Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Dragar

Samodejno tvorjenje programske kode za izpis obrazcev

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Kljub želji po brezpapirnemu poslovanju še vedno prejemamo velike količine papirnatih obrazcev, ki jih morajo različne službe pripravljati. Priprava obrazcev je lahko zamuden postopek, ki za nameček uporablja še veliko količino virov, kot sta procesorski čas in pomnilniški prostor. V diplomski nalogi proučite možnost izboljšanega samodejnega tvorjenja obrazcev, pri čemer ovrednotite količino porabljenih virov - procesorski čas, pomnilniški prostor in tudi stroškovno vrednost uporabljene programske opreme. Predlagajte svojo rešitev, ki jo tudi implementirajte.

Zahvalil bi se svojemu mentorju Andreju Brodniku, ki mi je pomagal pri izbiri teme in izdelavi diplomske naloge ter sodelavcem, ki so mi pomagali z zasnovo postopka. Zahvalil bi se tudi svojim staršem in partnerici za dolgoletno podporo med študijem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
XML	Extensible markup language	Razširljiv označevalni jezik
SQL	Structured query language	Strukturirani povpraševalni jezik
XSD	XML Schema Definition	Sheme dokumenta XML
PDF	Portable Document Format	Prenosni format dokumentov

Povzetek

Naloga vsebuje opis izdelave, postopek ovrednotenja in rezultate aplikacije, ki združuje dve metodi priprave dokumentov PDF v jeziku C# (*DevExpress XtraReports* in *iTextSharp*). Cilj naloge je aplikacija, ki omogoča preprosto izdelavo predloge za pripravo velikega števila obrazcev.

Z uporabo vsake metode posebej smo izdelali aplikaciji, ki pripravita enak testni obrazec v obliki PDF. S pridobljenim razumevanjem delovanja obeh orodij smo izdelali okvirni diagram poteka, ki za proces izdelave uporabi obe metodi hkrati. Iz diagrama je postopoma nastal prototip aplikacije, ki smo ga izboljšali in dopolnili do končnega izdelka. Na koncu je sledilo ovrednotenje in primerjanje z obema metodama, ki smo jih uporabili za izdelavo. Merili smo velikost, ki jo dokument zasede na disku (KB), in čas priprave testnega nabora dokumentov.

Končni izdelek je dosegel približno 3-krat manjšo velikost dokumenta in med 2- in 3-krat krajši čas za pripravo testnega nabora obrazcev.

Ključne besede: tvorjenje dokumentov, PDF, C#, iTextSharp, DevExpress.

Abstract

Thesis describes the process of developing and evaluating of an application, which combines two methods of generating PDF documents in C# programming language (iTextSharp library and DevExpress XtraReports program suite). Goal of the thesis was to develop an application, which allows us to create a template for mass document generation using a simple process.

We created two seperate applications, which prepare the same test PDF example, each using one of aformentioned methods. Based on the obtained understanding of both methods, we prepared a flow diagram, which combines iTextSharp and DevExpress tools in the same application. We developed the prototype from the flow diagram, optimized to the final form. The last step was evaluating the algorithm and compare the results to the methods we used to build our application. We messured the size of a generated document (in KB) and time to prepare the test document batch.

The final output was approximately 3 times smaller and the process was 2-3 times faster of the test batch preparation.

Keywords: document generation, PDF, documents, C#, iTextSharp, DevExpress.

Poglavje 1

Uvod

Večina nas pozna generične obrazce in dopise, ki jih dobivamo po pošti, kot so računi za internet, odločbe za davke, odmere dohodnine ipd. Večinoma so za vse prejemnike enaki, spreminjajo se samo zneski ali podatki o naslovniku. Takšne dokumente prejema večina državljanov vsake države. Glede na število prebivalcev Slovenije lahko sklepamo, da zaposleni na podjetjih ne sestavljajo vsakega obrazca ročno, ampak za to uporabljajo programska rešitev, ki sama ustrezno lepi podatke o posameznem naročniku na prazna mesta v predhodno sestavljeni predlogi.

Naročila za pripravo obrazcev so lahko v zelo velikih količinah, zato je ključnega pomena izbira pravega orodja, ki nam lahko izrazito izboljša čas priprave, ter velikost dobljenih datotek, ki se v večini primerov tudi arhivirajo. Seveda nam je v interesu obe količini ohranjati čim manjši.

Načinov za pripravo predlog je več. Za diplomsko nalogo bomo uporabili orodje *DevExpress XtraReports* in knjižnico *iTextSharp*, ki sta eni bolj razširjenih orodij za delo s PDF-dokumenti. Obe imata svoje prednosti in slabosti, ki jih bomo bolj podrobno opisali v poglavju 2.3. Pri naših izbranih orodjih se razlika odraža v enostavnosti uporabe, hitrosti priprave nabora dokumentov in velikosti nastale datoteke. Za primerjavo - povprečna velikost dokumenta z eno stranjo vsebine, ki nastane s knjižnico *iTextSharp*, je velika med 50 in 60 KB. Povprečna velikost z orodjem *DevExpress* pa je 110 KB.

Pri 100.000 dokumentih je razlika v skupni velikosti pripravljenih datotek med 4 in 5 GB.

Cilj diplomske naloge je delujoča aplikacija, ki združi delovanje zgoraj opisanih metod. Potek bo zajemal enostavno izdelavo predloge v grafičnem okolju orodja *DevExpress* in uporabo učinkovite knjižnice *iTextSharp* za hitro izdelavo ter majhno velikost končnih datotek.

V poglavju 2 opišemo motivacijo za našo aplikacijo, podporno okolje ASP.NET ter koncept in tehnologije za delo s predlogami. V sledečem poglavju 3 predstavimo našo rešitev, ki jo v poglavju 4 ovrednotimo. Na koncu podamo sklepne ugotovitve v poglavju 5.

Poglavje 2

Proces priprave predlog in dokumentov

V tem poglavju si bomo pogledali, katera orodja smo izbrali in zakaj. Spoznali bomo koncept priprave dokumentov z uporabo predlog, na testnem primeru pa bomo spoznali dve tehnologiji za delo s predlogami - orodje *DevExpress* in knjižnico *iTextSharp*.

2.1 Motivacija

Na trgu imamo širok nabor orodij, ki so namenjena pripravi dokumentov s pomočjo predloge. Veliko takšnih orodij je zaprtokodnih, kar pomeni, da jih ne moremo prilagajati po svojih potrebah. Izdelano aplikacijo pa želimo čimbolj prilagoditi za našo vrsto uporabe ter hočemo, da je izdelava predloge enostavna. V veliko podjetjih poznavalci vsebine niso programerji, kar zahteva precej komunikacije med zaposlenimi. Težavi se ognemo, če lahko predlogo izdela že poznavalec vsebine, programer pa jo le vključi v aplikacijo.

Izdani obrazci morajo biti arhivirani, zato mora biti tvorjenje učinkovito - velikost datotek mora biti majhna, saj tako pri velikem številu dokumentov lahko prihranimo veliko prostora.

Priprava velikega števila obrazcev mora biti hitra, saj so lahko roki za

izdelavo kratki, večkrat se s pripravo čaka na zadnji trenutek. Za implementacijo moramo zato izbrati ustrezna orodja in tehnologije, s katerimi bomo dosegli želene rezultate.

2.2 Podporno okolje

2.2.1 ASP.NET in C#

Pri izbiri okolja, s katerim bomo izdelali aplikacijo, imamo na voljo veliko možnosti. Izbrali smo okolje .NET in programski jezik C#, lahko pa bi aplikacijo prav tako izdelali v javi ali kakšnem drugem programskem jeziku.

C# je programski jezik, ki temelji na objektno orientiranem programiranju. Slovnica oz. sintaksa je zelo podobna programskim jezikom C, C++ in Java [7].

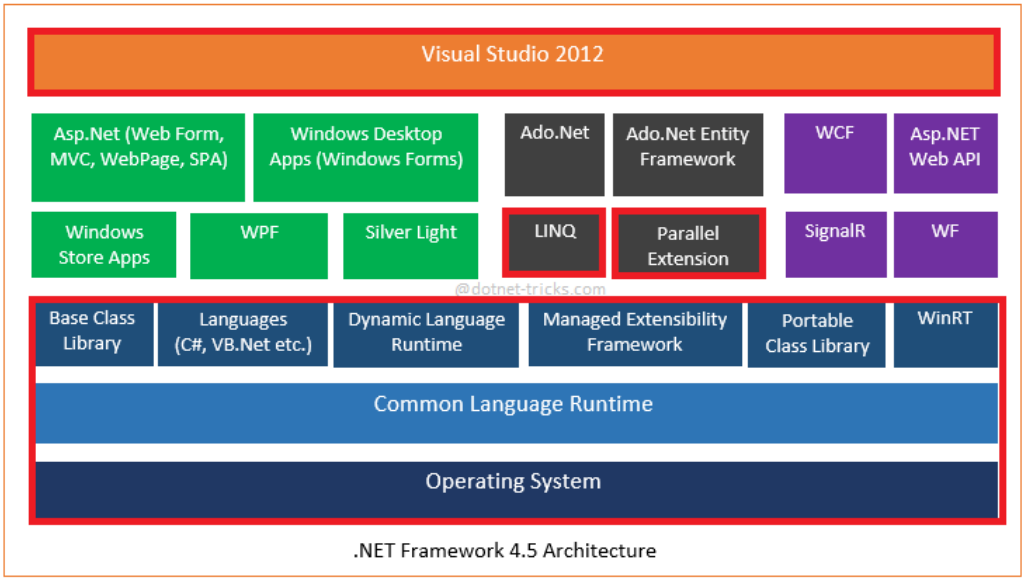
Jezik je bil zelo skrbno načrtovan na podlagi izkušenj iz razvoja drugih programskih jezikov. Lastnosti, ki so se pri ostalih jezikih izkazale za negativne, so na svoj način ustrezno dopolnili. Pomemben primer tega so generični tipi, katerih specifikacija je v C# bolj dodelana kot v javi, kar omogoča večjo varnost pri definiciji tipov, večjo preglednost in hitrejše delovanje [5] .

Uporablja izvajalsko okolje CLI, ogrodje .NET ter njegove različice, kot je Mono [12]. Razvilo ga je podjetje Microsoft v okviru razvoja ogrodja .NET. Prva različica jezika je bila izdana v januarju leta 2002 s programskim paketom Visual Studio .NET 2002 in okoljem .NET 1.0. Danes je zadnja verzija C# 7.0, .NET 4.6.2 ter paket Visual Studio 2017.

Podporno okolje *.NET framework* je ogrodje za razvoj programske opreme, ki ga lahko uporabljajo Windows in Linux operacijskimi sistemi (za delovanje na sistemin *Unix* uporabljamo okolje Mono [4]). Začetki izvirajo iz leta 2002, ko je z razvojem pričelo podjetje Microsoft, ki ga še danes vzdržuje ter izdaja nove različice. Trenutno najnovejša verzija je 4.6.2.

Okolje za delovanje uporablja več plasti. Prva plast je Visual Studio - grafično okolje, s katerim upravlja uporabnik. Koda, ki jo napišemo in

prevedemo, gre skozi prevajalnik jezika (v našem primeru C#), ki se izvaja pod nadzorom CLR (angl. *common language runtime*). To je Microsoftova implementacija okolja CLI, ki nadzoruje izvajanje in upravlja z delovnim pomnilnikom ter skrbi za upravljanje z napakami (angl. *exception handling*). Iz celotnega nabora zmožnosti, ki jih okolje ponuja, smo za izdelavo naše aplikacije uporabili jezik C# s prevajalnikom, osnoven nabor knjižnic in razredov ter gradnike, ki so na sliki 2.1 poudarjeni z rdečim okvirjem. Ogrodje smo izbrali zaradi širokega nabora že obstoječih knjižnic, ki pridejo z okoljem .NET. Vključuje že vsa potrebna orodja, ki jih bomo potrebovali za delo z datotekami XML [9].



Slika 2.1: Sestava ogrodja .NET po posameznih plasteh. (Vir: http://www.dotnettricks.com/img/netframework/.netframework4.5.png (20.8.2017))

2.2.2 XML

Extensible Markup Language (XML) je formalni jezik za opisovanje strukturiranih podatkov. Po obliki je zelo podoben jeziku HTML, saj sta oba

sestavljena iz značk (angl. *Tags*).

Za razliko od jezika HTML, XML sam po sebi nima nobene funkcionalnosti. Značke v HTML-ju imajo definiran pomen (npr. *body, div, img, href*), iz katerega spletni brskalnik, glede na pomen določene značke ustrezno sestavi podobo spletne strani ki se prikaže uporabniku. Pri sestavljanju XML dokumenta imena značk določamo sami, zaradi česar so njihova imena ponavadi tudi govoreča (iz imena lahko ugotovimo vsebino značke). Uporablja se predvsem v razvoju spletnih storitev in aplikacij kot struktura podatkov, ki se prenaša med strežnikom in uporabniki. Lahko ga tudi shranimo kot .xml datoteko za potrebo shranjevanja/izvažanja nastavitev aplikacije [6, 10, 11].

2.2.3 Pozaporedenje in razzaporedenje

V programu imamo podatke, nad katerimi delamo, shranjene v obliki spremenljivk oz. objektov. Če želimo informacijo objekta prenesti izven naše aplikacije (npr. iz strežnika uporabniku), potrebujemo obliko za zapis informacije, ki je poznana tako pošiljatelju kot prejemniku. Ena izmed oblik za strukturiranje podatkov je XML. V zaključnem izdelku smo uporabili XML strukturo za prenos podatkov o predlogi med orodjem *DevExpress* in knjižnico *iTextSharp*.

XML strukturo sestavimo s postopkom, imenovanim **pozaporedenje** (angl. *serialization*). Podatke v aplikaciji zaporedoma dodajamo strukturi v obliki značk, ki jih ustrezno poimenujemo. Obraten postopek, ko z branjem strukture oblikujemo objekt, se imenuje **razzaporedenje** (angl. *deserialization*).

V programskem okolju .NET imamo že vključene vse metode, ki so potrebne za izvedbo pozaporedenja in razzaporejanja. Nahajajo se v knjižnici *System.Xml* na razredu *XmlSerializer*. Metoda za razzaporedenje, uporabljena v naši aplikaciji, vrne objekt s podatki v enaki strukturi, kot jo ima XML struktura.

2.3 Priprava predlog

V tem razdelku bomo spoznali postopek programske priprave PDF-dokumentov s knjižnico *iTextSharp* in orodjem *DevExpress* v okolju .NET.

2.3.1 Koncept delovanja

Dokumenti, ki jih podjetja pogosto pripravljajo za vse prejemnike (npr. račun za internetne storitve), so si po vsebini zelo podobni. Oblika dokumenta in podatki, kot so pošiljatelj, obrazložitve, podpisniki in temu podobni, ostajajo enaki. Spreminjajo se samo deli obrazca, ki so specifični za naročnika, npr. ime, priimek, naslov, zneski za plačilo, podatki za obrazce UPN ipd.

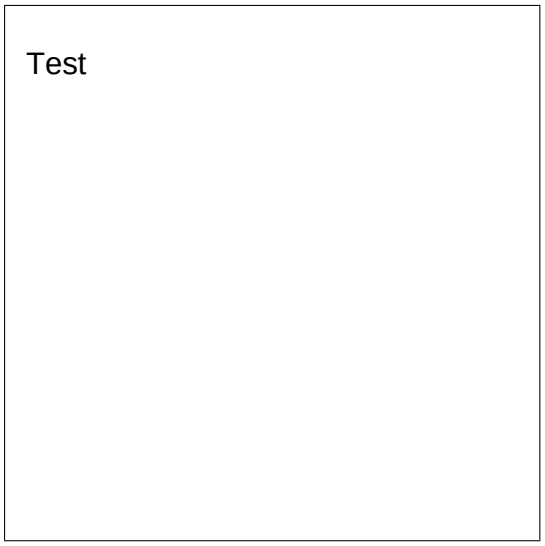
Za lažjo in hitrejšo pripravo takšnih obrazcev si izdelamo predlogo, ki bo ustrezno oblikovana in bo že zajemala vse zgoraj naštete stalne napise. Spreminjajoče podatke se ponavadi črpa iz podatkovne baze, orodju za pripravo dokumentov pa jih podamo kot seznam. Programska rešitev na koncu dokument shrani v poljubni obliki (v našem primeru PDF) in ga odloži na določeno lokacijo.

Postopek izdelave predloge obrazcev je v vsakem orodju drugačen. Pri izdelavi hibridne aplikacije smo za pripravo predloge uporabili orodje *DevExpress*, dokumente pa smo sestavljali s knjižnico *iTextSharp*. V naslednjem razdelku si bomo z uporabo obeh orodji ogledali postopek oblikovanja predloge in izdelave obrazca na preprostem testnem primeru.

2.3.2 Testni primer

Testni primer nam bo služil kot prikaz delovanja orodij *iTextSharp* in *DevExpress*.

Izdelali bomo predlogo v velikosti 300px krat 300px, ki bo v levem zgornjem kotu vsebovala napis Test s pisavo Arial. Predloga bo imela samo stalne napise, spreminjajočih podatkov v testnem primeru še ne bo in jih bomo vpeljali v kasnejših predlogah.



Slika 2.2: Izdelan obrazec testnega primera.

Z obema predlogama bomo pripravili še končni dokument (slika 2.2). Primer prikaza je seveda neuporaben za realno uporabo, nam bo pa pokazal princip, kako v obeh knjižnicah pridemo do rezultata in koliko vložka je za ta rezultat potrebnega.

2.3.3 Knjižnica *iTextSharp*

Knjižnica za programski jezik C# izvira iz orodja *iText*, ki je kot odprtokodni projekt nastal za programski jezik Java. Po nekaj verzijah kode je razvoj prevzelo novo nastalo podjetje *iText Software*, ki ponuja licenco za novejše različice, za katero dobimo vse posodobitve in uradno podporo [3].

V naši nalogi bomo uporabili majhen del knjižnice, ki je namenjen grajenju novih dokumentov. Orodje nam med drugim omogoča tudi branje, spreminjanje in izpolnjevanje že obstoječih PDF-datotek.

Oblikovanje temelji na predlogi, ki jo sestavljamo programsko s pomočjo širokega nabora gradnikov, ki jih ločujemo glede na tip vsebine, ki jo dodajamo na dokument. Gradniki so v knjižnici predstavljeni kot objekti, ki jim nastavljamo ustrezne lastnosti. Vsak ima vsebino, druge nastavitve pa se

razlikujejo glede na tip gradnika (npr. velikost, barva, stil pisave, poravnava, širina slike ipd).

Izdelavo predloge začnemo z objektom *Document* (program 2.3, vrstica 1), ki nam predstavlja glavni gradnik, kamor zlagamo vsebino (slike, besedilo). Prek njega nastavljamo tudi velikost dokumenta, odmike od robov, avtorja ipd. Vsak del besedila s skupnimi oblikovnimi lastnosti združimo v en odstavek in zapišemo v gradnik *Paragraph*, ki ga skupaj z vsemi lastnostmi (pozicija, širina, višina, tip in velikost pisave, poravnava) dodamo na dokument (program 2.3, vrstica 4). Pri dodajanju besedila moramo biti previdni z izbiro pisave. Več, kot jih na predlogi uporabimo, večja bo velikost vsake končne datoteke. Če gradniku *Paragraph* ne določimo lokacije, kjer ga želimo, se bo ta dodal na naslednje prazno mesto v dokumentu (za prejšnji element oz. v novo vrstico). Elemente lahko postavimo tudi na točno določeno pozicijo s pomočjo koordinat. Koordinatni sistem, ki ga *iTextSharp* uporablja, je enoten za cel dokument, izhodišče pa ima v levem spodnjem kotu.

```
1 Document doc = new Document(new Rectangle(200f, 200f), 5f, 5f, 5f,
    5f);
2 PdfWriter writer = PdfWriter.GetInstance(doc, new
    System.IO.FileStream("C:\Temp\test.pdf", FileMode.Create));
3 doc.Open();
4 Paragraph p = new Paragraph("Test");
5 doc.Add(p);
6 doc.Close();
```

Program 2.3: Priprava testnega primera s knjižnico *iTextSharp*

Ključne lastnosti, ki jih nastavimo gradniku *Paragraph*, so:

- ***Text*** - Besedilo, ki ga bomo prikazali na dokumentu.
- ***Alignment*** - Poravnava besedila (levo, desno, obojestransko, sredinsko).

- ***Font*** - Objekt tipa *iTextSharp.text.Font*. Vsebuje vrsto pisave (npr. Arial bold, Times New Roman regular ...), velikost pisave v točkovni enoti (angl. *Pixel*) in širino posamezne črke.

- ***UseSpacing*** - Parameter pove, ali želimo na koncu odstavka imeti razmak do naslednje vrstice ali ne.

Dokument, ki smo ga z gradniki sestavili, na koncu shranimo z uporabo objekta *PdfWriter* (program 2.3, vrstica 2), ki upravlja s fizično datoteko na disku. Dovoljuje nam ustvariti nove ali pa odpirati že obstoječe datoteke. Izdelavo zaključimo z zapiranjem dokumenta (program 2.3, vrstica 6).

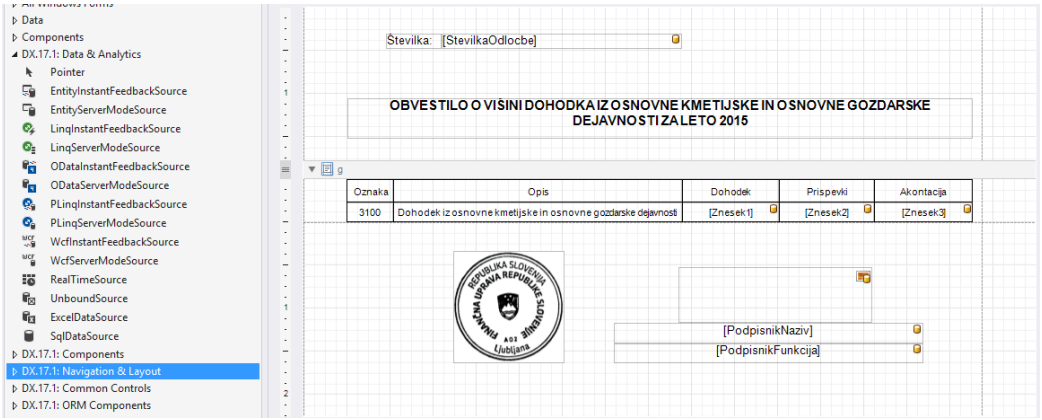
Predlogo sestavljamo torej povsem programsko. Vsebino stalnih napisov imamo lahko shranjeno v besedilnih datotekah. Vsebina ima prazna mesta, kamor vstavljamo ustrezne podatke, ki so na vsakem obrazcu drugačni. Odstavke iz besedilnih datotek dodajamo zaporedoma na dokument s pomočjo zanke. Vsaka iteracija skrbi za postavitev elementa z računanjem relativnih koordinat glede na že obstoječe elemente in polnjenje njegove vsebine. Ker se vsak obrazec sestavlja dinamično in obliko dokumenta določa količina vsebine, lahko pride do primerov, ki jih v testiranju nismo pričakovali. Iz tega razloga moramo dobro poznati vsebino in sestavo dokumentov ter zajeti čim več robnih primerov, ki lahko pri uporabi knjižnice nastanejo (npr. dolg spisek nepremičnin za davek na premoženje, kar lahko preseže dolžino ene strani).

Knjižnica *iTextSharp* ima zelo dobro optimizirano delovanje, kar pripomore k hitremu izvajanju in majhni velikosti končnih datotek. Izdelava in dopolnitve predlog pa so zahtevnejše, saj se vse izvaja programsko v kodi.

2.3.4 Orodje *DevExpress*

Orodje *DevExpress XtraReports* razvija in vzdržuje *Developer Express Inc.* Prodaja se kot licenca, ki jo je treba vsako leto obnoviti. Cena za prvo leto je $ 999, za vsako naslednje leto pa $ 499 [2].

Uporablja se za pripravo dokumentov s pomočjo predloge, ki jo izdelamo v grafičnem vmesniku, prikazanem na sliki 2.4, kjer z miško postavljamo gradnike na dokument. Za sestavljanje predloge imamo na levi strani orodja na voljo orodjarno, kjer najdemo širok nabor gradnikov, razdeljen po namenskih skupinah. Ločimo jih glede na vsebino, izbiramo lahko med gradniki za slike, besedila, tabele ipd. Vsebino jim lahko napolnimo že v predlogi (stalni podatki), lahko pa jim vrednost določamo med izvajanjem programa (spremenljivi podatki).



Slika 2.4: Grafično predstavljen dokument z gradniki (desno) in orodjarna (levo).

Največkrat uporabljeni elementi v poročilih so:

- ***XRLabel*** - Oznake se uporabljajo za dodajanje besedila na dokument. V gradnik lahko zapišemo tekst ali pa ga vežemo na atribut naše entitete ter dinamično določamo vsebino.

- ***XRPictureBox*** - Podoben elementu *XRLabel*, na dokumentu prikazuje sliko v obliki *ByteArray*. Prav tako lahko nastavljamo vsebino statično ali dinamično prek vhodnega objekta.

- ***XRTable*** - Standardna tabela, celice pa nam predstavlja množica elementov *XRLabel*. Prednost uporabe tabele prek posameznih oznak je

dinamično spreminjanje višine vrstic glede na vsebino ter lažje ohranjanje enotnosti oblike po celotni tabeli.

Predloga je sestavljena iz treh pasov - glava, telo in noga. Glava je vedno na vrhu poročila, noga na dnu, vmes pride ostala vsebina v obliki telesa dokumenta. Koordinatni sistem, ki se v tem orodju uporablja, je drugačen kot pri knjižnici *iTextSharp*. Vsak od teh pasov ima svoj koordinatni sistem, ki ima izhodišče v levem zgornjem kotu. Torej ima predloga lahko več kot eno točko T(0,0). Na sliki 2.4 so prikazani trije pasovi, torej imamo tri koordinatne sisteme z izhodiščno točko, ki jih bomo morali v naši nalogi ustrezno upoštevati.

Če želimo na poročilo dodajati podatke med izvajanjem programa, moramo pred dodajanjem poročila izdelati ustrezni podatkovni razred oz. entiteto, ki mora vsebovati vse podatke za izpolnitev obrazca. Bolj podrobno izdelavo razreda si bomo pogledali v poglavju 3.3.

Poročilo v projekt dodamo s pomočjo čarovnika, ki nam med izdelavo omogoči tudi povezavo podatkovnega razreda na predlogo (če jo bomo uporabili). S tem predlogi povemo, kakšne oblike (podatkovnega tipa) bo objekt, ki ga predloga dobi kot vir podatkov (angl. *data biniding*). Rezultat čarovnika je pripravljena prazna predloga standardnega A4-formata, ki ga lahko seveda tudi spremenimo na želeno velikost. Vsebino polnimo z besedilnimi elementi, slikami in tabelami, oblikujemo pa jo s premikanjem gradnikov na želeno mesto. Orodje nam med oblikovanjem pomaga tudi s pomočjo poravnave glede na že obstoječe elemente, pomagamo pa si lahko tudi z mrežo, ki je prikazana v ozadju, in ravnili ob robih strani. Stalne elemente z vsebino napolnimo prek lastnosti *Text* za besedilo oz. *Image* v primeru slike. Če pa želimo vsebino gradnika izpolnjevati za vsak obrazec posebej, ga dodamo na predlogo in pustimo praznega. Nastaviti pa mu je treba lastnost vira podatkov (angl. *data binding*), kjer bo dobili vsebino med izvajanjem programa. Vir nam predstavlja ustrezna lastnost podatkovnega razreda, ki smo ga povezali med kreiranjem predloge v čarovniku.

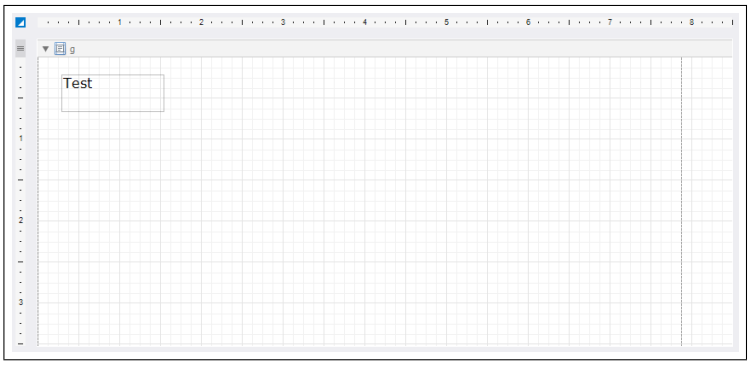
V kodi na ustreznem mestu izdelamo objekt naše predloge (program 2.5,

vrstica 1). Spreminjajočih podatkov na poročilu nimamo, zato nastavljanje podatkovnega vira ni potrebno. Če bi ga želeli nastaviti, bi uporabili lastnost *report.DataSource*. Programu moramo povedati samo še lokacijo, kamor nam dokument odloži (program 2.5, vrstica 5).

Naš testni primer smo pripravili še v orodju *DevExpress*. Na platno smo postavili besedilni gradnik in mu napolnili vsebino z besedo **Test** (slika 2.6).

```
1 TestReport report = new TestReport();
2 report.CreateDocument();
3 PdfExportOptions opts = new PdfExportOptions();
4 opts.ShowPrintDialogOnOpen = false;
5 report.ExportToPdf("E:\\test.pdf", opts);
```

Program 2.5: Priprava testnega primera z orodjem *DevExpress*



Slika 2.6: Izdelana predloga testnega primera v okolju *DevExpress*.

Predlogo, ki jo zgradimo v okolju *DevExpress*, lahko tudi izvozimo za zunanjo uporabo. Objekt *report* vsebuje metodo *ExportToXml*, ki nam sestavi strukturo XML in jo shrani na določeno lokacijo.

Dobimo strukturo, ki vsebuje podatke o predlogi z vsemi stalnimi podatki, ki smo jih definirali. Na shemi 2.7 vidimo strukturo, ki jo vrne naša predloga za testni primer. Vsi elementi so poimenovani *Item* z zaporedno številko. Značke *Item* ločimo na pasove glava (shema 2.7, vrstica 10), telo (shema 2.7,

vrstica 3) in noga (shema 2.7, vrstica 11). Prav tako so enako poimenovani gradniki za slike in besedilo (shema 2.7, vrstica 5). Tip gradnika, ki ga značka predstavlja, pa najdemo v lastnosti *ControlType*. Takšna struktura je pri pripravi razreda za razzaporedenje postopek precej otežila, zato jo bomo v poglavju 3.4 preoblikovali v obliko, ki bo lažja za obdelavo v hibridni aplikaciji.

Prednost orodja *DevExpress* je preprostost uporabe - za pripravo poročila ne potrebujemo programerskega znanja, zato lahko predlogo pripravijo poznavalci vsebine. To posledično sprosti delo programerjev in poveča natančnost poročil. Slabost je daljši čas priprave in večja velikost datoteke, ki jo zavzame na disku. Prav tako je za orodje potrebna precej draga licenca.

2.3.5 Podobnosti in razlike

Zelo hitro ugotovimo, da za uporabo orodja DevExpress skoraj ne potrebujemo znanja programiranja, zato lahko poročilo bolj natančno in hitreje pripravijo poznavalci vsebine in ne razvijalci. Veliko manj je tudi prostora za napake, saj deluje na konceptu *What You See Is What You Get*. Razvijalec v aplikacijo samo uvozi izdelano predlogo poročila ter entiteto, na ustreznem mestu uporabniškega vmesnika pa doda klic izdelave dokumentov. Slabost orodja se pokaže v daljšem času priprave in večji velikosti datotek.

Hitrosti se bomo bolj posvetili v poglavju 4, kjer izvedemo primerjavo z uporabo programskega časovnika. Razliko pa lahko vidimo že sedaj pri velikosti datotek zgoraj pripravljenih testnih primerov. V obeh testih smo pripravil enak dokument, opisan v poglavju 2.3.2. Orodje *DevExpress* je pripravilo datoteko velikosti 15,8 KB, *iTextSharp* pa zasede samo 4 KB, kar je skoraj 4-krat manj od preprostejše metode *DevExpress*.

```
<XtraReportsLayoutSerializer Ref="0"
    ControlType="Reports.TestReport" PageHeight="1100"
    PageWidth="850" Margins="0, 63, 0, 0">
<Bands>
<Item1 Ref="1" ControlType="DetailBand" Name="Telo"
    TextAlignment="TopLeft" Padding="0,0,0,0,100" Visible="false"
    HeightF="351.8678">
<Controls>
<Item1 Ref="2" Name="xrLabel1" ControlType="XRLabel"
    SizeF="126.0417,44.875" LocationFloat="29.16667, 20.79167"
    Font="Arial, 12pt, charSet=238" Text="Test">
<StylePriority Ref="3" UseFont="false" />
</Item1>
</Controls>
</Item1>
<Item2 Ref="4" Name="Glava" ControlType="TopMarginBand"
    Padding="0,0,0,0,100" TextAlignment="TopLeft" HeightF="0" />
<Item3 Ref="5" Name="Noga" ControlType="BottomMarginBand"
    Padding="0,0,0,0,100" BorderDashStyle="Dash"
    TextAlignment="TopLeft" HeightF="0">
<StylePriority Ref="6" UseBorderDashStyle="false" />
</Item3>
</Bands>
</XtraReportsLayoutSerializer>
```

Struktura 2.7: Struktura predloge testnega primera, izvožena iz orodja *DevExpress*.

Poglavje 3

Izdelava rešitve

V poglavju si bomo ogledali postopek izdelave programa za pripravo dokumentov, ki povezuje delovanje orodij *DevExpress* in *iTextSharp*.

3.1 Osnovna ideja aplikacije

Orodji *DevExpress* in *iTextSharp* imata precej nasprotne prednosti in slabosti. Orodje *DevExpress* ima zelo izpopolnjen vmesnik za izdelavo predlog, ki uporabniku omogoča preprosto pripravo oblike poročila brez znanja programiranja. Žal pa je orodje *DevExpress* od knjižnice *iTextSharp* bistveno manj učinkovito pri pripravi dokumentov, saj je velikost končnih datotek večja, čas priprave pa daljši. Da bi dobili, kar se da, učinkovito orodje, smo povezali obe orodji v en proces. *DevExpress* uporabimo za preprosto izdelavo predloge, medtem ko *iTextSharp* na podlagi predloge uporabimo za bolj učinkovito grajenje dokumentov.

Postopek izdelave poročila zajema naslednje korake (slika 3.1):

- **Priprava predloge z orodjem *DevExpress*** - V grafičnem vmesniku lahko preprosto in natančno izdelamo predlogo poročila, za katero ne potrebujemo programerskega znanja.

- **Izvoz predloge v XML** - Prek vgrajene metode orodja *DevExpress*

predlogo izvozimo v strukturo XML, ki jo uporabimo za sestavljanje dokumenta s knjižico *iTextSharp*.

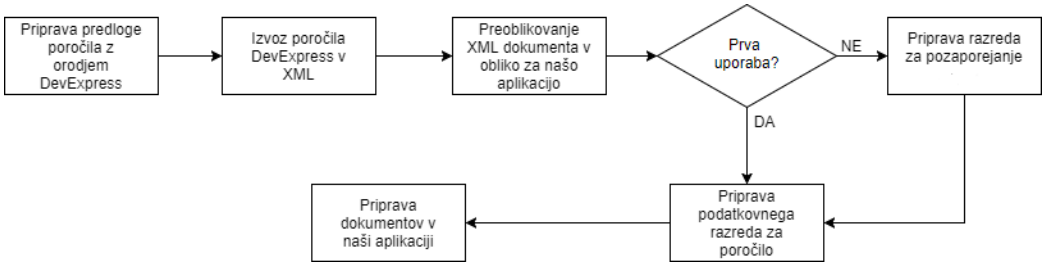
- **Preoblikovanje XML strukture v obliko za našo aplikacijo** - XML struktura, ki ga pripravi orodje *DevExpress*, zaradi zahtevnosti oteži postopek razzaporedenja. Korak izvedemo ročno vsakič, ko iz orodja *DevExpress* izvozimo novo ali spremenjeno/dopolnjeno poročilo v XML obliki. Strukturi poenotimo imena značk, da dobimo obliko drevesne strukture, ki jo lažje obdelamo v aplikaciji. Korak si bomo podrobneje pogledali v poglavju 3.4.

- **Priprava razreda za razzaporedenje** - Ko smo pripravljali aplikacijo, smo morali pripraviti še razred z enako strukturo kot XML, kamor shranimo podatke o predlogi. Korak izvedemo samo prvič ob pripravi aplikacije oz. ob spremembi orodja *DevExpress*. Na diagramu 3.1 je ta korak predstavljen kot kretnica - ali imamo razred za razzaporedenje že pripravljen ali ne.

- **Priprava razreda za poročilo** - Spreminjajoče podatke na našem poročilu (poglavje 3.2, rdeče označena polja na sliki 3.2) polnimo iz **podatkovnega vira**. V hibridni aplikaciji je vir shranjen v obliki objekta, ki zajema lastnosti (angl. *property*) s podatki, potrebnimi za izpolnitev enega poročila. Vir smo v aplikaciji izdelali kot C# razred.

- **Priprava dokumentov v naši aplikaciji** - Aplikaciji posredujemo XML s podatki o predlogi, napolnjen podatkovni objekt, lokacijo za odlaganje dokumenta in ime datoteke, ki se bo med postopkom tvorila.

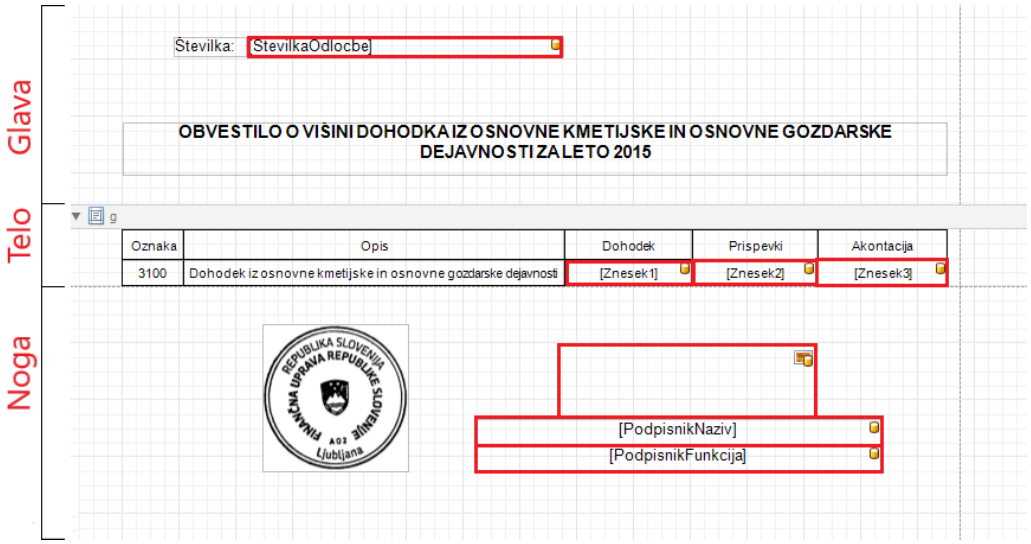
Celoten postopek je precej obširen in zahteven, vseeno je izdelava novega poročila precej lažja in hitrejša, kot če celo predlogo izdelamo s knjižnico *iTextSharp*. Implementacijo rešitve si poglejmo po korakih.



Slika 3.1: Diagram poteka za postopek hibridne aplikacije.

3.2 Priprava predloge z orodjem DevExpress in izvoz v XML

Implementacija aplikacije se začne s pripravo predloge v orodju *DevExpress*. Zgradili smo obliko obrazca, ki vsebuje besedilo, slike, tabelo in gradnike z dinamično določeno vrednostjo (slika 3.2). Elementi so razdeljeni na tri pasove: glava, telo in noga.



Slika 3.2: Predloga za dokumente v DevExpress orodju.

Na celotnem poročilu je uporabljena samo pisava *Arial*. Vsaka pisava, ki jo na poročilu uporabimo, pripomore k velikosti končne datoteke.

Predlogo izvozimo v strukturo XML po postopku, opisanem v poglavju 2.3.4. Potrebovali jo bomo v nadaljnjih korakih našega procesa.

3.3 Priprava podatkovnega razreda za poročilo

Kot smo opisali v poglavju 2.3.4, je za dinamično polnjenje polj na obrazcu potrebno definirati podatkovni tip oz. entiteto, ki poročilu predstavlja vir podatkov. Ta je v naši aplikaciji predstavljen kot razred. Naša predloga, ki smo jo izdelali v poglavju 3.2, vsebuje 7 gradnikov, ki jih v aplikaciji predstavimo s sedmimi lastnostmi (angl. *properties*) našega podatkovnega razreda, ki smo ga poimenovali *ReportTestEntity*. Lastnosti razreda se uparijo z vsebinskimi gradniki na predlogi poročila 3.2, ki so označeni z rdečim okvirjem:

- **StevilkaOdlocbe** (string)
- **Znesek1** (string)
- **Znesek2** (string)
- **Znesek3** (string)
- **PodpisnikFunkcija** (string)
- **PodpisnikNaziv** (string)
- **PodpisnikPodpis** (ByteArray)

Imena v entiteti morajo biti enaka tistim, ki smo jih pri izdelavi predloge v *DevExpress* orodju določili gradnikom kot podatkovni vir. Hibridna aplikacija uparja lastnosti te entitete z gradniki, ki vsebujejo značko *DataBinding* v XML strukturi. Za lažje uparjanje dodamo še seznam vseh imen lastnosti razreda, ki ga bomo uporabili v poglavju 3.6.

3.4 Preoblikovanje XML dokumenta

Kot smo opisali v poglavju 2.3.4, je struktura XML, ki nam jo pri izvozu vrne orodje *DevExpress*, precej zahtevna. Namesto prilagajanja aplikacije je bil lažji način preoblikovanje XML datoteke, da bo lahko razred za razzaporedenje kar se da preprost.

Značke niso več poimenovane po slogu *Item* z zaporedno številko (poglavje 2.3.4, struktura 2.7), ampak je ime značke enak njenemu tipu (nahaja se v lastnosti *ControlType*). Izvor strukture predstavlja značka *XtraReportsLayoutSerializer* (struktura 3.3, vrstica 1), ki poleg poddreves vsebuje še višino in širino dokumenta ter odmik vsebine od robov (angl. *margin*). Predloga je razdeljena na pasove, ki so predstavljeni kot značke *Band* (struktura 3.3, vrstica 3). Vsak pas ima podano skupno višino (*Height*), ki se uporablja za pravilno razporejanje vsebine na dokument. Lastnost *Name* vsebuje vrsto pasu (glava, telo ali noga). Seznam *Controls* (struktura 3.3, vrstica 4) vsebuje vsebinske gradnike *XRLabel* (struktura 3.3, vrstica 5) in *XRPictureBox* (struktura 3.3, vrstica 9). Besedilni gradnik (*XRLabel*) vsebuje podatke o tipu in velikosti pisave (*Font*) ter poravnavi besedila (*TextAlignment*). Slikovni gradnik (*XRPictureBox*) vsebuje sliko (*Image*), shranjeno v obliki *Byte Array*, lahko pa mu jo podamo iz podatkovnega objekta med izvajanjem aplikacije. Oba vsebinska gradnika vsebujeta še podatke o velikosti in poziciji na predlogi (*Size in Location*), lahko pa vsebujeta še znački za podatkovni vir in oblikovne lastnosti.

- *StylePriority (vrstica 7)* - Značka vsebuje oblikovne lastnosti besedilnega (*XRLabel*) ali slikovnega (*XRPictureBox*) gradnika. V aplikaciji uporabimo lastnost značke *UseBorders*, ki določa, če je gradnik obrobljen s črto (uporabimo za izdelavo tabele).

- *DataBindings (vrstica 6)* - Značko vsebujejo gradniki, katerim vsebino polnimo dinamično med izvajanjem programa. Podatke za besedilne ali slikovne gradnike, ki vsebujejo to značko, pridobivamo iz podatkovnega vira, ki je v aplikaciji predstavljen kot podatkovni objekt (razred,

pripravljen v poglavju 3.3). Podatek *DataMember* nam pove, iz katere lastnosti podatkovnega razreda pobiramo podatke - ime lastnosti podatkovnega razreda mora biti torej enaka vsebini polja *DataMember* v XML strukturi.

```
<XtraReportsLayoutSerializer PageHeight="" PageWidth="" Margins="" >
  <Bands>
    <Band ControlType="" Name="" Alignment="" Height="48">
      <Controls>
        <XRLabel Name="" ControlType="XRLabel" TextAlignment="" Size="" Location="" Font="" Text="">
          <DataBindings DataMember="" />
          <StylePriority UseBorders="true" Borders="Left, Right, Top, Bottom" />
        </XRLabel>
        <XRPictureBox Name="" ControlType="XRPictureBox" SizeF="" LocationFloat="" Image="*binaryData*" >
          <DataBindings DataMember="" />
        </XRPictureBox>
      </Controls>
    </Band>
  </Bands>
</XtraReportsLayoutSerializer>
```

Struktura 3.3: Primer preoblikovane XML strukture za predlogo iz poglavja 3.2.

Če primerjamo strukturo 3.3 in 2.7 vidimo, da smo namesto enega simbola *item* z različnimi vrednostmi podatka *ControlType* uvedli različne simbole. Na ta način bo precej lažje izvajati branje podatkov in sestavljanje PDF-dokumenta.

3.5 Razzaporedenje in priprava ustreznega razreda

Podatke o predlogi, ki smo jo izdelali z orodjem *DevExpress*, imamo shranjene v XML strukturi. Če jih želimo uporabiti v aplikaciji, jih moramo prebrati in shraniti v ustrezno strukturo s postopkom razzaporedenja, kot opisuje poglavje 2.2.3. Postopek potrebuje razred, ki ima enako obliko kot naša XML struktura. Priprave razredov se lahko lotimo ročno ali pa uporabimo program *XSD*, ki zna sam pretvoriti XML shemo v c# razrede. Več o postopku si lahko preberemo v uradni Microsoftovi dokumentaciji [8]. Rezultat je objekt z enako strukturo, kot je opisano v poglavju 3.4.

Razred pripravljamo iz izvožene predloge orodja *DevExpress*, na katero dodamo vse gradnike, ki jih želimo kasneje uporabljati v hibridni aplikaciji. To nam zagotovi, da ima razred vse strukture, ki se lahko pojavijo pri uvozu novih predlog, zaradi česar je ta korak potreben samo ob razvoju aplikacije.

3.6 Priprava poročil v hibridni aplikaciji

Do sedaj smo spoznali celoten postopek priprave virov, ki jih bomo potrebovali za našo aplikacijo. V tem razdelku se bomo posvetili delovanju aplikacije same. V grobem je razdeljena na dva glavna dela. Prvi del skrbi za razzaporedenje XML strukture in nam vrne objekt razreda **XtraReportsLayoutSerializer**, ki smo za pripravili v poglavju 3.5. Postopek je združen v metodo *Serializer*.

V metodi pripravimo objekt razreda za razzaporedenje, ki smo ga poimenovali *XtraReportsLayoutSerializer* (program 3.4, vrstica 3), kamor bomo shranili vsebino naše sheme. Vsebino XML strukture preberemo v delovni pomnilnik (program 3.4, vrstica 5) in izvedemo postopek razzaporedenja (program 3.4, vrstica 6). Metoda te podatke vrne kot rezultat tipa *XtraReportsLayoutSerializer*, napolnjenega z vsebino (program 3.4, vrstica 8).

Drugi del aplikacije predstavlja metoda *DocumentGenerator*, ki je za-

```
1 public XtraReportsLayoutSerializer Serializer(string path)
2 {
3    XtraReportsLayoutSerializer obj = null;
4    XmlSerializer serializer = new
         XmlSerializer(typeof(XtraReportsLayoutSerializer));
5    StreamReader reader = new StreamReader(path);
6    obj =
         (XtraReportsLayoutSerializer)serializer.Deserialize(reader);
7    reader.Close();
8    return obj;
9 }
```

Program 3.4: Metoda za razzaporedenje XML strukture.

dolžena za grajenje dokumentov s knjižnico *iTextSharp*. Kot parametra ji podamo podatkovni objekt in rezultat metode *Serializer* v obliki *XtraReportsLayoutSerializer*. Podatkovni objekt smo napolnili z umetnimi podatki, ki bodo na vseh dokumentih testnega nabora enaki. Za uporabo v realnem svetu bi podatkovni vir sestavili na podlagi podatkov iz podatkovne baze.

Metoda (program 3.5) zaporedno bere elemente iz *XtraReportsLayoutSerializer* objekta **obj** in jih dodaja na dokument po postopku, opisanem v poglavju 2.3.3. Z zanko po vrsti obdelamo vsak vsebinski gradnik. Začnemo z gradniki v pasu glava, ki se nahaja na vrhu strani. Nadaljujemo s pasom telo, noga pa se ne glede na ostalo vsebino nahaja na dnu strani.

Skozi strukturo se hibridna aplikacija sprehaja s pomočjo dveh zank. Zunanja zanka (program 3.5, vrstica 11) gre skozi seznam Bands, ki vsebuje vse pasove naše predloge. Vse gradnike v vsakem pasu pa obdelamo z drugo zanko (program 3.5, vrstica 17), ki vsebuje logiko za dodajanje besedilnih in slikovnih gradnikov. Pasovi v izvoženi XML datoteki niso nujno v pravilnem vrstnem redu, zato seznam ustrezno uredimo po vrstnem redu glava, telo, noga (program 3.5, vrstica 10).

Zunanja zanka skrbi tudi za pravilno postavitev gradnikov v posameznem

```
10 var bandList = SortBandList(obj.Bands);
11 foreach (Band band in bandList)
12 {
13    if (band.Controls == null)
14       break;
15    if(band.Name == "BottomMargin"  band.Name == "ReportFooter")
16       currentHeight = pageHeight - (float)band.HeightF;
17    foreach (var control in band.Controls)
18    {
19       if(control.GetType().Name.ToUpper() == "XRLABEL")
20       {
21       //Koda za dodajanje labele
22       }
23       else if((control.GetType().Name.ToUpper() == "XRPICTUREBOX")
24       {
25       //koda za dodajanje slike
26       }
27    }
28    currentHeight += (float)band.HeightF;
29 }
```

Program 3.5: Zanka za zaporedno dodajanje gradnikov na dokument.

pasu. S knjižnico *iTextSharp* elemente dodajamo po principu koordinatnega sistema - 2D ploskve z osmi X in Y. Vsak gradnik je določen s točko T(x,y), ki nam predstavlja pozicijo elementa v prostoru.

Kot pa smo videli v poglavjih 2.3.3 in 2.3.4, imata orodji drugačen pristop določanja koordinat na dokumentu. Izhodišče koordinatnega sistema v *iTextSharp* knjižnici (točka T(0,0)) se nahaja v levem spodnjem kotu, v orodju DevExpress pa je sistem drugačen. Vsak pas (glava, telo in noga) ima svoj koordinatni sistem, kjer je točka T(0,0) v levem zgornjem kotu. Vsak pas ima torej svoje izhodišče - svojo točko T(0,0), zato moramo pri

sestavljanju dokumenta beležiti skupno višino do sedaj dodanih pasov in jo ustrezno upoštevati pri dodajanju nadaljnjih gradnikov (program 3.5, vrstica 28). Vsak gradnik moramo zamakniti od vrha strani, da se bo nahajal v svojem pasu, in mu preslikati koordinato Y na nasprotno stran koordinatnega sistema. Primer: Predpostavimo, da smo na dokument že zložili vse elemente pasu glava in sedaj dodajamo gradnike na pas telo. Pri postavljanju gradnikov na dokument bo vrednost X osi ostala nespremenjena, os Y pa se bo prištela skupna višina pasu glava ter prezrcalila na nasprotno stran koordinatnega sistema. Računanje koordinat za postavitev besedilnega gradnika na dokument se izvaja v programu 3.7, za slikovne pa v programu 3.9. Izjema je pas noga. Ta se mora ne glede na višino pasu telo vedno nahajati na spodnjem delu strani. Koordinate gradnikov v tem pasu smo samo prezrcalili na pravo višino v dokumentu - da najdemo začetek pasu noga, višini strani odštejemo višino pasu noga (program 3.5, vrstica 16).

Logika za dodajanja besedilnih in slikovnih gradnikov je različna, vseeno pa imamo v obeh določene korake iste. Koda, za dodajanje besedilnih gradnikov, je razdeljena v programa 3.6 in 3.7, dodajanje slik pa je prikazano v programu 3.9.

Vsak gradnik, ki ga obdelujemo v aplikaciji, je tipa *object*, kar pomeni da okolje C# ne pozna točnega tipa in vsebine tega objekta, zato mu moramo najprej določiti podatkovni tip (angl. *cast*) (program 3.6, vrstica 32). Iz objekta lahko sedaj preberemo podatke, ki so potrebni za obdelavo gradnika. V vrsticah 33 do 43 (program 3.6) iz objekta beremo lastnosti za pisavo, velikost gradnika in poravnavo besedila. Ker so besedilni gradniki lahko tudi deli tabel, ustrezno nastavimo tudi lastnost *border*, ki gradniku doda črno obrobo - vrstica 42. Vsebino besedilnega gradnika lahko dobimo na dva načina. Lahko jo preberemo iz lastnosti objekta ali pa jo pridobimo iz podatkovnega objekta (program 3.6, vrstica 44). Metoda *GetLabelText* je opisana kasneje v poglavju in prikazana v programu 3.8. Metoda *AddLabelToPdf* skrbi za postavitev besedilnega gradnika na dokument (program 3.7). Gradniku določimo ustrezen tip (krepko ali navadno) in velikost pisave ter

```
30 if(control.GetType().Name.ToUpper() == "XRLABEL")
31 {
32    XRLabel labelObject = (XRLabel)control;
33    string[] pozicija = labelObject.LocationFloat.Split(',');
34    var font = labelObject.Font.Split(',');
35    float fontSize = ToFloat(font[1].Substring(0,
          font[1].IndexOf('p')));
36    string fontStyle = null;
37    if(font[font.Length - 1].Contains("style"))
38    {
39       fontStyle = font[2].Substring(font[2].IndexOf('=') + 1);
40    }
41    string[] size = labelObject.SizeF.Split(',');
42    bool border = labelObject.Borders != null;
43    int alignment = GetAlignment(labelObject.TextAlignment);
44    string text = GetLabelText(labelObject, dataObject);
45    AddLabelToPdf(cb, ToFloat(pozicija[0]), ToFloat(pozicija[1]),
          fontSize, fontStyle, text, ToFloat(size[0]),
          ToFloat(size[1]), border, alignment);
46 }
```

Program 3.6: Logika za obdelavo besedilnega gradnika.

koordinate za pozicijo na dokumentu (program 3.7, vrstice 49 do 61). V vrsticah 63 do 67 pripravimo objekt tipa *Rectangle*, ki bo nosil vsebino našega gradnika. V primeru, da je naš gradnik del tabele, mu v vrstici 69 nastavimo še črtno obrobo.

Vsebina je gradnikom lahko predpolnjena že v predlogi ali pa v času izvajanja aplikacije določena prek podatkovnega objekta. Če je predpolnjena v predlogi, bo gradnik hranil vsebino v lastnosti *Text* (program 3.8, vrstica 5), v nasprotnem primeru pa mora imeti izpolnjeno lastnost *DataBinding*, kjer je določena povezava na atribut iz podatkovnega objekta (program 3.8, vr-

stica 9). V poglavju 3.3 smo podatkovnemu viru dodali seznam, ki vsebuje imena vseh lastnosti razreda, s pomočjo katerega uparjamo gradnike z ustreznimi atributi ter iščemo morebitne napake v pripravi podatkovnega vira ali pa sheme predloge. Gradnike uparimo s podatkovnim virom prek lastnost *DataBinding*, ki se mora ujemati z imenom ustreznega atributa podatkovnega objekta (program 3.8, vrstica 13). Če pripadajočega para ne najdemo, sklepamo, da je med pripravo predloge oz. podatkovnega vira prišlo do napake (program 3.8, vrstica 10).

Slike na dokument dodajamo drugače kot besedilo. Logika je prikazana v programu 3.9. Kot pri dodajanju besedilnih gradnikov, najprej objektu gradnika določimo podatkovni tip, ki je tokrat *XRPictureBox* (program 3.9, vrstica 79). Pripravimo objekt tipa *img*, ki mu bomo nastavili potrebne lastnosti in ga dodali na dokument. Vsebino lahko kot pri besedilnih gradnikih preberemo kot statično vsebino iz predloge (program 3.9, vrstica 84) ali pa kot dinamično vsebino iz podatkovnega objekta (program 3.9, vrstica 82). Objektu *img* nastavimo še velikost ter pozicijo (program 3.9, vrstice 85 do 89) in ga dodamo na dokument (program 3.9, vrstica 90).

Dokument se po končanem postopku priprave odloži na lokacijo, ki jo aplikaciji nastavimo prek parametra.

```
47 public void AddLabelToPdf(PdfContentByte cb, float xPos, float
       yPos, float fontSize, string fontStyle, string textValue, float
       width, float heigth, bool border, int alignment = 0)
48 {
49    if (fontStyle == "Bold")
50       _font = _fontBold;
51    else
52       _font = _fontZL;
53
54    float llx = xPos;
55    float lly = pageHeight - (yPos + heigth) - currentHeight;
56    float urx = xPos + width;
57    float ury = pageHeight - yPos - currentHeight;
58
59    _font.Size = fontSize * 1.3333333f;
60    var paragraph = this.GetParagraph(textValue, false, _font,
          alignment);
61    paragraph.KeepTogether = true;
62
63    ColumnText ct = new ColumnText(cb);
64    Rectangle rect = new Rectangle(llx, lly, urx, ury);
65    ct.SetSimpleColumn(rect);
66    ct.AddElement(paragraph);
67    int status = ct.Go();
68
69    if (border)
70    {
71       rect.Border = iTextSharp.text.Rectangle.BOX;
72       rect.BorderWidth = 1;
73       rect.BorderColor = new BaseColor(0, 0, 0);
74       cb.Rectangle(rect);
75    }
76 }
```

Program 3.7: Logika za dodajanje besedilnega gradnika na dokument.

```
1  private string GetLabelText(XRLabel labelObject, ReportTestEntity
       dataObject)
2  {
3     if(labelObject.DataBindings == null)
4     {
5        return labelObject.Text;
6     }
7     else
8     {
9        string propName = labelObject.DataBindings.DataMember;
10       if (dataObject.GetType().GetProperty(propName).Equals(null))
11          throw new Exception("Neveljana oblika objekta!");
12
13       return
             (string)dataObject.GetType().GetProperty(propName).GetValue(dataObject);
14    }
15 }
```

Program 3.8: Metoda za polnjenje vsebine besedilnih gradnikov.

```
77 else if(control.GetType().Name.ToUpper() == "XRPICTUREBOX")
78 {
79    XRPictureBox pictureObject = (XRPictureBox)control;
80    Image img = null;
81    if (pictureObject.Image == null)
82       img = Image.GetInstance(GetObjectValue(pictureObject,
          dataObject));
83    else
84       img = Image.GetInstance(pictureObject.Image);
85    string[] pozicija = pictureObject.LocationFloat.Split(',');
86    string[] size = pictureObject.SizeF.Split(',');
87    img.SetAbsolutePosition(ToFloat(pozicija[0]), pageHeight -
       ToFloat(pozicija[1]) - currentHeight - ToFloat(size[1]));
88    img.ScaleAbsoluteHeight(ToFloat(size[1]));
89    img.ScaleAbsoluteWidth(ToFloat(size[0]));
90    doc.Add(img);
91 }
```

Program 3.9: Logika za dodajanje slikovnih gradnikov na dokument.

Poglavje 4

Ovrednotenje aplikacije

V tem poglavju bomo hibridno aplikacijo ovrednotili glede na:

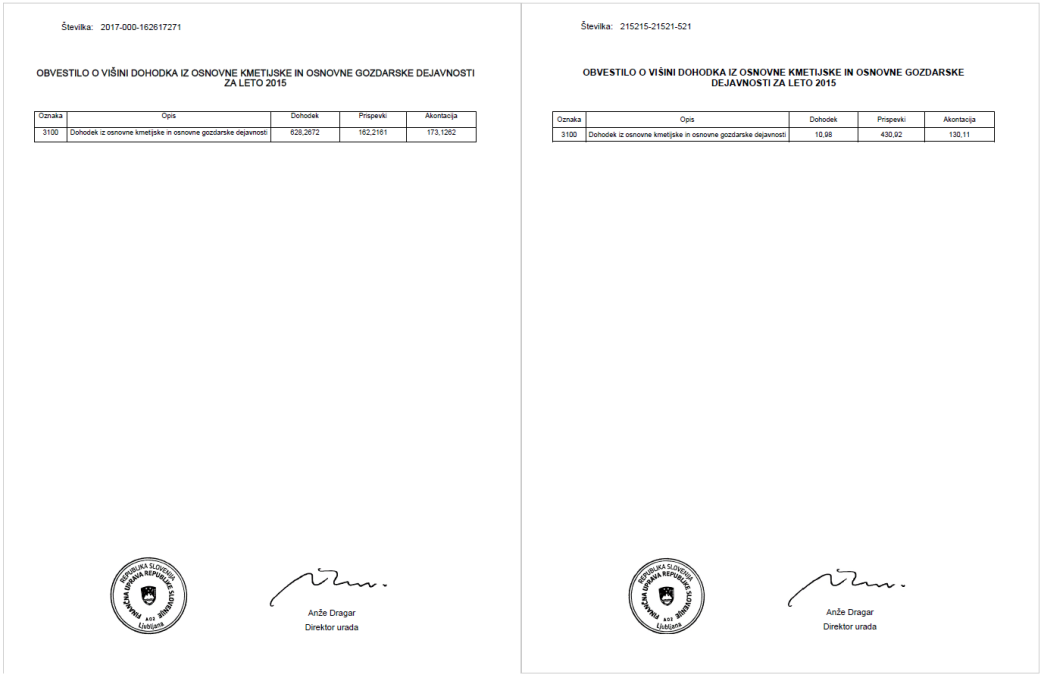
- podobnosti oblike doseženih rezultatov v primerjavi z izdelano predlogo,
- hitrosti priprave testnega nabora dokumentov in
- velikosti pripravljenih datotek (v KB).

Vsi rezultati bodo primerjani z orodjem *DevExpress*. Primerjava s knjižnico *iTextSharp* v našem primeru nima posebne vrednosti, saj je vložek za izdelavo predloge mnogo večji kot pri hibridni aplikaciji.

4.1 Preverjanje podobnosti izpisa

Podobnost dokumenta hibridne aplikacije bomo primerjali z rezultatom orodja *DevExpress*. Primerjavo bomo izvedli na predlogi, ki smo jo opisali v poglavju 3.2.

Razlike v dokumentih so minimalne (slika 4.1). Primer vidimo v vertikalni poravnavi besedila v tabeli, ki ga v naši različici knjižnice *iTextSharp* za besedilne gradnike ni mogoče nastaviti. Dokumenta seveda nista popolnoma enaka, vendar je dovolj že, da se toliko približamo originalu, da je dokument uporaben, kar v našem primeru tudi je.



Številka: 2017-000-162617271

OBVESTILO O VIŠINI DOHODKA IZ OSNOVNE KMETIJSKE IN OSNOVNE GOZDARSKE DEJAVNOSTI ZA LETO 2015

Oznaka	Opis	Dohodek	Prispevki	Akontacija
3100	Dohodek iz osnovne kmetijske in osnovne gozdarske dejavnosti	628,2672	162,2161	173,1262

Anže Dragar
Direktor urada

Številka: 215215-21521-521

OBVESTILO O VIŠINI DOHODKA IZ OSNOVNE KMETIJSKE IN OSNOVNE GOZDARSKE DEJAVNOSTI ZA LETO 2015

Oznaka	Opis	Dohodek	Prispevki	Akontacija
3100	Dohodek iz osnovne kmetijske in osnovne gozdarske dejavnosti	10,98	430,92	130,11

Anže Dragar
Direktor urada

Slika 4.1: Dokument pri uporabi metode *DevExpress* (desno) in hibridne aplikacije (levo).

4.2 Velikost datoteke in hitrost priprave

Hibridno aplikacijo smo primerjali z rezultati priprave poročil z orodjem *DevExpress*. Primerjavo metod smo izvedel na podlagi priprave testnega nabora 10.000 dokumentov z enako predlogo, podatki in pogoji (isti računalnik z enako obremenitvijo).

Testiranje smo izvajali na namiznem računalniku z naslednjimi specifikacijami:

- procesor Intel i7-6700, 4 jedra, 8 niti,
- 8 Gb delovnega pomnilnika,
- disk SSD Samsung Evo 850 in
- integrirana grafična kartica Intel HD 530.

Vsako testiranje obeh metod smo vedno ponovili 5-krat ter izračunal povprečje in standardni odklon vseh rezultatov.

```
1  Stopwatch t = new Stopwatch();
2  t.Start();
3  for (int i = 0; i < 10000; i++)
4  {
5     //Koda za generiranje
6  }
7  t.Stop();
8  t.Reset();
9  t.Start();
10 Parallel.For(0,10000, i =>
11 {
12    //Koda za generiranje
13 });
14 t.Stop();
```

Program 4.2: Testno okolje za merjenje časa priprave testnega nabora dokumentov.

Postopek smo izvedeli najprej z navadno zanko (program 4.2, vrstica 3), kjer se je priprava vsakega dokumenta izvajala zaporedno. Test smo ponovili še z uporabo vzporedne zanke (program 4.2, vrstica 10), kjer je vsaka nit procesorja pripravljala svoj dokument (teoretično 8 dokumentov na enkrat). Merjenje časa za pripravo nabora testnih dokumentov smo izvedeli z uporabo programske štoparice (program 4.2, vrstica 1), ki nam po zaključenem merjenju pove, koliko časa je program porabil za samo pripravo dokumentov v milisekundah.

Iz rezultatov primerjave lahko jasno vidimo prednosti hibridne aplikacije pred orodjem *DevExpress*, saj smo na prostoru prihranili 139 KB po dokumentu. Prav tako je bil čas priprave precej krajši. Navadna zaporedna priprava je bila hitrejša za cca 73 %, vzporedna pa za 80 %. Čas vzporedne

	DevExpress (std. odklon)	Hibridna aplikacija (std. odklon)
Zaporedna priprava (sekunde)	771,72 (8,67)	211,20 (12,14)
Vzporedna priprava (sekunde)	387,570 (9,69)	75,87 (7,4)
Velikost datoteke (KB)	201	62

Tabela 4.1: Rezultati testiranja.

priprave bi lahko še izboljšali, če bi imeli na voljo več procesorskih jeder.

Poglavje 5

Sklepne ugotovitve

Med izdelavo diplomskega dela je nastala aplikacija za univerzalno pripravo dokumentov. Prednost hibridne aplikacije je prihranek časa in velikost dobljenih dokumentov.

Hibridna rešitev za delovanje še vedno potrebuje licenco programskega paketa DevExpress, saj uporabljamo njihovo okolje za grafično izgradnjo predloge in izvoz v XML. V našem podjetju imamo to licenco zakupljeno za potrebe drugih projektov, zato v mojem primeru to ni težava. Če pa te licence nimamo na voljo, rešitev ni najcenejša (licenca stane okoli 1000 USD). Seveda je na voljo mnogo programskih rešitev, ki deloma nudijo enake funkcionalnosti kot *DevExpress*, in so brezplačne. Ena najbolje ocenjenih alternativ je *Windword AutoTag* [1].

Hibridna aplikacija je med testiranjem pokazala dobre rezultate v primerjavi s že obstoječimi rešitvami, ki smo jih uporabili pri izdelavi (*iTextSharp* in *DevExpress*). Rešitev je enostavna za uporabo, velikost dokumentov in hitrost delovanja pa presežeta moja pričakovanja pred izdelavo.

Nekaj možnih dopolnitev:

- Lažja izvedba celotnega postopka priprave za uporabnika (Poglavje 3.1): rešitev, izdelana v nalogi, še vedno potrebuje nekaj ročnih posegov uporabnika v aplikacijo (Izvoz XML datoteke, prenos datoteke v delovno področje aplikacije, preoblikovanje XML-ja). Ideja je, da se po pri-

pravljeni predlogi in razredih celoten postopek (poglavje 3.1) izvede samodejno s pritiskom gumba.

- Uporabniški vmesnik: rešitev s celim postopkom priprave bi združili v *Windows Forms* aplikacijo z uporabniškim vmesnikom, kjer bi uporabniki nastavljali lokacije za odlaganje pripraljenih dokumentov, vhodne podatke, spremljali delovanje ...

- Dodatne optimizacije delovanja: lahko bi še dodatno zmanjšali velikost končne datoteke in pohitrili delovanje aplikacije z uporabo hitrejših načinov sestavljanja dokumenta in izborom pisav, ki uporabijo manj prostora. Prav tako bi lahko uvedli stiskanje slik na velikost, ki se uporabi na dokumentu.

Literatura

[1] AlternativeTo.net. Alternativa za orodje DevExpress. Dosegljivo: http://alternativeto.net/software/xtrareports/, 2017. [Dostopano: 13. 6. 2017].

[2] DevExpress. Devexpress WinForms. Dosegljivo: https://www.devexpress.com/Products/NET/Controls/WinForms/, 2017. [Dostopano: 13. 6. 2017].

[3] ITEXT. Knjižnica iTextSharp. Dosegljivo: http://itextpdf.com/tags/itextsharp, 2017. [Dostopano: 18. 6. 2017].

[4] Mono Project. Mono. Dosegljivo: http://www.mono-project.com/docs/, 2017. [Dostopano: 30. 8. 2017].

[5] Jonathan Pryor. Comparing java and C sharp generics. Dosegljivo: http://www.jprl.com/Blog/archive/development/2007/Aug-31.html, 2017. [Dostopano: 6. 7. 2017].

[6] Tina Seiber. What is an XML file and what are its uses? Dosegljivo: http://www.makeuseof.com/tag/xml-file-case-wondering/, 2017. [Dostopano: 13. 6. 2017].

[7] Lokar Matija, Uranič Srečo. Programiranje 1. Dosegljivo: http://www.scpet.net/vss/xinha/plugins/ExtendedFileManager/demo_images/egradiva/Programiranje1-Lokar.pdf, 2016. [Dostopano: 30. 8. 2017].

[8] Eric Reitan, Luke Latham, tompratt AQ, yishengjin1413. Introducing XML serialization. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/framework/serialization/introducing-xml-serialization, 2017. [Dostopano: 17. 6. 2017].

[9] A. Troelsen. *Pro C Sharp 5.0 and the .NET 4.5 Framework*. Expert's voice in .NET. Apress, 2012.

[10] W3Schools. How can XML be used? Dosegljivo: https://www.w3schools.com/xml/xml_usedfor.asp, 2017. [Dostopano: 13. 6. 2017].

[11] W3Schools. Introduction to XML. Dosegljivo: https://www.w3schools.com/xml/xml_whatis.asp, 2017. [Dostopano: 13. 6. 2017].

[12] Wikipedia. Programski jezik C sharp. Dosegljivo: https://sl.wikipedia.org/wiki/Programski_jezik_C_sharp, 2017. [Dostopano: 13. 6. 2017].