UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Lovro Košmerl

Optimizacija sistema za upravljanje z izdajami na primeru aplikacije za zavarovalništvo

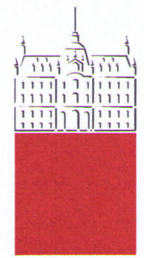
DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Mentor: doc. dr. Mojca Ciglarič

Ljubljana, 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si



Št. naloge: 01797/2012

Datum: 05.01.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **LOVRO KOŠMERL**

Naslov: **OPTIMIZACIJA SISTEMA ZA UPRAVLJANJE Z IZDAJAMI NA PRIMERU APLIKACIJE ZA ZAVAROVALNIŠTVO**

OPTIMIZATION OF A RELEASE MANAGEMENT SYSTEM: AN INSURANCE-RELATED CASE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Opišite začetno stanje sistema za upravljanje z izdajami v izbranem podjetju. Pojasnite, kako so se določale oznake novih različic izdaje in novih izdaj, kako je potekal nadzor nad vključeno izvorno kodo, kako je bilo nameščati izdaje v različna okolja z različnimi zahtevami, kako je potekalo testiranje aplikacije in kako se je vodila zgodovina. Pojasnite pomanjkljivosti in težave ter predlagajte, kako bi jih lahko odpravili. Izberite primerna orodja, ki omogočajo potrebne izboljšave sistema, ter nazadnje te izboljšave tudi realizirajte. Kritično komentirajte končni izdelek in izkušnje pri njegovi uporabi.

Mentor:

M. Ciglarič

doc. dr. Mojca Ciglarič

Dekan:

prof. dr. Nikolaj Zimic

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

diplomskega dela

Spodaj podpisani/-a   Lovro Košmerl,

z vpisno številko   63010071,

sem avtor/-ica diplomskega dela z naslovom:

Optimizacija sistema za upravljanje z izdajami na primeru aplikacije za zavarovalništvo

S svojim podpisom zagotavljam, da:

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

  doc. dr. Mojce Ciglarič

  in somentorstvom (naziv, ime in priimek)

  ____

- 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 22.3.2011   Podpis avtorja/-ice: _________________________

Zahvala

Iskreno se zahvaljujem mentorici doc. dr. Mojci Ciglarič za mentorstvo in nasvete pri izdelavi diplomske naloge. Zahvaljujem se sodelavcem, ki so mi pomagali optimizirati sistem za upravljanje z izdajami, predvsem Benjaminu Levsteku in Damirju Arhu. Zahvaljujem se tudi vsem svojim najbližjim, ki so mi stali ob strani skozi celoten študij, predvsem staršem, sestri Uršuli, prijatelju Blažu in ženi Petri.

Kazalo

Povzetek ... 1

Abstract ... 3

Uvod ... 5

Namen ... 5

1 Opis aplikacije in prvotnega stanja sistema za upravljanje z izdajami ... 6

1.1 Produkcijsko okolje in aplikacija ... 6

1.2 Prvotno stanje sistema za upravljanje z izdajami ... 7

1.2.1 Sistem za nadzor različic izvorne kode ... 7

1.2.2 Strežnik z zvezno integracijo ... 8

1.2.3 Orodje za gradnjo programske opreme ... 10

2 Problemi in pomanjkljivosti pri upravljanju z izdajami ... 14

2.1 Avtomatsko povečevanje oznake različice izdaje ... 14

2.2 Izdajanje stabilnih izdaj in nadzor nad vključeno izvorno kodo ... 15

2.3 Sinhrone spremembe podatkovnega modela baze in izdaje aplikacije ... 15

2.4 Nameščanje izdaj v različna okolja z različnimi zahtevami ... 16

2.5 Kvaliteta namestitve in nadzor nad nameščanjem ... 16

2.6 Avtomatsko testiranje aplikacije ... 17

2.7 Zgodovina namestitev različic izdaj ... 18

3 Odprava problemov in izboljšave ... 19

3.1 Uvedba modulizacije aplikacije ... 19

3.2 Upravljanje z izdajami izvorne kode na nivoju repozitorija strežnika SVN ... 24

3.3 Avtomatizacija povečevanja oznake različice izdaje ... 28

3.4 Usklajevanje sprememb podatkovnega modela baze skupaj z izvorno kodo ... 30

3.4.1 Izdelava orodja za pomoč pri izdelovanju in shranjevanju skript SQL ... 32

3.4.2 Zajem ustreznih skript SQL pri izdelavi izdaje ... 34

3.4.3 Testiranje ter zagon skript SQL ob namestitvi nove različice izdaje ... 37

3.4.4 Prehod iz starega na nov način dela ... 37

3.5 Vpeljava programske komponente Windows Installer ... 38

3.5.1 Izbira orodja za pripravo paketa MSI ... 39

3.5.2 Vpeljava funkcionalnosti izdelave paketa MSI v proces priprave izdaje ... 41

3.6 Vpeljava avtomatskega testiranja aplikacije ... 44

3.6.1 Ogrodje NUnit ... 44

3.6.2 Orodje Ranorex 46
4 Sklepne ugotovitve 49
Literatura 51

Seznam uporabljenih kratic

ASP.NET ogrodje za spletne aplikacije razvito s strani podjetja Microsoft, omogoča razvijanje dinamičnih spletnih strani, spletnih aplikacij in spletnih storitev

CCNET CruiseControl.NET; strežnik z zvezno integracijo implementiran z uporabo Microsoft .NET ogrodja
- skripta CCNET – nastavitvena skripta, kjer se definira posamezne projekte

CI Continuous integration; zvezna integracija

DLL Dynamic Link Library; dinamična povezovalna knjižnica; knjižnica, kjer se nahaja prevedena izvorna koda

GUID globalen enoličen identifikator

HTTP Hyper Text Transfer Protocol; protokol za komunikacijo med odjemalcem (spletnim brskalnikom) in strežnikom na svetovnem spletu

HTTPS Hypertext Transport Protocol Secure sockets; protokol HTTP, ki omogoča varno internetno povezavo

IIS Internet Information Services; internetne informacijske storitve

NAnt odprtokodno programsko orodje za avtomatizacijo prevajanja Microsoft .NET kode (orodje Ant prilagojeno za Microsoft .NET)
- skripta NAnt – nastavitvena skripta, kjer se definira posamezne procese

PDM Physical Data Model; fizični podatkovni model; model, ki opisuje entitete, za katere se želi hraniti podatke, in povezave med njimi

PROD produkcijska izdaja

RC predizdaja, kandidat za produkcijsko izdajo

RDP protokol za dostopanje do oddaljenega namizja

SQL strukturiran povpraševalni jezik za delo s podatkovnimi bazami
- skripta SQL – skupek ukazov SQL

SVN Apache Subversion; odprtokodni sistem za nadzor različic izvorne kode
- strežnik SVN – strežnik, kjer je postavljen sistem SVN
- naslov SVN – naslov, ki se nanaša na lokacijo na strežniku SVN
- ukaz SVN – ukaz, s katerim se na strežniku SVN dela z datotekami

VPN navidezno zasebno omrežje

WiX Windows Installer XML; orodje, ki gradi Windows namestitvene pakete iz izvorne kode XML

WXS izvorna koda orodja WiX

XML eXtensible Markup Language; format podatkov za izmenjavo strukturiranih dokumentov v spletu

Povzetek

Pri razvoju programskih produktov smo priča nenehnemu ciklu razvoja, testiranja ter nameščanja izdaj. Zaradi ponovljivosti je smiselno ta proces avtomatizirati in ga s tem pohitriti ter hkrati zmanjšati verjetnost za napako zaradi zmanjšanja človeškega vpliva. V kolikor gre za večji produkt in za večjo razvojno ekipo, je avtomatizacija procesa takorekoč nujna za zagotavljanje ustreznega nivoja kvalitete procesa priprave izdaje. Posledično se lahko razvijalci produkta osredotočijo predvsem na njegov razvoj in ne izgubljajo dragocenega časa s testiranjem aplikacije in pripravo izdaje.

Namen diplomskega dela je optimizacija obstoječega sistema za upravljanje z izdajami.

V začetnem delu diplomske naloge je opisana aplikacija in obstoječe stanje sistema za upravljanje z izdajami. Sledi opis lastnosti dobrega sistema in primerjava vsake izmed lastnosti z obstoječim sistemom. Izpostavljene so predvsem pomanjkljivosti obstoječega sistema, na podlagi katerih so zastavljeni cilji diplomskega dela. V osrednjem delu sledi podroben opis postopka doseganja vsakega izmed ciljev. Poudarek je predvsem na delu s strežnikom za nadzor izvorne kode, na delu s strežnikom z zvezno integracijo ter na orodju za gradnjo programske opreme. Na koncu sledi analiza optimiziranega sistema, kjer je razvidno kateri cilji so bili doseženi, hkrati pa so izpostavljene tudi slabosti ter predlogi za izboljšavo.

Ključne besede: upravljanje z izdajami, avtomatizacija, zvezna integracija, SVN

Abstract

In the development of software products, we witness the continuing cycle of development, testing and installation of releases. Because of repeatability it is reasonable to automate this process and thus speed it up while reducing the probability of an error due to the reduction of human influence. In the case of the larger software product and bigger development team, process automatization is almost essential to ensure an adequate level of quality of release management process. Consequently, product developers can focus on its development and do not lose valuable time testing applications and preparing releases.

Purpose of this thesis is to optimize the existing release management system.

In the first part of the thesis there is a description of application and the initial state of the release management. Following is a description of the characteristics of a good release management system and a comparison of each of the characteristics with the initial state of the system. Emphasis is on defects on initial state of the system, under which the objectives of the thesis are formed. In the middle part of the thesis there is detailed description of achieving each one of the objectives. The emphasis is on working with source control server, continuous integration server and the tools for building the code. At the end there is an analysis of optimized system, exposing achieved objectives, weaknesses and suggestions for improvement.

Key words: release management, automatization, continuous integration, SVN

Uvod

Tekom razvoja programske opreme postaja le-ta vedno bolj kompleksna. Tipično, še posebno pri razvoju produktov, smo priča nenehnemu ciklu razvoja, testiranja ter nameščanja izdaj, kar poudarja tudi [32]. Če upoštevamo še razvoj in rastočo kompleksnost okolij na katerih teče programska oprema, postaja jasno, da imamo veliko spremenljivk, katere se morajo med sabo prilegati, če želimo zagotoviti dolgoročen uspeh produkta oziroma projekta (vir: [1]).

S povečevanjem velikosti razvojne ekipe, naraščanjem kompleksnosti programske opreme in potrebe po pogostem nameščanju izdaj, se slej ko prej pojavi potreba po avtomatizaciji sistema za upravljanje z izdajami. Z avtomatizacijo se dolgoročno pohitri celoten cikel izdaje, hkrati pa se močno zmanjša možnost napake, saj se zmanjša vpliv človeškega faktorja. Pri velikih projektih je avtomatizacija takorekoč nujna.

Namen

Namen diplomske naloge je opisati optimizacijo sistema za upravljanje izdaj, ki sem jo izpeljal na večjem produktu (aplikacija za zavarovalništvo). Ob prihodu v podjetje je bil sistem upravljanja že realiziran. Vseboval je sistem za nadzor različic izvorne kode, strežnik z zvezno integracijo ter orodje za gradnjo programske opreme.

Ena večjih pomanjkljivosti na obstoječem sistemu je bila nezmožnost zagotavljanja stabilne izdaje v primeru nujnega popravka napake v izvorni kodi, saj je bila izvorna koda za razvoj ter za različico aplikacije shranjena na istem mestu ter posledično enaka.

Kljub temu je sprva ta sistem zadovoljil potrebe po upravljanju izdaj, sčasoma, z rastjo razvojne ekipe, ter s povečevanjem števila strank, pa so se v obstoječem sistemu pokazale še druge pomanjkljivosti, kot na primer: ni zgodovine namestitev, ni sistema za prilagoditve produkta potrebam strank, sočasnost sprememb PDM-ja ter izdaje aplikacije ni zagotovljena. V diplomski nalogi bom podrobneje opisal pomanjkljivosti ter rešitve, poudarek pa bo na strežniku z zvezno integracijo ter orodjem za gradnjo programske opreme.

Cilji diplome so narediti sistem, ki bo omogočal:

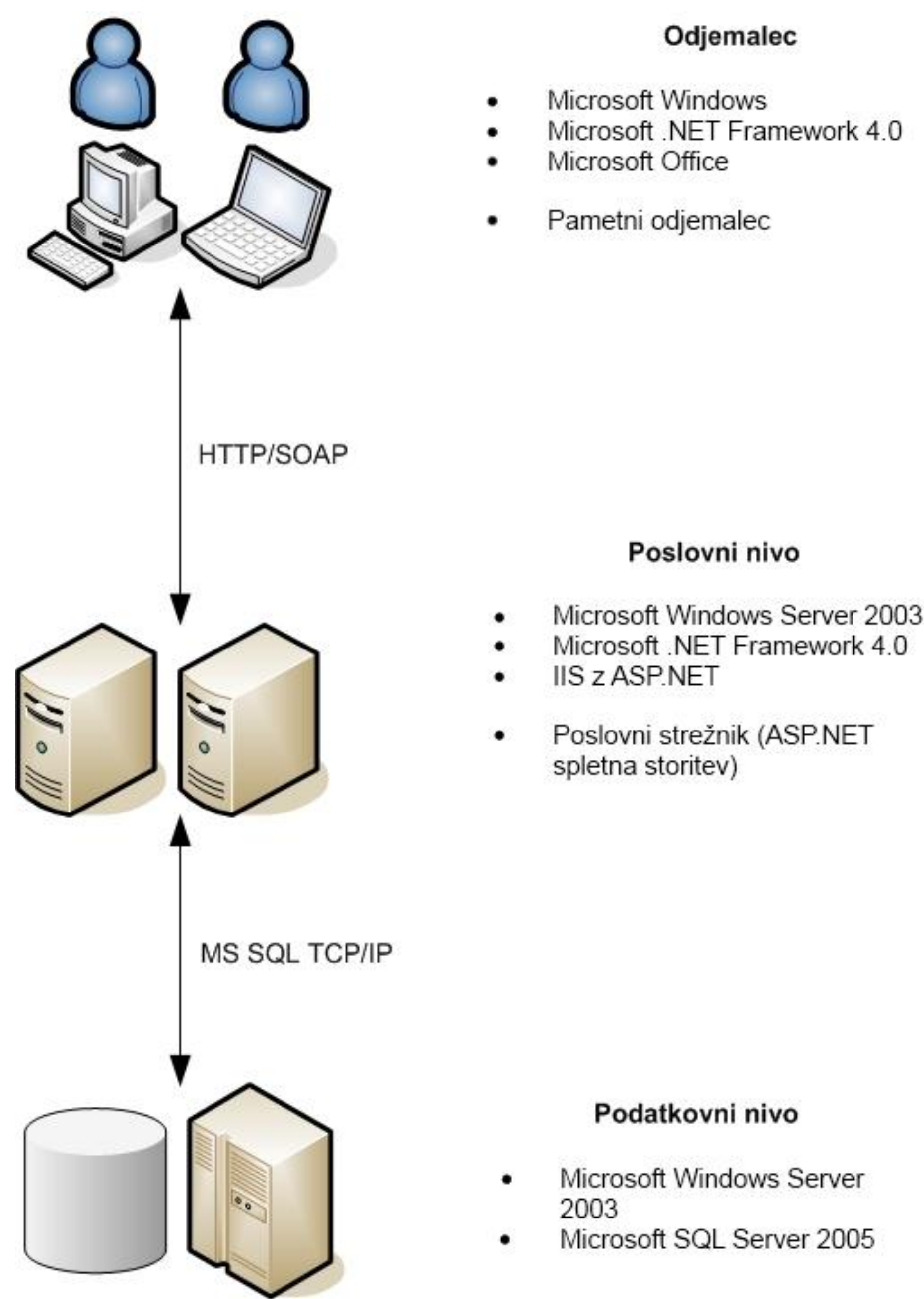
- avtomatsko povečevanje oznake različice izdaje
- prilagoditve produkta na podlagi potreb implementacij
- nadzor nad v izdajo vključeno kodo
- usklajevanje sprememb podatkovnega modela baze skupaj s kodo
- enostavnejši postopek namestitve izdaje
- avtomatsko testiranje aplikacije

1 Opis aplikacije in prvotnega stanja sistema za upravljanje z izdajami

Ob prihodu v podjetje je bil sistem upravljanja z izdajami že realiziran. V nadaljevanju sledi kratek opis postavitve aplikacije na produkcijskem okolju ter opis prvotnega stanja sistema za upravljanje z izdajami.

1.1 Produkcijsko okolje in aplikacija

Aplikacija se imenuje »Insure« in temelji na informacijskem sistemu s trinivojsko arhitekturo: odjemalec, poslovni nivo in podatkovni nivo (Slika 1).



Slika 1: Elementi arhitekture aplikacije »Insure«

Odjemalec je aplikacija, ki teče na delovnih postajah uporabnikov. Temelji na konceptu t.i. pametnega odjemalca. Ne potrebuje instalacije, avtomatično osvežuje izdaje odjemalca, ima izgled namizne aplikacije, binarne podatke prenaša preko protokola HTTP. Dodatne podrobnosti o pametnem odjemalcu so na voljo v [2]. Odjemalec je zasnovan na tehnologiji Microsoft .NET Windows Forms (več v: [3]) in teče na operacijskih sistemih družine Microsoft Windows, na katerih mora biti nameščeno ogrodje Microsoft .NET Framework 4.0 (več v: [4]). Priprava in tiskanje dokumentov sta podprta preko integracije s pisarniškim paketom Microsoft Office.

Poslovni nivo je predstavljen s poslovnim strežnikom, ki predstavlja jedro sistema in vsebuje vso poslovno funkcionalnost. Tehnično je zasnovan kot ASP.NET (več v: [5]) spletna storitev, ki teče znotraj internetnega strežnika IIS, z njo pa upravlja ASP.NET aplikacijski strežnik, ki je sestavni del Microsoft .NET platforme. Takšna arhitekturna zasnova ima vse prednosti spletnih storitev, poleg tega pa izkorišča zmogljivosti ASP.NET aplikacijskega strežnika, kot so na primer že vgrajeni varnostni elementi ter mehanizmi učinkovitega izkoriščanja sistemskih virov. Strežnik teče na operacijskih sistemih Microsoft Windows Server.

Podatkovni nivo temelji na uporabi relacijske podatkovne baze Microsoft SQL Server 2005 (več na: [33]).

Aplikacija je napisana v programskem jeziku C# in temelji na ogrodju Microsoft .NET Framework 4.0. Programski kodi odjemalca in poslovnega strežnika sta združeni vsaka v svojo rešitev. Rešitev je struktura za organizacijo projektov v programu Microsoft Visual Studio in je predstavljana z datoteko s končnico ».sln« (več v: [6]).

Razvijalci uporabljajo za razvoj aplikacije osebne računalnike z operacijskim sistemom Microsoft Windows 7. Glavno orodje za razvoj je Microsoft Visual Studio 2010 (več v: [7]). Za razvoj aplikacije potrebuje vsaj razvijalec lokalno kopijo produkta (iz repozitorija strežnika SVN).

1.2 Prvotno stanje sistema za upravljanje z izdajami

Razvijalci so izvorno kodo produkta »Insure« shranjevali na strežnik SVN. Ob potrebi za pripravo izdaje so preko nadzorne plošče v spletnem brskalniku zagnali proces izdelave izdaje, ki je v ozadju uporabil orodje za gradnjo programske opreme. Orodje je najprej osvežilo izvorno kodo z repozitorija strežnika SVN, nato je izvorno kodo v pravem zaporedju prevedlo, prevedeno aplikacijo preneslo na testni strežnik znotraj podjetja in na koncu pa izdelalo še paketno datoteko »version.zip«, ki je bila pripravljena za prenos k stranki.

Prvotni sistem za upravljanje z izdajami je torej že vseboval sistem za nadzor različic izvorne kode, strežnik z zvezno integracijo ter orodje gradnjo programske opreme.

1.2.1 Sistem za nadzor različic izvorne kode

Uporabljeni sistem za nadzor različic izvorne kode temelji na odprtokodnem sistemu Subversion (več v: [8]) oziroma SVN in sestoji iz repozitorija, kamor se shranjuje izvorno

kodo programske opreme. Njegova glavna prednost v primerjavi z navadnim datotečnim sistemom je hranjenje zgodovine posameznih sprememb za vsako od datotek. Posledično je kodo lažje nadzorovati, zelo enostavna pa je tudi povrnitev izvorne kode na prejšnje stanje v primeru napake. Zaradi enostavnega spajanja sprememb omogoča razvoj več razvijalcev hkrati na isti izvorni kodi, brez česar bi bil razvoj večjega produkta praktično nemogoč.

V podjetju se uporablja strežnik Visual SVN (podrobnosti v: [9]), ki je za Microsoft Windows okolje prilagojen strežnik SVN. Na strežniku je postavljen repozitorij »Product«, ki vsebuje izvorno kodo produkta »Insure«. Sprva se je za shranjevanje izvorne kode uporabljala le glavna linija razvoja imenovana »trunk«. Zaradi potreb po dostopu do strežnika tudi zunaj podjetja, se do njega dostopa s HTTPS avtentikacijo.

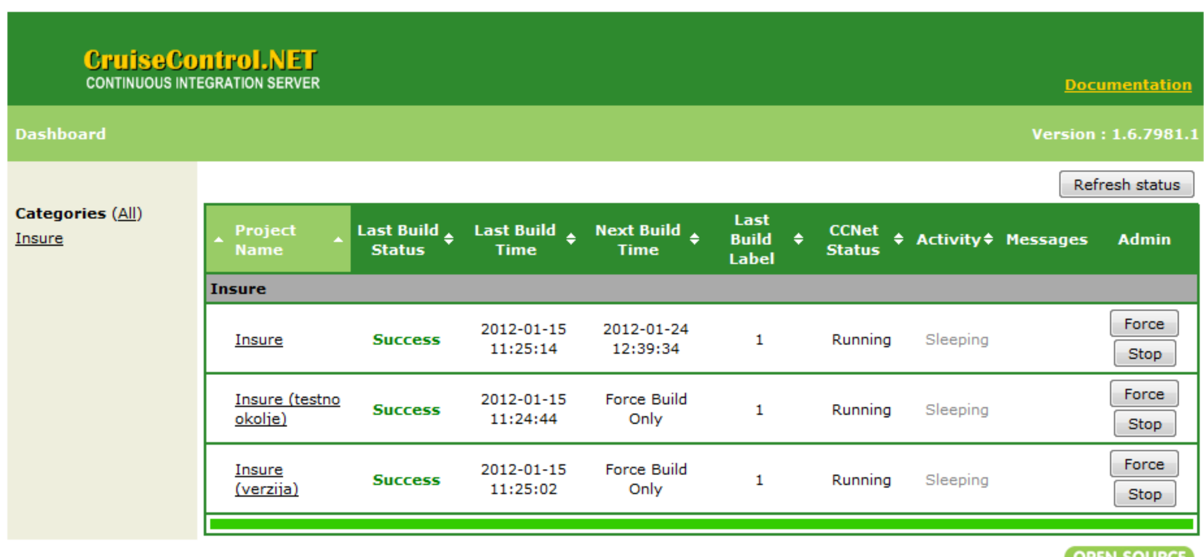
1.2.2 Strežnik z zvezno integracijo

V izrazu »zvezna integracija« se integracija nanaša na sestavljanje posameznih programskih komponent, zveznost pa se nanaša na odstonost časovnih omejitev (vir: [36]).

Strežnik z zvezno integracijo izvaja ponavljajoče procese, ki so v osnovi enostavni, vendar zaradi pogostosti pripomorejo h kvaliteti programske opreme in hkrati zmanjšujejo čas, ki je potreben za pripravo izdaje aplikacije. Tipičen primer enostavnega procesa je klic orodja za gradnjo programske opreme ob vsaki spremembi kode na repozitoriju strežnika SVN. Tako se napake pri prevajanju odpravlja sproti, kar je enostavnejše, kot odpravljanje večjega števila napak pred zahtevo za namestitev izdaje, kar je poudarjeno tudi v [10, 34, 35, 36].

V podjetju se za strežnik z zvezno integracijo uporablja odprtokodni program CruiseControl.NET (več v [11]) ali krajše CCNET, ki je nameščen na okolju z operacijskim sistemom Microsoft Windows. Sprva se ga je uporabljalo za avtomatsko prevajanje izvorne kode ob vsaki spremembi le-te na repozitoriju strežnika SVN, za pripravo testnega okolja znotraj podjetja ter za pripravo izdaje. Interakcija uporabnika s strežnikom je enostavna in intuitivna, saj se vse ukaze izvaja preko nadzorne plošče, ki je dostopna preko spletnega brskalnika (Slika 2).

V osnovi se lahko razdeli projekte v dve skupini: projekti, katere se lahko zažene samo na zahtevo (na primer z namenom namestitve izdaje) ter projekti, ki se lahko zaženejo tudi avtomatično (ob točno določeni uri, po uspešnem zaključku drugega projekta, ob zaznani spremembi izvorne kode na strežniku SVN).



Slika 2: nadzorna plošča strežnika CCNET

1.2.2.1 Upravljanje s projekti

Projekte na nadzorni plošči se dodaja in ureja v nastavitveni datoteki XML »ccnet.config«.

```
<project name="Insure">
    <category>Insure</category>
    <workingDirectory>$(WorkingDir)Product\Insure\</workingDirectory>
    <triggers>
      <intervalTrigger seconds="30" />
    </triggers>
    <modificationDelaySeconds>20</modificationDelaySeconds>
    <sourceControl type="svn">
        <autoGetSource>false</autoGetSource>
        <trunkUrl>https://svnserver/Product/Insure/trunk/</trunkUrl>
        <workingDirectory>.</workingDirectory>
        <executable>$(SvnExe)</executable>
        <username>CruiseControl</username>
        <password>$(SvnPassword)</password>
    </sourceControl>
    <tasks>
        <nant>
            <executable>$(NAntExe)</executable>
            <buildFile>insure.build</buildFile>
            <targetList>
                <target>build.ccnet</target>
            </targetList>
            <buildTimeoutSeconds>300</buildTimeoutSeconds>
        </nant>
    </tasks>
</project>
```

Slika 3: nastavitev projekta »Insure« v nastavitveni datoteki »ccnet.config«

Primer projekta na strežniku CCNET je projekt »Insure« (Slika 3). Projektu se najprej določi ime, katero se vidi na nadzorni plošči ter kategorijo, kar pripomore k večji preglednosti

nadzorne plošče. Preko sprožilcev se določi kaj proži zagon projekta. S sprožilcem »intervalTrigger« v kombinaciji z značko »sourceControl« povemo strežniku na koliko sekund naj preveri ali je prišlo do spremembe izvorne kode na glavni liniji razvoja produkta »Insure« na repozitoriju strežnika SVN. Z značko »modificationDelaySeconds« povemo strežniku koliko sekund naj preteče od proženja projekta na podlagi sprožilcev do dejanskega zagona projekta. V primeru proženja, se zaženejo naloge znotraj značke »tasks«, kjer povemo strežniku kateri ukaz naj izvede. V konkretnem primeru se z orodjem »NAnt« znotraj datoteke »insure.build« kliče ukaz »build-ccnet«. Čas izvajanja projekta je omejen z značko »buildTimeOutSeconds«.

1.2.3 Orodje za gradnjo programske opreme

Proces priprave izdaje je sestavljen iz več korakov, na primer: osveževanje izvorne kode iz repozitorija strežnika SVN, prevajanje izvorne kode v pravilnem vrstnem redu, prenos prevedene kode na določeno lokacijo, kjer se lahko aplikacijo testira, pakiranje prevedenih datotek v paketno datoteko. Ker je ta proces ponavljajoč dogodek, je smiselno uporabiti orodje, ki nam to delo olajša, pohitri ter zmanjša možnost napake.

V podjetju se za gradnjo programske opreme uporablja zastonjsko orodje »NAnt« (več o orodju v [12]). Vse korake, ki jih mora program izvesti zabeležimo v datoteki s končnico ».build«, ki temelji na sintaksi XML.

```
<?xml version="1.0"?>
<project name="" default="">
    <property name="" value="" />
    <target name="" description="" />
</project>
```

Slika 4: osnovna struktura skripte za orodje »NAnt«

Skripta je v osnovi sestavljena iz značk »project«, »property« in »target« (Slika 4). Značka »project« vsebuje ime projekta ter ime privzete tarče v skripti. Privzeta tarča se pokliče v primeru, ko orodju NAnt podamo le ime datoteke in ne tudi ime tarče. Značka »property« služi kot spremenljivka kamor se lahko shranjuje vrednosti. Značka »target« služi kot funkcija, ki ji določimo ime ter opis, znotraj nje pa izvedemo določeno funkcionalnost ali pa izvedemo klic neke druge značke »target«.

1.2.3.1 Skripta za prevajanje kode

Primer skripte je skripta za prevajanje kode produkta »Insure«, katero se zažene s klicem tarče »build.ccnet«. Skripta najprej razveljavi vse morebitne lokalne spremembe izvorne kode nato pa kodo osveži z repozitorija strežnika SVN. Naslednji korak je brisanje datotek v mapah »bin« in »obj«, ki so rezultat prejšnjega prevajanja izvorne kode, na koncu pa še prevede izvorno kodo (Slika 5).

```
<target name="build" description="Empties bin and debug folders and compiles the code">
    <call target="clean" />
    <call target="compile" />
</target>
<target name="build.ccnet" description="Reverts code, empties bin and debug folders and compiles the code">
    <call target="update.revert" />
    <call target="clean" />
    <call target="compile" />
</target>
<target name="update.revert" description="Reverts and updates source code from repository">
    <exec program="svn.exe">
        <arg line="revert . -R" />
    </exec>
    <exec program="svn.exe">
        <arg value="update" />
    </exec>
</target>
<target name="clean" description="Empties bin and obj folders">
    <delete failonerror="false">
        <fileset>
            <include name="**/bin/**" />
            <include name="**/obj/**" />
            <exclude name="**/*.refresh" />
        </fileset>
    </delete>
</target>
<target name="compile" description="Compiles WebService and Client">
    <msbuild project="WebService\WebService.sln" verbosity="minimal" />
    <msbuild project="Client\Client.sln" verbosity="minimal" />
</target>
```

Slika 5: skripta NAnt za prevajanje kode v skripti »insure.build«

Bistven del prikazane skripte je prevajanje kode, kar se izvede z značko »msbuild«, kateri kot parameter podamo datoteko Microsoft Visual Studio rešitve (.sln).

1.2.3.2 Skripta za postavitev testnega okolja

Predpogoj za postavitev testnega okolja znotraj podjetja je uspešno prevedena izvorna koda. Ker je na strežniku nastavljeno avtomatsko prevajanje izvorne kode ob vsaki spremembi le-te, je potrebno pri postavitvi testnega okolja poskrbeti le za prenos prevedene kode na pravo lokacijo, kar naredimo s klicem tarče »build.staging« v skripti »insure.build« (Slika 6).

```
<!-- staging functions -->
<target name="build.staging" description="Prepares staging">
    <call target="clean.staging.folders" />
    <call target="staging.copy.all" />
</target>
<target name="clean.staging.folders">
    <delete dir="${staging.folder.default}\${staging.folder.client}" failonerror="false" />
    <delete dir="${staging.folder.default}\${staging.folder.webservice}" failonerror="false" />

    <mkdir dir="${staging.folder.default}\${staging.folder.client}" failonerror="false" />
    <mkdir dir="${staging.folder.default}\${staging.folder.webservice}" failonerror="false" />
</target>
<target name="staging.copy.all" description="Copies Client and WebService">
    <property name="staging.target.dir" value="${staging.folder.default}\${staging.folder.client}" />
    <call target="staging.copy.client" />

    <property name="staging.target.dir" value="${staging.folder.default}\${staging.folder.webservice}" />
    <call target="staging.copy.webservice" />
</target>
<target name="staging.copy.client">
    <copy todir="${staging.target.dir}">
        <fileset basedir="Client\bin\Debug">
            <exclude name="**/*.vshost.exe" />
            <exclude name="**/en/**" />
            <include name="**/*.exe" />
            <include name="**/*.dll" />
            <include name="**/*.flt" />
            <include name="**/*.ini" />
        </fileset>
    </copy>
</target>
<target name="staging.copy.webservice">
    <copy todir="${staging.target.dir}">
        <fileset basedir="WebService">
            <include name="**/*.asax" />
            <include name="**/*.asmx" />
            <include name="**/*.aspx" />
            <include name="**/*.dll" />
            <include name="**/*.cs" />
            <include name="**/*.svc" />
        </fileset>
    </copy>
</target>
```

Slika 6: skripta NAnt za postavitev testnega okolja v skripti »insure.build«

1.2.3.3 Skripta za pripravo izdaje

Ob predpostavki, da je na strežniku nastavljeno avtomatsko prevajanje izvorne kode ob vsaki spremembi le-te, je potrebno za pripravo izdaje klicati le tarčo »version.insure« (Slika 7). Ta najprej nastavi ustrezno ikono ter logotip podjetja, nato na novo postavi testno okolje in na koncu cel program zapakira v paket »version.zip«. Iz skripte je razvidno, da se tako za testiranje kot tudi za pripravo izdaje uporablja testno okolje. Tak način delovanja je sprva zadostoval našim potrebam, saj smo aplikacijo testirali le razvijalci in to kar na svojih računalnikih. Posledično testno okolje sprva ni imelo večjega pomena.

```
<!-- version functions -->
<target name="version.insure" description="Builds version">
    <call target="set.icons" />
    <call target="build.staging" />
    <call target="version.insure.zip" />
</target>
<target name="version.insure.zip">
    <property name="version.zip.basedir" value="${staging.folder.default}" />
    <call target="version.zip" />
</target>
<target name="version.zip" description="Makes zip for a release">
    <zip zipfile="${version.zip.basedir}\Version.zip">
        <fileset basedir="${version.zip.basedir}">
            <exclude name="**/*.config" />
            <include name="**/*" />
        </fileset>
    </zip>
</target>

<!-- icons -->
<target name="set.icons">
    <property name="logo" value="logo.bmp" />
    <property name="icon" value="logo.ico" />
    <call target="set.icons.base" />
</target>
<target name="set.icons.base">
    <if test="${not property::exists('logo') or not property::exists('icon')}">
        <fail message="Properties 'logo' and 'icon' are not set!" />
    </if>

    <copy file="${local.path.folder.resources}\${logo}" tofile="Client\Resources\Bitmaps\logo.bmp" overwrite="true" />
    <copy file="${local.path.folder.resources}\${icon}" tofile="Client\Resources\Icons\app.ico" overwrite="true" />
    <copy file="${local.path.folder.resources}\${icon}" tofile="Client\Client\App.ico" overwrite="true" />
    <msbuild project="Client\Client.sln" verbosity="minimal" />
</target>
```

Slika 7: skripta NAnt za pripravo izdaje v nastavitveni datoteki »insure.build«

2 Problemi in pomanjkljivosti pri upravljanju z izdajami

Prvotni sistem upravljanja z izdajami je imel več pomanjkljivosti. Nekaj jih je bilo prisotnih že od vsega začetka zaradi napačne zasnove sistema, druge pa so posledica rasti produkta. V nadaljevanju sledi opis vsake od pomanjkljivosti.

2.1 Avtomatsko povečevanje oznake različice izdaje

Oznaka različice izdaje služi kot identifikator določenega stanja programske opreme. V procesu označevanja se programski opremi dodeli enolične številke ter imena. Načeloma se dodeljene številke povečujejo in s tem odražajo razvoj programske opreme (vir: [13]).

Skupaj z različicami izdaj je smiselno voditi seznam dodelav ter popravkov narejenih na programski opremi. Preko tega seznama lahko stranka spremlja razvoj produkta. Ustrezno povečevanje različice izdaje je koristno tudi za razvijalce, saj preko nje lažje ugotovijo katero stanje izvorne kode je nameščeno na določenem okolju.

V podjetju se različico produkta »Insure« označuje s štirimi številkami, ki so ločene med sabo s pikami (x.y.z.w). Trenutno različico produkta si interno shranjujemo v datoteko »version.txt«, ki je poleg ostalega produkta shranjena na repozitorij strežnika SVN. Datoteka služi kot osnova na podlagi katere se nastavi ustrezno oznako v vseh ostalih programskih komponentah. Proces povečevanja različice je sestavljen iz branja datoteke »version.txt«, povečevanja različice, zapisa nove različice v vse datoteke, ki opisujejo programske komponente (datoteke »AssemblyInfo.cs«) ter zapisa nazaj v datoteko »version.txt«. Na koncu se vse spremembe shranijo na repozitorij strežnika SVN.

Produkt »Insure« je bil sprva nameščen pri eni sami stranki, zato je bilo ugotavljanje katero stanje programske opreme je nameščeno pri stranki trivialno.

Povečevanje oznake različice je bilo sicer podprto preko paketne datoteke, vendar ni bilo vključeno v sistem za izdelavo izdaj. Paketno datoteko se je ročno zagnalo na lokalnem okolju, kjer je bila shranjena lokalna kopija izvorne kode produkta »Insure«. Po končanem procesu povečevanja oznake, je bilo potrebno na repozitorij strežnika SVN ročno shraniti spremembe na datotekah »version.txt« in »AssemblyInfo.cs«. Oznako različice se je povečalo le ob večjih spremembah, ki smo jih želeli poudariti. Dokumenta, kjer bi si beležili seznam sprememb ob vsaki novi različici izdaje, se ni pisalo.

Ob pridobitvi nove stranke pa se je izkazalo, da je smiselno oznake povečevati ob vsaki namestitvi. Zaradi pozabljenega zagona paketne datoteke je prišlo do situacije, ko sta bili pri obeh strankah isti oznaki različice izdaje, kljub temu, da je bilo stanje programske opreme različno. To je povzročilo zmedo in paniko med razvijalci, saj smo sprva pomislili, da je bila na okolje ene izmed strank nameščena napačna različica izdaje.

Poleg slabšega internega nadzora nad različicami izdaj, se je problem nepovečane oznake različice izdaje odrazil tudi kot nezadovoljstvo stranke, saj stranka v primeru nepovečane oznake ni vedela ali dela že z novo ali še s staro različico izdaje.

2.2 Izdajanje stabilnih izdaj in nadzor nad vključeno izvorno kodo

Podjetje je za razvoj produkta »Insure« na repozitoriju strežnika SVN uporabljalo eno samo linijo razvoja, kamor so se shranjevale vse spremembe na produktu, tako nove funkcionalnosti, kot tudi odprave napak. Iz te linije so se izdelovale tudi vse izdaje.

Pogosto je prišlo do situacije, ko se je shranila izvorna koda z novo funkcionalnostjo na repozitorij strežnika SVN, ki še ni bila dovolj testirana s strani razvijalca. Prav tako je lahko na novi funkcionalnosti delalo več razvijalcev hkrati in posledično so se morale na repozitorij shranjevati delne rešitve. V primeru, da je ravno ob enem izmed takih trenutkov stranka naletela na kritično napako na produkcijskem okolju, je bilo potrebno čim hitreje odpraviti napako v izvorni kodi, ter na novo namestiti izdajo. Zaradi ne dovolj testiranih sprememb in pa delnih rešitev saj so več ali manj vsi razvijalci pregledovali spremembe na repozitoriju, ocenjevali potencialnost napake ter testirali aplikacijo. V nekaterih primerih je bilo potrebno spremembe tudi razveljaviti in jih po namestitvi na novo shraniti v repozitorij strežnika SVN.

Kljub pazljivosti se je nekajkrat z odpravo ene napake na produkcijskem okolju pojavilo več novih napak. Posledično se je porabilo veliko časa za odkrivanje napak in za pripravo ter nameščanje več izdaj.

2.3 Sinhrone spremembe podatkovnega modela baze in izdaje aplikacije

V sklopu razvoja produkta se spreminja tako izvorna koda kot tudi podatkovni model relacijske baze. Tipično spremembe podatkovnega modela narekujejo razvijalci aplikacije, ki potrebujejo za njen razvoj ustrezno podporo na podatkovnem nivoju. Zaradi konsistentnosti podatkovnega modela baze je smiselno, da podatkovni model spreminja čim manj oseb, vsekakor pa ne vsi razvijalci.

Pravilno delovanje aplikacije na produkcijskem okolju se lahko zagotovi le v primeru, ko sta namestitev aplikacije in prilagajanje podatkovnega modela med sabo usklajena. V določenih primerih se lahko prilagodi podatkovni model baze na način, ki omogoča delo tako z staro kot tudi z novo različico izdaje.

V podjetju se za vzdrževanje podatkovnega modela uporablja program Sybase Power Designer (več v: [14]), v katerem je shranjena celotna struktura podatkovne baze. Za vse spremembe v fizičnem podatkovnem modelu (PDM) je odgovoren njen skrbnik. Prilagoditev PDM je dokaj enostavna – skrbnik vnese spremembe preko uporabniškega vmesnika programa, ta pa kot rezultat generira skripto SQL, ki se jo poimenuje na način iz katerega je nedvoumno določeno zaporedje izvajanja skript.

Sprva se je spreminjalo PDM pred namestitvijo nove izdaje aplikacije. Tak pristop je možen zaradi omogočenega dostopa do produkcijske baze stranke. Skrbnik PDM je prejel navodilo za spremembo, katero je vnesel v program Sybase Power Designer, ta pa mu je vrnil ustrezno skripto SQL. Po potrebi je skripto dopolnil tako, da je bil zagon skripte varen tudi brez namestitve nove različice izdaje aplikacije. Nato je skripto ustrezno poimenoval ter zagnal na

vseh podatkovnih bazah, tako znotraj podjetja kot pri stranki. Po končanem delu je razvijalcu sporočil, da je PDM urejen, ta pa je takrat svojo spremembo izvorne kode shranil na repozitorij strežnika SVN. Ob izdelavi naslednje izdaje, sta bili ustrezno prilagojeni obe komponenti, tako PDM kot izvorna koda. V primeru, da je bilo potrebno namestiti novo različico izdaje hkrati z ustrezno skripto SQL, se je moral skrbnik PDM prilagoditi in zagnati skripto takoj po namestitvi izdaje. Kljub pazljivosti je že prišlo do situacije, ko se je na produkcijsko okolje namestilo novo različico izdaje pred zagonom ustrezne skripte SQL.

Tak pristop poenostavi delo skrbniku PDM in je mogoč samo pod pogojem, da ima podjetje dostop neposredno do podatkovne baze stranke, kar pa seveda ni najbolj pogosta praksa. Prav tako sprememba PDM nima neposredne povezave s spremembo v izvorni kodi. V primeru iskanja razloga za spremembo PDM je vse odvisno od dokumentacije, ki jo vodi skrbnik PDM, ta pa je lahko nepopolna.

Z rastjo produkta, razvojne ekipe ter števila strank, postaja verjetnost za napako pri omenjenem pristopu prevelika. Prav tako je zelo verjetno, da stranka zaradi varovanja podatkov ne bo omogočila neposrednega dostopa do podatkovne baze.

2.4 Nameščanje izdaj v različna okolja z različnimi zahtevami

Pri razvoju produkta se teži k temu, da se ga implementira pri več različnih strankah. Če je produkt zelo obsežen, mora biti do neke mere prilagodljiv, saj stranke večinoma ne želijo spremeniti delovnega procesa.

Informacijski sistem za podporo zavarovalnic je zelo obsežen in podpira veliko procesov. Osnovni procesi so v različnih zavarovalnicah pogosto enaki. Na primer definicija zavarovanj, produktov, sklepanje pogodb, dodajanje in izbira zavarovanca, zavarovalca, itd. Obstajajo pa procesi, ki so za vsako zavarovalnico specifični.

Produkt mora biti v osnovi zasnovan na način, ki omogoča prilagoditve glede na zahteve strank. Temu primerno mora biti urejen tudi sistem za izdelavo izdaj, ki mora nedvoumno vedeti za katero stranko se pripravlja izdaja, saj lahko le tako zagotovi ustrezne prilagoditve produkta.

V podjetju je produkt »Insure« nastal na osnovi potreb ene same zavarovalnice. Posledično podpora za prilagoditve ni bila implementirana. S pridobitvijo nove stranke, se je izkazala potreba po prilagajanju produkta.

2.5 Kvaliteta namestitve in nadzor nad nameščanjem

Namestitev izdaje produkta je trivialna operacija, saj so v naprej znani vsi potrebni koraki ter cilji. V primeru napake pri namestitvi, je potrebno v najkrajšem možnem času zagotoviti prejšnje stanje. Zaradi ponovljivosti je smiselno ta postopek čim bolj avtomatizirati.

Vsaka zavarovalnica ima med drugimi zaposlene tudi skrbnike informacijskega sistema, ki urejajo vse potrebno za delovanje sistema. Zaradi varovanja podatkov je zunanji dostop do produkcijskega okolja pogosto onemogočen. V vsakem primeru je smiselno, da nameščanje

izdaj opravljajo skrbniki in ne podjetje, ki razvija produkt. Posledično mora biti postopek nameščanja toliko bolj poenostavljen ter avtomatiziran, da se človeški vpliv ter z njim verjetnost za napako zmanjša na najmanjšo možno vrednost.

Na zavarovalnici, kjer imamo dostop do oddaljenega namizja preko VPN povezave, smo produkt »Insure« nameščali sami. Produkt smo imeli zapakiran v paket »version.zip«, katerega smo na produkcijskem okolju razširili ter na koncu zagnali paketno datoteko, katera je poskrbela za prenos datotek na predvideno lokacijo. Za varnostno kopijo trenutne različice izdaje je moral poskrbeti tisti, ki je izdajo nameščal. Postopek je dokaj enostaven za osebo, ki izdajo namešča pogosto, vsekakor pa ni primeren za predajo v roke skrbnikov informacijskega sistema na strani zavarovalnice.

2.6 Avtomatsko testiranje aplikacije

Izvorna koda produkta se konstantno spreminja. Lahko gre za izdelavo nove funkcionalnosti ali pa za dodelavo stare, v vsakem primeru obstaja verjetnost za nastanek napake. Stranke so po pričakovanjih najbolj tolerantne do napak, ki se pojavijo pri novih funkcionalnostih. Malo manj tolerantne so do napak, ki nastanejo s prilagajanjem funkcionalnosti, nikakor pa ne morejo tolerirati napak, ki nastanejo kot posledica dodelave, ki se konkretne funkcionalnosti ne tiče neposredno, ali pa napak, ki so, kar se tiče njih, nastale brez kakršne koli zahteve za spremembo.

V splošnem ima vsak razvijalec nalogo, da vsako spremembo v izvorni kodi testira na lokalnem okolju preden jo prenese na skupni repozitorij strežnika SVN. Večinoma gre za test preko uporabniškega vmesnika aplikacije. Lahko se zgodi, da zaradi časovne stiske razvijalec sprememb ne testira dovolj podrobno, zato je potrebno pred vsako pripravo izdaje narediti test vsaj osnovnih funkcionalnosti aplikacije. Ročno testiranje preko uporabniškega vmesnika je časovno potratno. Poleg tega postane sčasoma zaradi enostavnosti, količine testov in ponovljivosti nezanimivo, kar povečuje verjetnost nepopolnega testiranja. Posledično je smiselno teste uporabniškega vmesnika avtomatizirati.

Poleg testiranja uporabniškega vmesnika, je smiselno testirati tudi posamezne funkcionalnosti v izvorni kodi. V splošnem se razvijalcem ne zdi smiselno pisati teste za nove funkcionalnosti, saj poleg dodatne tolerance s strani stranke v trenutku razvoja ne vidijo razloga zakaj bi napisali test za funkcionalnost, ki sama po sebi deluje. Kljub temu je teste najbolj smiselno napisati takoj, saj takrat razvijalec natančno pozna na novo izdelano funkcionalnost, dolgoročno pa nam taki testi preprečijo napake, ki se pojavijo pri spreminjanju obstoječe funkcionalnosti. Smiselno je uvesti avtomatsko testiranje s pomočjo ogrodja za testiranje enot. Odprtokodni primer takega programa je NUnit (podrobnosti v: [15]). Paziti moramo le, da testi ne postanejo sami sebi namen.

V podjetju se je testiranje pred namestitvijo izdaje opravljalo ročno, kar je privedlo do marsikatere napake na produkcijskem okolju ter posledično nezadovoljstva stranke.

2.7 Zgodovina namestitev različic izdaj

Produkt se konstanto spreminja. Lahko gre za novo funkcionalnost ali pa za predelavo oziroma izboljšavo stare funkcionalnosti. Ob vsaki taki spremembi je nesmiselno namestiti novo izdajo pri vsaki stranki. Ko se pojavi potreba po namestitvi nove izdaje, je sprememb, katere niso bile še nikoli na produkcijskem okolju, veliko. Vse te spremembe so glavni kandidati za nastanek napake. Kljub testiranju lahko pride do situacije, ko se po namestitvi nove izdaje izkaže, da ima le-ta preveč napak. V takem primeru je potrebno na produkcijsko okolje namestiti prejšnjo različico izdaje. To ne predstavlja nobenega problema, če imamo na za vsako stranko urejeno zgodovino namestitev izdaj.

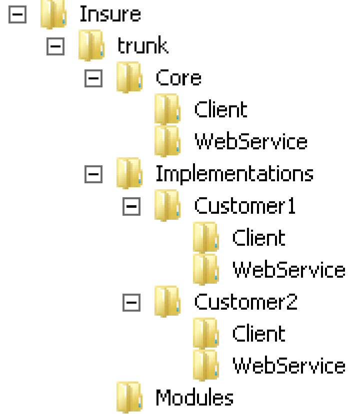
V podjetju smo nameščali izdaje ročno in prave zgodovine izdaj ni bilo. Za varnostno kopijo trenutne različice izdaje je moral poskrbeti tisti, ki je izdajo nameščal. V primeru, da je to pozabil storiti, je bil proces pridobitve prejšnje izdaje dokaj zamuden in kompleksen.

3 Odprava problemov in izboljšave

Prvotni sistem upravljanja z izdajami je imel več pomanjkljivosti, ki so bile opisane v prejšnjem poglavju. V nadaljevanju sledi opis več korakov s katerimi smo omenjene pomanjkljivosti odpravili.

3.1 Uvedba modulizacije aplikacije

V podjetju smo bili zaradi pridobitve novih strank prisiljeni preurediti produkt na način, s katerim ga lahko enostavno prilagodimo potrebam strank, hkrati pa obdržimo enotno jedro aplikacije. Posledično sem celoten produkt »Insure« razdelil na več vsebinskih sklopov: jedro, moduli, implementacije. Skladno sem dodal 3 nove mape na datotečnem sistemu: »Core«, »Modules« in »Implementations« in na repozitoriju strežnika SVN (Slika 8).



Slika 8: Nova struktura map za produkt »Insure« na repozitoriju strežnika SVN

Celotni produkt sem sprva prenesel v mapo »Core«. Za vsako stranko sem naredil novo t.i. implementacijo produkta - prazno rešitev v sklopu programa Microsoft Visual Studio, ki vsebuje reference na programske komponente jedra in je, podobno kot jedro, razdeljena na dve programski komponenti: prilagoditve za odjemalca ter prilagoditve za poslovni strežnik. Vsaka od njiju je označena s posebnim atributom preko katerega jedro prepozna programsko komponento kot implementacijo in ima definiran dogodek, ki po uspešno končanem prevajanju izvorne kode prenese prevedeno programsko komponento na ustrezno mesto v jedru. Vsaka implementacija mora vsebovati poleg rešitve tudi skripto NAnt, ki se jo rabi za avtomatsko prevajanje kode ob vsaki spremembi ter za pripravo izdaj. Tako kot pri skripti za prevajanje jedra, je tudi tukaj (slika 9) bistven del skripte prevajanje kode, kar se izvede z značko »msbuild«, kateri kot parameter podamo datoteko Microsoft Visual Studio rešitve.

```
<project name="Insure.Customer1" default="build">
    <target name="build">
        <call target="clean" />
        <call target="compile" />
    </target>
    <target name="build-ccnet">
        <call target="build" />
    </target>
    <target name="clean">
        <delete failonerror="false">
            <fileset>
                <include name="**/bin/**" />
                <include name="**/obj/**" />
            </fileset>
        </delete>
    </target>
    <target name="compile">
        <msbuild project="Insure.Customer1.sln" verbosity="minimal" />
    </target>
</project>
```

Slika 9: skripta NAnt »Insure.Customer1.build« za prevajanje implementacije stranke 1

Za gradnjo implementacij imamo pripravljeno predlogo »Microsoft Visual Studio Template« (več v: [16]), preko katere so izdelane vse rešitve implementacij s čimer imamo zagotovljeno podobnost osnovne strukture ter ustrezno skripto NAnt.

V jedru smo dodali funkcionalnost, ki zna v času delovanja brati programske komponente, ki so s posebnim atributom označene kot implementacija. V vsaki programski komponenti je ena ali več prilagoditev, vsaka od njih mora ali izhajati iz določenega objekta ali pa mora implementirati določen vmesnik iz jedra. Poleg tega mora imeti vsaka prilagoditev še poseben atribut po meri s katerim določimo rang. V primeru, da jedro najde več prilagoditev za isto zadevo, izbere tisto, ki ima največji rang.

Ob zahtevi po prilagoditvi s strani stranke, se med sabo primerja zahtevano spremembo ter trenutno delovanje funkcionalnosti. Na podlagi kratke analize se odloči kateri način je bolj splošen in spada v osnovne funkcionalnosti produkta in kateri bo prenesen kot izjema v implementacijo. Seveda se lahko pride tudi do sklepa, da sta oba načina izjemi in tako prenesemo vsakega v svojo implementacijo.

Določeno prilagoditev lahko zahteva več različnih strank zaradi česar smo podprli tudi možnost izdelave modulov. Modul ima skoraj vse lastnosti enake kot implementacija. Razlika je le v atributu po meri, s katerim določimo rang prilagoditve, katerega modul nima. Posledično modul sam po sebi nima nobenega vpliva na delovanje aplikacije. Modul deluje le v kombinaciji z implementacijo, kjer se ga referencira in doda rang prilagoditve.

Zaradi splošnosti modulov jih prevajamo vedno, tako ob spremembi izvorne kode kot tudi ob pripravi izdaje. Zato sem napisal ustrezno skripto, ki najde skripto NAnt vsakega modula ter kliče privzeto tarčo, ki modul prevede (Slika 10). Bistveni del skripte je zanka »foreach«, ki se sprehodi po vseh mapah znotraj mape »Modules« in išče vse NAnt skripte (datoteke s končnico »build«). Za vsako najdeno datoteko nato požene želeno tarčo, če jo seveda skripta vsebuje.

```xml
<target name="build.modules" description="Finds all modules build files and calls desired target" >
    <property name="module.target" value="" unless="${property::exists('module.target')}" />

    <foreach item="File" property="filename">
        <in>
            <items>
                <include name="${InsureModulesFolder}\**\*.build" />
            </items>
        </in>
        <do>
            <property name="ModuleHasTarget" value="false"/>
            <property name="ModuleHasTarget" value="true" if="${module.target == ''}"/>
            <if test="${ModuleHasTarget=='false'}">
                <foreach item="Line" in="${filename}" property="line">
                    <property name="ModuleHasTarget" value="true" if="${string::contains(line, module.target)}"/>
                </foreach>
            </if>

            <nant buildfile="${filename}" target="${module.target}" failonerror="false" if="${ModuleHasTarget=='true'}" />
        </do>
    </foreach>
</target>
```

Slika 10: skripta NAnt za iskanje ter prevajanje vseh modulov

Uredil sem tudi nastavitveno datoteko za strežnik CCNET, ki sproži prevajanje modulov ob spremembi izvorne kode na repozitoriju strežnika SVN ter ob uspešni prevedbi jedra (Slika 11). S sprožilcem »projectTrigger« sem nastavil odvisnost projekta »Insure Modules (trunk)« od projekta »Insure (trunk)«, torej v primeru uspešno izvedenega projekta »Insure (trunk)« se bo zagnal še projekt za prevajanje modulov.

```xml
<project name="Insure Modules (trunk)">
    <category>Insure</category>
    <workingDirectory>$(WorkingDir)Product\Insure\</workingDirectory>
    <triggers>
        <intervalTrigger seconds="60" />
        <projectTrigger project="Insure (trunk)">
            <triggerStatus>Success</triggerStatus>
            <innerTrigger type="intervalTrigger" seconds="30" buildCondition="ForceBuild" />
        </projectTrigger>
    </triggers>
    <modificationDelaySeconds>20</modificationDelaySeconds>
    <sourcecontrol type="svn">
        <trunkUrl>https://svnserver/Product/Insure/trunk/Modules/</trunkUrl>
        <workingDirectory>.</workingDirectory>
        <executable>$(SvnExe)</executable>
        <username>CruiseControl</username>
        <password>$(SvnPassword)</password>
    </sourcecontrol>
    <tasks>
        <nant>
            <executable>$(NAntExe)</executable>
            <buildFile>complete.build</buildFile>
            <targetList>
                <target>build.modules</target>
            </targetList>
            <buildTimeoutSeconds>120</buildTimeoutSeconds>
        </nant>
    </tasks>
</project>
```

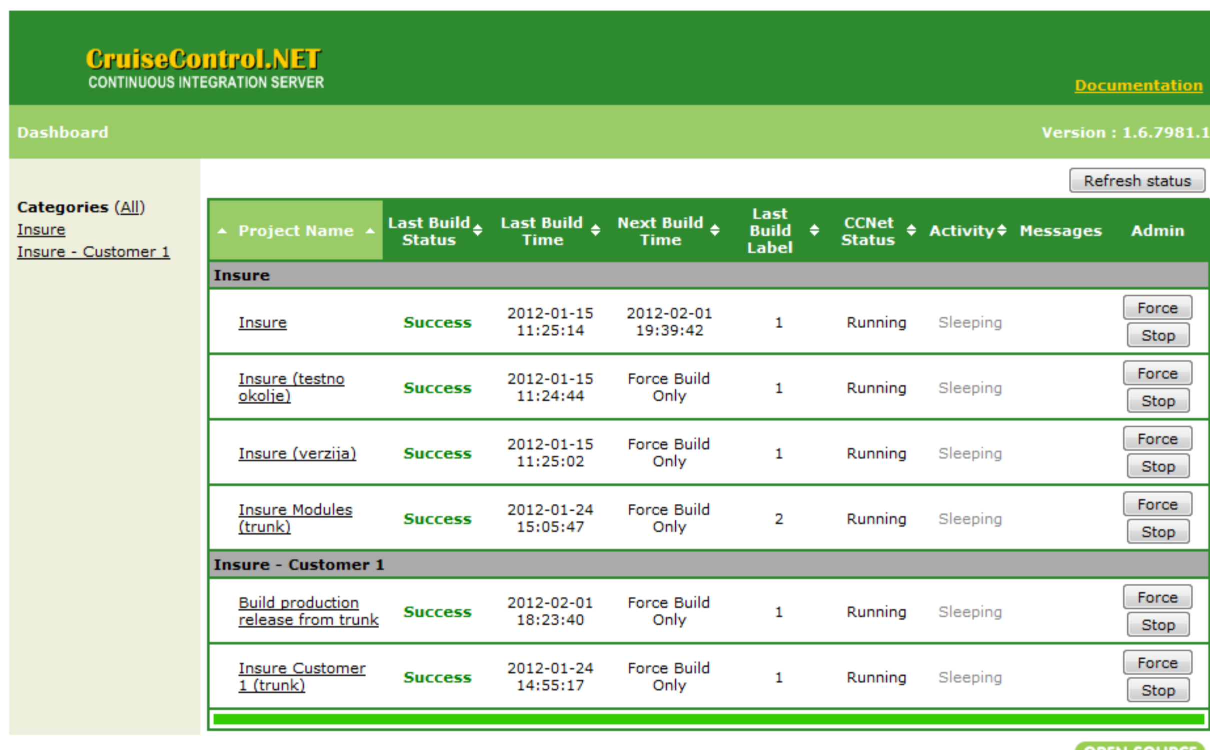
Slika 11: nastavitev projekta »Insure Modules (trunk)« v nastavitveni datoteki »ccnet.config«

Uredil sem tudi prevajanje implementacij in sicer ob vsaki spremembi izvorne kode na repozitoriju strežnika SVN ter ob uspešni prevedbi modulov (le-ti so sproženi z uspešno prevedbo jedra) (Slika 12).

```xml
<project name="Insure Customer 1 (trunk)">
    <category>Insure - Customer  1</category>
    <workingDirectory>$(WorkingDir)Product\Insure\Implementations\Customer1\</workingDirectory>
    <sourcecontrol type="svn">
        <trunkUrl>https://svnserver/Product/Insure/trunk/Implementations/Customer1/</trunkUrl>
        <workingDirectory>.</workingDirectory>
        <executable>$(SvnExe)</executable>
        <username>CruiseControl</username>
        <password>$(SvnPassword)</password>
    </sourcecontrol>
    <triggers>
        <intervalTrigger seconds="60" />
        <projectTrigger project="Insure Modules (trunk)">
            <triggerStatus>Success</triggerStatus>
            <innerTrigger type="intervalTrigger" seconds="30" buildCondition="ForceBuild" />
        </projectTrigger>
    </triggers>
    <modificationDelaySeconds>20</modificationDelaySeconds>
    <tasks>
        <nant>
            <executable>$(NAntExe)</executable>
            <buildFile>customer1.build</buildFile>
            <targetList>
            <target>build-ccnet</target>
            </targetList>
            <buildTimeoutSeconds>600</buildTimeoutSeconds>
        </nant>
    </tasks>
</project>
```

Slika 12: nastavitev projekta »Insure Customer 1 (trunk)« v nastavitveni datoteki »ccnet.config«

Zaradi različnih implementacij se sedaj izdaje z istim jedrom ter različno prilagoditvijo za stranko med sabo razlikujejo. Posledično sem na strežniku CCNET naredil za vsako stranko nov projekt za pripravo produkcijske izdaje (Slika 13).



Project Name	Last Build Status	Last Build Time	Next Build Time	Last Build Label	CCNet Status	Activity	Messages	Admin
Insure								
Insure	Success	2012-01-15 11:25:14	2012-02-01 19:39:42	1	Running	Sleeping		Force Stop
Insure (testno okolje)	Success	2012-01-15 11:24:44	Force Build Only	1	Running	Sleeping		Force Stop
Insure (verzija)	Success	2012-01-15 11:25:02	Force Build Only	1	Running	Sleeping		Force Stop
Insure Modules (trunk)	Success	2012-01-24 15:05:47	Force Build Only	2	Running	Sleeping		Force Stop
Insure - Customer 1								
Build production release from trunk	Success	2012-02-01 18:23:40	Force Build Only	1	Running	Sleeping		Force Stop
Insure Customer 1 (trunk)	Success	2012-01-24 14:55:17	Force Build Only	1	Running	Sleeping		Force Stop

Slika 13: Novo stanje pregleda projektov na strežniku CCNET po uvedbi modulizacije

Projekt »Build production release from trunk« preko skripte NAnt poleg jedra prevede še ustrezno implementacijo in s tem ustrezno prilagodi izdajo.

```
<target name="deploy.build.prod.customer1" description="Builds Customer 1 production deploy">
    <call target="version.revision.increment" />
    <call target="set.icons.customer1" />
    <call target="build-ccnet" />
    <call target="build.modules" />
    <call target="build.Insure.Customer1" />
    <call target="build.staging.customer1.prod" />
    <call target="version.insure.customer1.zip" />
</target>
```

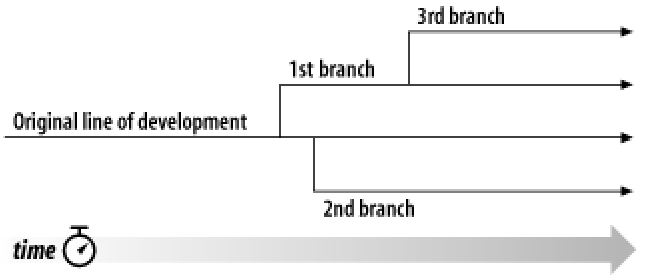
Slika 14: skripta NAnt za pripravo produkcijske izdaje za stranko 1

Iz prikazane skripte (Slika 14) je lepo viden pravilen tok dogodkov: prevod jedra, prevod modulov, prevod implementacije, postavitev testnega produkcijskega okolja ter priprava paketa za prenos izdaje na produkcijsko okolje.

V primeru potrebe po namestitvi aplikacije v testno okolje pri stranki, se preko strežnika CCNET zažene kar projekt za pripravo produkcijske izdaje. Razlika pri pripravi je predvsem v testiranju, ki ga v primeru izdaje za testno okolje praktično ni. Tak pristop je sprejemljiv zaradi dejstva, da se je na testno okolje nameščalo s točno določenim namenom testiranja v naprej določene funkcionalnosti, zato napake na drugih področjih niso bile moteče. Stranki je bila v takem primeru najbolj pomembna hitra odzivnost.

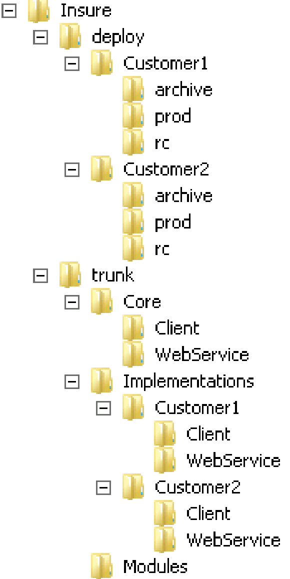
3.2 Upravljanje z izdajami izvorne kode na nivoju repozitorija strežnika SVN

Uporaba sistema za nadzor različic izvorne kode je pri razvijanju obsežne programske opreme praktično nujna. Koda je dostopna vsem razvijalcem, osveževanje in prenos sprememb na strežnik pa je enostavno. Med drugim nam sistem omogoča uporabo t.i. vej (Slika 15). Veja je vzporedna linija razvoja programske opreme, katero lahko spreminjamo brez vpliva na ostale linije razvoja. Več o tem lahko preberemo v [17, 18, 19]. Kljub ločenosti, je prenos sprememb iz ene na drugo linijo enostaven. Uporaba vej je smiselna tako pri razvoju kot tudi pri pripravi različice izdaje.



Slika 15: uporaba vej v sistemu za nadzor različic izvorne kode (vir: [19])

Pripravo različice izdaje lahko v grobem razdelimo na dva dela: priprava izdaje zaradi novih funkcionalnosti in priprava izdaje zaradi napak na trenutni različici izdaje. Pri načinu dela, ki smo ga imeli na začetku v podjetju, so se nameščale nove funkcionalnosti hkrati z odpravo napak, kar je med drugim lahko privedlo tudi do izdaje z ne dovolj dobro testirano novo funkcionalnostjo. Posledično smo se v podjetju odločili, da bomo proces izdelave izdaj razširili s pomočjo uporabe vej na nivoju repozitorija strežnika SVN (Slika 16). Na repozitoriju sem naredil novo mapo »deploy«, kjer so za vsako stranko na voljo 3 mape: »archive«, »prod« in »rc«. V mapi »archive« se nahaja zgodovina produkcijskih različic izdaje, v mapi »prod« se nahaja trenutna produkcijska različica izdaje, v mapi »rc« pa se nahaja kandidat za novo produkcijsko različico izdaje.



Slika 16: drevesna struktura map na repozitoriju strežnika SVN

Razvoj produkta še vedno poteka na glavni liniji razvoja. Po predhodnem dogovoru s stranko, ki želi uporabljati novo razvite funkcionalnosti produkta, se naredi kopija glavne linije, ki se shrani v vejo »rc«.

```
<target name="deploy.make.new" description="Makes deploy for product">
    <call target="svn.revert" />

    <property name="NewDeploySvnPath"
              value="${svn.path.product}/${svn.folder.deploy}/${implementation}/${svn.folder.deploy.rc}" />

    <!-- delete existing RC deploy -->
    <echo message="Deleting deploy '${NewDeploySvnPath}' from svn." />
    <exec program="svn.exe" failonerror="false">
        <arg line="delete ${NewDeploySvnPath} -m &quot;Deleted core deploy folder: ${NewDeploySvnPath}.&quot;" />
    </exec>

    <!-- Make new Insure deploy -->
    <echo message="Making new ${svn.folder.deploy.rc} deploy." />
    <exec program="svn.exe">
        <arg line="copy ${svn.path.product}/trunk ${NewDeploySvnPath} --parents
                   -m &quot;Deployed ${svn.folder.deploy.rc} by nant script.&quot;" />
    </exec>

    <!-- Switch to new deploy -->
    <property name="Switch_SvnPath" value="${NewDeploySvnPath}" />
    <property name="Switch_LocalPath" value="${local.path.checkout}\" />
    <call target="svn.switch.and.revert" />

    <echo message="deploy.make.new - COMPLETED" />
</target>
```

Slika 17: skripta NAnt za izdelavo nove veje kandidata za produkcijo izdajo

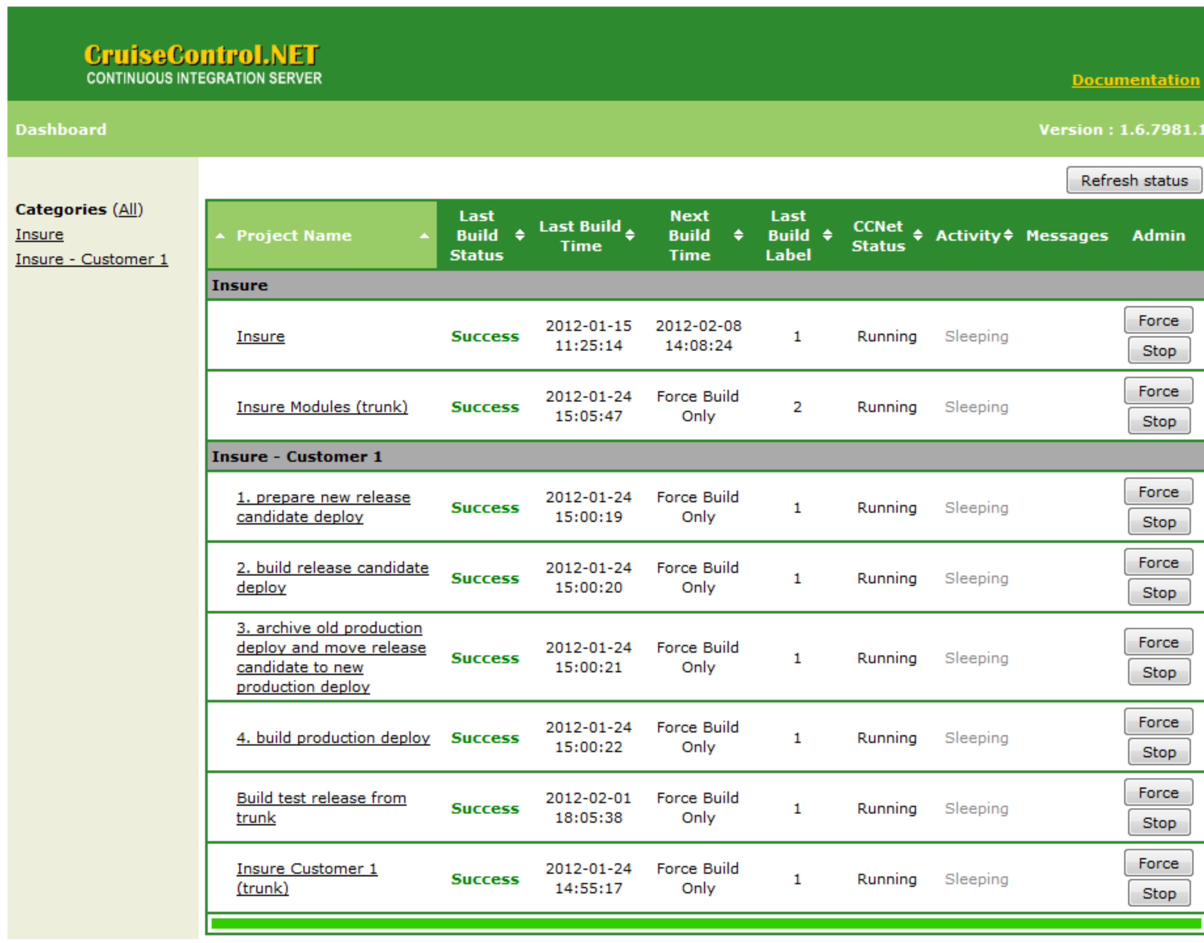
Skripta (Slika 17) najprej pobriše morebitno obstoječo vejo »rc« za dano implementacijo. Pomembno je, da se nastavi atribut »failonerror=false«, saj izdelovanje nove veje ne sme pasti, če prejšnja veja »rc« ne obstaja. Glavni del skripte je kopiranje glavne linije razvoja v vejo »rc«, za kar se uporabi kar ukaz SVN »copy«. Na koncu je potrebno lokalni kopiji izvorne kode zamenjati vir na repozitoriju strežnika SVN, za kar se uporabi ukaz SVN »switch«.

V tej veji je torej pripravljen kandidat za novo produkcijsko izdajo, na kar se opozori tudi razvijalce. V primeru delno shranjenih rešitev oziroma nepopolnega testiranja katere od funkcionalnosti, je potrebno funkcionalnost dodatno testirati ali pa onemogočiti. Na to vejo se lahko shranjujejo le še morebitni popravki preneseni iz glavne linije razvoja in nikakor ne nove funkcionalnosti. Kandidat za izdajo se nato preko strežnika CCNET prevede in namesti na ustrezno testno okolje pri stranki. Hkrati se pripravi dokument, ki ima zabeležene vse nove funkcionalnosti, ki so nastale na produktu od trenutno nameščene produkcijske različice izdaje naprej. Naloga stranke je, da nove funkcionalnosti preveri ter ustrezno testira. V primeru napake, se le-to preprosto odpravi na glavni liniji razvoja ter nato preko spajanja prenese še na vejo kandidata za produkcijo izdajo. Po potrditvi ustreznosti kandidata za izdajo, se na repozitoriju strežnika SVN arhivira prejšnjo produkcijsko različico izdaje ter kandidata prenese iz veje »rc« v vejo »prod« (Slika 18). Za obe operaciji se uporabi ukaz SVN »move«, ki izbrano vejo premakne na nov naslov SVN.

```
<target name="deploy.archive.prod.and.rename.rc.to.prod" description="Archives production deploy and moves RC to PROD.">
    <call target="svn.revert" />

    <!-- setting properties -->
    <call target="get.deploy.archive.name" /><!--returns deploy archive name based on deploy creation date and time -->
    <property name="SvnImplementationPath" value="${svn.path.product}/${svn.folder.deploy}/${implementation}" />
    <property name="ArchiveCoreDeploySvnPath"
              value="${SvnImplementationPath}/${svn.folder.archive}/${svn.folder.deploy.prod}_${UniqueArchiveName}" />
    <property name="RcCoreDeploySvnPath" value="${SvnImplementationPath}/${svn.folder.deploy.rc}" />
    <property name="ProdCoreDeploySvnPath" value="${SvnImplementationPath}/${svn.folder.deploy.prod}" />

    <!-- archiving -->
    <echo message="Archiving '${ProdCoreDeploySvnPath}' => '${ArchiveCoreDeploySvnPath}'." />
    <exec program="svn.exe">
        <arg line="move ${ProdCoreDeploySvnPath} ${ArchiveCoreDeploySvnPath}
                   -m &quot;Archiving '${ProdCoreDeploySvnPath}' => '${ArchiveCoreDeploySvnPath}'.&quot;" />
    </exec>

    <!-- moving RC to PROD -->
    <echo message="Moving '${RcCoreDeploySvnPath}' => '${ProdCoreDeploySvnPath}'." />
    <exec program="svn.exe">
        <arg line="move ${RcCoreDeploySvnPath} ${ProdCoreDeploySvnPath}
                   -m &quot;Moving '${RcCoreDeploySvnPath}' => '${ProdCoreDeploySvnPath}'.&quot;" />
    </exec>
</target>
```

Slika 18: skripta NAnt za arhiviranje veje stare produkcijske izdaje ter premik veje »rc« na vejo »prod«

Produkcijska veja se nato preko strežnika CCNET in skripte NAnt prevede ter namesti na produkcijsko okolje. V primeru napake se, podobno kot pri odpravi napake na veji kandidata za produkcijsko izdajo, napako najprej odpravi na glavni liniji razvoja in nato preko spajanja prenese popravek še na vejo produkcijske izdaje. Opisani postopek se imenuje »cherry-picking« in je opisan v [37]. V našem podjetju se je tak princip spajanja izkazal kot najboljši, saj si s tem zagotovimo popravek napake v glavni liniji razvoja, poleg tega pa smo iz izkušenj ugotovili, da je lažje spajati iz novejše v starejšo kodo, kot pa obratno.



Project Name	Last Build Status	Last Build Time	Next Build Time	Last Build Label	CCNet Status	Activity	Messages	Admin
Insure								
Insure	Success	2012-01-15 11:25:14	2012-02-08 14:08:24	1	Running	Sleeping		Force Stop
Insure Modules (trunk)	Success	2012-01-24 15:05:47	Force Build Only	2	Running	Sleeping		Force Stop
Insure - Customer 1								
1. prepare new release candidate deploy	Success	2012-01-24 15:00:19	Force Build Only	1	Running	Sleeping		Force Stop
2. build release candidate deploy	Success	2012-01-24 15:00:20	Force Build Only	1	Running	Sleeping		Force Stop
3. archive old production deploy and move release candidate to new production deploy	Success	2012-01-24 15:00:21	Force Build Only	1	Running	Sleeping		Force Stop
4. build production deploy	Success	2012-01-24 15:00:22	Force Build Only	1	Running	Sleeping		Force Stop
Build test release from trunk	Success	2012-02-01 18:05:38	Force Build Only	1	Running	Sleeping		Force Stop
Insure Customer 1 (trunk)	Success	2012-01-24 14:55:17	Force Build Only	1	Running	Sleeping		Force Stop

Slika 19: novo stanje pregleda projektov na strežniku CCNET

Z upravljanjem z izdajami na nivoju repozitorija strežnika SVN, smo izgubili možnost priprave izdaje iz glavne linije razvoja za namestitev na testni strežnik pri stranki, zaradi česar sem naredil nov projekt na strežniku CCNET »Build test release from trunk« (Slika 20). Ta pokliče ustrezno tarčo v skripti NAnt in pripravi ustrezno izdajo.

```
<target name="deploy.build.test.customer1" description="Builds test release for Customer 1 from trunk">
    <call target="version.revision.increment" />
    <call target="set.icons.customer1" />
    <call target="build-ccnet" />
    <call target="build.modules" />
    <call target="build.Insure.Customer1" />
    <call target="build.staging.customer1.test" />
    <call target="version.insure.customer1.zip" />
</target>
```

Slika 20: skripta NAnt za pripravo testne izdaje iz glavne linije razvoja za stranko 1

Z upravljanjem izdaj izvorne kode smo naredili velik korak. Po novem ni več bojazni za namestitev nepreizkušene funkcionalnosti na produkcijsko okolje. Razvijalci imajo tako bolj proste roke, predvsem pa jim ni potrebno skrbeti, kdaj bo katera od funkcionalnosti nameščena v produkcijski izdaji. V primeru odkrite napake, se lahko izdajo s popravkom namesti razmeroma hitro brez nepotrebnih pritiskov.

Posledično se je povečal tudi nadzor nad novo vključenimi funkcionalnostmi. Te se lahko pred novo različico izdaje dobro testira in odpravi morebitne napake še predno pridejo v stik s stranko. Za podjetje je zelo dobrodošla tudi vpeljava kandidata za produkcijsko izdajo, s katero se del odgovornosti za napake preloži na stranko, stranka pa ima kljub dodatnemu delu dober občutek zaradi vključenosti v razvoj produkta.

Poleg omenjenega smo si zagotovili tudi zgodovino namestitev različic izdaj. Kljub več nivojem testiranja se namreč lahko zgodi, da je potrebno, zaradi kritičnih napak na novi različici izdaje, na produkcijskem okolju namestiti nazaj prejšnjo različico izdaje. Bistvene so predvsem izdaje, ki vsebujejo nove funkcionalnosti. Glede na to, da ob izdelavi produkcijske izdaje z novimi funkcionalnostmi staro različico izdaje arhiviramo na repozitoriju strežnika SVN, je priprava ter namestitev starejše različice izdaje enostavna.

3.3 Avtomatizacija povečevanja oznake različice izdaje

Zaradi večjega nadzora nad programsko opremo je smiselno, da ima vsaka različica izdaje svojo oznako, preko katere se lahko natančno določi katere nove funkcionalnosti ter popravki so vključeni v izdajo. V primeru, da je programska oprema nameščena pri eni sami stranki, se natančno označevanje ne zdi najbolj pomembno, saj je postopek določanja nameščenih funkcionalnosti enostaven. Z naraščanjem števila strank se izkaže natančnost označevanja kot nujna.

V podjetju se različico produkta »Insure« označuje s štirimi številkami, ki so ločene med sabo s pikami (x.y.z.w). Zaradi zagotavljanja različnosti oznake različice pri vsaki izmed strank, se ob vsaki kopiji glavne linije razvoja v vejo »rc« poveča tretja številka v oznaki za stopnjo ena, zadnja pa se postavi na vrednost 0. Ob vsaki pripravi nove različice izdaje pa se povečuje zadnja številka za stopnjo 1. Prvi dve številki sta namenjeni večjim spremembam na produktu in se jih nastavlja ročno.

Povečevanje oznake različice izdaje lahko razdelimo na dva dela:

- Branje datoteke »version.txt«, kjer je shranjena trenutna oznaka različice in povečevanje oznake ter zapis nove oznake nazaj v datoteko
    - Povečevanje tretje številke: x.y.z.w => x.y.(z+1).0
    - Povečevanje zadnje številke: x.y.z.w => x.y.z.(w+1)
- Zapis oznake v vse datoteke, ki opisujejo programske komponente (datoteke »AssemblyInfo.cs«) in shranjevanje sprememb na repozitorij strežnika SVN

Funkcionalnost povečevanja oznake različice izdaje sem zaradi lažje vključitve v skripto za pripravo izdaje napisal kar s pomočjo skripte NAnt. Kot je opisano zgoraj, je tudi skripta razdeljena na dva dela.

```
<target name="increment.revision.number" description="Increment the revision number and write to version.txt file">
    <exec program="svn.exe">
        <arg line="update ${version.filename} -q" />
    </exec>
    <loadfile file="${version.filename}" property="build.version.string" />

    <property name="major" value="${version::get-major(version::parse(build.version.string))}" />
    <property name="minor" value="${version::get-minor(version::parse(build.version.string))}" />
    <property name="build" value="${version::get-build(version::parse(build.version.string))}" />
    <property name="revision" value="${version::get-revision(version::parse(build.version.string))+1}" />
    <property name="build.version.string" value="${major}.${minor}.${build}.${revision}" />

    <echo message="${build.version.string}" file="${version.filename}" />
    <exec program="svn.exe">
        <arg line="commit ${version.filename} -q --message &quot;Version ${build.version.string} set by build script.&quot;" />
    </exec>
</target>
```

Slika 21: skripta NAnt, ki poveča »revision« v oznaki različice v datoteki »version.txt«

Prikazana skripta (slika 21) najprej osveži datoteko iz repozitorija strežnika SVN in vsebino datoteke prebere v spremenljivko »build.version.string«. Sledi ustrezno povečevanje oznake. Skriptni jezik NAnt omogoča delo z objekti tipa »Version« (podrobnosti o objektu »Version« v: [20]), kar poenostavi povečevanje oznak, ter poveča berljivost skripte. Na koncu preko značke »echo« zapiše oznako različice nazaj v datoteko ter spremembo shrani tudi na repozitorij strežnika SVN.

```
<target name="version.set" description="Stamp the version info onto assemblyinfo.cs files">
    <property name="list.filenames"  value="" />
    <property name="assembly.version.string"
                  value="[assembly: AssemblyVersion(&quot;' + build.version.string + '&quot;)]" />
    <property name="assembly.file.version.string"
             value="[assembly: AssemblyFileVersion(&quot;' + build.version.string + '&quot;)]" />

    <foreach item="File" property="filename">
        <in>
            <items>
                <include name="**\AssemblyInfo.cs" />
            </items>
        </in>
        <do>
            <exec program="svn.exe">
                <arg line="update ${filename} -q" />
            </exec>
            <property name="list.filenames"  value="${list.filenames} &quot;${filename}&quot;" />

            <loadfile file="${filename}" property="file" encoding="UTF-8" />
            <foreach item="Line" in="${filename}" property="file.line">
                <if test="${string::starts-with(file.line, '[assembly: AssemblyVersion(')}">
                    <property name="file" value="${string::replace(file, file.line, ' + assembly.version.string +')}" />
                </if>
                <if test="${string::starts-with(file.line, '[assembly: AssemblyFileVersion(')}">
                    <property name="file" value="${string::replace(file, file.line, ' + assembly.file.version.string + ')}" />
                </if>
            </foreach>
            <echo message="${file}" file="${filename}" />
        </do>
    </foreach>
    <exec program="svn.exe">
        <arg line="commit -q --message &quot;Version ${build.version.string} is set by build script.&quot; ${filename_list} " />
    </exec>
</target>
```

Slika 22: skripta NAnt, ki zapiše novo oznako različice v datoteke »AssemblyInfo.cs«

Skripta (slika 22) ima nalogo, da poišče vse datoteke »AssemblyInfo.cs« ter v njih najde zapis z oznako različice ter ga zamenja z novo oznako. Funkcionalnost je implementirana z dvojno

»foreach« zanko. V prvi iteraciji prebere vsako najdeno datoteko in jo osveži z repozitorija strežnika SVN, v drugi pa vrstico po vrstici išče zapis z oznako različice ter jo zamenja z novo oznako.

Skripto za povečevanje oznake različice izdaje sem uvrstil v skripto, kjer se pripravlja izdajo (Slika 23). V prikazani skripti je pomembno, da se kliče povečevanje oznake pred prevajanjem jedra.

```
<target name="deploy.build.prod.customer1" description="Builds Customer 1 production deploy">
    <call target="version.revision.increment" />
    <call target="set.icons" />
    <call target="build-ccnet" />
    <call target="build.modules" />
    <call target="build.Insure.Customer1" />
    <call target="build.staging" />
    <call target="version.insure.zip" />
</target>
```

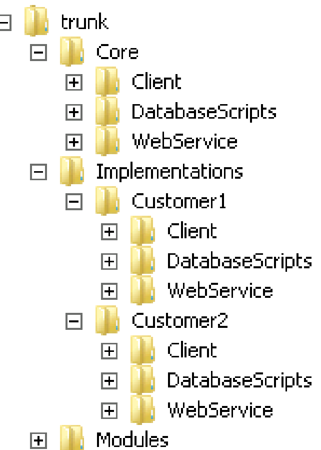
Slika 23: skripta NAnt, ki poveča oznako različice izdaje ter prevede vejo »prod« za stranko 1

3.4 Usklajevanje sprememb podatkovnega modela baze skupaj z izvorno kodo

Razvoj produkta narekuje poleg sprememb izvorne kode tudi spremembe v podatkovnem modelu baze. Aplikacija lahko deluje pravilno le v primeru, ko sta obe omenjeni komponenti med sabo usklajeni.

Sprva smo v podjetju vsako prilagoditev produkta najprej uredili na nivoju PDM in sicer na vseh podatkovnih bazah, tako znotraj podjetja kot tudi pri stranki in šele nato v izvorni kodi. Z večanjem razvojne ekipe in s pridobitvijo novih strank tak pristop ni bil več primeren. Vse stranke namreč ne omogočajo neposrednega dostopa do podatkovne baze, hkrati pa se je povečalo tudi število zahtev po sočasni spremembi PDM ter aplikacije.

Rešitev problema sem implementiral s shranjevanjem skript SQL skupaj z ustrezno izvorno kodo na repozitorij strežnika SVN. V ta namen sem na repozitoriju dodal mape »DatabaseScripts«, kamor se bodo shranjevale skripte SQL (Slika 24). Zelo pomembno je, da sem omenjene mape dodal tudi v vseh implementacijah za stranke, saj je lahko skripta SQL namenjena samo nekaterim izmed strank.



Slika 24: nova struktura na repozitoriju strežnika SVN po dodanih mapah »DatabaseScripts«

Z omogočenim shranjevanjem skript skupaj z izvorno kodo se poveča nadzor nad spremembami PDM, saj lahko neposredno določimo, kaj je bil vzrok za spremembo.

Delo skrbnika PDM postane s to spremembo bolj odgovorno, po drugi strani pa se mu zmanjša obseg dela, saj mu ni potrebno zagnati skripte SQL na vseh podatkovnih bazah v podjetju ter pri vseh strankah, saj za zagon skript skrbi tisti, ki izdajo namešča.

Enako kot prej razvijalec produkta »Insure« odda zahtevo za spremembo skrbniku PDM, ki preko programa Power Designer ustrezno spremeni PDM in pridobljeni skripti SQL doda glavo in nogo preko katere se v podatkovno bazo zapiše podatek o uspešno zagnani skripti. Namesto zagona skripte na vseh podatkovnih bazah, jo pošlje preko elektronske pošte razvijalcu, ki je spremembo zahteval. Ta jo skupaj s spremenjeno izvorno kodo shrani na ustrezno mesto na repozitorij strežnika SVN. Posledično je pridobitev razloga za spremembo PDM trivialna, skrbniku pa ni potrebno več skrbeti za arhiviranje in izdelovanje varnostnih kopij skript SQL.

Kljub na videz dobri rešitvi ta še zdaleč ni popolna. Izgubili smo namreč zelo pomemben del pri spreminjanju PDM in sicer pravilno zaporedje izvajanja skript SQL. Res je, da skrbnik spreminja PDM in izdaja ustrezne skripte v pravem vrstnem redu, vendar nam nič ne zagotavlja, da se bodo skripte v istem zaporedju shranile na repozitorij strežnika SVN. Vsaka dodelava v produktu »Insure« je po svoje unikaten izdelek in lahko traja od nekaj minut do nekaj dni. V primeru ene same stranke in majhne ekipe je enostavno preprečiti napačno zaporedje zagona skript na produkcijskem strežniku, saj praktično vsak v ekipi ve kdaj se izdeluje kakšna večja sprememba na produktu. V primeru večje ekipe in večjega števila strank, pa je tak nadzor praktično nemogoč. Zlahka pride do situacije, ko se pri stranki namesti novo različico izdaje, pri čemer le-ta ne vsebuje vseh potrebnih skript SQL.

Spremembe v PDM lahko v osnovi razdelimo na dva dela: na spremembe, ki so neodvisne od prejšnjega stanja PDM in na spremembe, ki so od stanja PDM odvisne. Če se doda v PDM novo tabelo gre za spremembo neodvisno od prejšnjega stanja (ob predpostavki, da tabela z enakim imenom v PDM še ne obstaja), če pa se v neko tabelo doda nov stolpec, je ta sprememba odvisna od prejšnjega stanja, saj mora očitno ta tabela že obstajati v PDM. Posledično ima skrbnik bolj odgovorno delo, saj mora v programu Power Designer za vsako

spremembo voditi evidenco s katero skripto je bila sprememba implementirana. Informacijo o odvisnosti izdelane skripte mora posredovati razvijalcu, ki mora pred shranjevanjem skripte ter izvorne kode na repozitorij strežnika SVN preveriti ali so vse potrebne skripte že shranjene na pravem mestu. V primeru, da predhodne skripte na strežniku ni, mora s shranjevanjem počakati oziroma v primeru nujnosti prositi skrbnika PDM, da ustrezno prilagodi skripto. Ob prehodu na nov način dela je bilo potrebno določiti neko osnovno skripto, ki služi kot predpogoj za vse naslednje izdelane skripte.

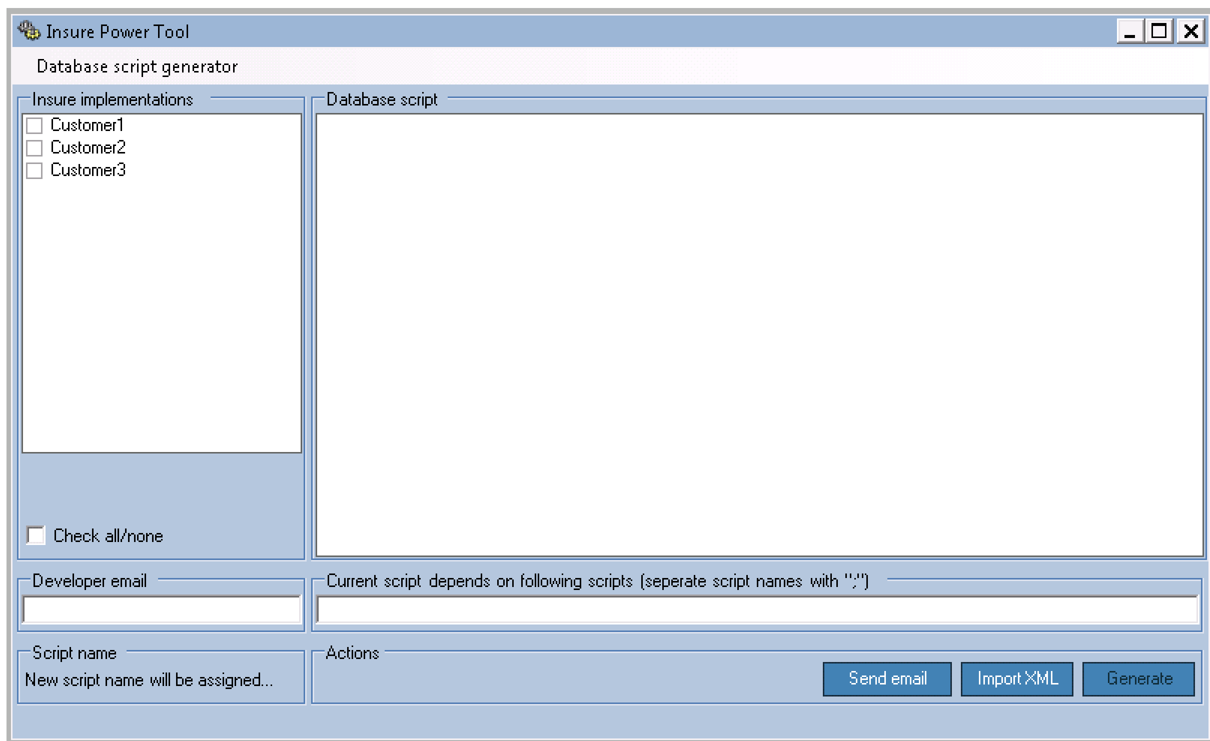
Poleg večje odgovornosti s preverjanjem do sedaj shranjenih skript mora razvijalec po pridobitvi skripte le-to shraniti na pravilno mesto na repozitorij strežnika SVN. Če je skripta splošna in je namenjena vsem implementacijam, jo mora shraniti v mapo »DatabaseScripts« znotraj jedra, če pa gre za skripto namenjeno samo nekaj strankam, jo mora shraniti na več mest, v mapo »DatabaseScripts« znotraj mape vsake stranke. Pri tem obstaja možnost napake pri shranjevanju na prava mesta, s čimer lahko razvijalec povzroči napako na produkcijskem okolju.

Skripte SQL se lahko uporablja tudi za spremembe, ki se ne tičejo PDM, na primer dodatni zapis v šifrant. V primeru, da morajo biti zapisi v šifrantu prevedeni v ustrezni jezik, je potrebno izdelati za različne stranke različne skripte SQL, razvijalec pa mora vsako posebej shraniti na ustrezno mesto na repozitorij strežnika SVN.

Več kot očitno je to kar velika odgovornost za razvijalca, zato je smiselno za shranjevanje skript na ustrezna mesta izdelati orodje, ki bo razbremenilo razvijalca ter zmanjšalo možnost za napako.

3.4.1 Izdelava orodja za pomoč pri izdelovanju in shranjevanju skript SQL

Orodje sem izdelal z namenom razbremenitve tako razvijalca kot skrbnika PDM. Opravlja enostavno delo, ki je pri ročnem delu zaradi ponovljivosti pričakovan razlog za napako.



Slika 25: ekran programa »Database script generator«

Tako kot pri prejšnjem postopku, se tudi tukaj začne sprememba z zahtevo s strani razvijalca. Ta mora poleg zahtevane spremembe PDM določiti še katerim strankam je sprememba namenjena. Večinoma gre za spremembo podatkovnega modela, ki je namenjen vsem strankam. Skrbnik PDM enako kot prej s programom Power Designer ustrezno spremeni PDM. Namesto, da izdelano skripto sam poimenuje in jo posreduje razvijalcu, jo vstavi v orodje v okno »Database script«. V okno »Current script depends on following scripts« vpiše vse zahtevane skripte, ki morajo biti že shranjene na repozitorij strežnika SVN in so pogoj za uspešno izvedbo konkretne skripte. Na podlagi informacije, ki jo dobi s strani razvijalca, ustrezno izbere stranke, katerim je skripta namenjena. Če je prilagoditev za vsako stranko specifična, mora skrbnik izdelati skripto za vsako posebej. Na koncu se v polje »Developer email« vnese še elektronski naslov razvijalca in vnos potrdi preko gumba »Send email«. V tem koraku se najprej skripti doda glava in noga, nato pa se iz mape jedra prebere vsebino datoteke »LastScriptId.txt«, kjer je zapisano ime zadnje izdelane skripte. Ime ima obliko X.XX_<trenutna_oznaka_različice_produkta>_YY. Konstanta »X.XX« predstavlja trenutno verzijo skript SQL in je enaka »2.00« (v starem sistemu za upravljanje z izdajami, so imele skripte verzijo »1.00«). Vmesni del je enak oznaki različice produkta »Insure«, kjer je vsak posamezni del zapisan z vodilnimi ničlami na širino treh mest, zadnji del »YY« pa predstavlja števec izdelanih skript znotraj iste oznake različice. Ime nove skripte se določi glede na prebrano vrednost in glede na trenutno oznako različice produkta ter se zapiše nazaj v datoteko. Na koncu orodje izdela datoteko XML ter jo pošlje razvijalcu na vpisani elektronski naslov.

Razvijalec uporabi na svojem računalniku enako orodje z drugačnim pooblastilom. Orodje mu omogoči gumb »Import XML«, s katerim uvozi datoteko XML, ki mu jo je poslal skrbnik. V orodju se mu izpolnijo vsa polja z namenom, da preveri ustreznost skripte in da pregleda ali

so izbrane prave stranke. Po uvozu se mu omogoči gumb »Generate«, ki ob kliku najprej preveri vsebino polja »Current script depends on following scripts«. Če polje ni prazno, se za vsako vpisano skripto preveri ali že obstaja na lokaciji kamor naj bi se odložila nova skripta. Če iskane skripte ni, potem orodje javi razvijalcu, da shranjevanje nove skripte ni možno zaradi odvisnosti skripte od ostalih skript, ki še niso shranjene na repozitorij strežnika SVN. Če vse zahtevane skripte SQL obstajajo ali pa je skripta neodvisna od drugih skript, orodje shrani skripto SQL na pravo mesto. V primeru vseh izbranih strank, se bo skripta zapisala v mapo »DatabaseScripts« znotraj jedra, v primeru izbire le določenih strank, pa se bo skripta zapisala pri vseh izbranih strankah v mapi »DatabaseScripts«. Nato se vsako skripto shranjeno na trdi disk označi za dodajanje v repozitorij strežnika SVN.

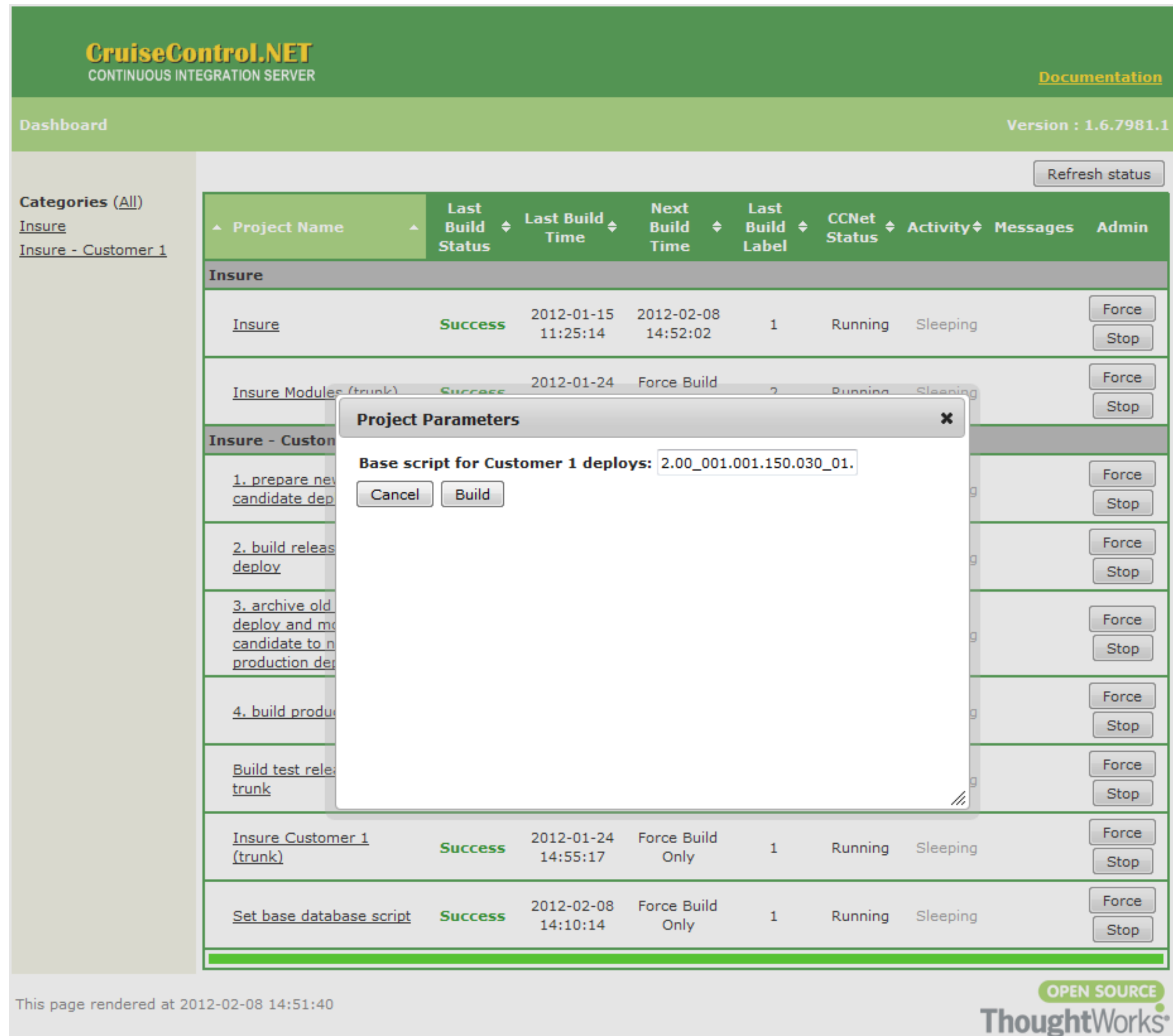
V primeru, da so vse predhodne skripte ustrezno nameščene, lahko novo skripto SQL razvijalec shrani na repozitorij strežnika SVN skupaj z ustrezno testirano izvorno kodo.

3.4.2 Zajem ustreznih skript SQL pri izdelavi izdaje

Vsaka skripta SQL vsebuje glavo in nogo v kateri so zapisani podatki, ki se po uspešni izvedbi skripte zabeležijo v podatkovni bazi na vseh okoljih pri vsaki stranki. Ta funkcionalnost nam omogoča pregled vseh zagnanih skript na konkretni podatkovni bazi in hkrati služi za preprečevanje ponovnih zagonov že zagnanih skript.

Kljub temu ni smiselno zajemati v vsako izdajo vse skripte SQL, ki se jih tekom let nabere kar veliko. Določitev meje za zajem skript za posamezno stranko je lahko dokaj trivialna naloga, saj ima lahko stranka le osnovna okolja, na primer: produkcijsko okolje, testno okolje, okolje za testiranje kandidata za produkcijo. V takem primeru bi lahko vodili za vsako okolje podatek o zadnji nameščeni skripti. Problem se pojavi, če ima stranka več testnih okolij na katera se izdaje ne nameščajo redno. Nesmiselno je za vsako stranko voditi podatek za N okolij.

V podjetju smo se odločili, da bomo za vsako stranko našli najnovejšo skripto, ki je nameščena na vseh podatkovnih bazah stranke in jo zapisali v datoteko »BaseDatabaseScript.txt« v mapo »DatabaseScripts« poleg skript SQL. Funkcionalnost zajemanja skript pri pripravi izdaje zajame le skripte, katerih ime je večje od tistega zapisanega v datoteki. Ob večini izdaj bodo tako zajete že nameščene skripte, vendar z dokaj pogostim osveževanjem datoteke to ne predstavlja prevelikega problema. Osveževanje datoteke z imenom najnovejše skripte sem implementiral preko strežnika CCNET (Slika 26).



Slika 26: Prikaz okna za vpis osnovne skripte za stranko 1

Skripta prikaže ob zagonu okno v katerega se vpiše ime skripte. Ob pritisku na gumb »Build« se pokliče tarča »copy.database.scripts.save.base.script« v skripti NAnt »insure.build« (Slika 27), še pred tem pa nastavi parametra »base.script« in »implementation« na ustrezno vrednost.

```xml
<project name="Set base database script">
    <category>Insure - Customer  1</category>
    <workingDirectory>$(WorkingDir)Product\Insure\Core\</workingDirectory>
    <tasks>
        <nant>
            <executable>$(NAntExe)</executable>
            <buildFile>insure.build</buildFile>
            <targetList>
                <target>copy.database.scripts.save.base.script</target>
            </targetList>
            <buildArgs>-D:base.script=$[Base script for Customer 1 deploys] -D:implementation=Customer1</buildArgs>
            <buildTimeoutSeconds>20</buildTimeoutSeconds>
        </nant>
    </tasks>
    <parameters>
        <textParameter name="Base script for Customer 1 Customer 1 deploys">
            <description>Base script for Customer 1 Customer 1 deploys.</description>
            <required>true</required>
            <default></default>
        </textParameter>
    </parameters>
</project>
```

Slika 27: nastavitev projekta »Set base database script« v nastavitveni datoteki »ccnet.config«

Omenjeno funkcionalnost sem podprl še v skripti NAnt v jedru produkta.

```xml
<target name="copy.database.scripts" description="Copies new scripts to release folder">
    <property name="scripts.dir" value="${deploy.folder.default}\${implementation.release.folder}\scripts" />
    <call target="copy.database.scripts.read.base.script" />

    <!-- core scripts -->
    <property name="copyFromDir" value="DatabaseScripts\" />
    <property name="copyToDir" value="${scripts.dir}" />
    <call target="copy.database.scripts.files" />

    <!-- implementation scripts -->
    <property name="copyFromDir" value="${implementationFolder}\DatabaseScripts\" />
    <property name="copyToDir" value="${scripts.dir}" />
    <call target="copy.database.scripts.files" />
</target>
<target name="copy.database.scripts.files" description="Copies .sql files to specified location">
    <fileset id="scripts" basedir="${copyFromDir}">
        <include name="*.sql" />
    </fileset>
    <foreach item="File" property="copyFilename">
        <in>
            <items refid="scripts" />
        </in>
        <do>
            <call target="copy.database.scripts.file" if="${path::get-file-name(copyFilename) &gt; base.script}" />
        </do>
    </foreach>
</target>
<target name="copy.database.scripts.file" description="Copies specified file to a specified location">
    <copy todir="${copyToDir}">
        <fileset basedir="${copyFromDir}">
            <include name="${copyFilename}" />
        </fileset>
    </copy>
</target>
```

Slika 29: skripta NAnt za kopiranje ustreznih skript v mapo, kjer se pripravlja izdajo

V skripti (Slika 29) NAnt sta bistvena predvsem dva dela. Prvi je klic tarče »copy.database.scripts.read.base.script«, ki prebere iz datoteke »BaseDatabaseScript.txt« osnovno ime skripte. Drugi pa je zajem vseh skript SQL, ki pa je ločen na splošne skripte (shranjene znotraj jedra) ter na specifične skripte (shranjene znotraj konkretne implementacije). Za vsak nabor skript se preko imena preverja ali je potrebno skripto kopirati ali ne. Tako primerjanje je mogoče zato, ker so imena vseh skript enako dolga ter zaradi konstantnega povečevanja oznake različice produkta.

3.4.3 Testiranje ter zagon skript SQL ob namestitvi nove različice izdaje

Pred koncem priprave izdaje je potrebno kljub previdnosti pri shranjevanju skript narediti še test zagona skript SQL. V ta namen imamo pripravljeno posebno podatkovno bazo, na kateri je najnovejša naložena skripta SQL starejša ali kvečjemu enaka kot najstarejša skripta v paketu za izdajo. Za primeren test je potrebno najprej zagnati vse skripte od zadnje naložene pa vse do najstarejše skripte v paketu za izdajo, s čimer poustvarimo okolje pri stranki, nato pa zaženemo še vse skripte iz paketa za izdajo. V primeru neuspešnega zagona, je napaka ali v skripti sami ali pa v spregledani odvisnosti skript. Vsekakor se nove izdaje ne sme namestiti pred ureditvijo skript SQL, saj se bo napaka zelo verjetno ponovila pri stranki.

3.4.4 Prehod iz starega na nov način dela

Pri starem načinu dela je večino dela opravil skrbnik PDM, ostali so mu le podali zahtevo, in čakali na sporočilo, da je bila zadeva urejena. Nov način dela vključuje v delo s skriptami SQL več ljudi. Vsak, ki potrebuje spremembo na podatkovni bazi, ima sedaj večjo odgovornost. Zaradi spremembe načina dela pri več osebah, je bilo smiselno prehod izpeljati postopno. V nasprotnem primeru bi se čas procesa izdelave skript močno povečal, v primeru kakšne napake v postopku pa bi na oddelku nastala zmeda.

V prvi fazi sem vpeljal nov način poimenovanja skript ter shranjevane skript SQL na ustrezno mesto na repozitoriju strežnika SVN. Delo se je spremenilo le skrbniku PDM, ki je začel uporabljati orodje »Database script generator«. Zahtevano premembo je vpisal v program Power Designer, ki mu je generiral skripto SQL, katero je ustrezno prilagodil ter vpisal v orodje. Nato je ustrezno izbral stranke katerim je bila skripta namenjena ter izpolnil polje »Current script depends on following scripts«, kjer je zabeležil od katerih skript je bila nova skripta SQL odvisna. Zaradi postopnega prehoda je bil omogočen gumb »Generate«, ki je skripte shranil na pravo mesto na lokalnem okolju ter jih označil za dodajanje na repozitorij strežnika SVN. Tako kot pred novim načinom dela, je skrbnik skripte zagnal na vseh podatkovnih bazah, tako znotraj podjetja kot tudi pri strankah ter na koncu, v primeru, da ni prišlo do napake, je skripte shranil na repozitorij strežnika SVN.

V drugi fazi sem vklopil zajem skript SQL v paket za nameščanje nove različice izdaje. Najprej sem pri vsaki stranki preveril na vseh podatkovnih bazah katera je najstarejša zadnja zagnana skripta. Pridobljeno ime skripte sem vpisal kot »osnovno skripto« preko strežnika CCNET. Skrbnik PDM je izdelano skripto SQL še vedno zagnal na podatkovnih bazah znotraj podjetja z namenom testiranja skripte same ter aplikacije s strani razvijalca, ni pa zagnal skript pri strankah. Posledično je bilo potrebno pri vsaki namestitvi izdaje zagnati še

vse nove skripte SQL v paketu. Ta faza razbremeni delo skrbniku PDM, več dela pa mora opraviti tisti, ki izdajo namešča.

V tretji fazi se je pričelo shranjevati skripte SQL skupaj s prilegajočo izvorno kodo na repozitorij strežnika SVN. Skrbniku sem v orodju »Database script generator« onemogoči gumb »Generate«, omogočil pa sem mu gumb »Send mail«, preko katerega pošlje mail z datoteko XML razvijalcu, ki je spremembo zahteval. Le-ta datoteko uvozi, preveri ustreznost skripte SQL, klikne gumb »Generate« ter skripto skupaj z izvorno kdo shrani na repozitorij strežnika SVN. V tej fazi se spremeni način dela praktično samo razvijalcem.

3.5 Vpeljava programske komponente Windows Installer

Namestitev izdaje je sprva potekala z zagonom paketne datoteke, katera je vsebovala ukaz za kopiranje datotek produkta na točno določeno lokacijo. Zaradi možnih napak pri namestitvi je bilo potrebno pred vsako namestitvijo izdelati varnostno kopijo, ki je omogočala dokaj hitro razveljavitev sprememb. Gre za odgovorno delo kljub temu, da je postopek dokaj enostaven.

Nekoliko več dela je ob prvi namestitvi pri stranki. Za vsako okolje je potrebno spisati ustrezno skripto za kopiranje datotek produkta na določene lokacije ter jo shraniti v paketno datoteko. Na internetnem strežniku IIS je potrebno dodati novo aplikacijo, ki kaže na mapo produkta »WebService«.

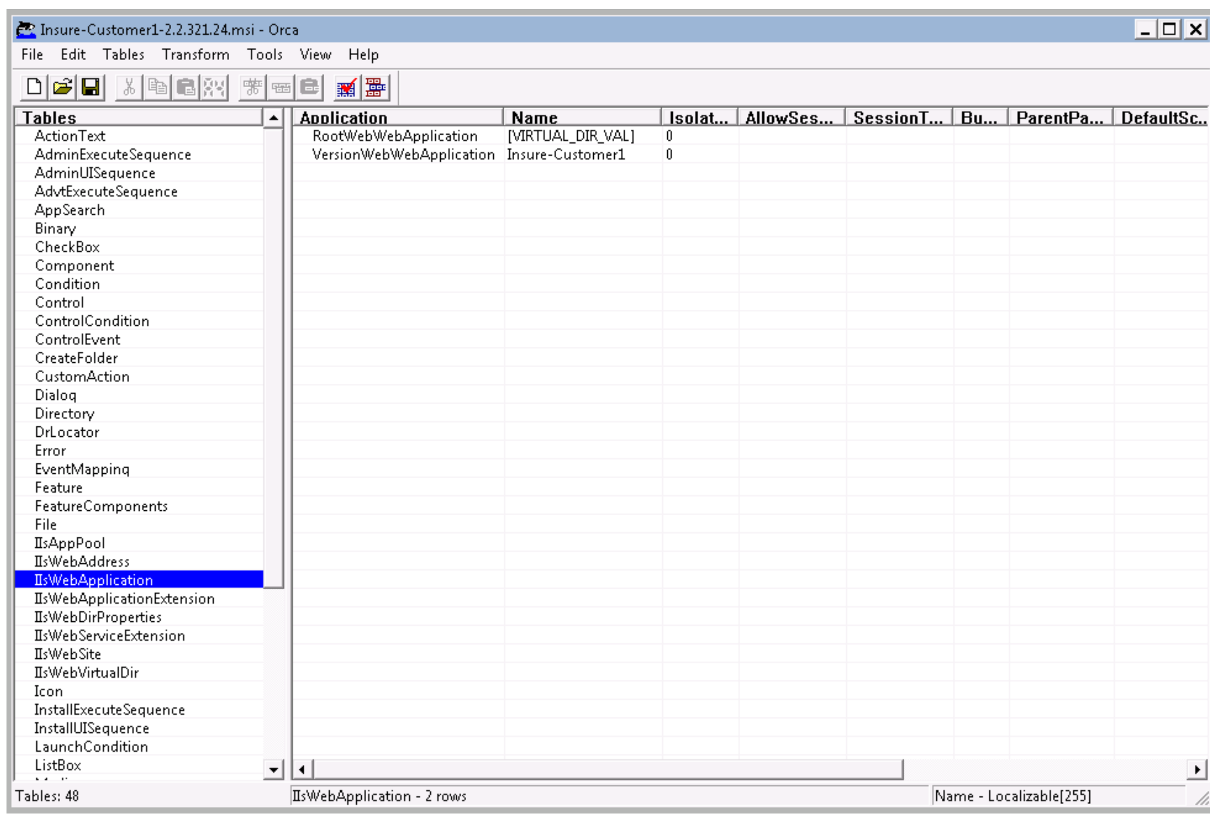
Kljub enostavnosti je možnosti za nastanek napake pri nameščanju izdaje preveliko in zato ni primerno za predajo v roke skrbnikov informacijskega sistema na strani zavarovalnice. Zaradi pridobivanja novih strank, ki imajo omejen dostop do produkcijskega okolja, se je težnja po poenostavitvi procesa nameščanja ter posledično predaje namestitve osebju znotraj zavarovalnice povečala.

Glede na to, da se produkt razvija in teče na operacijskih sistemih Microsoft Windows, smo se v podjetju za nameščanje ter konfiguriranje odločili za uporabo Microsoftove programske komponente z imenom »Windows Installer«. To je komponenta operacijskega sistema, ki se jo uporablja za namestitve, vzdrževanje in odstranjevanje programske opreme na operacijskih sistemih Microsoft Windows. Uporaba programske komponente »Windows Installer« zagotavlja učinkovito namestitev in konfiguracijo produkta. Več o programski komponenti si lahko preberemo v [21].

Prednosti uporabe programske komponente Windows Installer je več. Omogoča dinamično konfiguriranje in hkrati zagotavlja konsistentnost namestitve, saj se vsi koraki izvedejo po točno določenem postopku. Zelo uporabna je tudi lastnost razveljavitve sprememb v primeru napake pri namestitvi, ki povrne sistem v prejšnje stanje.

Uporabniški vmesnik je vsem dobro poznan in posledično je predaja procesa namestitve stranki preprosta. Po namestitvi se novo nameščena aplikacija pojavi na seznamu trenutno nameščenih programov (orodje »Add or Remove Programs« v operacijskem sistemu Microsoft Windows), kar omogoča popravljanje oziroma ponovno konfiguriranje ter po potrebi odstranitev aplikacije (vir: [22]).

Programska komponenta »Windows Installer« temelji na navodilih, ki so shranjena v datoteki .MSI. Datoteka .MSI je v osnovi relacijska podatkovna baza, ki vsebuje vse potrebne informacije za namestitev. Vsebuje podatke katere datoteke je potrebno prenesti, katere mape je potrebno kreirati, katere ključe v registru je potrebno dodati in podobno (vir: [23, 32]).



Slika 30: prikaz strukture datoteke .MSI s programom »Orca«

Slika 30 prikazuje grafični vmesnik za izdelavo in urejanje paketov .MSI imenovan »Orca«, o katerem si lahko več preberemo v [24]. Na levi imamo seznam tabel, na desni pa vrstice v tej tabeli. V tabelah so shranjene vse informacije potrebne za pravilno namestitev programa. Trenutno izbrana tabela vsebuje navodilo za dodajanje nove aplikacije na strežnik IIS.

Windows Installer zagotavlja veliko akcij za opravljanje procesa namestitve in v večini primerov so le-te zadostne za izvedbo procesa izdelave paketa. Kljub temu se lahko pojavi potreba po poljubni akciji kot na primer: zagon nekega drugega programa tekom namestitve ali klic funkcije iz knjižnice DLL. Poljubne akcije se lahko kličejo kadarkoli v procesu. Pri njihovi izdelavi je potrebno upoštevati nekaj pravil, ki so definirana v orodju s katerim pripravljamo paket MSI, več o pravilih pa si lahko preberemo v [25] in [29].

3.5.1 Izbira orodja za pripravo paketa MSI

Program »Orca« omogoča predvsem enostavno pregledovanje in prilagajanje obstoječega MSI paketa, ni pa primerno za avtomatizacijo procesa izdelave paketa pri izdelavi izdaje. Za ta namen obstaja več orodij. V podjetju izbirali predvsem med brezplačnimi ter primernimi za delo s programom Visual Studio 2010: Visual Studio Installer, InstallShield 2010 Limited Edition in Windows Installer XML. Vsako orodje bo opisano le na kratko, saj podrobna

analiza presega okvire te diplomske naloge. Podrobnosti primerjave orodij si lahko preberemo v [26, 27].

3.5.1.1 Visual Studio Installer

»Visual Studio Installer« je na voljo v programih Microsoft Visual Studio in sicer v izdajah Professional, Premium in Ultimate. Za izdelavo paketa se uporablja uporabniški vmesnik. Je enostaven za uporabo vendar so njegove funkcionalnosti za izdelavo paketa MSI omejene. Za vse nadaljnje dodelave se lahko uporabi program »Orca«. Zaradi zahtevane kompleksnosti s strani produkta, avtomatizacija postopka izdelave s tem orodjem ni mogoča. Poleg tega so se pri Microsoftu odločili, da orodja ne bodo vključili v naslednjo različico programa Visual Studio, saj ga bo nadomestil program »InstallShield 2010 Limited Edition«.

3.5.1.2 InstallShield 2010 Limited Edition

»InstallShield 2010 Limited Edition« je brezplačna različica programa »InstallShield« in je po registraciji na voljo v programu Microsoft Visual Studio. V osnovi nadomešča Visual Studio Installer, pretvorba iz enega v drugega pa je s pomočjo orodja za uvoz enostavna. Tudi to orodje omogoča izdelavo paketa preko uporabniškega vmesnika, z njim pa se da narediti skoraj vse, kar se da narediti v programu Visual Studio Installer. Morebitni problem predstavlja omejitev dialogov, saj brezplačna izdaja ne podpira možnosti izdelave dodatnega poljubnega dialoga. Prav tako ne podpira dela s podatkovno bazo. Omenjeni možnosti sta seveda podprti v plačljivi različici orodja, katere cena zraste v primeru več licenc preko 2000,00€. Posledično se v podjetju za to orodje nismo odločili.

3.5.1.3 Windows Installer XML (WiX)

»Windows Installer XML« je odprto kodni projekt podjetja Microsoft, ki podpira vse funkcionalnosti programske komponente Windows Installer in je na voljo preko Microsoftove spletne strani. Kot ostali dve opisani orodji tudi s tem delo poteka v programu Microsoft Visual Studio vendar ta za izdelavo paketa ne uporablja uporabniškega vmesnika. Vse se definira preko datotek z XML sintakso. Krivulja učenja pri tem orodju je v primerjavi z ostalima orodjema veliko bolj strma, saj je potrebno imeti že za izdelavo preprostega paketa veliko znanja, prav tako pa je časovno potratna tudi vsaka prilagoditev. Orodje ima poleg neprijaznega vmesnika še eno večjo pomanjkljivost in sicer ne omogoča avtomatskega nameščanja programov, ki so predpogoj za delovanje programa, ki ga nameščamo. Pri nameščanju produkta »Insure« bi rabili namestitev ogrodja Microsoft .NET Framework 4.0 v kolikor to še ni nameščeno.

Kljub kompleksnosti ima orodje veliko dobrih lastnosti. Je brezplačno, zato se ga lahko brez skrbi glede licenc namesti na okolja več razvijalcev ter tudi na strežnik, kjer je nameščen strežnik CCNET. Vse delo poteka preko urejanja XML datotek, zato je spremembe enostavno nadzirati s pomočjo sistema SVN. Orodje omogoča delo s podatkovnimi bazami in klice poljubnih akcij izdelanih z uporabo .NET izvorne kode. Izdelava paketov je omogočena preko okolja z orodno vrstico, kar poenostavi integracijo z orodjem NAnt in posledično tudi z avtomatizacijo preko strežnika CCNET.

3.5.2 Vpeljava funkcionalnosti izdelave paketa MSI v proces priprave izdaje

V podjetju smo se odločili za uporabo orodja WiX, o katerem si lahko več preberemo v [28]. Podpira praktično vse funkcionalnosti programske komponente Windows Installer, ki jih potrebujemo in je hkrati brezplačno. Čas potreben za izdelavo prvega paketa je bil zaradi strme krivulje učenja res malo daljši, vendar imamo sedaj potrebno znanje, s katerim se morebitne dodelave opravi dokaj hitro.

Funkcionalnost za pripravo paketa ima izdelano svojo skripto NAnt, katero pokličem iz skripte NAnt za pripravo celotne izdaje (Slika 31). Ob klicu se nastavijo naslednji podatki: verzija produkta, ime stranke, GUID, jezik v katerem bo potekala namestitev, pot kje se nahajajo podatki, ki se morajo prenesti v paket in pot kamor se odloži paket.

```
<target name="version.msi">
  <call target="version.get" />
  <nant buildfile="${deploy.buildfile}">
    <properties>
      <property name="productversion" value="${version.major}.${version.minor}.${version.build}.${version.revision}" />
      <property name="implementation" value="${deploy.implementation}" />
      <property name="upgradecode" value="${GUID}" />
      <property name="culture" value="${deploy.culture}" />
      <property name="rootpath" value="${deploy.folder.default}\${release.folder}" />
      <property name="path.msi.output" value="${deploy.folder.default}\${release.folder}\${deploy.folder.setup}" />
    </properties>
  </nant>
</target>
```

Slika 31: klic skripte NAnt za pripravo paketa MSI

Skripta za pripravo paketa MSI (Slika 32) se v osnovi deli na: prevajanje in pripravo poljubnih akcij, sprejem in nastavljanje osnovnih informacij o produktu, priprava izvornih datotek XML za WiX na podlagi podanih map in datotek ter prevajanje in združitev vsega v paket MSI.

```
<target name="build">
    <call target="clean"/>
    <msbuild project="Insure.Install.CustomActions\Insure.Install.CustomActions.csproj" verbosity="minimal" />
    <call target="init.productinfo"/>
    <call target="harvest.WebService"/>
    <call target="harvest.Client"/>
    <call target="build.msi"/>
</target>
```

Slika 32: skripta NAnt za pripravo paketa MSI

Prevajanje poljubnih akcij prevede projekt izdelan s programom Microsoft Visual Studio, pri izdelavi katerega je potrebno upoštevati nekaj pravil in sicer: projekt mora vsebovati referenco na »Microsoft.Deployment.WindowsInstaller«, objekt znotraj projekta mora imeti nastavljen atribut »CustomAction« ter mora biti definiran kot statičen. Natančna navodila kako narediti poljubno akcijo si lahko preberemo v [25] in [29]. Programska komponenta Windows Installer ne more dostopati direktno do knjižnic DLL, zato je potrebno vsako od njih predelati v paket, ki je z njo kompatibilna. To naredimo s klicem orodja »MakeSfxCA.exe«, ki je del razvojnega ogrodja WiX. Predelavo se pokliče znotraj projekta »Insure.Install.CustomActions.csproj« v dogodku, ki se zgodi takoj po uspešno prevedeni

izvorni kodi. Konkretna poljubna akcija je napisana v programskem jeziku »C#«, njena naloga pa je priprava nastavitvenih datotek za odjemalca in strežnik.

V koraku sprejema in nastavljanja osnovnih informacij paketa se prebere vse parametre, s katerimi je bila poklicana skripta NAnt in se z njimi zgradi datoteko XML, katero kasneje upošteva proces izdelave paketa MSI (Slika 33).

```
<target name="init.productinfo">
  <echo file="ProductInfo.wxi">&lt;?xml version="1.0" encoding="utf-8"?&gt;
    &lt;Include&gt;
      &lt;?define UpgradeCode = "${installer.upgradecode}" ?&gt;
      &lt;?define Implementation = "${installer.implementation}" ?&gt;
      &lt;?define ProductVersion = "${installer.productversion}" ?&gt;
      &lt;?define ProductVersionZeroPadded = "${functions::zero-pad-version(installer.productversion)}" ?&gt;
      &lt;?define RootPath = "${installer.rootpath}" ?&gt;
    &lt;/Include&gt;
  </echo>
</target>
```

Slika 33: skripta NAnt za pripravo nastavitvene datoteke XML z osnovnimi podatki

Priprava podatkov produkta se izvede s pomočjo orodja »Heat.exe, ki je del razvojnega ogrodja WiX (Slika 34).

```
<target name="harvest.Client">
  <property name="harvest.path" value="${installer.rootpath}\Source\Client"/>
  <property name="harvest.componentgroup" value="ClientComponent"/>
  <property name="harvest.directoryreference" value="FOLDER_CLIENT"/>
  <property name="harvest.variablename" value="var.ClientPath"/>
  <property name="harvest.outputfile" value="Features\ClientComponent.wxs"/>
  <property name="harvest.xslt" value=""/>
  <call target="harvest"/>
</target>

<target name="harvest">
  <exec program="${path.heat}">
    <arg value="dir"/>
    <arg path="${harvest.path}"/>
    <arg value="-cg"/>
    <arg value="${harvest.componentgroup}"/>
    <arg value="-wixvar"/>
    <arg value="-srd"/>
    <arg value="-sreg"/>
    <arg value="-ke"/>
    <arg value="-gg"/>
    <arg value="-g1"/>
    <arg value="-dr"/>
    <arg value="${harvest.directoryreference}"/>
    <arg value="-var"/>
    <arg value="${harvest.variablename}"/>
    <arg value="-o"/>
    <arg path="${harvest.outputfile}"/>
    <arg value="-t" if="${string::ends-with(property::get-value('harvest.xslt'), 'xslt')}"/>
    <arg path="${harvest.xslt}" if="${string::ends-with(property::get-value('harvest.xslt'), 'xslt')}"/>
  </exec>
  <includewxi sourcefile="${project::get-base-directory()}\${harvest.outputfile}" includefile="../Includes.wxi" />
</target>
```

Slika 34: skripta NAnt za pripravo sheme XML produkta

Orodju se kot parameter poda mapo, kjer se nahajajo datoteke, ter parametre s katerimi določamo podrobnosti kako naj interpretira datoteke ter mape znotraj podane mape. Kot izhod generira na podlagi sheme XML »wix.xsd« datoteko XML s končnico »wxs«, ki je izvorna datoteka v sistemu WiX, podobno kot je datoteka ».cs« izvorna datoteka za programski jezik C#.

Izvorne datoteke »wxs« prejme kot vhod WiX prevajalnik »candle.exe«, ki je del razvojnega ogrodja WiX (Slika 35). Vsako izvorno datoteko prevede in zgradi na podlagi sheme XML »objects.xsd« datoteko XML s končnico »wixobj«. Vsebina datoteke je sestavljena iz elementov <table/>, <row/> in <field/>, ki predstavljajo neobdelane podatke, ki bodo vstavljeni v podatkovno bazo Windows Installer.

```
<target name="build.msi">
  <loadtasks assembly="${path.wix}\Microsoft.Tools.WindowsInstallerXml.NAntTasks.dll" />
  <candle
    exedir="${path.wix}"
    out="${path.obj}\"
    rebuild="true"
    extensions="WixNetFxExtension;WixIIsExtension;WixUtilExtension;WixUIExtension"
    warningsaserrors="true">
    <sources basedir="${project::get-base-directory()}">
      <include name="Product.wxs" />
      <include name="Features\ClientComponent.wxs" />
      <include name="Features\WebServiceIISComponent.wxs" />
      <include name="Features\WebServiceComponent.wxs" />
      <include name="UI\FeatureTree_Extended.wxs" />
      <include name="UI\IisWebSitesDlg.wxs" />
    </sources>
  </candle>
  <!-- Set cultures to either 'en-US' or 'something;en-US' -->
  <property name="temp.cultures" value="en-US" />
  <property name="temp.cultures" value="${installer.culture};${temp.cultures}"
            unless="${property::get-value('installer.culture') == 'en-US'}" />
  <light
    exedir="${path.wix}"
    out="${path.msi.output}\${filename.output}.msi"
    warningsaserrors="true"
    cultures="${temp.cultures}"
    suppressices="ICE61"
    extensions="WixNetFxExtension;WixIIsExtension;WixUtilExtension;WixUIExtension"
    rebuild="true">
    <sources basedir="${path.obj}">
      <include name="*.wixobj" />
    </sources>
    <localizations basedir="${project::get-base-directory()}">
      <include name="Languages\${installer.culture}.wxl"
               unless="${property::get-value('installer.culture') == 'en-US'}" />
      <include name="Languages\en-us.wxl" />
    </localizations>
  </light>
</target>
```

Slika 35: skripta NAnt prevajanje in združitev vsega v paket MSI

Po uspešnem prevajanju vseh izvornih datotek je potrebno uporabiti WiX povezovalnik »light.exe«, ki je del razvojnega ogrodja WiX. Le-ta prejme kot vhod seznam prevedenih datotek. Med njimi najde vhodno sekcijo, nato pa z rekurzijo preko sklicev poveže vse objekte med sabo. Na koncu za vsako datoteko pridobi različico, jezik in zgoščeno vrednost ter še ostale potrebne standardne akcije, ki so potrebne za uspešno izvedbo namestitve.

3.6 Vpeljava avtomatskega testiranja aplikacije

Skupaj s konstantnim spreminjanjem izvorne kode nastajajo tudi napake. Zaradi lažjega vzdrževanja je smiselno izdelati teste, ki nam na določen interval testirajo aplikacijo. Kratkoročno je pisanje testov zamudno in na videz nepomembno delo, dolgoročno pa nam pripomorejo k hitrejšem zaznavanju in odpravi napak še preden se spremembo prenese na produkcijsko okolje pri stranki.

Testiranje lahko v osnovi delimo na dva dela: testiranje na nižjem nivoju, kjer se testira čim bolj osnovne funkcije ter testiranje na višjem nivoju, kjer se testira funkcionalnosti čim bližje uporabniškemu vmesniku.

Pri izdelavi testov si je smiselno pomagati z že izdelanimi orodji oziroma ogrodji. V podjetju smo se odločili za uporabo ogrodja NUnit, saj je to zelo pogosto uporabljeno ogrodje med razvijalci .NET aplikacij in je hkrati brezplačno. Poleg tega smo za testiranje aplikacije izbrali še plačljivo orodje Ranorex, za katero je podjetje že kupilo licence za razvoj zaradi potreb na drugem oddelku.

3.6.1 Ogrodje NUnit

NUnit je odprtokodno ogrodje namenjeno testiranju aplikacij razvitih z Microsoft .NET orodji. Temelji na ideji izdelovanja testov na nižjem nivoju, torej testiranju posameznih enot oziroma funkcij. Več o orodju si lahko preberemo v [15]. Tako testiranje je enostavno, saj za posamezno funkcijo vemo kakšni so vhodi v funkcijo in kakšen je pričakovan izhod. Posledično je enostavno tudi vzdrževanje. V primeru spremembe posamezne funkcije je potrebno ustrezno popraviti še test.

Pri izdelavi testov je potrebno upoštevati nekaj osnovnih pravil. Za vsak objekt, ki ga želimo testirati naredimo nov objekt namenjen pisanju testov, ki ima referenco na objekt »NUnit.Framework«. Objektu nastavimo atribut »[TextFixture]«, posamezni testni metodi pa nastavimo atribut »[Test]«. Znotraj metode naredimo novo instanco objekta, ki ga želimo testirati, in pokličemo želeno funkcijo znotraj objekta. Ob klicu nastavimo testne vhodne parametre in ob izhodu definiramo kakšen je pričakovan rezultat. Zaradi večje preglednosti je smiselno za pisanje testov narediti nov projekt znotraj rešitve v programu Microsoft Visual Studio. Natančna navodila si lahko preberemo v [30], primer pa si lahko ogledamo v [38].

Zaradi enostavnosti integracije NUnit ogrodja z Microsoft Visual Studio lahko posamezne teste vsak razvijalec poganja takoj po izdelavi, še preden jih shrani na repozitorij strežnika SVN. Ob zagonu testa nam ogrodje preveri ali je bil ob klicu funkcije vrnjen pričakovan

rezultat in nam v primeru enakosti test označi kot uspešen in v primeru neenakosti označi kot neuspešen.

Izvedba testov na nižjem nivoju je zelo hitra, zato se lahko teste zažene ob vsaki spremembi izvorne kode na repozitoriju strežnika SVN. V primeru, da razvijalec naredi napako pri spreminjanju neke funkcije ali pa pozabi test ustrezno prilagoditi, bo napako v roku nekaj minut javil strežnik CCNET. Verjetnost, da pride napaka na produkcijsko okolje se tako močno zmanjša.

Za namene testiranja aplikacije ob izdelavi izdaje sem napisal skripto Nant (Slika 36). Le-ta prevede tako projekt, ki ga testiramo kot tudi projekt, ki vsebuje teste. Nato preprosto pokliče program NUnit preko datoteke »nunit-console-x86.exe«, kot parameter pa mu poda knjižnico DLL, ki je rezultat prevajanja projekta, ki vsebuje teste. Poleg tega sem podal kot parameter še pot kamor se shrani rezultat prevajanja testov v obliki datoteke XML, katero zna prikazati strežnik CCNET (Slika 37).

```
<target name="compile.test.query" description="Builds query tests">
    <msbuild project="Insure.Data\Insure.Data.csproj" verbosity="minimal" />
    <msbuild project="Insure.Data.Tests\Insure.Data.Tests.csproj" verbosity="minimal" />
</target>
<target name="test.query" description="Runs query tests">
    <exec program="nunit-console-x86.exe">
        <arg value="Insure.Data.Tests\bin\Debug\Insure.Data.Tests.dll" />
        <arg value="/xml=Insure.Data.Tests\bin\Debug\Insure.Data.Tests.dll-Results.xml" />
    </exec>
</target>
```

Slika 36: skripta NAnt za prevajanje in zagon testa poizvedb SQL

NUnit Test Results

Summary

Assemblies tested: 1
Tests executed: 2221
Passes: 2221
Fails: 0
Ignored: 16

Assembly Test Details:

Test Fixture	Status	Progress
AdministracijaQueriesTest		(34/34)
ConnectionTest		(1/1)
DefinicijaProduktovQueriesTest		(64/64)
DefinicijaZavarovanjQueriesTest		(116/116)
DenarniTokQueriesTest		(229/229)
IQueriesBaseTest		(18/18)
IzvrsbeQueriesTest		(37/37)
Method_PodatkiProviziji		(9/9)
Method_PodatkiProvizijZiv		(4/4)
PozavarovanjeQueriesTest		(36/36)
ProvizioniranjeQueriesTest		(25/25)
QueriesFactoryTest		(3/3)
RegisterOsebQueriesTest		(147/147)
SkodeQueriesTest		(286/286)
SkodeRegresiQueriesTest		(5/5)
StoredProcedureQueriesTest		(2/2)

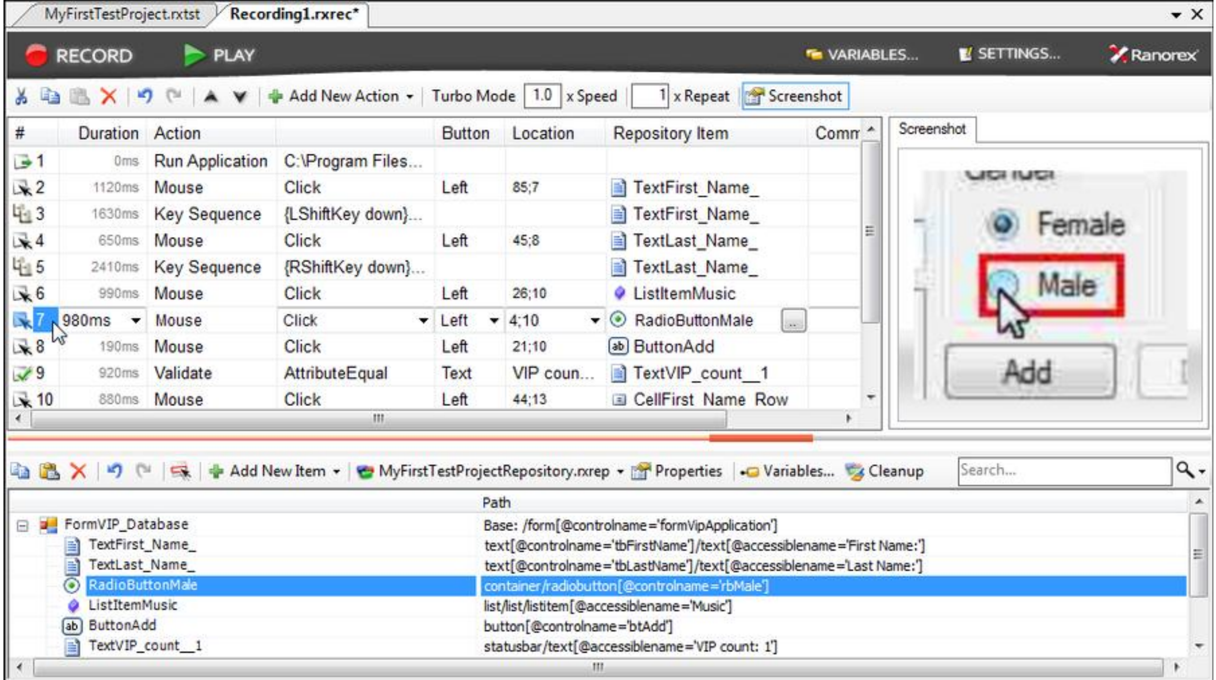
Slika 37: Prikaz rezultat testiranja poizvedb SQL na podlagi izhodne datoteke na strežniku CCNET

V podjetju smo s pomočjo ogrodja NUnit napisali teste za vse poizvedbe SQL. Poizvedb je vse skupaj približno 2200, čas izvedbe testov pa je približno ena minuta.

3.6.2 Orodje Ranorex

Z orodjem Ranorex se testira aplikacijo na višjem nivoju, saj se testi izvajajo direktno preko uporabniškega vmesnika. Orodje temelji na izredno natančnem prepoznavanju in identificiranju elementov uporabniškega vmesnika. Izdelovanje testov je enostavno. V orodju »Ranorex Studio« se odpre nov projekt v katerem se na začetku nastavi katero aplikacijo se bo testiralo, nato pa se s klikom na gumb »Record« začne snemanje. Ob zagonu se odpre izbrana aplikacija, orodje pa prične zaznavati in shranjevati v svoj repozitorij vse interakcije z uporabniškim vmesnikom. V tem koraku je potrebno simulirati normalno uporabo aplikacije in sicer funkcionalnost, ki jo želimo testirati. Orodje iz testa časovno izniči vse premike miške in počasno tipkanje ter s tem pohitri test. Podrobnosti o orodju ter o možnih načinov testiranja z si lahko preberemo v [31].

Po končanem snemanju se lahko pogleda podrobnosti posameznega koraka, ki smo ga izvedli tekom snemanja testa (Slika 38).



Slika 38: Prikaz podrobnosti posameznega koraka v izvedenem testu z orodjem Ranorex

Preko gumba »Play« sprožimo prevajanje in zagon testa, ki odpre aplikacijo in ponovi vse korake, katere smo naredili mi, ko smo test snemali. Ob prevajanju se izdela tudi zagonska datoteka, preko katere lahko zaženemo posneti test tudi izven programa »Ranorex Studio«.

Testiranje preko uporabniškega vmesnika je zelo intuitivno, saj se testira aplikacijo na način, kot jo uporablja stranka. Nekoliko težje je vzdrževanje testov, saj orodje temelji na imenih posameznih kontrol na ekranu aplikacije. V primeru, da se spreminja ime ekrana ali posameznega elementa uporabniškega vmesnika, je potrebno to ustrezno popraviti tudi v testu. Če je teh sprememb veliko je smiselno premisliti, ali se popravke uredi ročno ali pa se ponovno posname test.

Slaba lastnost testiranja na višjem nivoju je počasnost, zato se ti testi ne morejo zaganjati ob vsaki spremembi izvorne kode na repozitoriju strežnika SVN, ampak ob določenih intervalih ali pa celo samo ponoči. Posledično razvijalec v primeru povzročene napake ne dobi odziva dovolj hitro in lahko pride do situacije, kjer se namesti določena funkcionalnost na produkcijsko okolje stranke brez ustreznega testiranja.

Integracija testov s strežnikom CCNET je dokaj enostavna. Za vsak izdelan test se iz skripte NAnt s pomočjo značke »<exec>« pokliče njegovo zagonsko datoteko. Strežnik CCNET nato spremlja kakšen izhod vrne test. Če je vrednost 0, potem je zanj to znak, da je test uspel, če pa vrne vrednost različno od nič, je to znak, da je pri testu prišlo do napake. Problem tega pristopa je v zahtevi, da je na računalniku, kjer teže strežnik CCNET naložena licenca za zaganjanje testov izdelanih s programom Ranorex Studio.

V podjetju smo se odločili za drugačen pristop. Ne želimo namreč nameščati preveč stvari na strežnik, kjer teže proces CCNET, poleg tega pa je treba licenco dokupiti. Zato imamo za razvoj in zagon testov Ranorex na voljo poseben računalnik. Strežnik CCNET za zagon testov

pokliče zagonsko datoteko, ta pa preko spletne storitve zažene ustrezne teste na računalniku, kjer je nameščena licenca za zaganjanje in izdelavo testov. Posledično je zagon testov malo bolj kompleksen, kar se odrazi predvsem pri javljanju morebitnih napak pri posameznih testih nazaj na strežnik CCNET.

4 Sklepne ugotovitve

V diplomskem delu sem si zastavil več ciljev s katerimi bi optimiziral obstoječi sistem za upravljanje z izdajami. Vsakega izmed njih sem brez večjih težav tudi dosegel (Tabela 1).

Pomanjkljivosti	Rešitve
ročno povečevanje različice izdaje	avtomatizacija povečevanja oznake različice izdaje
slab nadzor nad izvorno kodo vključeno v izdajo	upravljanje z izdajami izvorne kode na nivoju repozitorija strežnika SVN
asinhrone spremembe PDM in izdaje aplikacije	shranjevanje skript SQL skupaj z izvorno kodo na repozitorij strežnika SVN in zagon teh skript ob namestitvi izdaje
ni podpore za prilagoditve aplikacije potrebam stranke	uvedba modulizacije aplikacije
kompleksen postopek namestitve aplikacije z veliko možnosti za pojavo napake	vpeljava programske komponente Windows Installer
ročno testiranje aplikacije pred izdelavo izdaje	vpeljava avtomatskega testiranja aplikacije
ni zgodovine namestitev različic izdaj	upravljanje z izdajami izvorne kode na nivoju repozitorija strežnika SVN

Tabela 1: prikaz pomanjkljivosti in rešitev realiziranih v diplomski nalogi

Tabela 1 prikazuje seznam pomanjkljivosti, s katerimi smo se soočali v podjetju pred optimizacijo sistema za upravljanje z izdajami, ter seznam rešitev s katerimi sem pomanjkljivosti odpravil.

Prednosti izdelanega sistema je več:

- Ob vsakem shranjevanju izvorne kode na repozitorij strežnika SVN se preko strežnika CCNET sproži prevajanje kode ter zagon testov. Zaradi dobre odzivnosti sistema so rezultati na voljo v roku nekaj minut.
- Podprta je funkcionalnost za prilagajanje produkta potrebam strank, hkrati pa ostaja večina funkcionalnosti produkta vsem strankam skupna, kar olajša vzdrževanje.
- Zaradi uporabe vej na repozitoriju strežnika SVN je odprava kritičnih napak na produkcijskem okolju enostavna in hitro izvedljiva.
- Shranjevanje skript SQL na repozitorij strežnika SVN olajša delo skrbniku PDM, hkrati pa omogoča namestitev skript SQL skupaj z aplikacijo.
- Zaradi uporabe programske komponente Windows Installer je postopek namestitve zelo enostaven zaradi česar lahko nameščanje aplikacije prepustimo stranki.

Kljub doseženim ciljem, ima sistem tudi slabosti:

- Avtomatično testiranje preko uporabniškega vmesnika je dolgotrajno, zato se ti testi ne zaganjajo ob vsaki spremembi izvorne kode na repozitoriju strežnika SVN. Posledično lahko pride do situacije, kjer se izdela izdajo brez ustreznega testa aplikacije.

- Napisane skripte SQL nimajo ustreznih razveljavitvenih skript, ki bi vzpostavile na podatkovni bazi stanje kakršno je bilo pred zagonom skripte. Razveljavitev izdaje aplikacije in vzpostavitev prejšnjega stanja je enostavno z vidika aplikacije same, na podatkovni bazi pa je potrebno veliko ročnega dela.
- Za enkrat še ni podprto avtomatično zaganjanje skript SQL ob namestitvi aplikacije z uporabo programske komponente Windows Installer. Skripte se le odloži na določeno mesto, za zagon skript pa mora poskrbeti tisti, ki verzijo namešča.

Optimizacija sistema je potekala približno 2 leti. V vmesnem času so se pojavila nova orodja za nadzor izvorne kode in novi strežniki z zvezno integracijo. S preučitvijo orodij bi lahko verjetno obstoječi sistem še dodatno optimiziral.

Glavna nevarnost sistema sta njegov obseg in kompleksnost. Sistem je sestavljen iz veliko različnih orodij in tehnologij, kar lahko povzroča težave v primeru potrebe po spremembi sistema.

Literatura

[1] WIKIPEDIA, Release management
Dostopno na: http://en.wikipedia.org/wiki/Release_management
Dostopano januarja 2012

[2] MSDN, Smart Client Architecture and Design Guide
Dostopno na: http://msdn.microsoft.com/en-us/library/ff647359.aspx
Dostopano januarja 2012

[3] MSDN, Windows Forms
Dostopno na: http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx
Dostopano januarja 2012

[4] MSDN, .NET Framework 4.0
Dostopno na: http://msdn.microsoft.com/en-us/netframework/aa569263
Dostopano januarja 2012

[5] MSDN, ASP.NET Overview
Dostopno na: http://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx
Dostopano januarja 2012

[6] MSDN, Solution (.sln) File
Dostopno na: http://msdn.microsoft.com/en-us/library/bb165951%28v=vs.80%29.aspx
Dostopano januarja 2012

[7] MSDN, Visual Studio
Dostopno na: http://msdn.microsoft.com/en-us/vstudio/aa718325
Dostopano januarja 2012

[8] APACHE, Subversion
Dostopno na: http://subversion.apache.org/
Dostopano decembra 2011

[9] VISUALSVN, Visual SVN Server
Dostopno na: http://www.visualsvn.com/server/
Dostopano oktobra 2011

[10] M. Fowler and M. Foemmel, Continuous Integration
Dostopno na: http://martinfowler.com/articles/continuousIntegration.html
Objavljeno 1.5.2006

[11] THOUGHTWORKS, CruiseControl.NET
Dostopno na: http://confluence.public.thoughtworks.org/display/CCNET
Dostopano novembra 2011

[12] SOURCEFORGE, NAnt Home Page
Dostopno na: http://nant.sourceforge.net/
Dostopano decembra 2011

[13] WIKIPEDIA, Software versioning
Dostopno na: http://en.wikipedia.org/wiki/Software_versioning
Dostopano februarja 2012

[14] SYBASE, Power Designer
Dostopno na: http://www.sybase.com/products/modelingdevelopment/powerdesigner
Dostopano februarja 2012

[15] NUNIT, NUnit
Dostopno na: http://www.nunit.org/

Dostopano decembra 2011

[16] MSDN, Visual Studio Templates
Dostopno na: http://msdn.microsoft.com/en-us/library/6db0hwky%28v=vs.80%29.aspx
Dostopano februarja 2012

[17] WIKIPEDIA, Branching (software)
Dostopno na: http://en.wikipedia.org/wiki/Branching_%28software%29
Dostopano februarja 2012

[18] APACHE, Subversion Best Practices
Dostopno na:
http://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html
Dostopano decembra 2011

[19] SVNBOOK, Branching and Merging
Dostopno na: http://svnbook.red-bean.com/en/1.1/ch04.html
Dostopano decembra 2011

[20] MSDN, Version Class
Dostopno na: http://msdn.microsoft.com/en-us/library/hdxyt63s%28vs.71%29.aspx
Dostopano februarja 2012

[21] MSDN, Overview of Windows Installer
Dostopno na:
http://msdn.microsoft.com/en-us/library/windows/desktop/aa370566%28v=vs.85%29.aspx
Dostopano januarja 2012

[22] PCFASTER, Windows Installer Utility
Dostopno na: http://www.pcfaster.net/Windows_Installer_Utility_Article.html
Dostopano januarja 2012

[23] WINDOWSNETWORKING, MSI Packaging Tools
Dostopno na:
http://www.windowsnetworking.com/articles_tutorials/msi-packaging-tools.html
Dostopano januarja 2012

[24] MSDN, Orca.exe
Dostopno na: http://msdn.microsoft.com/en-us/library/aa370557.aspx
Dostopano februarja 2012

[25] MSDN, Windows Installer - Custom Actions
Dostopno na:
http://msdn.microsoft.com/en-us/library/windows/desktop/aa368066%28v=vs.85%29.aspx
Dostopano januarja 2012

[26] MSDN, Choosing a Windows Installer Deployment Tool
Dostopno na: http://msdn.microsoft.com/en-us/library/ee721500.aspx
Dostopano oktobra 2011

[27] WORDPRESS, Visual Studio Installer vs. InstallShield LE vs. WiX
Dostopno na:
http://eduardralph.wordpress.com/2011/10/12/visual-studio-installer-vs-installshield-le-vs-wix/
Dostopano februarja 2012

[28] SOURCEFORGE, Windows Installer XML Overview
Dostopno na: http://wix.sourceforge.net/manual-wix2/overview.htm

Dostopano januarja 2012

[29] LOSTECHIES, WIX and Custom Actions
Dostopno na:
http://lostechies.com/gabrielschenker/2010/05/18/wix-and-custom-actions/
Dostopano januarja 2012

[30] NUNIT, NUnit Quick Start
Dostopno na: http://www.nunit.org/index.php?p=quickStart&r=2.6
Dostopano decembra 2011

[31] RANOREX, Test Automation
Dostopno na: http://www.ranorex.com/test-automation-features.html
Dostopano februarja 2012

[32] V. Lestideau, N. Belkhatir, P.-Y. Cunin, Towards automated software component configuration and deployment, Proceedings of PDTSD’02, 2002

[33] MICROSOFT, SQL Server
Dostopno na: http://www.microsoft.com/sqlserver/en/us/default.aspx
Dostopano februarja 2012

[34] P. M. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, 2007

[35] A. Pavlo, P. Couvares, R. Gietzel, A. Karp, I. D. Alderman, M. Livny, C. Bacon, The NMI Build & Test Laboratory: Continuous Integration Framework for Distributed Computing Software, Proceedings of LISA '06: 20th Large Installation System Administration Conference, 2006

[36] J. Holck, N. Jorgensen, Continuous Integration and Quality Assurance: a case study of two open source projects
Dostopno na: http://journals.sfu.ca/acs/index.php/ajis/article/view/145/125
Dostopano februarja 2012

[37] P. Burba, Understanding the internals of Subversion's merge tracking feature
Dostopno na: http://www.open.collab.net/community/subversion/articles/merge-info.html
Dostopano marca 2012

[38] M. Georgescu, D. Milodin, PhD, Open Source Tools to Assist in the Development of Software Applications
Dostopno na: http://opensourcejournal.ro/2010-Volume02/number02/paper004-fullpaper.pdf
Dostopano januarja 2012