Tallinna Ülikool

Informaatika Instituut

PHP koodimisstandard PSR

Seminaritöö

Autor : Manuel Vulp

Juhendaja : Jaagup Kippar

Tallinn 2014

Sisukord

Sissejuhatus 4

1 Mis on koodimisstandard? 5

2 Miks on koodimisstandardid vajalikud? 6

3 PHP Koodimisstandard PSR 7

4 PSR-0 8

4.1 Alakriipsud nimedes 8

5 PSR-1 9

5.1 Failid 9

5.2 Nimeruumid, klasside ja meetodite nimed 9

6 PSR-2 10

6.1 Failid 10

6.2 Read 10

6.3 Tabuleerimine 10

6.4 Võtmesõnad 10

6.5 Nimeruumide asetus 10

6.6 Klasside laiendamine ja implementeerimine 11

6.6.1 Näide illustreerimaks eelnevat juttu 11

6.7 Muutujad 11

6.7.1 Näide illustreerimaks eelnevat juttu 11

6.8 Meetodid 11

6.9 Deklaratsioonid ***abstract***, ***final*** ja ***static*** 12

6.10 Meetodite ja funktsioonide kasutamine 13

6.10.1 Näide illustreerimaks eelnevat juttu 13

6.11 Juhtimisstruktuurid 13

6.11.1 Tingimuslaused *if*, *elseif* ning *else* 14

6.11.2 Kontrollstruktuur *case* ja *switch* 14

6.11.3 Kontrollstruktuur *while* ja *do while* 15
6.11.4 Kontrollstruktuur *for* ja *foreach* 15
6.11.5 Kontrollstruktuur *try* ja *catch* 15
6.12 Sulgemised (*Closure*) 16
7 Koodimisstandardi rakendamine reaalses projektis. 17
7.1 PSR-0 kasutatud osad 17
7.2 PSR-1 kasutatud osad 18
7.3 PSR-2 kasutatud osad 18
7.4 Analüüs tarkvara „Varsaaru“ kohta PSR põhjal 21
7.4.1 Koodi kirjutamise järjepidevus 22
Kokkuvõte 23
Kasutatud kirjandus 24

Sissejuhatus

Üle 50 aasta on programmeerijad kirjutanud koodi arvutisse. Arvestades praeguse ajastu kiiret tehnikaarengu staadiumit on programmeerimiskeelte koguarvu suhteliselt võimatu täpselt määrata. Kindel võib olla, et neid leidub tuhandeid. Igal keelel on mingi omapära ja olenevalt kas ajaloolistel või praktilistel põhjustel on ka tekkinud paljudele neist koodimisstandardid.

Käesoleva seminaritöö eesmärk on tutvustada inimestele programmeerimiskeele PHP koodimisstandardit PSR (PHP Standard Recommendations). Lisaks juhtida tähelepanu ka sellele, et kood peaks igas keeles teatud malli järgima ehk näeks ühesugune välja. Töö lugemisel tasub ka mõelda selle peale, et sama põhimõttega mallid eksisteerivad ka teiste keelte jaoks, mida tasub uurida. Põhjusi, miks järgida keele standardeid on mitmeid ning neid on ka kirjeldatud selles töös.

Käesoleva teema valimise põhjus on üheselt määratletav – kuna ülikooli õppekavas ei ole ette nähtud PHP õppimises koodimisstandardi tutvustamist ja selle järgimist nõutud, oleks inimesel, kes on kas segaduses või omab kahtlusi, kuidas PHP koodi visuaalselt kirjutada, vastust siit tööst lihtne leida. Suure osa programmeerimiskeele õppimisest moodustab iseseisev õppimine. Tihtipeale tuleb informatsioon pigem internetist kui kusagilt õpikust, kus pea iga erinev vastus on omamoodi kirjutatud ning see tõstatab küsimuse – milline neist on õige? Eriti algajale on sellele vastust vägagi keeruline leida, sest millistele allikatele põhineda, kui pole ette antud, mis on õige. Siinkohal tuleb ka kindlasti ära mainida, et paljudes keeltes on ka mitmeid koodimisstandardeid (nagu ka PHP).

1 Mis on koodimisstandard?

Koodimisstandardiks võib pidada vastava programmeerimiskeele kokkulepitud suunitlusi (mida nimetatakse ka „heaks praktikaks“), kuidas kood peaks välja nägema. Visuaalselt ennekõike. Selle alla kuuluvad enamjaolt :

- Kommentaarid – visuaalne kuju
- Tühikute kasutamine – kas kasutada tühikut või sakki ning mitu sümbolit see peab võtma
- Muutujate deklaratsioonid – visuaalne kuju
- Klasside ja meetodite deklaratsioonid – visuaalne kuju
- Tingimuslaused – visuaalne kuju
- Programmeerimise põhimõtted

2 Miks on koodimisstandardid vajalikud?

Kui alustatakse programmeerimist, kujuneb tihtipeale välja mingi kindel viis koodi kirjutamiseks, mida hakatakse järgima. Alguses tehakse projekti tihtipeale üksi mistõttu ei tundugi koodimisstandard nii vajalik olema. Aga mis saab siis kui projektid suurenevad ja nende kallal hakkavad töötama mitmed inimesed? Väga harva valdab ühe projekti terve elutsükli ajal seda ainult üks inimene või meeskond. See on paratamatus, et liikmed vahetuvad või tuleb inimesi juurde projekti suurenemise tõttu. Tihtipeale umbes 50% või rohkem projekti maksumusest moodustub koodi haldamine ja selle edasi arendamine peale algset arendust. Kui meeskonnaga liitub aga võõras liige peab ta koodiga tutvuma. Kui kood pole aga kirjutatud mingisugusel standardsel viisil võib sellega tutvumine võtta omajagu aega olenevalt inimesest. Teiseks on tihtipeale raske harjuda ka ümber oma hetkel oleva mugavustsoonis välja kujunenud kirjutamisviisist. Kui aga projekti on algusest peale arendatud kindlale koodimisstandardile on koodi haldamine märgatavalt lihtsam; seda nii haldamise kui ka loetavuse (arusaamise) eesmärkidel. Näiteks kui mingi rakendus on vabavaraline siis selle kirjutama eeldus on see, et kood on puhtalt, arusaadavalt ning loetavalt kirjutatud põhjusega, et teised inimesed, kes pole eales autoriga koos töötanud ega tea arendusest midagi, oleksid ikkagi suutelised koodi lugema ning sellest aru saama. (Glass, 2002) (Grubb, 2003)

3 PHP Koodimisstandard PSR

Käesoleva töö uuritavas koodimisstandardil on 3 osa – 0, 1, 2, mis kõik põhinevad üksteisel ning ei ole mõeldud eraldi väljavõetult kasutatavana. Põhjus, miks PSR on populaarsust koguv ning väärt järgimist, on selle eestseisjad. Viis, kuidas need standardid on väljakujunenud, ei ole lihtsalt kellegi otsus, et nii peab olema, vaid hääletusmeetodil viidud läbi küsitlused iga aspekti kohta kodeerimisviisis. Hääletajateks on aga käesoleva PHP ajastu osade oluliste projektide juhid või juhtprogrammeerijad (ligikaudu 30). (PHP Framework Interop Group, 2013)

Ülevaade PSR standarditest :

- PSR-0 – automaatlaadimise standardid
- PSR-1 – koodimisstandardi baas
- PSR-2 – koodi stiili juhend

4 PSR-0

Keskendub automaatlaadimisele. Järgnevas tekstis kasutatav sõna „*projekt*“ viitab arendatava projekti struktuursele asukohale. Korrektne struktuur nimeruumide klasside laadimisele koosneb järgnevatest komponentidest :

- \Väljaandja\
- Nimeruum\
- Klassi nimi

Tähtis on, et nimeruumi kuju vastab ka reaalsele rakenduse struktuurile ning, et arendatavas rakenduses asuks järgneval kujul struktuur, kuhu imporditavad klassid laadida :

projekt\lib\vendor

Näide :

Järgnev nimeruum

\Väljaandja\Alamkataloog\Alamkataloog\Klass

peab vastama struktuurile

projekt\lib\vendor\Väljaandja\Alamakataloog\Alamkataloog\Klass.php

4.1 Alakriipsud nimedes

Alakriipsud teisendamine struktuuri asukohaks PSR kohaselt, jääb kuju samaks välja arvatud viimane alakriipsuga tähistatud osa klassi- või nimeruumist – selles tuleb asendada kõik alakriipsud tagurpidise kaldkriipsuga.

Näide :

Järgnev klassi- või nimeruum

\Väljaandja\alam_kataloog\klass

Peab vastama struktuurile

projekt\lib\vendor\Väljaandja\alam_kataloog\Klass.php

5 PSR-1

Keskendub koodi standardelementidele ja standardkoodi mähistele. PSR-1 järgib PSR-0 olevaid kodeerimisstandardeid.

5.1 Failid

Faili alguses asuvad PHP märgendid peavad olema kas ühel kujul kahest : kas <?php või <?=.

Failide kodeering PHP koodi jaoks peab olema UTF-8 **ilma** baidi märgiseta (BOM). (Ishida, 2013)

5.2 Nimeruumid, klasside ja meetodite nimed

Klasside nimed peavad olema kokku kirjutatud sõnad, mis algavad suurte tähtedega

Näide : *KlassiNimi*

Juhul kui tegu on varasema PHP versiooniga kui 5.3.0, mis ei toeta nimeruume, tuleb klassi nimi defineerida läbiva suurtähega ning alakriipsudega :

Näide : *Autor_Klassi_Nimi*

Klassides asuvad konstandid peavad olema läbivsuurtähega ning alakriipsuga :

Näide : *const KONSTANDI_NIMI = 'väärtus';*

Meetodite nimed peavad olema kokku kirjutatud sõnad läbiva suurtähega välja arvatud esimene sõna, mis kirjutatakse väikse tähega :

Näide : *public function meetodiNimi() {}*

6 PSR-2

Keskendub koodi välimusele ning sisaldab ka muutujate defineerimist. PSR-2 järgib PSR-1 olevaid kodeerimisstandardeid.

6.1 Failid

Iga rida peab lõppema Unix LF (U+000A) stiilis sümboliga.

Iga PHP faili lõpus peab olema reavahetus.

Kui fail sisaldab ainult PHP koodi, ei tohi koodi lõpus olla sulgevat sümbolit ?>

6.2 Read

Ridade pikkus ei tohi olla limiteeritud, kuid maksimaalne piir peaks olema kuni 120 tähemärki. Reaalselt peaks rea pikkus olema kuni 80 tähemärki ning seda ületavad read tuleks jagada mitmele reale, kus ühegi pikkus ei ületa 80 sümbolit.

Rea lõppudes ei tohi olla tühikuid.

Tühjasid ridasid võib lisada koodi, kui eesmärk on koodiblokkide eraldamine lugemise lihtsuse eesmärkidel.

Ühelgi real ei tohi olla üle ühe avaldise.

6.3 Tabuleerimine

Tabuleerimiseks ei tohi kasutada *tab* sümbolit, vaid peab tabuleerima 4 tühiku kaupa. Põhjus on selles, tabulaatori sümbol on eraldi tõlgendatav vastavalt süsteemi või programmi konfiguratsioonile mistõttu ei pruugi failid olla üheselt loetavad.

6.4 Võtmesõnad

PHP võtmesõnad, sealhulgas ka sõnad *true*, *false* ja *null*, peavad olema läbiva väikse tähega. Nimekirja PHP võtmesõnadest võib leida nende enda ametlikult kodulehelt : http://php.net/manual/en/reserved.keywords.php

6.5 Nimeruumide asetus

PHP faili alguses peavad olema kõigepealt defineeritud nimeruumid. Pärast nimeruume tuleb lisada tühiku ja seejärel klasside kasutamise deklaratsioonid. Seejärel veel üks tühik ning siis kood.

6.6 Klasside laiendamine ja implementeerimine

Laienduse ja implementeerimise definitsioonid peavad asuma samal real klassi nimega. Klassi ümbritsevad loogelised sulud peavad olema üksi eraldi ridadel nii, et koodi ja sulgude vahel pole tühikuid. Juhul kui implementatsioone on nii mitmeid, et lubatud rea pikkuses läheb sümbolite arv üle, võib need defineerida järgnevatele ridadele nii, et iga rea peal on üks implementatsioon. Sealhulgas ka kõige esimene implementatsioon!

6.6.1 Näide illustreerimaks eelnevat juttu

class Klass *extends* Laiendus *implements*

\Esimene,

\Teine

{

PHP kood

}

6.7 Muutujad

Muutujate kuju peab olema sõnad kokku kirjutatud, läbiva suurtähega välja arvatud esimene sõna. Sõna *var* ei tohi kasutada muutujate deklareerimiseks. Muutujate avalikkus peab olema defineeritud selleks mõeldud sõnadega, mitte alakriipsuga.

6.7.1 Näide illustreerimaks eelnevat juttu

class Klass

{

~~*var $muutujaNimi = 'väärtus';*~~

private $muutujaNimi = 'väärtus';

}

6.8 Meetodid

Meetodite definitsioon on sarnane muutujate omaga: kirjutatakse läbiva suurtähega välja arvatud esimene sõna ning muutujate avalikkust peab olema defineeritud ning selleks mõeldud sõnadega, mitte alakriipsuga. Pärast meetodi nime ei tohi olla tühikut. Meetodi parameetrid peavad olema eraldatud komadega ning pärast koma peab olema üks tühik. Rea sümboli liimiti ületavate parameetrite korral tuleb iga parameeter panna järgmisele reale ning sama rea lõppu

ka koma. Üldjuhul tuleb panna meetodi loogelised sulud eraldi ridadele üksi, kuid kui meetodi parameetrite kogus on tähemärkides pikem, kui lubatud rea pikkus, siis lähevad parameetrid eraldi ridadele ning parameetrite sulgev sulg ning loogeline esimene loogeline sulg on samal real.

Näide illustreerimaks eelnevat juttu

```
public function meetodiNimi($muutuja)
{
   // PHP kood
}
```

Või

```
public function meetodiNimi(
        $muutuja,
        $muutuja2,
        $muutuja3
){
   // PHP kood
}
```

6.9 Deklaratsioonid ***abstract***, ***final*** ja ***static***

Eelnimetatud deklaratsioonide asukohtade puhul kehtivad järgmised reeglid :

Muutujate nimede puhul peab kõigepealt olema defineeritud avalikkus, seejärel staatilisus (kui eksisteerib)

Näide *: public static $muutuja = 'väärtus';*

Abstraktsete meetodite puhul peab kõigepealt olema defineeritud abstraktsus ning seejärel avalikkus.

Näide : *abstract protected function meetod();*

Lõplike meetodite puhul peab kõigepealt olema defineeritud lõplikkus, seejärel avalikkus ning lõpuks staatilisus.

Näide :

final public static function meetod()

{

}

6.10 Meetodite ja funktsioonide kasutamine

Peamine asi, mida silmas pidada on see, et tühikuid ei tohi olla kuskil peale argumentide eraldamiseks mõeldud tühikud. Juhul kui argumente on nii palju, et realimiit on ületatud, tuleb iga argument kirjutada eraldi reale.

6.10.1 Näide illustreerimaks eelnevat juttu

```
esimene();

$teine->kolmas();

Neljas::viies($kuues, $seitsmes);

Kaheksas::üheksas(
        $kümnes,
        $üheteistkümnes,
        $kaheteistkümnes
);
```

6.11 Juhtimisstruktuurid

Juhtimisstruktuuride puhul peab silmad pidama järgmisi reegleid :

- Pärast iga juhtsõna peab olema tühik
- Argumendid defineeritakse komadega. Pärast iga koma peab olema tühik ning enne ega pärast ümarsulge ei tohi olla tühikuid.
- Avav loogeline sulg peab olema samal real eelneva tühikuga.
- Juhtstruktuuri sisu peab olema järgmisel real.
- Iga koodiplokk peab olema taandatud 4 tühiku võrra.
- Juhul kui tegu on *elseif*-iga siis tuleb eelneva *if*-i sulgev sulg olema samal real, seejärel tühik, seejärel *elseif* millele järgneb tühik ning uus avanev loogeline sulg.
- Sulgev loogeline sulg peab olema järgmisel real koodi blokist.

6.11.1 Tingimuslaused *if, elseif* ning *else*

Loetavuse eesmärkidel tuleb *elseif* kokku kirjutada. Kõik tingimussõnad võiksid välja näha ühe sõnana.

Näide :

```
if  ($tõene) {
   // PHP kood
} elseif  ($väär) {
   // PHP kood
} else {
   // PHP kood
}
```

6.11.2 Kontrollstruktuur *case* ja *switch*

Tingimussõna *case* kirjutatakse järgmisele reale *switch*-ist ning tabuleeritakse 4 tühiku võrra. *Case* ja argumendi vahel on tühik ning pärast argumenti tuleb ilma tühikuta koolon. *Case* sisu kirjutatakse järgmisele reale ning tabuleeritakse eelnevast 4 tühiku võrra edasi. Case tuleb alati vaikimisi lõpetada sõnaga *break* või *return* ning juhul kui seda taotuslikult ei ole, tuleb eraldi kommenteerida, et *break*-i ei tohi vastavasse kohta tulla.

Näide :

```
switch ($tingimus) {
   case $esimene:
      // PHP kood
      break;
   case $teine:
      // PHP kood
      return;
   Case $kolmas
```

```
    // tingimust ei tohi lõpetada
}
```

6.11.3 Kontrollstruktuur *while* ja *do while*

Struktuursõna *while* kood kirjutatakse järgmisele reale avasulust ning tabuleeritakse 4 tühikut. kui tegu on tegu *do while*-iga, on ülesehitus sama nagu *while*-ga. Ainult *while* tuleb kirjutada seda lõpetava suluga samale reale.

Näide :

```
do {
   // PHP kood
} while ($tingimus);
```

6.11.4 Kontrollstruktuur *for* ja *foreach*

Struktuursõna for ja avasulg kirjutatakse samale reale. Koodiblokk kirjutatakse järgmisele reale.

Näide :

```
for ($i = 0; $i < 5; $++) {
   // PHP kood
}
foreach ($massiiv as $võti => $väärtus) {
   // PHP kood
}
```

6.11.5 Kontrollstruktuur *try* ja *catch*

Struktuursõnad, tingimused ,avasulg ja lõpusulg kirjutatakse samale reale.

Näide :

```
try {
   // PHP kood
} catch (Viga $v) {
```

```
    // PHP kood
}
```

6.12 Sulgemised (*Closure*)

Pärast muutuja defineerimist tuleb panna sõna *function* ja avasulgude vahele tühik. Juhul kui kasutatakse argumentide ümbernimetamist, tuleb ka see (sõna *use*) eraldada tühikutega. Juhul kui argumente on rohkem, kui realimiit ette näeb, tuleb kirjutada nad järgmistele ridadele tabuleeritud 4 tühikuga. Kehtib mõlema puhul (*function* ja *use*).

Näide:

```
$muutuja = function (
    $muutujaÜks,
    $muutujaKaks,
    $muutujaKolm
) use (
    $esimene,
    $teine,
    $kolmas
) {
    // PHP kood
};
```

7 Koodimisstandardi rakendamine reaalses projektis.

Järgnev näide on toodud tarkvara „Varsaaru“ arendamisest. Tegu on koolipraktikal arendatud tarkvaraga, mis võeti vastu suvel 2012.

Kuigi koodimisstandardi järgimine võib tunduda lihtsa tegevusena, ei ole see paraku sedamoodi üheselt teostatav ilma komplikatsioonideta. Muidugi „Varsaaru“ arendamisel tuleb ka silmas pidada seda, et tegu oli koolipraktikal arendatava projektiga, kus arendajad ei olnud arvestata kogemusega, mis lihtsustab märgatavalt arendamise sujuvust. Sellest hoolimata tuleb saab välja tuua paralleele, mis kehtivad nii kogenenud arendajate kui ka algajate puhul. Esiteks hõlmab koodimisstandard suuremat osa ettetulevatest juhtumistest stiililiselt. Need varieeruvad rohkelt ning iga koodirea peale ei minda koheselt uurima stiili käsiraamatust, kas rida sai õigesti kirjutatud. Kui osadel juhtudel ei tule isegi selle peale, et see rida võiks olla stiliseeritud erinevalt kui seda enda mõte ette nägi siis teistel juhtudel jällegi saab laiskus või aja puudumine jagu tegelikust vajadusest vaadata stiliseerimisjuhendit.

Veel üheks probleemiks said ka harjumused. Kõige kogenuma rühmaliikme eeskujul tõime me sisse ka osasid, mis olid PSR-st erinevad. Isegi kui meie arendushetkel ei tekkinud sellega probleeme siis pärast mõne kuulist pausi, kui oli vaja haldamis- ja parandustöid tegema, tekitasid need rohkelt peavalu. Tekkis palju küsimusi, miks üks või teine asi niimoodi tehtud on. Nii PSR kui ka teiste koodi stiliseerimisjuhendite eesmärk ei ole mitte ainult õpetada programmeerijatele kirjutama puhtamat ning loetavamat koodi vaid ka kirjutama sellist koodi, mida on lihtne hallata. Kui peaks tekkima küsimusi, kuidas või miks mõni rida on just sellisena kirjutatud siis saab sellele vastuse vastavast koodijuhendi käsiraamatust. Nii on ka väga lihtne kaasata uusi inimesi projekti – öelda neile, millist juhendit kasutati, lasta see läbi lugeda ning sellest piisabki, et uue liikme kirjutatud kood näeks samasugune välja nagu terve ülejäänud rakenduse koodibaas.

Nagu eelnevalt mainitud, ei võetud projektis „Varsaaru“ kõiki PSR kodeerimisjuhendi soovitusi kasutusele. Järgnevas osas on välja toodud detailselt, mida kasutati ning mida ei kasutatud vastavatest PSR alamosadest.

7.1 PSR-0 kasutatud osad

Meie projektis kasutatud PHP raamistikuks oli võetud FuelPHP, mille automaatlaadimise klass kasutab PSR sätestatud standardeid.

Näide FuelPHP enda klassist :

Asukoht: vendor/Fuel/Upload/Upload.php

Nimeruum: \Fuel\Upload\Upload

Näide meie lisatud klassist:

Asukoht: vendor/Smarty/Smarty.php

Nimeruum: \Smarty\Smarty

7.2 PSR-1 kasutatud osad

Selles osas tuli sisse esimene suurem erinevus võrreldes PSR standardiga ning meie kirjutatud koodiga. Meil oli kasutatud meetodite defineerimisel alakriipsuga eraldatud sõnu meetodite nimedes kuigi PSR näeb ette meetodite nime kirjutamist läbiva suurtähega välja arvatud esimene sõna, mis kirjutatakse väikse tähega.

PSR*: actionIndex*

Varsaaru*: action_index*

Kuna meil endal oli harjumus PHP koodi sellise metoodikaga kirjutada, siis tundus loogiline ka nii jätkata. Muid erinevusi koodis PSR alampunktiga ei olnud.

7.3 PSR-2 kasutatud osad

Kõige rohkem erinevusi tuli sisse PSR-2 stiilijuhendi soovitustega. Järgnevas nimekirjas on täpsemalt kirjutatud lahti, mida kasutasime erinevalt:

PSR: Reapikkus ei tohiks ületada 80 tähemärki ning limiit peaks olema 120 tähemärki.
Varsaaru: Leidus rohkelt ridu, mille tähemärkide kogus ületas 80 tähemärki. Leidus isegi ridu, mille sümbolite kogus ületas 120 tähemärki. Põhjenduseks võib ilmselt tuua ekraani resolutsiooni suuruse. Kuna hea resolutsiooni korral mahuvad 120+ tähemärgi pikkused read ära siis ei tundunud selle järgimine nii vajalik olema.

PSR: Tabuleerimiseks tuleb kasutada 4 tühikut, mitte *tab* sümbolit.
Varsaaru: Püsivus projektis puudus: failides oli segamini kasutatud *tab* sümbolit ja 4 tühikut. Kohati oli isegi *tab* sümboli pikkuseks 2 tähemärki, suuremalt jaolt siiski 4. Põhjuseks oli erinevate redigeerimistööriistade kasutamine. Osadel neist on vaikimisi määratud *tab* pikkuseks 2 tähemärki. Vahel seadistasime jälle ise ära 4 tähemärgi pikkuse peale. Samas kui failis oli

eelnevalt juba alustatud näiteks 2 tähemärgi pikkuse tabulaatoriga, siis jätkasime ka juba kasutusel olevat kirjutusviisi.

PSR: PHP konstandid *true, false* ja *null* peavad olema kirjutatud väikeste tähtedega.
Varsaaru: Konstandid *true, false* ning *null* olid kirjutatud koodis läbisegamini kord suurte tähtedega, kord väikestega. Näide, kuidas eelnevalt oli harjumus sisse jäänud kirjutada vastaval viisil ning nii see kandus ka edasi projekti koodibaasi. Erinevatel programmeerijatel on erinevad harjumused ja kui keegi ühest stiili välja ei paku, kirjutatakse koodi mõlemal viisil.

PSR: Meetodi parameetride vahel peab olema pärast iga koma tühik.
Varsaaru: Soovitust sai kasutatud kohati. Mitmetes kohtades ei olnud parameetrite vahel tühikuid. PSR soovituslik viis polnud harjumuseks saanud ning sai läbisegi kirjutatud mõlemat pidi.

PSR: Kontrollstruktuur *if*, *elseif*, *else* soovituslik väljanägemine.
Varsaaru: Näidisstruktuur:

```
if($exists)
{
  // Koodiblokk
}
else
{
  // Koodiblokk
}
```

Nagu näha on eksitud siin kõikvõimalike PSR reeglite vastu:

- Pärast *if* peaks olema tühik, mida meil kuskil koodis kasutusel ei ole.
- Kontrollstruktuuri lõpetavad ja alustavad sulud peaksid olema samal real kontrollstruktuuri tingimusega.

Põhjus, miks kontrollstruktuur selline „Varsaaru“ koodibaasis välja nägi tuleneb sellest, et kuna meetodite kirjutamisel tuleb loogelised sulud eraldi ridadele panna ning meetodi nime ja parameetrite vahel ei ole tühikut, siis tundus loogiline järgida sama ülesehitust kirjutamisel.

PSR: Kontrollstruktuur *switch* soovituslik väljanägemine

Varsaaru: Näidisstruktuur:

```
switch($type)
{
  case 'archive':
    // Koodiplokk
    break;

  case 'unarchive':
    // Koodiplokk
    break;
}
```

Siin on stiili kohapealt eksitud samamoodi nagu ka *if* kontrollstruktuuris: tühik puudub *switch* ja parameetri vahel ning loogelised sulud on eraldi ridadel. Teiseks on iga *case* vahel reavahetus. Kuigi PSR-s on sätestatud, et reavahetuse võib panna kohtadesse, kus oleks vaja mingi loogilise bloki eristavust välja tuua, ei oleks see siinkohal juba sellepärast vajalik, et iga *case* ongi erinev loogiline osa ning on selgelt lõpetatud *break*-iga (või teistel juhtudel *returniga*).

7.4 Analüüs tarkvara „Varsaaru“ kohta PSR põhjal

Nagu näha, siis algselt planeeritud koodimisstandardi kasutuselevõtust ei tulnud midagi väga suurt välja. Peamiseks probleemiks saigi kas laiskus või eelnevad harjumused. Välja võib tuua ka järelduste tegemise eelnevate kogemuste põhjal. Näiteks kontrollstruktuur *if* puhul, kui sai see kirjutatud samamoodi nagu ka meetodeid kirjutatakse puhtalt oletuste põhjal, et nii võiks loogiline olla.

Vahepeal tekkisid tarkvara „Varsaaru“ arendamisesse ka pikemad pausid, mis on analüüsi mõttes hea, sest siis saab selgelt välja tuua tarkvara elutsükli erinevad etapid. Kuidas toimus koodi haldamine, mis ei olnud arendatud PSR standardi järgi. Arenduse hetkel ei tundunud koodimisstandardist kõrvalekaldumine olema mitte midagi suurt või märgatavat, sest tarkvara oli pidevas arendamises ning sellega sai kokku puututud peaaegu iga päev. Küll aga pärast mitmeid pikemaid pause muutus eelneva kirjutatust arusaamine järjest keerulisemaks. Programmi elutsükli jooksul tuli kaks suuremat pausi. Kummagi jooksul ei tehtud koodi parandamist ning sai arendatud edasi vigast koodi. Seda võib juba pidada esimeseks suuremaks veaks, mis tehtud sai. Kuna arendatud tarkvara ei olnud nii mahukas, oleks tulnud terve enda kirjutatud koodibaas ümber kirjutada korrektsele kujule. See oleks ettenägelikult olnud õige tegu järgnevate haldus- või arendustsüklite jaoks. Koodi ümberkirjutamist aga ei tehtud nii et järgnevad haldus- ja arendustööd süvendasid juba eelnevalt halvasti stiliseeritud koodi. Mida kauem tarkvara sai arendatud ja mitmeid pause oli möödas, hakkas koodi haldamine juba häirivaks muutuma. Peamine põhjus oli selles, et samal ajal toimub konstantne areng programmeerijatel: nii koodi kirjutamisoskus kui ka korrektselt stiliseeritud koodi kirjutamine muutuvad fundamentaalseteks osadeks igapäeva arenduses ning harjumus konstruktiivselt kritiseerida või üritada parandada vigast koodi muutub aktuaalseks. Seda ennekõike sellepärast, et olles aktiivne programmeerija, on terve „Varsaaru“ elutsükli jooksul valmis tehtud rohkelt projekte ning iga uue projektiga tuleb soov ennast parandada ning lisaks ka eelnevaid vigu mitte korrata. Kuna arendustsükkel „Varsaarul“ puhul oli palju pikem kui sellele tegelikult oleks pidanud olema, siis olid vead arenduse lõpu poole järjest rohkem esilekerkivad.

Kui projekti peaks veel hiljem keegi teine inimene või meeskond edasi arendama siis tekiks kindlasti ka neil väga rohkelt küsimusi, miks üks või teine asi vastavalt kirjutatud on.

7.4.1 Koodi kirjutamise järjepidevus

Kui koodimisstandardit poleks järgitud projekti (mida võib öelda, et ka edukalt ei tehtud) arendamisel, oleks tulnud olla järjepidev enda järgitud koodi stiili kasutamises. Ka PSR-s on sätestatud, et juhul kui arendatav või hallatav kood ei näe välja nii nagu PSR-s on öeldud, siis ei pea hakkama üritama tervet koodibaasi ümber kirjutama sellele kujule nagu PSR ette näeb, vaid tutvuda kirjapandud stiiliga ning seda jätkata.

„Varsaaru“ arendamise selge probleem oli see, et puudus konstantsust. Nagu ka detailanalüüsi mitmetest punktidest võis välja lugeda, sai kirjutatud ühte asja mitmel eri viisil. Kuigi me kaldusime PSR standardist kõrvale, ei oleks tulemus üldsegi projekti arendamisele nii negatiivset mõju avaldanud kui me oleks järginud mingit konkreetsed stiilijuhendit kirjutamisel. See koodijuhend ei oleks ilmtingimata pidanud isegi eksisteerima laiemas plaanis vaid kas või antud projekti raames. Selle oleks pidanud tarkvara dokumentatsiooni kirja panema, millise stiiliga kood on kirjutatud, et teaks kuidas koodi ka edasi kirjutada.

Kokkuvõte

Kuigi pealtnäha võib kood tihtipeale suhteliselt sarnane välja näha, on detaile siiski rohkelt nagu ka käesolevas töös näha on. Ühe koodimisstandardi järgimise ära harjumisega võib minna tükk aega, sest iga tühiku asukoht on tähtis. Isegi kui alguses tundub see liigse pedantsusena, on sellel tähtis tagamõte – tagada koodi hea hoolduse võimalus ning lihtsam loetavus. Kui projekt saab suuremaks ning koodi kirjutatakse rohkem, ei saa klient ega ka programmeerija lubada endale koodi halva stiili pärast kohanemiseks rohkem aega. Koodi haldamine on tihtipeale niigi keeruline ülesanne, ei ole mingit põhjust seda veel keerulisemaks teha. Siinkohal tasub ka mainida, et kuigi käesolev töö on PHP koodimisstandardi kohta, on igal keelel oma levinum(ad) koodimisstandard(id), mida tuleks järgida.

Kasutatud kirjandus

(4. Oktoober 2013. a.). Allikas: PHP Framework Interop Group: http://www.php-fig.org/

Autoloading Standard. (25. September 2013. a.). Allikas: PHP Framework Interop Group: http://www.php-fig.org/psr/psr-0/

Basic Coding Standard. (3. Oktoober 2 2013. a.). Allikas: PHP Framework Interop Group: http://www.php-fig.org/psr/psr-1/

Coding Style Guide. (8. Oktoober 2013. a.). Allikas: PHP Framework Interop Group: http://www.php-fig.org/psr/psr-2/

Glass, R. (2002). *Facts and Fallacies of Software Engineering.*

Grubb, P. (2003). *Software Maintenance: Concepts and Practice.* World Scientific Publishing Company.

Ishida, R. (13. Jaanuar 2013. a.). *The byte-order mark (BOM) in HTML*. Kasutamise kuupäev: 18. Oktoober 2013. a., allikas W3: http://www.w3.org/International/questions/qa-byte-order-mark

Java Code Conventions. (1997). California.