in *toString* preprosto naštejeta vse pod-syde:

```
public String getValueAsString() {
  String result="";
for (Sydable subSyd : getSubSyds())
 result+=subSyd.getName()+": "+subSyd.getValueAsString();
return result;
}

@Override
public String toString() {
 String result=super.toString();
 for (Sydable subSyd : getSubSyds())
  result+=subSyd.toString();
 return result;
 }
```

Metoda *calculateReliability* izračuna skupno zanesljivost *R* sestavljenega syda po naslednji formuli:

$$R = 1 - \prod_{i=1}^{n} (1 - r_i)$$

kjer je *n* število pod-sydov sestavljenega syda, r_i pa posamezna zanesljivost i-tega pod-syda. Pri tem se upošteva, da je vrednost r_i enaka 0 za tiste pod-syde, ki niso pripravljeni, kar posledično pomeni, da taki pod-sydi nimajo vpliva na skupno zanesljivost. Drugače povedano: identifikatorji, ki jih ni moč dognati, prebrati iz sistema, se pri računanju skupne zanesljivosti popolnoma ignorirajo.

Matematično je mogoče dokazati, da je zgornja relacija komutativna in asociativna (iz česar sledi, da različne razporeditve pod-sydov znotraj hierarhije sestavljenega syda ne vplivajo na izid) in omejena na zaprtem intervalu [0,1].

Metoda *calculateReliability*:

```
static protected double calculateReliability(List<Sydable> subSyds) {
   double r=1;
 for (Sydable subSyd : subSyds) {
  r*=1-subSyd.getReliability();
   }
   return 1-r;
 }
```

Metoda *compareTo* vrne skupno podobnost *S* dveh sestavljenih sydov, izračunano s pomočjo uteženih primerjanj pod-sydov po formuli:

$$S = \frac{1 - \prod_{i=1}^{n} (1 - s_i r_i)}{R}$$

kjer je *n* število pod-sydov sestavljenega syda, s_i je posamezna podobnost (kot rezultat rekurzivno klicane metode *compareTo*) obeh i-tih pod-sydov, r_i posamezna zanesljivost i-tega pod-syda, *R* pa je skupna zanesljivost sestavljenega syda.
Pri tem se upošteva, da je vrednost r_i enaka 0 za tiste pod-syde, ki niso pripravljeni, kar posledično pomeni, da taki pod-sydi nimajo vpliva na skupno podobnost. Drugače povedano: identifikatorji, ki jih ni moč dognati, prebrati iz sistema, se pri primerjanju dveh sestavljenih sydov popolnoma ignorirajo.
Ob tem pa *R* nikoli ne sme imeti vrednosti 0, sicer se javi napaka `IllegalArgumentException`.

Matematično je mogoče dokazati, da je zgornja relacija komutativna in asociativna (iz česar sledi, da različne razporeditve pod-sydov znotraj hierarhije sestavljenega syda ne vplivajo na izid) in omejena na zaprtem intervalu [0,1].

Dobro je tudi vedeti, da se pri izračunu uporabljajo faktorji zanesljivosti *tega* (*this*) syda, ne pa parametrovi, kar bi lahko imelo za posledico razliko pri primerjanju trenutnega syda s kakim shranjenim arhivskim sydom, če je v vmesnem času prišlo do kake spremembe pri definiciji faktorjev zanesljivosti.

Metoda *compareTo*:

```
@Override
public double compareTo(Sydable syd) {
  if (super.compareTo(syd)==0) return 0; // check if ready
  if (getReliability()==0) throw new IllegalArgumentException();
  double s=1;
  for (int i=0; i<getSubSyds().size(); ++i) {
    Sydable syd1=getSubSyds().get(i);
    Sydable syd2=getClass().cast(syd).getSubSyds().get(i);
    if (syd1.isReady()==null)
      throw new NullPointerException("'"+syd1.getName()+"' ready NULL");
    if (syd2.isReady()==null)
      throw new NullPointerException("'"+syd2.getName()+"' ready NULL");
    double si=syd1.compareTo(syd2);
    double ri=syd1.getReliability();
```

```
      s*=1-si*ri;
    }
    return (1-s)/getReliability();
  }
```

Metodi *calculateReliability* in *compareTo* razreda *AbstractContrainerSyd* sta dobro premišljeni, tako da uporabnika, ki želi implementirati nov sestavljen syd, popolnoma razbremenita truda, saj se lahko zanese, da bo delovanje celote povsem zadovoljivo, če bo le poskrbel, da bodo posamezni faktorji zanesljivosti pod-sydov pravilno podani. Ti metodi zagotovita, da bo skupna zanesljivost sestavljenega syda povsem realistična in da bo rezultat primerjave dveh sestavljenih sydov odražal ravno pravo mero vseh pod-sydov. Poleg tega lahko računamo tudi na vse samoumevne osnovne zakonitosti: način, kako sestavimo syd iz pod-sydov, ne vpliva na delovanje teh dveh metod, se pravi, vrstni red podajanja pod-sydov ni važen, prav tako ne igra vloge, kako grupiramo pod-syde na drugih hierarhičnih nivojih. Ravno tako vemo, na primer, da bo rezultat primerjave dveh sestavljenih sydov, kjer so vsi pod-sydi, ki so pripravljeni, med seboj identični, dala vedno rezultat 1, in nasprotno, da bo rezultat vedno 0, kadar bodo vsi pripravljeni pod-sydi med seboj popolnoma različni.

3.4.2. Primer implementacije sestavljenega syda

Implementacija sestavljenega syda je resnično preprosta.
Vse, kar je potrebno narediti, je nadgraditi AbstractContainerSyd in implementirati konstruktor.

Poglejmo si to na primeru syda *Syd*, ki je zamišljen kot glavni, korenski (*root*) syd naše knjižnice:

```
public class Syd extends AbstractContainerSyd {

 public Syd() {
  super("SYD", new HwSyd(), new SwSyd());
  }
}
```

To je vse.
Konstruktor kliče super konstruktor, torej konstruktor razreda *AbstractContainerSyd*, kateremu poda ime syda in elemente sestavljenega syda. Elementa sta dva, *HwSyd* in *SwSyd*, ki sta tudi sama sestavljena syda, implementirana na povsem enak način kot sam *Syd*.

3.4.3. Primer uporabe sestavljenega syda

Poglejmo si še primer uporabe takega sestavljenega syda:

```
    // inspection TEST
    Syd syd=new Syd();
    if (!syd.inspect()) {
      System.out.println("Syd inspection failure.");
    }
    System.out.println("\n\n"+syd.toString());

    // (de)serialization TEST
    Syd sydClone=null;
    try {
      sydClone=AbstractSyd.deserialize(syd.serialize());
      System.out.println("\n\n"+sydClone.getValueAsString());
    } catch (IOException e) {} catch (ClassNotFoundException e) {
      System.out.println("(de)serialization problem.");
    }
    System.out.println("\n\nSyd Comparison: "+syd.discreteCompareTo(sydClone));
```

Uporaba je enostavna.
Syd je potrebno ustvariti s klicem konstruktorja, nato je treba poklicati metodo *inspect*, s čimer syd pripravimo za uporabo. Zdaj ga lahko tudi smiselno izpišemo, ker je njegova vrednost že nastavljena.
Lahko ga serializiramo in ta "podpis" shranimo v podatkovno bazo ali pa, recimo, pošljemo na nek oddaljen strežnik.
Kasneje lahko podpis spet deserializiramo nazaj v syd objekt.
Na koncu vidimo še primerjavo originalnega syda s svojo kopijo (*sydClone*).

Zgornji testni program izpiše:

```
SYD [value='
HW:
  BIOS Data: null
  MAC Address:
    00:1D:E0:6F:B7:25
    00:1E:37:88:EF:8D
  Hard Disk ID: null
  CPU Data: null
SW:
  OS:
    OS Name: Linux
    OS Version: 2.6.24-19-generic
    OS Architecture: amd64
  Java:
    Java Specification Version: 1.6
    Java VM Version: 10.0-b22
    Java VM Specification Version: 1.0
  Computer Name: null', reliability=0.9807090625, ready=true, OS support=[]]
HW [value='
  BIOS Data: null
  MAC Address:
    00:1D:E0:6F:B7:25
    00:1E:37:88:EF:8D
  Hard Disk ID: null
  CPU Data: null', reliability=0.9, ready=true, OS support=[]]
  BIOS Data [value='null', reliability=0.0, ready=false, OS support=[]]
  MAC Address [value='
    00:1D:E0:6F:B7:25
    00:1E:37:88:EF:8D', reliability=0.9, ready=true, OS support=[WINDOWS, LINUX,
MAC]]
  Hard Disk ID [value='null', reliability=0.0, ready=false, OS support=[]]
  CPU Data [value='null', reliability=0.0, ready=false, OS support=[]]
SW [value='
```

```
  OS:
    OS Name: Linux
    OS Version: 2.6.24-19-generic
    OS Architecture: amd64
  Java:
    Java Specification Version: 1.6
    Java VM Version: 10.0-b22
    Java VM Specification Version: 1.0
  Computer Name: null', reliability=0.8070906250000001, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
  OS [value='
    OS Name: Linux
    OS Version: 2.6.24-19-generic
    OS Architecture: amd64', reliability=0.775, ready=true, OS support=[WINDOWS,
LINUX, MAC]]
    OS Name [value='Linux', reliability=0.5, ready=true, OS support=[WINDOWS, LINUX,
MAC]]
    OS Version [value='2.6.24-19-generic', reliability=0.1, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
    OS Architecture [value='amd64', reliability=0.5, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
  Java [value='
    Java Specification Version: 1.6
    Java VM Version: 10.0-b22
    Java VM Specification Version: 1.0', reliability=0.1426250000000001, ready=true,
OS support=[WINDOWS, LINUX, MAC]]
    Java Specification Version [value='1.6', reliability=0.05, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
    Java VM Version [value='10.0-b22', reliability=0.05, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
    Java VM Specification Version [value='1.0', reliability=0.05, ready=true, OS
support=[WINDOWS, LINUX, MAC]]
  Computer Name [value='null', reliability=0.0, ready=false, OS support=[WINDOWS,
LINUX, MAC]]



HW:
  BIOS Data: null
  MAC Address:
    00:1D:E0:6F:B7:25
    00:1E:37:88:EF:8D
  Hard Disk ID: null
  CPU Data: null
SW:
  OS:
    OS Name: Linux
    OS Version: 2.6.24-19-generic
    OS Architecture: amd64
  Java:
    Java Specification Version: 1.6
    Java VM Version: 10.0-b22
    Java VM Specification Version: 1.0
  Computer Name: null


Syd Comparison: IDENTICAL
```

4. IMPLEMENTACIJA SYDOV

Naš cilj je, da bi knjižnica syd tekla na vseh najbolj razširjenih operacijskih sistemih: Windows, Linux (Unix) in MacOS. Upali smo, da bomo z uporabo programskega jezika Java, ki je znana prav po načelu "write once, run anywhere", to dosegli brez večjih težav. Žal pa se je izkazalo, da ravno na tem področju podpora Jave šepa, če se milo izrazimo [6].

Na začetku smo v originalnih knjižnicah Jave našli razred *java.lang.System*, ki ponuja metodo *getProperties*, s katero lahko izvemo določene trenutne sistemske nastavitve: predvsem o Javi (verzija, prodajalec, verzija specifikacije, instalacijska mapa, prevajalnik, ipd.) ter zelo skopo o operacijskem sistemu (ime, arhitektura, verzija). S tem bi lahko že zgradili nek minimalni "posnetek" identifikacije računalnika, a bi bil morda premalo določujoč (ker obstaja preveč kopij iste verzije Jave in operacijskega sistema, sploh v nekem lokalnem obsegu) in pa časovno nekoliko manj obstojen (zaradi menjave verzij Jave ali OS).

Kasneje smo odkrili, da nove verzije Jave (šele od verzije 1.6 naprej) ponujajo metodo *java.net.NetworkInterface.getHardwareAddress*, ki naj bi vrnila MAC naslov danega mrežnega vmesnika. To nas je zelo razveselilo, saj je MAC naslov eden najbolj zanesljivih identifikatorjev.

Uporabna je tudi metoda *java.net.InetAddress.getLocalHost*, ki vrne objekt *InetAddress*, nad katerim lahko nato s klicem metode *getHostName* dobimo ime računalnika.

Vse ostale podatke o sistemu, programski ali strojni opremi, pa je treba dobiti od drugod. Našli smo nekaj Javinih odprtokodnih knjižnic, ki pa delujejo prek JNI (Java Native Interface) vmesnika in posredno uporabljajo kodo, napisano v C-ju ipd. - a so namenjene izključno le Windows sistemom. To pa pomeni, da se tu vsa lepota Jave kot "write once, run anywhere" jezika povsem izgubi.

V tem poglavju si bomo podrobneje pogledali posamezne implementacije osnovnih, atomičnih sydov. Knjižnica syd vključuje naslednje *atomične syde*, grupirane v naslednje **sestavljene syde**:

Syd
- **HwSyd**
  - *BiosDataSyd*
  - *MacAddressSyd*
  - *HardDiskIdSyd*
  - *CpuDataSyd*
- **SwSyd**
  - **OsSyd**
    - *OsNameSyd*
    - *OsVersionSyd*
    - *OsArchitectureSyd*
  - **JavaSyd**

JavaSpecVersionSyd
JavaVmVersionSyd
JavaVmSpecVersionSyd
ComputerNameSyd

Pa si oglejmo vsak atomični syd posebej.

4.1. BiosDataSyd

Dokončna implementacija tega syda je presegla časovne okvire tega dela in je prepuščena prihodnosti.

4.2. MacAddressSyd

MacAddressSyd uporablja kot identifikator MAC naslov mrežne kartice [7], ki je eden najbolj določujočih in zanesljivih identifikatorjev, kar jih je moč pridobiti iz računalniškega sistema. Mrežna kartica se dandanes nahaja že praktično v vsakem računalniku, večkrat celo več kot ena naenkrat. Vsaka mrežna kartica naj bi imela univerzalno unikaten MAC naslov, ki ga je možno pridobiti na enoten in enostaven način v Javi.

Vendar pa se menda lahko zgodi, da so MAC naslovi kakega manj znanega proizvajalca kar enaki za vso serijo izdelanih mrežnih kartic. Poleg tega obstajajo poročila o načinih, kako je možno MAC naslov celo spremeniti. Resnici na ljubo pa v praksi do teh primerov skorajda nikoli ne pride.

Implementacija syda *MacAddressSyd* je razmeroma preprosta. Vsebuje spremenljivko *macAddresses*, ki je seznam vseh MAC naslovov danega računalniškega sistema.

Metoda *inspect* uporablja metodo *java.net.NetworkInterface.getNetworkInterfaces*, ki vrne seznam vseh mrežnih kartic računalniškega sistema, nato pa za vsako od teh pokliče metodo *getHardwareAddress*, ki vrne njen MAC naslov, katerega potem shrani.

Metoda *compareTo* pa se sprehodi čez seznama shranjenih MAC naslovov obeh sydov in primerja vsakega z vsakim. Če pride do kakega zadetka, vrne 1, sicer pa 0.

4.3. HardDiskIdSyd

Dokončna implementacija tega syda je presegla časovne okvire tega dela in je prepuščena prihodnosti.

4.4. CpuDataSyd

Dokončna implementacija tega syda je presegla časovne okvire tega dela in je prepuščena prihodnosti.

4.5. OsNameSyd, OsVersionSyd, OsArchitectureSyd

Vsi ti trije sydi so implementirani s pomočjo metode *java.lang.System.getProperty*, zato uporabljajo skupen abstraktni razred *AbstractSystemPropertySyd*.

Le-ta definira spremenljivki *propertyName* in *value* ter metodo *inspect*, ki kliče metodo *getProperty*, ter metodo *compareTo*, ki vrne 1, kadar sta vrednosti povsem enaki, sicer pa 0.

Vrednosti *propertyName* teh treh sydov so, po vrsti: `"os.name"`, `"os.version"`, `"os.arch"`.
Zanesljivost imata prvi in zadnji 0.5, drugi pa 0.1.

4.6. JavaSpecVersionSyd, JavaVmVersionSyd, JavaVmSpecVersionSyd

Tudi ti trije sydi temeljijo na klicu metode *java.lang.System.getProperty* in prav tako uporabljajo *AbstractSystemPropertySyd*.

Vrednosti *propertyName* teh treh sydov so, po vrsti: `"java.specification.version"`, `"java.vm.version"`, `"java.vm.specification.version"`.
Zanesljivost imajo vsi 0.05.

4.7. ComputerNameSyd

Implementacija tega syda je že nadrobno opisana v razdelku 3.3.2.

5. SKLEP

V tem delu smo poglobljeno preučili parametre podobnosti in različnosti računalniških sistemov ter raziskali tako njihovo posamezno določljivost kot tudi učinke njihovega sestavljanja. Rezultat tega je knjižnica syd, ki je zelo primerno orodje za ugotavljanje istovetnosti računalniških sistemov.
Njene glavne odlike so:

- preudarna zasnova, ki naslovi vse potrebne značilnosti sistemskih identifikatorjev;
- močno jedro abstraktnih razredov, ki omogoča pravilno in realistično uporabo sestavljenih identifikatorjev;
- popolna odprtost in nastavljivost ter res preprosta razširljivost z novimi identifikatorji.

Zasnova knjižnice temelji na povsem izvirnih idejah in razmišljanjih ter ne skuša namerno posnemati nobene podobne rešitve, ki bi utegnila obstajati v svetu.

Poleg tega gre pri tem za skrajno preprostost: poiskati je treba čimveč določujočih sistemskih parametrov, pri čemer se skuša upoštevati vse, kar je v računalniku; ker pa ne obstaja en sam popolnoma določujoč parameter, je potrebno združiti več parametrov skupaj.

Že hiter površinski pregled informacij na internetu nam razkrije, da se s to problematiko že dolgo srečuje marsikdo, a kljub temu še vedno ni mogoče najti nobene dobre, uporabne rešitve. To nam potrjuje, da gre za perečo zadevo, ki pa nima enostavne praktične rešitve, čeprav je sama ideja tako preprosta. Splošno ime za eno od glavnih aplikacij tega je *Aktivacija produkta* (*Product Activation*) [8].

Pri enem od teh primerov smo se prepričali, da so poskusili priti do takega sestavljenega identifikatorja s pregledovanje osmih različnih identifikatorjev strojne opreme [9]. Razlikovanje med dvema primerkoma tega sestavljenega identifikatorja so uporabili za določanje istovetnosti sistema in posledično za odklenitev (aktivacijo) programske opreme. Vendar pa smo tudi prebrali, da je tak pristop preveč pogosto zavrnil istovetnost sistema po pomoti in to je povzročalo kar nekaj hude krvi pri uporabnikih.

Naša obravnava sestavljenih identifikatorjev, predvsem mislimo pri tem na algoritem izračuna skupne zanesljivosti in skupne podobnosti pri primerjavi dveh sestavljenih identifikatorjev, pa je hkrati zadosti robustna in premišljena, kot tudi zadosti razširljiva, da daje uporabniku upanje, da mu ne bo treba izgubljati nepotrebnega časa zavoljo napačne ocene istovetnosti. Seveda se pričakuje, da bo potrebno za vsako novo problemsko domeno, kjer bi se knjižnica syd uporabljala, na začetku nekoliko premisliti in stestirati ter uglasiti nastavitve posameznih identifikatorjev (sploh faktorje zanesljivosti), vendar si drznemo optimistično verjeti, da ta proces nikakor ne bo postal trn v peti uporabnikom, kot se, sodeč po nezadovoljstvu uporabnikov zgoraj omenjenih rešitev, lahko tako hitro pripeti pri bolj togih in zaprtih sistemih.

Zdaj pa naštejmo še nekaj smernic, kamor bi se lahko knjižnica syd s pridom razvijala v

prihodnosti:

- natančna kalibracija nastavitev (operacija *compareTo*, faktor zanesljivosti...) identifikatorjev glede na pridobljene praktične izkušnje in statistične pokazatelje;
- implementacija novih osnovnih identifikatorjev (grafična kartica, SCSI in IDE adapter, CD/DVD enota...);
- podpora vsem identifikatorjem na vseh operacijskih sistemih;
- izdelava lastnih "native" knjižnic s pomočjo JNA rešitev [10].

Prva točka zahteva le nekoliko več časa in večji nabor primerov uporabe knjižnice. Predvidevamo, da se bo sčasoma ta kalibracija skozi uporabo povsem neopazno zgodila.

Druga in tretja točka sta nekoliko bolj vprašljivi, saj ni nobenega zagotovila, da vsi našteti osnovni sistemski identifikatorji sploh obstajajo, kaj šele, da jih je mogoče tudi programsko pridobiti. Potrebna je bolj globoka raziskava na "nizkem" tehničnem nivoju strojne opreme in sistemskih klicev.

Zadnja točka je bolj enostavna. Potrebno je preučiti uporabo JNA knjižnice in jo uporabiti za implementacijo tistih identifikatorjev, ki zahtevajo "native" pristop.

6. PRILOGE

Izvorna koda knjižnice syd.

6.1. Sydable.java

```
package syd;

import java.io.IOException;
import java.io.Serializable;
import java.util.Set;

/**
 * SYstem iDentification interface.
 *
 * @author Borut Lesjak
 */
public interface Sydable extends Serializable {

  /**
   * The name of the Syd entity, e.g. "MAC Address".
   *
   * @return Syd entity name
   */
  String getName();

  /** Operating systems. */
  enum OS {
    WINDOWS, LINUX, MAC
  }

  /**
   * Which operating systems are supported by this Syd entity?
   *
   * @return OS support set
   */
  Set<OS> getOsSupport();

  /**
   * Inspects the system to obtain this Syd entity's value. <br>
   * *Must* be implemented (*with* a call to AbstractSyd.setReady) and called at
   * least once or the compareTo and serialize methods (as defined in
   * AbstractSyd) will throw IllegalArgumentException. <br>
   * Usually calls a third-party method, provided in a separate native library.
   * <br>
   * Note that before a Syd instance has been successfully inspected, its
   * reliability is 0. <br>
   * Returns <code>true</code> if inspection went well and <code>false</code>
   * if something went wrong (e.g. MAC Address could not be obtained).
   *
   * @return inspection success
   */
```

```
boolean inspect();

/**
 * Is this Syd entity ready to be used (i.e. compared to other instances)?
 * <br>
 * Ready flag is set when the inspect method has been called successfully and
 * reset (<code>false</code>) if the inspect method call was not
 * successful. If the inspect methods has not been called at all, the ready
 * flag is <code>null</code>. <br>
 * Note that before a Syd instance has been successfully inspected, its
 * reliability is 0. <br>
 *
 * @return Syd entity status or <code>null</code>
 */
Boolean isReady();

/**
 * The value of the Syd entity, as a String, e.g. "01:23:45:67:89:ab".
 *
 * @return Syd entity value as String
 */
String getValueAsString();

/**
 * Determines the reliability factor of this Syd entity instance, a real value
 * between 0 and 1, inclusively, 0 being completely unreliable and 1 being
 * completely determining. <br>
 * Note that before a Syd instance has been successfully inspected, its
 * reliability is 0. <br>
 * For example, ComputerNameSyd should have a reliability of no more than
 * 0.05, since it is extremely easy to change the name of the computer or find
 * another computer with the same name by chance. On the other hand, changing
 * the MAC address is practically impossible and they are expected to be
 * globally unique, so the reliability should be at least 0.95. <br>
 * Reliability factor is used when calclulating the total comparison result of
 * two container syds by comparing the sub-syds.
 *
 * @return reliability factor
 */
double getReliability();

/**
 * Compares this Syd entity instance to another, specified by the
 * <code>syd</code> parameter. <br>
 * Returns <code>similarity</code>, a continuous real value between 0 and
 * 1, inclusively: 0 when comparing two completely different values and 1 for
 * two identical values. <br>
 * All implementations *should* necessarily call the AbstractSyd.compareTo to
 * check all the basics (null, type compatibility, ready flag). <br>
 * Use <code>discretizeSimilarity</code> to convert this rather
 * uninformative value into a readble, easy-to-understand enum value. <br>
 * Note that AbstractSyd provides
 * {@link AbstractSyd#discreteCompareTo(Sydable)} method which delegates to
 * the <code>compareTo</code> and <code>discretizeSimilarity</code>
 * methods. <br>
 * The semantics of comparison and discretization depends on each individual
 * Syd entity implementaion. For example, two MAC addresses can only be either
 * IDENTICAL or DIFFERENT, whereas two versions of OS may as well be SAME (as
 * opposed to IDENTICAL) if they only differ by a minor revision or build
 * number, or SIMILAR (as opposed to DIFFERENT) if they differ by a major
 * version number, but are still of the same kind. <br>
 *
 * @param syd another Syd entity instance to compare to (must not be
 *        <code>null</code>)
 * @return comparison result
```

```
   * @throws NullPointerException if parameter is <code>null</code>
   * @throws ClassCastException if the parameter type is not identical to this
   *         Syd entity's type
   * @throws IllegalArgumentException if inspect has not been called yet for
   *         either operand
   */
  double compareTo(Sydable syd);

  /** A measure of two Syd entity instances discrete similarity. */
  enum Similarity {
    IDENTICAL, SAME, SIMILAR, DIFFERENT
  }

  /**
   * Discretize a continuous similarity value to a enum similarity value. <br>
   * The default discretization intervals are set like this (in AbstractSyd, if
   * a Sydable implementation chooses to extend it):
   *
   * <pre>
   * - IDENTICAL: 1.00
   * - SAME: 0.90-0.99
   * - SIMILAR: 0.50-0.89
   * - DIFFERENT: 0.00-0.49
   * </pre>
   *
   * Each individual Syd entity implementaion may override the default settings,
   * for example, two MAC addresses can only be either IDENTICAL (1.00) or
   * DIFFERENT (0.00-0.99). <br>
   * Currently set thresholds may be obtained by methods
   * getThresholdIdentical(), getThresholdSame(), getThresholdSimilar(), defined
   * in AbstractSyd.
   *
   * @param continuousSimilarity continuous similarity value
   * @return enum similarity value or <code>null</code>
   * @throws IllegalArgumentException if continuousSimilarity not within [0,1]
   */
  Similarity discretizeSimilarity(double continuousSimilarity);

  /**
   * Serializes this Syd entity instance to a byte stream. <br>
   *
   * @return serialized instance
   * @throws IllegalArgumentException if inspect has not been called yet
   * @throws IOException if serialization fails
   */
  byte[] serialize() throws IOException;
}
```

6.2. AbstractSyd.java

```
package syd;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

```java
import java.util.Arrays;
import java.util.Set;
import java.util.TreeSet;

/**
 * Abstract Syd.
 *
 * @author Borut Lesjak
 */
public abstract class AbstractSyd implements Sydable {

  /**
   * The name of the Syd entity, e.g. "MacAddress".
   */
  private String name;

  /**
   * Is this Syd entity ready to be used (i.e. compared to other instances)?
   * <br>
   * Ready flag is set when the inspect method has been called successfully and
   * reset (<code>false</code>) if the inspect method call was not
   * successful. If the inspect methods has not been called at all, the ready
   * flag is <code>null</code>. <br>
   * Note that before a Syd instance has been successfully inspected, its
   * reliability is 0. <br>
   */
  private Boolean ready=null;

  /**
   * Reliability factor. <br>
   *
   * @see Sydable#getReliability()
   */
  private double reliability;

  /**
   * Similarity discretization thresholds.
   *
   * @see Sydable#discretizeSimilarity(double)
   */
  private double[] discretizationThresholds={1, 0.9, 0.5};

  /**
   * Class constructor.
   *
   * @param name name of the Syd entity
   * @param reliability reliability factor (within [0,1])
   * @throws IllegalArgumentException if reliability is out of range
   * @see Sydable#getReliability()
   */
  public AbstractSyd(String name, double reliability) {
    this.name=name;
    setReliability(reliability);
  }

  /**
   * Class constructor.
   *
   * @param name name of the Syd entity
   * @param reliability reliability factor
   * @param thresholdIdentical discretization threshold for IDENTICAL
   * @param thresholdSame discretization threshold for SAME
   * @param thresholdSimilar discretization threshold for SIMILAR
   * @throws IllegalArgumentException if reliability is out of range
   * @see Sydable#getReliability()
```

```
   * @see Sydable#discretizeSimilarity(double)
   */
  public AbstractSyd(String name, double reliability,
    double thresholdIdentical, double thresholdSame, double thresholdSimilar) {
    this(name, reliability);
    this.discretizationThresholds[0]=thresholdIdentical;
    this.discretizationThresholds[1]=thresholdSame;
    this.discretizationThresholds[2]=thresholdSimilar;
  }

  public String getName() {
    return this.name;
  }

  public Boolean isReady() {
    return this.ready;
  }

  /** Full OS support. */
  public static final Set<OS> fullOsSupport=new TreeSet<OS>(Arrays.asList(
    OS.WINDOWS, OS.LINUX, OS.MAC));

  public Set<OS> getOsSupport() {
    return new TreeSet<OS>();
  }

  /**
   * Sets the ready flag. <br>
   * Must be called after or from the inspect method implementation or the
   * compareTo and serialize methods will throw IllegalArgumentException.
   *
   * @param ready ready status
   */
  protected void setReady(boolean ready) {
    this.ready=ready;
  }

  public double getReliability() {
    if (isReady()==null) return 0; // syd has not been inspected yet
    if (!isReady()) return 0; // syd has not been successfully inspected
    return this.reliability;
  }

  /**
   * Sets the reliability factor.
   *
   * @param reliability reliability factor (within [0,1])
   * @throws IllegalArgumentException if reliability is out of range
   */
  protected void setReliability(double reliability) {
    if (reliability<0||reliability>1) throw new IllegalArgumentException();
    this.reliability=reliability;
  }

  /**
   * Returns the threshold for Similarity.IDENTICAL.
   *
   * @return Similarity.IDENTICAL threshold
   */
  public double getThresholdIdentical() {
    return getDiscretizationThresholds()[0];
  }

  /**
   * Returns the threshold for Similarity.SAME.
```

```
 *
 * @return Similarity.SAME threshold
 */
public double getThresholdSame() {
  return getDiscretizationThresholds()[1];
}

/**
 * Returns the threshold for Similarity.SIMILAR.
 *
 * @return Similarity.SIMILAR threshold
 */
public double getThresholdSimilar() {
  return getDiscretizationThresholds()[2];
}

@Override
public String toString() {
  return getName()+" [value='"+getValueAsString()+"', reliability="
    +getReliability()+", ready="+isReady()+", OS support="
    +getOsSupport().toString()+"]";
}

public Similarity discretizeSimilarity(double continuousSimilarity) {
  if (continuousSimilarity>1||continuousSimilarity<0)
    throw new IllegalArgumentException("continuousSimilarity="
      +continuousSimilarity);
  if (continuousSimilarity>=getDiscretizationThresholds()[0])
    return Similarity.IDENTICAL;
  if (continuousSimilarity>=getDiscretizationThresholds()[1])
    return Similarity.SAME;
  if (continuousSimilarity>=getDiscretizationThresholds()[2])
    return Similarity.SIMILAR;
  return Similarity.DIFFERENT;
}

/**
 * Checks the basics: the parameter must no be null and the types must be
 * assignment compatible.
 *
 * @param syd syd to compare to
 * @return <code>1</code> if both syds are ready, <code>0</code> otherwise
 */
public double compareTo(Sydable syd) {
  if (syd==null) throw new NullPointerException(); // must not be null
  if (!getClass().isInstance(syd)) throw new ClassCastException(); // type
  if (isReady()==null)
    throw new IllegalArgumentException("this.inspect not called");
  if (syd.isReady()==null)
    throw new IllegalArgumentException("syd.inspect not called");
  if (!isReady()||!syd.isReady()) return 0;
  return 1;
}

/**
 * Delegates to {@link #compareTo(Sydable)} and
 * {@link #discretizeSimilarity(double)}.
 *
 * @param syd see {@link Sydable#compareTo(Sydable)}
 * @return see {@link Sydable#discretizeSimilarity(double)}
 */
public Similarity discreteCompareTo(Sydable syd) {
  return discretizeSimilarity(compareTo(syd));
}
```

```
  public byte[] serialize() throws IOException {
    if (isReady()==null)
      throw new IllegalArgumentException("inspect not called");
    ByteArrayOutputStream baos=new ByteArrayOutputStream();
    ObjectOutputStream oos=new ObjectOutputStream(baos);
    oos.writeObject(this);
    oos.flush();
    return baos.toByteArray();
  }

  /**
   * Deserializes a Syd entity instance from a byte stream. <br>
   *
   * @param byteStream byte stream to deserialize from (must not be
   *        <code>null</code>)
   * @param <T> deserialized instance target class
   * @return deserialized instance
   * @throws NullPointerException if parameter is <code>null</code>
   * @throws IOException if deserialization goes wrong
   * @throws ClassNotFoundException if deserialization goes wrong
   */
  @SuppressWarnings("unchecked")
  public static <T extends Sydable> T deserialize(byte[] byteStream)
    throws IOException, ClassNotFoundException {
    if (byteStream==null) throw new NullPointerException();
    ByteArrayInputStream bais=new ByteArrayInputStream(byteStream);
    ObjectInputStream ois=new ObjectInputStream(bais);
    Object object=ois.readObject();
    return (T)object;
  }

  /**
   * @return the discretizationThresholds.
   */
  private double[] getDiscretizationThresholds() {
    return discretizationThresholds;
  }
}
```

6.3. AbstractContainerSyd.java

```
package syd;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

/**
 * Abstract Container Syd. <br>
 * An abstract container for sub-syds. Provides universal methods for obtaining
 * OS support, doing inspection and comparison and converting to string value.
 *
 * @author Borut Lesjak
 */
public class AbstractContainerSyd extends AbstractSyd {
```

```
/** Sub-syd list. */
List<Sydable> subSyds=new ArrayList<Sydable>();

/**
 * Class constructor. <br>
 *
 * @param name container syd name
 * @param subSyds sub-syds array
 */
public AbstractContainerSyd(String name, Sydable... subSyds) {
  super(name, 0);
  this.subSyds.addAll(Arrays.asList(subSyds));
}

/**
 * Inspects all the sub-syds and checks their ready status and returns
 * <code>true</code>, if at least one of them is ready. <br>
 * Also calculates and sets the overall reliability.
 *
 * @return ready status
 */
public boolean inspect() {
  boolean ready=false;
  for (Sydable subSyd : getSubSyds()) {
    ready|=subSyd.inspect();
  }
  setReliability(calculateReliability(getSubSyds()));
  setReady(ready);
  return ready;
}

/**
 * Returns an intersection of all sub-syd OS supports.
 */
@Override
public Set<OS> getOsSupport() {
  Set<OS> result=new TreeSet<OS>(AbstractSyd.fullOsSupport);
  for (Sydable subSyd : getSubSyds())
    result.retainAll(subSyd.getOsSupport()); // set intersection
  return result;
}

/**
 * Returns a concatenation of all sub-syd string values.
 */
public String getValueAsString() {
  String result="";
  for (Sydable subSyd : getSubSyds())
    result+=subSyd.getName()+": "+subSyd.getValueAsString();
  return result;
}

@Override
public String toString() {
  String result=super.toString();
  for (Sydable subSyd : getSubSyds())
    result+=subSyd.toString();
  return result;
}

/**
 * Calculates an overall realiability for this container Syd. <br>
 * Uses the following formula:
 *
```

```
   * <pre>
   * R=1-Product(1-Ri)
   * </pre>
   *
   * where Ri stands for the individual reliability of i-th sub-syd. <br>
   * It can be proven that the above function is commutative and associative
   * (i.e. different groupings within the syd containers do not affect the
   * outcome) and bounded within the [0,1] real interval. <br>
   *
   * @param subSyds sub-syd list
   * @return an overall container reliability
   */
  static protected double calculateReliability(List<Sydable> subSyds) {
    double r=1;
    for (Sydable subSyd : subSyds) {
      r*=1-subSyd.getReliability();
    }
    return 1-r;
  }

  /**
   * Returns the overall (continuous) similarity, calculated from weighted
   * sub-syd instances comparisons. <br>
   * Uses the following formula:
   *
   * <pre>
   * S=(1-Product(1-Si*Ri))/R
   * </pre>
   *
   * where Si stands for the individual similarity of the two i-th sub-syds
   * instances, Ri stands for the individual reliability of i-th sub-syd, and R
   * stands for the overall reliability of this container. R must never be 0, or
   * the IllegalArgumentException is thrown. <br>
   * It can be proven that the above function is commutative and associative
   * (i.e. different groupings within the syd containers do not affect the
   * outcome) and bounded within the [0,1] real interval. <br>
   * Note that reliability factors of *this* syd are used, not of the parameter,
   * which can make a difference when comparing a current syd with an archived
   * one, if any of the reliability factors have been changed in between.
   *
   * @throws IllegalArgumentException if this container's reliability is 0
   */
  @Override
  public double compareTo(Sydable syd) {
    if (super.compareTo(syd)==0) return 0; // check if ready
    if (getReliability()==0) throw new IllegalArgumentException();
    double s=1;
    for (int i=0; i<getSubSyds().size(); ++i) {
      Sydable syd1=getSubSyds().get(i);
      Sydable syd2=getClass().cast(syd).getSubSyds().get(i);
      if (syd1.isReady()==null)
        throw new NullPointerException("'"+syd1.getName()+"' ready NULL");
      if (syd2.isReady()==null)
        throw new NullPointerException("'"+syd2.getName()+"' ready NULL");
      double si=syd1.compareTo(syd2);
      double ri=syd1.getReliability();
      s*=1-si*ri;
    }
    return (1-s)/getReliability();
  }

  /**
   * @return the subSyds.
   */
  public List<Sydable> getSubSyds() {
```

```
    return subSyds;
  }
}
```

6.4. AbstractSystemPropertySyd.java

```
package syd;

import java.util.Set;

/**
 * Abstract Syd for System Property values.
 *
 * @author Borut Lesjak
 */
public class AbstractSystemPropertySyd extends AbstractSyd {

  /** System property name. */
  private String propertyName;

  /** OS Version value. */
  private String value;

  /**
   * Class constructor.
   *
   * @param name name of the Syd entity
   * @param reliability reliability factor
   * @param propertyName System property name, e.g. "os.version".
   */
  public AbstractSystemPropertySyd(String name, double reliability,
    String propertyName) {
    super(name, reliability);
    this.propertyName=propertyName;
  }

  @Override
  public double compareTo(Sydable syd) {
    if (super.compareTo(syd)==0) return 0; // check if ready
    if (getValue().compareTo(syd.getValueAsString())==0) return 1;
    return 0;
  }

  @Override
  public Set<OS> getOsSupport() {
    return AbstractSyd.fullOsSupport;
  }

  public String getValueAsString() {
    return getValue();
  }

  public boolean inspect() {
    setReady(false);
    if (getPropertyName()==null) return false;
    String value=System.getProperty(getPropertyName());
    if (value==null) return false;
```

```
    setValue(value);
    setReady(true);
    return true;
  }

  /**
   * @return the propertyName.
   */
  private String getPropertyName() {
    return propertyName;
  }

  /**
   * @return the value.
   */
  protected String getValue() {
    return value;
  }

  /**
   * @param value the value.
   */
  protected void setValue(String value) {
    this.value=value;
  }
}
```

6.5. Syd.java

```
package syd;

import syd.hw.HwSyd;
import syd.sw.SwSyd;

/**
 * Global container Syd.
 *
 * @author Borut Lesjak
 */
public class Syd extends AbstractContainerSyd {

  /**
   * Class constructor.
   */
  public Syd() {
    super("SYD", new HwSyd(), new SwSyd());
  }
}
```

6.6. HwSyd.java

```
package syd.hw;

import syd.AbstractContainerSyd;

/**
 * HW Syd. <br>
 * Composed of HW sub-syds.
 *
 * @author Borut Lesjak
 */
public class HwSyd extends AbstractContainerSyd {

  /**
   * Class constructor.
   */
  public HwSyd() {
    super("\nHW", new BiosDataSyd(), new MacAddressSyd(), new HardDiskIdSyd(),
      new CpuDataSyd());
  }
}
```

6.7. SwSyd.java

```
package syd.sw;

import syd.AbstractContainerSyd;
import syd.sw.java.JavaSyd;
import syd.sw.os.OsSyd;

/**
 * SW Syd. <br>
 * Composed of SW sub-syds.
 *
 * @author Borut Lesjak
 */
public class SwSyd extends AbstractContainerSyd {

  /**
   * Class constructor.
   */
  public SwSyd() {
    super("\nSW", new OsSyd(), new JavaSyd(), new ComputerNameSyd());
  }
}
```

6.8. OsSyd.java

```
package syd.sw.os;

import syd.AbstractContainerSyd;

/**
 * OS Syd. <br>
 * Composed of OS sub-syds.
 *
 * @author Borut Lesjak
 */
public class OsSyd extends AbstractContainerSyd {

 /**
  * Class constructor.
  */
 public OsSyd() {
  super("\n  OS", new OsNameSyd(), new OsVersionSyd(),
      new OsArchitectureSyd());
 }
}
```

6.9. JavaSyd.java

```
package syd.sw.java;

import syd.AbstractContainerSyd;

/**
 * Java Syd. <br>
 * Composed of Java sub-syds.
 *
 * @author Borut Lesjak
 */
public class JavaSyd extends AbstractContainerSyd {

 /**
  * Class constructor.
  */
 public JavaSyd() {
  super("\n  Java", new JavaSpecVersionSyd(), new JavaVmVersionSyd(),
      new JavaVmSpecVersionSyd());
 }
}
```

6.10. BiosDataSyd.java

```
package syd.hw;

import syd.AbstractSyd;

/**
 * BIOS Data Syd.
 *
 * @author Borut Lesjak
 */
public class BiosDataSyd extends AbstractSyd {

 /**
  * Class constructor.
  */
 public BiosDataSyd() {
  super("\n  BIOS Data", 0.95);
  }

 public String getValueAsString() {
  return null;
  }

 public boolean inspect() {
    setReady(false);
    return false;
  }
}
```

6.11. CpuDataSyd.java

```
package syd.hw;

import syd.AbstractSyd;

/**
 * CPU Data Syd.
 *
 * @author Borut Lesjak
 */
public class CpuDataSyd extends AbstractSyd {

 /**
  * Class constructor.
  */
 public CpuDataSyd() {
  super("\n  CPU Data", 0.95);
  }

 public String getValueAsString() {
  return null;
  }
```

```
 public boolean inspect() {
    setReady(false);
    return false;
  }

}
```

6.12. HardDiskIdSyd.java

```
package syd.hw;

import syd.AbstractSyd;

/**
 * Hard Disk Id Syd.
 *
 * @author Borut Lesjak
 */
public class HardDiskIdSyd extends AbstractSyd {

 /**
  * Class constructor.
  */
 public HardDiskIdSyd() {
  super("\n  Hard Disk ID", 0.85);
  }

 public String getValueAsString() {
  return null;
  }

 public boolean inspect() {
    setReady(false);
    return false;
  }

}
```

6.13. MacAddressSyd.java

```
package syd.hw;

import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
```

```java
import java.util.Set;

import syd.AbstractSyd;
import syd.Sydable;

/**
 * MAC Address Syd.
 *
 * @author Borut Lesjak
 */
public class MacAddressSyd extends AbstractSyd {

  /** MAC Address list. */
  List<byte[]> macAddresses=new ArrayList<byte[]>();

  /**
   * Class constructor.
   */
  public MacAddressSyd() {
    super("\n  MAC Address", 0.9);
  }

  @Override
  public Set<OS> getOsSupport() {
    return AbstractSyd.fullOsSupport;
  }

  /**
   * Convert a byte value to a hex string representation.
   *
   * @param b
   * @return hex string
   */
  private String byteToHex(byte b) {
    String dataHexDigit=Integer.toHexString(b);
    while (dataHexDigit.length()<2) {
      dataHexDigit="0"+dataHexDigit;
    }
    dataHexDigit=dataHexDigit.substring(dataHexDigit.length()-2);
    return dataHexDigit.toUpperCase();
  }

  /**
   * Convert a MAC address to a hex string representation.
   *
   * @param macAddress MAC address
   * @return hex string
   */
  private String macAddressToString(byte[] macAddress) {
    String result="";
    for (byte b : macAddress) {
      if (!result.isEmpty()) result+=":";
      result+=byteToHex(b);
    }
    return result;
  }

  public String getValueAsString() {
    String result="";
    for (byte[] addr : getMacAddresses())
      result+="\n    "+macAddressToString(addr);
    return result;
  }

  public boolean inspect() {
```

```
    // Compose a list of all NICs' MAC addresses
    try {
      Enumeration<NetworkInterface> nis=NetworkInterface.getNetworkInterfaces();
      while (nis.hasMoreElements()) {
        NetworkInterface networkInterface=nis.nextElement();
        byte[] addr=networkInterface.getHardwareAddress();
        if (addr==null) continue;
        getMacAddresses().add(addr);
      }
    } catch (SocketException e) {
      setReady(false);
      return false;
    }
    setReady(true);
    return true;
  }

  /**
   * Check if the two MAC addresses are equal.
   *
   * @param macAddress1
   * @param macAddress2
   * @return equality
   */
  private boolean macAddressEquals(byte[] macAddress1, byte[] macAddress2) {
    if (macAddress1.length!=macAddress2.length) return false;
    for (int i=0; i<macAddress1.length; ++i)
      if (macAddress1[i]!=macAddress2[i]) return false;
    return true;
  }

  @Override
  public double compareTo(Sydable syd) {
    if (super.compareTo(syd)==0) return 0; // check if ready

    // Compare every MAC address from one list to every one on the other list
    for (byte[] thisMacAddress : getMacAddresses()) {
      for (byte[] sydMacAddress : ((MacAddressSyd)syd).getMacAddresses()) {
        // Any match means IDENTICAL
        if (macAddressEquals(thisMacAddress, sydMacAddress)) return 1;
      }
    }
    return 0;
  }

  /**
   * @return the macAddresses.
   */
  private List<byte[]> getMacAddresses() {
    return macAddresses;
  }
}
```

6.14. JavaSpecVersionSyd.java

```
package syd.sw.java;
```

```
import syd.AbstractSystemPropertySyd;

/**
 * Java Specification Version System Property Syd.
 *
 * @author Borut Lesjak
 */
public class JavaSpecVersionSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public JavaSpecVersionSyd() {
  super("\n    Java Specification Version", 0.05,
   "java.specification.version");
  }
}
```

6.15. JavaVmSpecVersionSyd.java

```
package syd.sw.java;

import syd.AbstractSystemPropertySyd;

/**
 * Java VM Specification Version System Property Syd.
 *
 * @author Borut Lesjak
 */
public class JavaVmSpecVersionSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public JavaVmSpecVersionSyd() {
  super("\n    Java VM Specification Version", 0.05,
   "java.vm.specification.version");
  }

}
```

6.16. JavaVmVersionSyd.java

```
package syd.sw.java;

import syd.AbstractSystemPropertySyd;
```

```
/**
 * Java VM Version System Property Syd.
 *
 * @author Borut Lesjak
 */
public class JavaVmVersionSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public JavaVmVersionSyd() {
  super("\n    Java VM Version", 0.05, "java.vm.version");
  }

}
```

6.17. OsArchitectureSyd.java

```
package syd.sw.os;

import syd.AbstractSystemPropertySyd;

/**
 * OS Architecture System Property Syd.
 *
 * @author Borut Lesjak
 */
public class OsArchitectureSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public OsArchitectureSyd() {
  super("\n    OS Architecture", 0.5, "os.arch");
  }
}
```

6.18. OsNameSyd.java

```
package syd.sw.os;

import syd.AbstractSystemPropertySyd;

/**
 * OS Name System Property Syd.
 *
 * @author Borut Lesjak
```

```
 */
public class OsNameSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public OsNameSyd() {
  super("\n    OS Name", 0.5, "os.name");
  }
}
```

6.19. OsVersionSyd.java

```
package syd.sw.os;

import syd.AbstractSystemPropertySyd;

/**
 * OS Version System Property Syd.
 *
 * @author Borut Lesjak
 */
public class OsVersionSyd extends AbstractSystemPropertySyd {

 /**
  * Class constructor.
  */
 public OsVersionSyd() {
  super("\n    OS Version", 0.1, "os.version");
  }
}
```

6.20. ComputerNameSyd.java

```
package syd.sw;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Set;

import syd.AbstractSyd;
import syd.Sydable;

/**
 * Computer Name Syd.
 *
 * @author Borut Lesjak
```

```
 */
public class ComputerNameSyd extends AbstractSyd {

  /** Computer name. */
  private String computerName;

  /**
   * Class constructor.
   */
  public ComputerNameSyd() {
    super("\n  Computer Name", 0.05);
  }

  @Override
  public Set<OS> getOsSupport() {
    return AbstractSyd.fullOsSupport;
  }

  public String getValueAsString() {
    return getComputerName();
  }

  public boolean inspect() {
    setReady(false);
    try {
      setComputerName(InetAddress.getLocalHost().getHostName());
    } catch (UnknownHostException e) {
      return false;
    }
    setReady(true);
    return true;
  }

  @Override
  public double compareTo(Sydable syd) {
    if (super.compareTo(syd)==0) return 0; // check if ready

    if (getComputerName().equalsIgnoreCase(
      ((ComputerNameSyd)syd).getComputerName())) return 1;
    return 0;
  }

  /**
   * @return the computerName.
   */
  private String getComputerName() {
    return computerName;
  }

  /**
   * @param computerName the computerName.
   */
  private void setComputerName(String computerName) {
    this.computerName=computerName;
  }
}
```

6.21. CurrentTimeSyd.java

```
package syd.test;

import java.util.Arrays;
import java.util.Set;
import java.util.TreeSet;

import syd.AbstractSyd;
import syd.Sydable;

/**
 * @author Borut Lesjak
 */
public class CurrentTimeSyd extends AbstractSyd {

  /** Current system time. */
  private long currentTime=0;

  /**
   * Class constructor.
   */
  public CurrentTimeSyd() {
    super("\n  Time", 1);
  }

  @Override
  public double compareTo(Sydable syd) {
    if (super.compareTo(syd)==0) return 0; // check if ready
    long time1=getCurrentTime();
    long time2=getClass().cast(syd).getCurrentTime();
    long difference=Math.abs(time1-time2);
    if (difference==0) return 1;
    if (difference<100) return getThresholdSame(); // within 100 ms
    if (difference<1000) return getThresholdSimilar(); // within 1 second
    return 0;
  }

  @Override
  public Set<OS> getOsSupport() {
    return new TreeSet<OS>(Arrays.asList(OS.WINDOWS, OS.LINUX));
  }

  public String getValueAsString() {
    return ""+getCurrentTime();
  }

  public boolean inspect() {
    setCurrentTime(System.currentTimeMillis());
    setReady(true);
    return true;
  }

  /**
   * @return the currentTime.
   */
  private long getCurrentTime() {
    return currentTime;
  }

  /**
   * @param currentTime the currentTime.
```

```
   */
  private void setCurrentTime(long currentTime) {
    this.currentTime=currentTime;
  }
}
```

6.22. SydTest.java

```
package syd;

import java.io.IOException;

import syd.test.CurrentTimeSyd;

/**
 * Syd Test.
 *
 * @author Borut Lesjak
 */
public class SydTest {

  /**
   * Test.
   *
   * @param args
   */
  public static void main(String[] args) {
    // inspection TEST
    Syd syd=new Syd();
    System.out.println("\n\n"+syd.toString());
    if (!syd.inspect()) {
      System.out.println("Syd inspection failure.");
    }
    System.out.println("\n\n"+syd.toString());

    // (de)serialization TEST
    Syd sydClone=null;
    try {
      sydClone=AbstractSyd.deserialize(syd.serialize());
      System.out.println("\n\n"+sydClone.getValueAsString());
    } catch (IOException e) {} catch (ClassNotFoundException e) {
      System.out.println("(de)serialization problem.");
    }
    System.out.println("\n\nSyd Comparison: "+syd.discreteCompareTo(sydClone));

    // comparison TEST, increasing current time
    CurrentTimeSyd s1=new CurrentTimeSyd();
    s1.inspect();
    CurrentTimeSyd s2=new CurrentTimeSyd();
    s2.inspect();
    System.out.println("\n\nComparison: "+s1.discreteCompareTo(s2));
    delay(10);
    s2.inspect();
    System.out.println("Comparison: "+s1.discreteCompareTo(s2));
    delay(100);
    s2.inspect();
    System.out.println("Comparison: "+s1.discreteCompareTo(s2));
```

```
    delay(1000);
    s2.inspect();
    System.out.println("Comparison: "+s1.discreteCompareTo(s2));
  }

  static private void delay(long milliseconds) {
    try {
      Thread.sleep(milliseconds);
    } catch (InterruptedException e) {}
  }
}
```

7. VIRI

[1] S. Južnič, *Identiteta*, Knjižna zbirka Teorija in praksa, 1993, uvod.

[2] Computer, Wikipedia.
http://en.wikipedia.org/wiki/Computer

[3] Universally Unique Identifier, Wikipedia.
http://en.wikipedia.org/wiki/UUID

[4] The Source for Java Developers, Sun Microsystems.
http://java.sun.com/

[5] Java (programming language), Wikipedia.
http://en.wikipedia.org/wiki/Java_(programming_language)

[6] B. Lesjak, T. Dobravec, "Pridobivanje določujočih sistemskih parametrov," v zborniku šestnajste mednarodne Elektrotehniške in računalniške konference ERK 2007, Portorož, Slovenija, sept. 2007, str. 50-52.

[7] MAC address, Wikipedia.
http://en.wikipedia.org/wiki/MAC_address

[8] Product activation, Wikipedia.
http://en.wikipedia.org/wiki/Product_activation

[9] Windows Product Activation, Wikipedia.
http://en.wikipedia.org/wiki/Windows_Product_Activation

[10] Java Native Access (JNA).
https://jna.dev.java.net/

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Borut Lesjak,

z vpisno številko 24011642,

sem avtor/-ica diplomskega dela z naslovom:

Ugotavljanje istovetnosti računalniških sistemov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  doc.dr.Tomaža Dobravca

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 14.9.2008 Podpis avtorja/-ice: _____