Matematički fakultet
Univerzitet u Beogradu

Master rad

„Algoritmi za poređenje sadržaja Autocad datoteka korišćenjem C#-a“

Ivan Pavlović

Beograd
2011. godina

„Algoritmi za poređenje sadržaja Autocad datoteka korišćenjem C#-a“

Ivan Pavlović

Članovi komisije:

dr Dušan Tošić, mentor
dr Miroslava Antić
dr Miroslav Marić

Sadržaj

Predgovor

U savremenoj arhitekturi korišćenje softvera u projektovanju je postao standardni način rada. Na velikim projektima radi tim projektanata, zbog čega se javlja mnogo revizija jednog projekta. Da bi se lakše pratile izmene, neophodno je napraviti automatsko poređenje datoteka. Poređenje je neophodno i za automatsko spajanje dve različite datoteke koje su nastale od iste revizije.

U prvom poglavlju se govori o programskom jeziku C#, jer je to jezik u kojem će biti napisani programi za realizaciju algoritama. C# je izabran jer je objektno orijentisan jezik u kojem se jednostavno razvijaju aplikacije i ima dobru podlogu u bibliotekama koje su razvijane od strane mnogih korisnika.

U drugom poglavlju se govori o opštim stvarima vezanim za Autocad. Autocad je izabran jer je to najpopularniji softverski prozvod među projektantima.

U trećem poglavlju se govori o sadržaju Autocad datoteka i pojmovima koje je neophodno razumeti da bi mogla da se razvije aplikacijja.

U četvrtom poglavlju se govori o samim algoritmima za poređenje, prikazivanju rezultata poređenja i implementaciji u C#-u. Navedeni kodovi, koji se odnosi na vađenje podataka iz klasa biblioteka za rad sa DWG datotekama, se mogu smatrati pseudokodovima.

Peto poglavlje se odnosi na primenu algoritama za poređenje u spajanju datoteka.

Zahvaljujem se profesoru Dušanu Tošiću koji je prihvatio da bude mentor pri izradi ovog rada, pomogao pri formulisanju teme, i svojim sugestijama pomogao poboljšanju kvaliteta ovog rada.

Zahvaljujem se članovima komisije: profesorki Miroslavi Antić i profesoru Miroslavu Mariću koji su pročitali rad i svojim sugestijama doprineli uobličavanju rada.

Zahvaljujem se i kolegama iz firme Bexel Consulting, koji su me projektima Document Management i Document Comparer motivisali da izaberem ovu temu, kao i na stručnoj podršci i toleranciji kada sam izostajao sa posla za vreme višemesečnog rada na ovoj temi.

Programski jezik C-#

1.1. Opšte o C#-u

C# je programski jezik koji podržava više paradigmi: imperativnu, deklarativnu, funkcionalnu, generičku i objektno orijentisanu. To je strogo tipizirani programski jezik. Razvio ga je Microsoft u sklopu .Net platforme, a kasnije je odobren kao standrad od strane Ecma (ECMA-334) i ISO (ISO/IEC 23270). C# je jedan od jezika koji je razvijen za CLI. CLI (engl. Common Language Infrastructure) je opšta jezička infrastruktura koja opisuje izvršni kod i okruženje za njegovo izvršenje, koje čini .Net Framework.

C# je predviđen da bude jednostavan, moderan, objektno orijentisani jezik opšte upotrebe. Vođa razvojnog tima je Anders Hejlsberg. Najnovija verzija je C# 4.0 od 12. aprila 2010.

Ime C# je inspirisano muzičkom notom cis, tj c povišeno, slično kao kod C++-a, gde je to c uvećano za jedan. Pošto se originalni muzički simbol ♯ ne nalazi na tastaturi, u pisanju imena se koristi #. Isti simbol se koristi i u drugim .NET jezicima kao što su J# (nastao od Jave 1.1), A# (nastao od Ade), funkcionalni jezik F# i drugi.

1.2. Istorija

Za vreme razvoja .NET Framework-a, klasne biblioteke su pisane korišćenjem kompajlera za verziju programskog jezika C (Simple Managed C - SMC), gde korisnik ne mora da vodi računa o oslobađanju memorije. U januaru 1999. godine Anders Hejlsber je formirao tim za pravljenje novog jezika koji je vremenom dobio ime Cool (C-like Object Oriented Language). Menadžeri Microsofta su hteli da zadrži ime „Cool”, ali su odustali zbog problema sa zaštitnim znakom. U julu 2000. godine .NET projekat je javno objavljen, jezik je preimenovan u C# i sve klasne biblioteke su prebačene u C#.

Vodeći dizajner i glavni arhitekta je Anders Hejlsberg, koji je prethodno radio na Turbo Paskalu (Turbo Pascal), jednoj verziji programskog jezika Delfi (Delphi) i na jeziku

J++ (Visual J++). U jednom intervjuu, Anders je rekao da su mane C++-a, Jave, Delfija i Smaltolka (Smalltalk) dovele do dizajna C#-a.

U početku je C# dosta podsećao na Javu, ali od verzije C# 2.0, u novembru 2005. C# i Java su počeli da se razvijaju u različitim pravcima i prestali toliko da liče.

Jedna od najvećih razlika nastala je pojavom parametarskih klasa u oba jezika, koje su implementirane na različit način. C# koristi konkretizaciju kako bi obezbedio da generički objekti prve klase mogu da se koriste kao bilo koja druga klasa, uz generisanje koda koji se izvršava pri učitavanju klasa. Parametarske klase u Javi su sintaksna mogućnost i ne utiču na generisani bajtovski kod, zato što kompajler izvršava brisanje tipova na generičkim informacijama posle provere njihove ispravnosti.

C# je dodao nekoliko važnih mogućnosti: prilagodio se i funkcionalnom stilu programiranja, koje je dostiglo vrhunac u C#-u 3.0, kada je uveden LINQ (Language Integrated Query) sa lambda izrazima, ekstenzijama i anonimnim klasama. LINQ je skup metoda koji olakšava pravljenje upita. U mnogim situacijima to pojednostavljuje pisanje i čitanje koda.

C# je imao više verzija, koje su prikazane u sledećoj tabeli:

Verzija	**Datum**	**.NET Framework**	**Visual Studio**
C# 1.0	Januar 2002.	.NET Framework 1.0	Visual Studio .NET 2002
C# 1.2	April 2003.	.NET Framework 1.1	Visual Studio .NET 2003
C# 2.0	Novembar 2005.	.NET Framework 2.0	VIsual Studio 2005
C# 3.0	Novembar 2007.	.NET Framework 2.0 .NET Framework 3.0 .NET Framework 3.5	Visual Studio 2008 Visual Studio 2010
C# 4.0	April 2010.	.NET Framework 4.0	Visual Studio 2010

U sledećoj tabeli je prikazan kratak pregled saržaja u novim verzijama:

	C# 2.0	C# 3.0	C# 4.0	C# 5.0
Nove mogućnosti	• Parametarske klase • Parcijalni tipovi • Anonimne metode • Iteratori • Nulabilni tipovi • Privatni dodeljivači • Delegati	• Implicitno dodeljivanje tipa lokalnim promenljivama • Inicijalizator objekata i kolekcija • Auto-implementirani propertiji • Anonimni tipovi • Ekstenzije • Upitni izrazi • Lambda izrazi • Drvo izraza	• Dinamičko vezivanje • Imenovane i opcione argumente • Generičke kovarijanse i kontravarijanse	• Asinhrone metode • Kompajler kao servis

1.3. Mogućnosti

Po dizajnu C# je programski jezik koji direktno reflektuje osnovni CLI. Većina tipova C#-a odgovara tipovima implementiranim u CLI okruženju. Ipak, u specifikaciji C#-a se ne pominje da C# kompajler mora da generiše kôd koji odgovara CLI-u. Teorijski, C# kompajler bi mogao da generiše mašinski kôd kao i tradicionalni kompajleri za C++ ili Fortran.

Neka od važnijih svojstava C#-a su:

- Ne postoje globalne promenljive ili funkcije. Sve metode i članovi moraju da budu deklarisani u nekoj klasi. Statičke metode javnih klasa mogu da budu zamena za globalne promenljive i funkcije.

- Lokalne promenljive, za razliku od C-a i C++-a, ne mogu da budu sakrivene promenljivama u nekom unutrašnjem bloku. Sakrivanje promenljivih je obično zbunjivalo programere u C++ kodu.
- C# podržava tip boolean, koji može imati samo vrednosti true i false. Naredbe if i while zahtevaju izraze, koji implementiraju operator true, kao što je tip boolean. Za razliku od C-a ili C++-a, gde se bool dobija od integer-a ili pokazivača, u C#-u integer ne može da bude true ili false. To sprečava greške koje su se često javljale u C-u i C++-u, tipa if(a = b) (korišćenje dodele = umesto poređenja jednakosti ==).
- U C#-u, pokazivači mogu da se koriste samo u blokovima koji su markirani sa unsafe i programi sa unsafe kodom moraju da imaju specijalne dozvole da bi se pokrenuli. Većini objekata se pristupa pomoću „bezbednih“ referenci, koje ili referenciraju „živi“ objekat u memoriji, ili predefinisanu vrednost null. Nemoguće je dobiti referencu na „mrtvi“ objekat (objekat koji je obrisan iz memorije), ili dobiti referencu na nasumični objekat u memoriji. Nebezbedni pokazivač može da pokaže na instancu vrednosnog tipa, niza, niske, ili bloka u memoriji alociranoj na steku. Kôd koji nije markiran kao nebezbedan (unsafe), i dalje može da manipuliše pokazivačima kroz System.IntPtr tip, ali ne može da ih dereferencira.
- Upravljana memorija ne može biti eksplicitno oslobođena; umesto toga, automatski je kupi skupljač smeća (garbage collector). Skupljač smeća uklanja problem curenja memorije tako što više nije na programeru odgovornost da oslobodi memoriju.
- Kao dodatak na try...catch blok, C# ima try...finally, koji garantuje izvršavanje nekog koda u finally bloku.
- Višestruko nasleđivanje nije podržano, ali klase mogu da implementiraju neograničano interfejsa. Ovo je odluka vodećeg arhitekte da bi pojednostavio arhitektonske zahteve CLI-a.

- C#, kao i C++ (za razliku od Jave), podržava preopterećivanje (overloading) operatora.
- Jedine implicitne konverzije su one koje se smatraju bezbednim, kao što je proširivanje integer-a. Ne postoji implicitna konverzija između integer-a i boolean-a, niti između članova enumeracije i integer-a (osim za 0, koja implicitno može da se konvertuje u bilo koji tip enumeracije). Bilo koja korisnički definisana konverzija mora da se označi kao eksplicitna ili implicitna, za razliku od C++ konstruktora za kopiranje i operatora konverzije, kojima je podrazumevano da su implicitni. Od verzije 4.0, C# podržava dinamičke tipove podataka, što zahteva proveru tipova samo za vreme izvršavanja programa.
- Članovi enumeracije su stavljeni u sopstveni prostor.
- C# nudi svojstva koja enkapsuliraju dodele i vraćanje podataka (getter i setter) u jedan atribut klase.
- Trenutna verzija C#-a je 4.0 i ima 77 rezervisanih reči.

1.4. Kategorije tipova podataka

C# ima ujedinjen tip podataka. To znači da svi tipovi podataka, uključujući i primitivne, kao što je integer, predstavljaju podklase klase System.Object. Na primer, svaki tip nasleđuje metodu ToString().

Postoje dve kategorije tipova podataka:

1. Vrednosni tipovi
2. Referentni tipovi

Vrednosni tipovi su čiste agregacije podataka. Instance vrednosnih tipova nemaju referentni identitet, niti poređenje po referenci. Vrednosni tipovi su nasleđeni iz klase System.ValueType, uvek imaju podrazumevanu vrednost, i uvek mogu da se kreiraju i kopiraju. Ograničenja nad ovim tipovima su to što ne mogu jedni druge da naslede (ali mogu da implementiraju interfejse) i ne mogu da imaju eksplicitni podrazumevani

konstruktor (konstruktor bez parametara). Primeri vrednosnih tipova su int (označeni 32-bitni ceo broj), float (32-bitni IEEE broj sa pokretnim zarezom), char System.DateTime (vreme sa preciznosti u nanosekundama), kao i enum (enumeratorski tip) i struct (korisnički definisane strukture).

Kod referentnih tipova svaka instanca je različita od bilo koje druge, čak iako su podaci obe instance isti. To je reflektovano u podrazumevanim poređenjima jednakosti referentnih tipova, koji gledaju da li je referenca ista, umesto da li je struktura ista, osim ako se ne definiše suprotno (kao na primer za System.String). U opštem slučaju, nije uvek moguće kreirati instancu, ili porediti vrednosti dve postojeće instance, ali je moguće napraviti javne konstruktore ili implementirati odgovarajuće interfejse (ICloneable ili IComparable). Primeri referentnih tipova su object (bazna klasa svih drugih klasa u C#-u), System.String (niz Unicode karaktera), i System.Array (bazna klasa svih nizova).

Obe kategorije mogu da se prošire korisnički definisanim tipovima.

Parametarske klase su dodate u verziji C# 2.0. One koriste tipove parametara koji omogućavaju da se klase i metode dizajniraju tako da ne određuju tip sve dok se klasa ili metoda ne instanciraju. Glavna prednost je to što mogu da se koriste generički tipovi parametara za kreiranje klasa i metoda, koje se mogu koristiti bez trošenja vremena na konvertovanje tipova. Na primer:

```
public class GenericList<T>
{
    void Add(T input) { }
}

class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Deklarisanje liste tipa int.
        GenericList<int> list1 = new GenericList<int>();
```

```
        // Deklarisaje liste tipa string.
        GenericList<string> list2 = new GenericList<string>();

        // Deklarisanje liste klase tipa ExampleClass.
        GenericList<ExampleClass> list3 = new
GenericList<ExampleClass>();
    }
}
```

Preprocesorske direktive (iako C# nema pravi preprocesor) su bazirane na C preprocesoru i omogućavaju programerima da definišu simbole, ali ne i makroe. Uslovi kao #if, #endif, i #else su takođe obezbeđeni. Direktive (kao što je #region) daju editoru naznaku koji deo koda je za sklapanje. Na primer:

```
public class Foo
{
    #region Procedures
    public void IntBar(int firstParam) {}
    public void StrBar(string firstParam) {}
    public void BoolBar(bool firstParam) {}
    #endregion

    #region Constructors
    public Foo() {}
    public Foo(int firstParam) {}
    #endregion
}
```

Znak // se koristi na početku linije da označi da je cela linija komentar. To je preuzeto iz C++-a.

```
public class Foo
{
    // komentar
    public static void Bar(int firstParam) {}   // drugi komentar
}
```

Takođe, postoji i način da se više redova obeleži kao komentar korišćenjem /* na početku i */ na kraju komentara. To je preuzeto iz C-a.

```
public class Foo
{
    /* Komentar
       u vise linija  */
    public static void Bar(int firstParam) {}
}
```

Dokumentacija u C#-u je slična Javinom Javadoc-u, ali bazirana je na XML-u. Trenutno su podržane dve metode pravljenja dokumentacije od strane C# kompajlera.

Komentari dokumentacije u jednoj liniji počinju sa ///

```
public class Foo
{
    /// <summary>Rezime metode.</summary>
    /// <param name="firstParam">Opis.</param>
    /// <remarks>Napomene o metodi</remarks>
    public static void Bar(int firstParam) {}
}
```

Komentari dokumentacije, iako su definisani u verziji 1.0, nisu podržani do verzije .NET 1.1. Ovi komentari počinju sa /** a završavaju se sa */

```
public class Foo
{
    /** <summary>Rezime.</summary>
     *  <param name="firstParam">Opis.</param>
     *  <remarks>Napomene</remarks> */
    public static void Bar(int firstParam) {}
}
```

Veoma je bitno voditi računa o razmacima u XML dokunetaciji kada se koristi /** tehnika.

Ovaj kod:

```
/**
 * <summary>
 * Rezime.</summary>*/
```

nije isti kao

```
/**
 * <summary>
   Rezime.</summary>*/
```

Sintaksa pravljenja dokumentacije je određena aneksom ECMA C# standarda. Isti standard definiše pravila za procesiranje ovakvih komentara i njihovu transformaciju u XML dokument sa preciznim pravilima mapiranja CLI identifikatora u njihove elemente u dokumentaciji. Ovo omogučava bilo kom razvojnom alatu da nađe dokumentaciju za bilo koji simbol u kodu na određeni, dobro definisani način.

U specifikaciji C#-a se nalazi minimalni skup tipova i klasnih biblioteka za koje se očekuje da kompajler mora da ima na raspolaganju. U praksi, C# se najčešće koristi sa delom implementacije CLI, koja je standardizovana kao ECMA-335 Common Language Infrastructure (CLI).

1.5. Primer programa

Primer „Zdravo svete“:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Zdravo svete!");
    }
}
```

Rezultat ovog programa je ispisivanje teksta „Zdravo svete!“ na konzoli. Svaka linija ima neki cilj.

```
using System;
```

Kôd iznad 'kazuje' kompajleru da koristi System kao kandidata za prefiks tipova koji se koriste u kodu. Kompajler pokušava da nađe System.Console korišćenjem System prefiksa iz using dela. Korišćenje using naredbe omogućava programeru da naznači sve kandidate prefiksa za vreme kompilacije, umesto da uvek koristi puna imena.

```
class Program
```

Iznad je definicija klase. Sve što je između zagrada opisuje klasu Program.

```
static void Main()
```

Iznad je deklaracija člana klasne metode gde počinje izršavanje koda.

```
Console.WriteLine("Zdravo svete!");
```

Poziva metodu koja ispisuje „Zdravo svete!“ na konzoli.

2. Autocad

Autocad je jedan od najpoznatijih računarskih programa za računarsko projektovanje. Autor programa je kompanija Autodesk, koja nudi preko 75 specijalizovanih softverskih alata i pomagala za različita ekspertska područja (mašinogradnja, elektrotehnika, elektronika, građevinarstvo, arhitektura, kartografija, geodezija, vatrozaštita itd).

2.1. Kratak opis rada sa Autocad-om

Autocad je sofisticirani projektantski alat široke, univerzalne namene, koji podržava dvodimenzionalno projektovanje. On je praktično zamenio klasično projektovanje na papiru, odnosno tablu za crtanje, šestar i lenjir. Omogućuje trodimenzionalno modelovanje složenih objekata koji se u „modelnom prostoru“ (engl. model space) mogu proizvoljno zumirati, naginjati, okretati, prikazivati u projekcijama, pogledima i presecima iz svih smerova, sa perspektivnim efektom ili bez njega, proizvoljno osvetljavati i renderovati. Trodimenzionalni prikaz imitira fotografiju virtuelnog objekta, koji postoji samo u memoriji računara.

Za razliku od alternativnih softverskih proizvoda za 2D i 3D modelovanje, Autocad je specifičan po sofisticiranom sistemu merila, visokoj preciznosti koja može ići ispod nanometra i sistemu kotiranja razmere. Podržava automatsko računanje koje zadovoljava i najstrože tehničke standarde. Uz izvesne uslove, kotiranje je asocijativno, tj. automatski sledi izmene geometrije obrađivanog modela.

Radni prostor Autocada čini prostor za trodimenzionalno modelovanje i proizvoljan broj radnih prostora koji se mogu koristiti u režimima „papir“ i „model“. U režimu „model“, na radnim listovima se mogu otvarati projekcije i pogledi (engl. viewport) na trodimenzionalni model napravljen u prostoru za modeliranje. U režimu „papir“, radni prostori nemaju nikakve korelacije sa trodimenzionalnim modelom i u tom se režimu pogledi (ako su uopšte kreirani) ne mogu aktivirati. Modelni i papirni prostor se u načelu koriste odvojeno, odnosno ne organizuju se u istom radnom prostoru.

2.2. Trodimenzionalni režim

Prostor za modelovanje i radni prostori u režimu „model“ čine „modelni prostor“ u kojem se definiše (konstruiše, menja, kotira i opisuje) jedan trodimenzionalni model, koji međutim može biti vrlo složen i sadržati veliki broj sastavnih elemenata. U radnim prostorima, u režimu „model“, mogu se otvarati projekcije na bilo koju ravan, ili preseci i pogledi iz bilo kog smera na trodimenzionalni model. Prostor za modelovanje i projekcije u radnim prostorima automatizovano sarađuju, tako da se svaka promena u bilo kom radnom prostoru, ili na modelu u prostoru za modeliranje, reflektuje na model, tj. automatski se ažurira na svim ostalim radnim listovima. Prostor za modelovanje je neograničen, a model se može neograničeno zumirati.

2.3. Dvodimenzionalni režim

U režimu „papir“, radni prostori predstavljaju nezavisne papire među kojima nema nikakve povezanosti, pa se koriste na način uobičajen u klasičnom „papirskom“ projektovanju za crtanje dvodimenzionalnih projekcija i preseka. Takve su projekcije „pljosnate“, tj. ne mogu se okretati u prostoru, i nemaju promenljiv ugao gledanja. Na svakom radnom listu se može prikazati drugi objekat, ili drugi element složenog objekta, čiji je sastavni crtež takođe sadržan u jednom ili više radnih prostora. Skup radnih prostora u režimu „papir“ predstavlja „papirni prostor“ Autocada (engl. paper space). Pogledi i projekcije trodimenzionalnog modela ne mogu se aktivirati u papirnom prostoru, a objekti ucrtani u papirni prostor uopšte se ne prikazuju u prostoru za trodimenzionalno modelovanje.

2.4. Rad u slojevima

U oba režima, u svim radnim prostorima i pogledima ili projekcijama, crtanje se izvodi na proizvoljnom broju prozirnih slojeva (engl. layers). Pojedini objekti ili grupe crtežnih elemenata (kote, šrafure, pomoćne konstruktivne linije isl.) mogu se iscrtati u zasebnim slojevima. Svaki sloj se može zasebno formatirati (debljina, vrsta i boja linija i dr.) i po potrebi sakriti, tj. na njemu sadržani objekti mogu se učiniti nevidljivim u jednom ili više radnih prostora.

2.5. Blokovi

Blokovi su objedinjene grupe objekata, ili crtežnih elemenata, koji čine zasebnu celinu i ponašaju se kao jedan element ili objekat. Blokovi, koji su napravljeni u jednom radnom prostoru ili projektu, mogu se koristiti u drugom radnom prostoru ili projektu, mogu se kopirati, brisati itd. kao jedan element. Mogu se ponovo rastaviti na elemente od kojih su sačinjeni.

Često korićene elemente ili objekte (vijci, instalacijski elementi, mašinski sklopovi, arhitektonski detalji, nameštaj itd.) zgodno je spremiti kao imenovane blokove, pa se po potrebi mogu pozvati i uklopiti u bilo koji radni prostor ili projekat. Kolekcije blokova štede konstruisanje istih konstruktivnih detalja i značajno skraćuju vreme projektovanja.

Skromna kolekcija takvih elemenata sadržana je i u samom Autocadu.

Od verzije Autocad 2006 novost su „dinamički blokovi“. Naime, bloku se mogu pridružiti dinamička svojstva i na temelju tih svojstava (na primer, umesto bloka vrata u arhitekturi koji ima više dimenzija, smerova otvaranja, količine detalja) zameniti to sve jednim blokom koji ima sva ta svojstva i menja ih jednim klikom, umesto više komandi koje su se do sada koristile da se dobije željeni izgled bloka. Jedan dinamički blok vrata može sadržati preko 500 kombinacija vrata.

3. Autocad datoteke

3.1 Rad sa DWG datotekama

Autocad datoteke se čuvaju u DWG (drawing) formatu. Datoteke .bak (rezervna kopija slike), .dws (standard slike), .dwt (šablon slike) i .sv$ (privremeno automatsko čuvanje slike) su takođe deo DWG formata. (Wikipedia)

DWG datoteke se ponašaju kao baza podataka, tako da sve što važi za rad sa bazama podataka, važi i za rad sa datotekama (na primer, svaka operacija mora da bude deo neke transakcije).

Sadržaj datoteka je, najčešće, veoma složen, ali je moguće koristiti postojeće biblioteke

3.2. Osnovni pojmovi

Bilo koji objekat, koji se može nacrtati u Autocad-u, je entitet (Entity). Dakle svi objekti imaju skup zajedničkih osobina, koje posle nadograđuju i stvaraju nove tipove.

Hendl (eng. Handle) je 64-bitni ceo broj i predstavlja jedinstveni identifikator svakog entiteta. Svaki entitet ima tačno jedan hendl.

3.3. Tipovi objekata

Neki tipovi objekata imaju fiksne vrednosti, a neki vrednosti koje zavise od crteža. Sledeći tipovi objekata imaju fiksne vrednosti:

Tip objekta	**Vrednost**	**Tip objekta**	**Vrednost**
UNUSED	0	RAY	0x28
TEXT	1	XLINE	0x29
ATTRIB	2	DICTIONARY	0x2A
ATTDEF	3		0x2B
BLOCK	4	MTEXT	0x2C
ENDBLK	5	LEADER	0x2D
SEQEND	6	TOLERANCE	0x2E

INSERT	7	MLINE	0x2F
MINSERT	8	BLOCK CONTROL OBJ	0x30
	9	BLOCK HEADER	0x31
VERTEX (2D)	0x0A	LAYER CONTROL OBJ	0x32
VERTEX (3D)	0x0B	LAYER	0x33
VERTEX (MESH)	0x0C	STYLE CONTROL OBJ	0x34
VERTEX (PFACE)	0x0D	STYLE	0x35
VERTEX (PFACE FACE)	0x0E		0x36
POLYLINE (2D)	0x0F		0x37
POLYLINE (3D)	0x10	LTYPE CONTROL OBJ	0x38
ARC	0x11	LTYPE	0x39
CIRCLE	0x12		0x3A
LINE	0x13		0x3B
DIMENSION (ORDINATE)	0x14	VIEW CONTROL OBJ	0x3C
DIMENSION (LINEAR)	0x15	VIEW	0x3D
DIMENSION (ALIGNED)	0x16	UCS CONTROL OBJ	0x3E
DIMENSION (ANG 3-Pt)	0x17	UCS	0x3F
DIMENSION (ANG 2-Ln)	0x18	VPORT CONTROL OBJ	0x40
DIMENSION (RADIUS)	0x19	VPORT	0x41
DIMENSION (DIAMETER)	0x1A	APPID CONTROL OBJ	0x42
POINT	0x1B	APPID	0x43
3DFACE	0x1C	DIMSTYLE CONTROL OBJ	0x44
POLYLINE (PFACE)	0x1D	DIMSTYLE	0x45
POLYLINE (MESH)	0x1E	VP ENT HDR CTRL OBJ	0x46
SOLID	0x1F	VP ENT HDR	0x47
TRACE	0x20	GROUP	0x48
SHAPE	0x21	MLINESTYLE	0x49
VIEWPORT	0x22	OLE2FRAME	0x4A

ELLIPSE	0x23	(DUMMY)	0x4B
SPLINE	0x24	LONG_TRANSACTION	0x4C
REGION	0x25	LWPOLYLINE	0x4D
3DSOLID	0x26	HATCH	0x4E
BODY	0x27	XRECORD	0x4F
ACDBPLACEHOLDER	0x50		
VBA_PROJECT	0x51		
LAYOUT	0x52		

Prefiks 0x označava da je broj koji sledi prikazan u heksadekadnom sistemu.

Sledeći tipovi podataka imaju promenljive vrednosti:

ACAD_TABLE CELLSTYLEMAP DBCOLOR DICTIONARYVAR

DICTIONARYWDFLT GROUP

HATCH

IDBUFFER

IMAGE

IMAGEDEF

IMAGEDEFREACTOR

LAYER_INDEX

LAYOUT

LWPLINE MATERIAL MLEADERSTYLE

OLE2FRAME

PLACEHOLDER PLOTSETTINGS

RASTERVARIABLES SCALE

SORTENTSTABLE

SPATIAL_FILTER

SPATIAL_INDEX TABLEGEOMETRY TABLESTYLES

VBA_PROJECT VISUALSTYLE

WIPEOUTVARIABLE

XRECORD

Da bismo odredili tip objekta koji se ne može naći u tabeli, nego ima promenljivu vrednost, treba od trenutne vrednosti oduzeti 500. Time se dobija indeks u klasnoj listi, iz koje se određuje tip objekta. Na primer, ako neki objekat ima vrednost 501, to znači da je na drugoj poziciji u klasnoj listi. Polje classdxfname, u klasnoj listi, sadrži informaciju o imenu tipa objekta.

4. Poređenje datoteka

4.1. Tipovi algoritama

Postoje nekoliko tipova algoritama za poređenje Autocad datoteka:

1. Poređenje po hendlu
2. Geometrijsko poređenje
3. Kombinovano poređeje

4.1.1 Poređenje po hendlu

Poređenje po hendlu se koristi kada su datoteke nastale od iste datoteke. Tada je velika verovatnoća da će sve datoteke imati veliki broj entiteta sa istim hendlovima. Poređenje po hendlu je brzo jer se odmah uparuju kandidati. Pri poređenju datoteke nisu ravnopravne. Treba unapred odrediti koja datoteka je original, a koja je datoteka sa izmenama.

Rezultati poređenja mogu biti: „isti“, „dodat“, „izbrisan“ i „izmenjen“.

Rezultat „dodat“ znači da entitet nije postojao u originalnoj datoteci, a da u datoteci izmene postoji.

Rezultat „izbrisan“ znači da u originalnoj postoji, a u datoteci izmene ne postoji.

Rezultat „izmenjen“ znači da postoji u obe datoteke, ali da postoje određene razlike.

Rezultat „isti“ znači da postoji u obe datoteke i da ne postoje nikakve (značajne) razlike.

Algoritam poređenja po hendlu izgleda ovako:

Proći kroz sve entitete u datoteci izmene, i za svaki entitet proveriti da li postoji entitet sa istim hendlom u originalnoj datoteci. Ako postoji, upariti entitete, uporediti ih i na osnovu rezultata poređenja dobiti rezultat. Ako hendl ne postoji u originalnoj datoteci, onda je rezultat „dodat“. Za sve entitete koji nisu upareni iz originalne datoteke rezultat je “izbrisan“.

Pri poređenju uparenih entiteta može se desiti da su isti, i onda je to i rezultat, ali ako su različiti, to može da se protumači na više načina. Ako su entiteti istog tipa, onda je rezultat „izmenjen“, ali ako su tipovi različiti, onda je entitet iz originalne datoteke izbrisan, a u datoteci izmene dodat novi entitet kojem se slučajno desilo da dobije isti handle. Iako je verovatnoća da se to desi veoma mala, pošto datoteke imaju veliki broj entiteta, to se ipak može očekivati.

Pošto se vrši po jedan prolaz kroz datoteke, složenost je O(m+n), gde su m i n brojevi entiteta u datotekama (ako se podaci organizuju na pravi način, pristup entitetu po hendlu je direktan).

Mana ovog algoritma je to što ne vraća rezultat „isti“ za dva entiteta ako imaju različit hendl, iako su isti po svim ostalim svojstvima (hendl je nebitan sa gledišta korisnika).

4.1.2. Geometrijsko poređenje

Geometrijsko poređenje se koristi kada se pretpostavlja da datoteke imaju sličnosti, ali nisu nastale od iste datoteke, ili ako su se hendlovi promenili prilikom kopiranja (npr. u drugu datoteku). Ovaj algoritam je sporiji od poređenja po hendlu, ali je precizniji, jer za iste entitete vraća rezultat „isti“, bez obzira na vrednost hendla. Velika razlika je u tome što geometrijsko poređenje vraća rezultate „dodat“ i „izbrisan“ čak iako se radi o entitetu sa istim hendlom ako je promenjena geometrija na bilo koji način (na primer možda samo malo pomeren). Rezultat „izmenjen“ jedino vraća kada je geometrija entiteta ista, a nešto drugo je izmenjeno (sloj, boja, tip linije...).

I u geometrijskom poređenju je bitno da se odredi originalna datoteka i datoteka izmene.

Algoritam geometrijskog poređenja izgleda ovako:

Proći kroz sve entitete u datoteci izmene i uporediti ih sa svim entitetima iz originalne datoteke. Ako su dva entiteta ista, vraća se rezultat „isti“, a ako entitet iz datoteke izmene nije isti ni sa jednim entitetom iz originalne datoteke, onda je rezultat „dodat“. Entiteti iz originalne datoteke, koji su isti sa nekim entitetom iz datoteke izmene,

ne učestvuju više u poređenju. Ako imamo entitet EI2 koji je nastao u datoteci izmene kopiranjem nekog entiteta EI1 i nalazi se na istoj lokaciji, EI2 će biti prepoznat kao „dodat“ jer neće imati upareni entitet u originalnoj datoteci. Kada se prođe kroz sve entitete iz datoteke izmene, oni entiteti iz originalne datoteke, koji nisu upareni ni sa jednim entitetom iz datoteke izmene, daju rezultat „izbrisan“. EO1 je entitet iz originalne datoteke koji ima istu geometriju kao i EI1 i EI2.

Kao što je već rečeno, ako je geometrija dva entiteta ista, a imaju druge izmene, onda je rezultat „izmenjen“ i ti elementi takođe više ne učestvuju u poređenju. Ovde je problematična situacija ako se izvrši neka izmena na EI2 (na primer promena boje). Prolazak kroz datoteku je nasumičan; ako se prvo porede EI1 i EO1, rezultat bi bio „isti“ (za EI1 i EO1) i „dodat“ (za EI2). Ako se prvo poredi EI2 sa EO1, onda bi rezultati bili „izmenjen“ (za EI2 i EO1), i „dodat“ (za EI1).

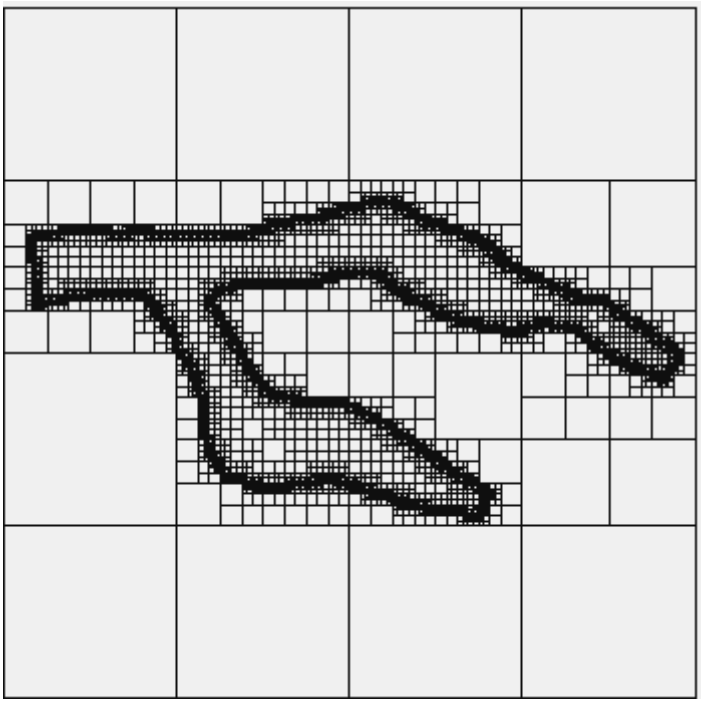
U slučaju da elementi učestvuju u poređenju iako su upareni i da rezultat „isti“ ima prioritet, onda bi rezultati bili jenoznačni. Pri međusobnom poređenju EI1, EI2 i EO1, rezultati bi bio „isti“ (za EI1, EO1) i „dodat“ (za EI2), bez obzira kojim redosledom se prvo porede. Ako se prvo porede EI1 i E01, to je očigledno. Ako se prvo poredi EI2 sa E01, rezultat je „izmenjen“ (za EI2 i EO1), ali kada se izvrši poređenje EI1 sa EO1, zbog prioriteta rezultata „isti“, EI1 i EO1 imaju rezultat „isti“, a EI2 dobija rezultat „dodat“.

Zaključuje se da ni jedna opcija ne daje sa sigurnošću tačno šta se desilo, a zbog jednostavnosti algoritma je bolja opcija da izmenjeni entiteti ne učestvuju u poređenju. Iako „istorijski“ ne daje tačne rezultate (što generalno važi za geometrijski algoritam), logički je svejedno šta je rezultat, jer korisnik saznaje gde postoje promene (da umesto jednog, sada imaju dva elementa, od kojih na jednom ima nekih izmena), i iz konteksta u kojem su nastale promene može da zaključi šta one znače.

U najgorem slučaju dve datoteke mogu biti potpuno različite i onda će se izvršavati poređenje entiteta svaki sa svakim. Složenost ovo algoritma je O(m*n).

Postoji dobra optimizacija ovog algoritma. Posmatraćemo dvodimenzioni crtež (optimizacija važi i za veće dimenzije). Pošto Autocad radi sa koordinatama, može se

koristiti kvadratno drvo (Quadtree) za grupisanje entiteta (za tri dimenzije to bi bilo kockasto drvo Octree).



(Quadtree - slika preuzeta sa
http://www.staff.ncl.ac.uk/qiuhua.liang/Research/grid_generation.html)

Crtež se može upisati u kvadrat, koji se rekurzivno može podeliti na četiri manja kvadrata do dubine n. Dubina n znači da ima ukupno 4^n listova. Svaki kvadrat dubine n (u daljem teksu list), obuhvata određen broj elemenata. Jedan element se može naći u jednom ili više listova. Ima smisla porediti samo elemente u istom listu, jer ako nisu u istom listu, geometrija im nije ista. U najgorem slučaju, jedan element može biti u jednom listu, a svi drugi u drugom listu, dok su ostali listovi prazni. Tada složenost ostaje ista. U najboljem slučaju, kada su elementi ravnomerno raspoređeni, dobija se lepo ubrzanje algoritma.

Na primer: neka se ide do dubine 4. Tada ima 256 listova. Neka je broj elemenata 256000. Ako bi svaki list imao po 1000 elemenata, bilo bi 1000*1000 poređenja po listu, a pošto ih ima 256 to je 256 000 000 poređenja, a to je 256 puta brže od osnovnog algoritma. Naravno, u praksi je skoro nemoguće da se dobije tačno 1000 elemenata po listu, ali ovaj uslov može približno biti ispunjen pa je ubrzanje približno prethodno navedenom. Bez

obzira što je složenost i dalje kvadratna, to je dobro ubrzaje. Umesto algoritma koji bi se izvršavao četiri minuta (sata...), izvršiće se za jednu sekundu (minut...).

Opšti slučaj više teži najboljem, nego najgorem slučaju. Vreme raspoređivanja entiteta u drvetu je zanemarljivo u odnosu na vreme poređenja do određene dubine drveta.

Na primer: drvo dubine 10 bi imalo 1 048 576 listova. Na osnovu prethodno rečenog, može se zaključiti da bi, u najboljem slučaju, to dovelo do ubrzanja od 1048576 puta, ali to nije tako. Algoritam bi se previše usporio zbog samog kreiranja drveta i raspoređivanja elemenata. Jedan element bi se javljao u mnogo listova, tako da bi i to dodatno usporilo algoritam. Dakle, preporučena dubina zavisi od mnogo faktora, od kojih je najvažniji prosečan broj elemenata crteža u oblasti za koju se porede datoteke.

4.1.3. Kombinovano poređenje

Kombinovano poređenje je poboljšanje algoritma poređenje po hendlu. Poređenje po hendlu je više teorijske prirode, tako da u praksi skoro uvek treba koristiti kombinovano poređenje, umesto čistog poređenja po hendlu.

Prvi deo algoritma za kombinovano poređenje je isti kao poređenje po hendlu. Treba proći kroz sve entitete u datoteci izmene i za svaki entitet proveriti da li postoji entitet sa istim hendlom u originalnoj datoteci. Ako postoji, izvršiti operacije iste kao u poređenju po hendlu.

Drugi deo algoritma se svodi na geometrijsko poređenje. Sve elemente koji nisu upareni po hendlu, ili koji su upareni ali su različitih tipova, rasporediti u kvadratno drvo i izvršiti operacije geometrijskog algoritma.

Ovaj algoritam je dobar zato što je poređenje po hendlu brzo, tako da je u opštem slučaju dobro koristiti ovaj algoritam. Velika je verovatnoća da će se porediti datoteke istog porekla, a ako to nije slučaj, dobiće se tačni rezultati zbog geometrijskog poređenja, koje sledi nakon poređenja po hendlu. Takođe, ovaj algoritam ima preciznost geometrijskog algoritma jer pronalazi iste entitete iako nisu upareni po hendlu. Pri tom, ovaj algoritam dobro pronalazi i izmenjene entitete koji su upareni po hendlu.

U najboljem slučaju ovaj algoritam ima složenost algoritma poređenja po hendlu O(m+n), a u najgorem O(m+n+m*n) ⇔ O(m*n).

Ipak, postoje situacije kada je čist geometrijski algoritam bolji od ovog, tako da ne može biti apsolutna zamena za geometrijski algoritam. Na primer: ako se za dve datoteke zna da nisu istog porekla i da imaju veliki broj elemenata istog tipa, može se desiti da elementi istog tipa imaju elemente različitog porekla, ali isti hendl, tada se bolji rezultati dobijaju primenom čisto geometrijskog algoritma.

4.2. Poređenje entiteta

Poređenje entita se sastoji iz dva dela. Prvi deo je poređenje opštih svojstava entiteta koja su zajednička za sve entitete. Od važnijih to su: sloj, boja, tip linije, širina linije. Bilo koji korak poređenja može da bude opcioni (na primer entiteti različitih boja mogu da se smatraju istim).

Entiteti se nalaze u istom sloju ako slojevi u kojima se nalaze imaju isto ime. Iako i slojevi imaju hendl, ime ih jedinstveno opisuje, a sa gledišta korisnika je ime bitnije. Na primer: može da se desi da slojevi „stolice“ u originalnoj datoteci, i „zidovi“ u datoteci izmene imaju isti hendl, ali da u datoteci izmene postoji sloj „stolice“ koji ima novi hendl. U obe datoteke postoji po jedan sloj „stolice” i reč je o istim slojevima. Ako je neko stavio projekciju stolica u sloj „zidovi“, najverovatnije je načinio grešku koja će biti vidljiva posle poređenja.

Boja entiteta je opisana pomoću dva svojstva. Prvo je metod dodeljivanja boje (preuzmi boju od bloka, od sloja, koristi korisnički izabranu boju), a drugo je izabrana boja. Prvo se porede metodi dodeljivanja boje, ako su različiti onda se smatra da boja nije ista, bez obriza da li je vrednost boje ista. Slično, ako su metodi isti, smatra se da je boja ista, bez obzira da li su boje različite. Na primer: neka se isti metod koristi kod entiteta u izmenjenoj datoteci i originalnoj datoteci, i neka se nalaze u istom sloju, ali ti slojevi imaju lokalno promenjene boje – rezultat je da su boje entiteta iste. Razlika u boji bi se uočila pri poređenju slojeva.

Poređenje tipa linije i širine linije je trivijalno. Ako su vrednosti iste, onda su i tip i širina linije isti.

Drugi deo deo poređenja je poređenje geometrije. Geometrija se razlikuje kod svih tipova entiteta, tako da se poređenje za svaki tip mora implementirati na način koji je specifičan za taj tip.

4.3. Implementacija u C#-u

Za implementaciju kompletnog softvera za poređenje Autocad datoteka je neophodno implementirati više celina:

1. Grafički interfejs za učitavanje datoteka za poređenje i biranje opcija za poređenje.

2. Logika i grafički interfejs za prikazivanje rezultata poređenja i učitanih DWG datoteka.

3. Implementacija samih algoritama

4.3.1. Grafički interfejs

Grafički interfejs u C# pravi se jednostavno, korišćenjem ugrađenog dizajnera u Visual Studio-u. Trenutno postoje dva koncepta u kreiranju grafičkog interfejsa:

1. Windows Forms je stari, klasičan način pravljenja interfejsa

2. WPF (Windows Presentation Foundation) je moderniji, fleksibilniji način pravljenja interfejsa

I jedan i drugi koncept se mogu primeniti za pravljenje ovog softvera. Poređenje Windows Forms i WPF je posebna, široka tema i njome se ovde nećemo baviti.

Za čuvanje opcija najbolje je korišćenje XML-a. Sve opcije je poželjno čuvati u lokacijama na kojima je garantovano da svaki korisnik ima pravo da piše (na primer: ApplicationData).

4.3.2. Prikazivanje DWG datoteka

DWG datoteke su veoma složene i za njihovo prikazivanje (pomoću aplikacije napisane u C#-u) je najbolje koristiti postojeće biblioteke. Prikazivanje gotove datoteke pomoću boljih biblioteka je jednostavno. Dovoljno je predvideti kontrolu za crtanje (najpogodniji je Panel), dati lokaciju datoteke i napraviti osnovna grafička podešavanja koja ta biblioteka zahteva u specifikaciji.

Pošto je nekada potrebno prikazati dva ili više crteža odjednom, preporučljivo je napraviti klasu (u daljem tekstu DWGViewer) koja nasleđuje klasu UserControl i na osnovu lokacije datoteke, ili same datoteke, izvršiti neophodna podešavanja i automatski prikazati sliku. Takođe mogu se implementirati i razne akcije kao što je: zumiranje slike, pomeranje kamere, selektovanje elemenata... Sve ove operacije su obično ugrađene u biblioteku i neophodno je samo osnovno znanje matematike za njihovu implementaciju.

Rezultat poređenja datoteka se ne može predstaviti tekstualno, jer osim hendla koji korisniku ništa ne znači, ne postoji ništa što intuitivno i jedinstveno opisuje entitet. Najbolji način je da se grafički prikaže rezultat.

Dve datoteke su ulazni podaci, porede se entiteti datoteka, a rezultat ima povratne vrednosti: „isti“, „izmenjen“, „izbrisan“ i „dodat“, i to za svaki entitet. Pošto je koncept takav da postoje datoteka izmene i originalna datoteka, pri čemu se posmatra kakvo je stanje datoteke izmene u odnosu na originalnu datoteku, najprirodnije je da prikaz u osnovi bude datoteka izmene. Svaki rezultat bi imao svoju predefinisanu boju. Obično su to bela za rezultat „isti“, plava za „izmenjen“, crvena za „izbrisan“ i zelena za „dodat“, ali najbolje je korisniku dozvoliti da kroz opcije izabere odgovarajuće boje.

Najveći problem u ovom konceptu je kako prikazati podatke na traženi način. Pošto DWGViewer očekuje datoteku, najbolje je napraviti novu datoteku koja će u sebi imati kopiju originalnih entiteta u odgovarajućoj boji. Nova datoteka se pravi tako što se isti, dodati i izmenjeni entiteti kopiraju iz datoteke izmene, a izbrisani entiteti se kopiraju iz originalne datoteke, jer se informacije o njima ne nalaze u datoteci izmene. To znači da će izmenjeni entiteti imati lokaciju i oblik entiteta iz datoteke izmene, a ne kao originalni entitet, što je skoro uvek dovoljna informacija.

Prikaz može da se napravi tako što se uporedo prikažu datoteka izmene, originalna datoteka i rezultat poređenja. Selektovanjem entiteta iz bilo kog pogleda, selektuju se odgovarajući entiteti u ostalim pogledima, što nedvosmisleno upućuije na to izmenom kog entiteta je nastao novi entitet. Pri kopiranju entiteta iz jedne datoteke u drugu, entitet dobija novi hendl, tako da je neophodno mapirati hendlove originalnog entiteta i kopije, da bi postojala veza izmedju njih. Za mapiranje ja najbolje koristiti Dictionary<Handle, Handle> i to napraviti dve kolekcije koje će mapirati hendlove u oba smera: original – kopija i kopija – original. Na taj način se lako, pomoću hendla, mogu dobiti svi podaci entiteta iz bilo koje datoteke.

U praksi se pokazalo da je korisno prikazati datoteku izmene i originalnu datoteku bez rezultata, i to na dva načina:

1. Jednu pored druge (side by side)
2. Preklapanjem (overlay)

Kada su datoteke jedna pored druge, korisnik ima jasnu predstavu kako datoteke izgledaju. Kada uoči promenu, jasno mu je na šta se ta promena odnosi i u kom je delu crteža, ali same promene se teže uočavaju. Sve komande, koje korisnik izvrši, treba da se pozivaju u oba crteža na isti način (zumiranje, transliranje...). Ovde se javlja potreba za poravnjanjem crteža.

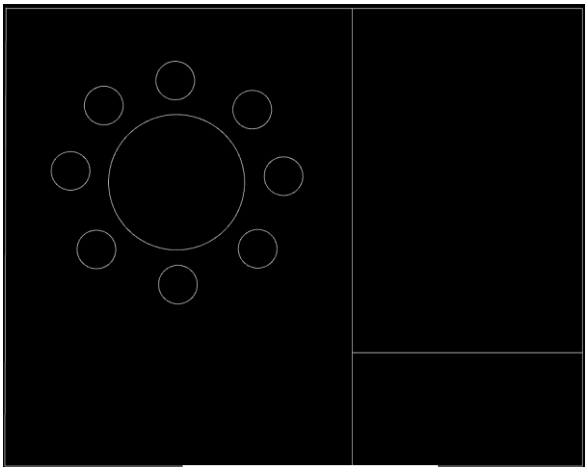
Dešava se da su crteži potpuno (ili delimično) isti, ali da su svi entiteti datoteke izmene translirani za određeni vektor. U opštem slučaju se ne može automatski odrediti koji je to vektor i zato korisniku treba omogućiti da izabere po dve tačke na oba crteža koje bi trebalo da imaju iste koordinate. Pošto je korisniku teško da pogodi precizno tačku, bolje je ponuditi da izabere po dve linije koje nisu paralelne. Kako se Autocad crteži obično sastoje od velikog broj linija, velika je verovatnoća da će u svakom ozbiljnijem projektu moći da se nađu dva para takvih pravih. Poravnjanje ima primenu i u automatskom poređenju.

Kada se slike preklapaju (original ispod, a izmena iznad), menjanjem vidljivosti delova izmene se lako mogu uočiti, ali je manja preglednost obe datoteke. I ovde sve komande treba da se izvršavaju na isti način.

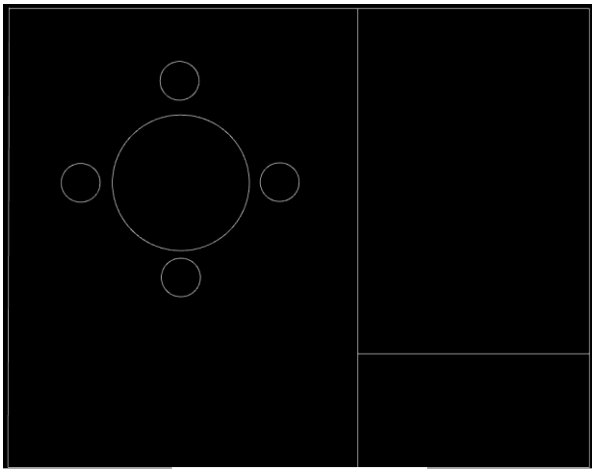
4.3.3. Implementacija algoritama

Pri implementaciji algoritama, pored poštovanja samog algoritma, potrebno je voditi računa i o podacima neophodnim za prikaz rezultata i specifičnostima biblioteka za rad sa Autocad datotekama.

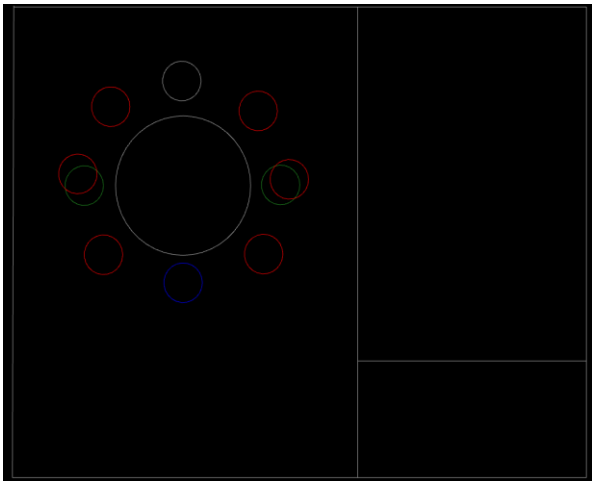
Pre izvršavanja algoritma za poređenje, neophodo je pripremiti podatke. Pošto se DWG ponaša kao baza podataka, neohodno je napraviti lokalnu klasu (MyEntity) koja će imati podatke koji se često koriste i u prikazu i samim algoritmima.



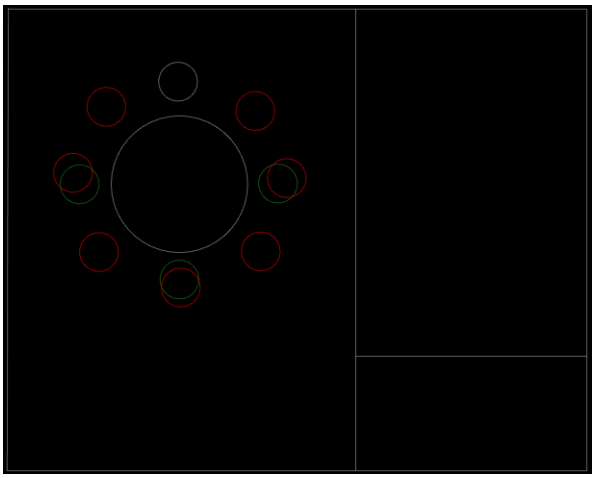
Originalni crtež



Izmenjeni crtež



Rezultat poređenja kombinovanim algoritmom



Rezultat poređenja geometrijskim algoritmom

Za svaki entitet postoji informacija o pravougaoniku u koji je upisan. Ako dva entiteta nemaju iste pravougaonike, onda ni entiteti nisu isti, a ako imaju iste pravougaonike, onda ima smisla porediti geometrijska svojstva. U velikom broju slučajeva

pravougaonici neće biti isti, tako da to može znatno da doprinese ubrzanju algoritma. Takođe, to je jedan od podataka koji treba čuvati u klasi MyEntity. Pored toga, treba čuvati hendl, širinu linije, tip linije, boju, tip entiteta, id objekta u memoriji. Poželjno je napraviti i metodu (ne u klasi MyEntity, već u posebnoj menadžer klasi koja je zadužena za transakcije) koja će, po potrebi, otvarati bazu i vratiti originalni Entitet da bi se dobili podaci koji nisu lokalno sačuvani (na primer geometrijska svojstva). Ako se vodi računa o brzini izvršavanja programa, onda tu metodu treba što manje koristiti, a što više potrebnih podataka staviti u MyEntity.

Primer klase MyEntity sa osnovnim podacima (popunjavanje u konstruktoru zavisi od biblioteke za rad sa DWG datotekama. U primeru je pseudokod):

```
public class MyEntity
  {
    public LineType LineType { get; set; }
    public LineWeight LineWeight { get; set; }
    public Color Color { get; set; }
    public ColorMethod ColorMethod { get; set; }
    public Point3D MinPoint { get; set; }
    public Point3D MaxPoint { get; set; }
    public string Layer { get; set; }

    public MyEntity(Entity originalEntity)
    {
      this.LineType = originalEntity.LineType;
      this.LineWeight = originalEntity.LineWeight;
      this.ColorMethod = originalEntity.Color.ColorMethod;
      this.Color = originalEntity.Color.RGBColor;
      this.LineType = originalEntity.Extents.MinPoint;
      this.LineType = originalEntity.Extents.MaxPoint;
      this.Layer = originalEntity.Layer.Name;
```

```
    }
}
```

Sve instance MyEntity treba organizovati u kolekciju MyEntityCollection koja bi trebalo da sadrži dve interne kolekcije:

- Dictionary<Handle, MyEntity> entitiesByHandle
- Dictionary<Type, Dictionary<Handle, MyEntity>> entitiesByType

Kolekcija entitiesByHandle služi da se pomoću hendla direktno pristupi entitetu. To je veoma bitno pri poređenju po hendlu. Kolekcija entitiesByType grupiše entitete po tipu, što ubrzava algoritam geometrijskog poređenja jer ima smisla porediti samo entitete istog tipa, a ova kolekcija ubrzava pristup entitetima istog tipa. Kolekcija nije Dictionary<Type, MyEntity> jer uvek treba imati i direktan pristup po hendlu. Klasa treba da implementira interfejs IEnumerable, gde iteracija treba da bude po kolekciji etitiesByHandle i treba da implementira odgovarajuće indeksere.

Deo implementacije klase DWGEntityCollection (Prikazana je implementacija interfejsa IEnumerable i indeksera):

```
public class DWGEntityCollection : IEnumerable<KeyValuePair<Handle, MyEntity>>
{
    private Dictionary< Type, Dictionary<Handle, MyEntity>> entitiesByType;
    private Dictionary<Handle, MyEntity> entitiesByHandle;

    public ICollection<Handle> Keys
    {
        get { return entitiesByHandle.Keys; }
    }
    public ICollection<MyEntity> Values
    {
        get { return entitiesByHandle.Values; }
```

```
}

public MyEntity this[Handle handle]
{
   get
   {
      if (entitiesByHandle.ContainsKey(handle))
         return entitiesByHandle [handle];
      else
         return null;
   }
   set
   {
      entitiesByHandle [handle] = value;
   }
}

public Dictionary<Handle, MyEntity> this[string groupKey]
{
   get
   {
      Dictionary<Handle, MyEntity> tempDictionary;
      if (entitiesByType.TryGetValue(groupKey, out tempDictionary))
         return tempDictionary;
      return null;
   }
   set
   {
      entitiesByType [groupKey] = value;
```

```
        }
    }
}
```

Klasu koja sadrži DWG bazu, kolekciju entiteta, i operacije nad njima, možemo nazvati MyDWG. Operacije mogu biti: učitavanje baze na datoj lokaciji, poziv popunjavaja lokalnih entiteta i kolekcija, ...

Posle učitavanja originalne datoteke i datoteke izmene, kao i popuvanja odgovarajućih instanci MyDWG klase, sve je pripremljeno za poređenje (pretpostavka je da su opcije poređenja učitane pri pokretanju aplikacije).

Implementacija algoritama treba da bude u posebnoj klasi. Metoda koja vrši poređenje, treba da prihvati dve instance MyDWG, a kao rezultat generiše klasu CompareResult.

DWGCompareResult može da bude klasa ili struktura koja sadrži informacije koji entitet pripada kojoj grupi izmena. Klasa bi izgledala ovako:

```
class DWGCompareResult
{
    public List<MyEntity> UndefinedEntities { get; set; }
    public List<MyEntity> UnchangedEntities { get; set; }
    public List<MyEntity> AddedEntities { get; set; }
    public List<MyEntity> RemovedEntities { get; set; }
    public Dictionary<MyEntity, CompareResult > ChangedEntities {get; set;}
}

class CompareResult
{
    bool GeometryChanged {get; set;}
    bool LayerChanged {get; set;}
    bool ColorChanged {get; set;}
```

```
    bool LineTypeChanged {get; set;}
    bool LineWeightChanged {get; set;}
  }
```

CompareResult je klasa koja sadrži informacije kakve izmene su nastale na entitetima. Kolekcija ChangedEntities sadrži informacije o izmenjenim entitetima i njihovim izmenama. Ostali tipovi izmena ne zahtevaju nikakve dodatne informacije.

DWGCompareResult se popunjava na način koji je već opisan u delu o algoritmima. Poređenje entiteta će ovde biti detaljnije objašnjeno.

Za poređenje entiteta je najbolje koristiti patern Factory. Cilj Factory paterna je da implementira interfejs za kreiranje objekta, ali da dozvoli potklasama da izaberu koja klasa da se instancira. To je potrebno za poređenje geometrijskih svojstava. Bazna klasa bi bila EntityComparer. U njoj se poredi sve osim geometrijskih svojstava. Klasa bi trebalo da ima poziv i za poređenje geometrijskih svojstava, koje će se izvršiti u posebnim klasama, napravljenim za svaki tip (na primer PolylineComparer).

Sledeći primer je implemetacija poređenja entiteta na primeru PolylineComparer. Svi drugi tipovi entiteta se porede na isti način, samo što se implementacija PolylineComparer zamenjuje klasom poređenjenja za tip koji se poredi.

```
class PolylineComparer : EntityComparerBase<Polyline>
  {
    public bool Compare(Polyline x, Polyline y)
    {
      if (x.NumberOfVertices != y.NumberOfVertices)
        return false;
      for (int i = 0; i < x.NumberOfVertices; i++)
      {
        if (!GeometryCompare.Point3dCompare(x.GetPoint3dAt(i), y.GetPoint3dAt(i)))
          return false;
      }
```

```
            if (!Tolerances.CheckLength(x.Elevation, y.Elevation)) return false;

            return true;
        }
        public override bool CompareEx(Entity x, Entity y)
        {
            return Compare(x as Polyline, y as Polyline);
        }

    }

public interface IEntityComparer
 {
      CompareResult Compare(Entity entityX, Entity entityY);
 }

public abstract class EntityComparerBase: IEntityComparer
   {
      public CompareResult Compare(Entity entityX, Entity entityY)
      {
         #region Geometry compare
         if (!GeometryCompare.Point3dCompare(entityX.Extent.MaxPoint,
                  entityY.Extent.MaxPoint)) compareResult.GeometryChanged = true;

         if (!GeometryCompare.Point3dCompare(entityX.Extent.MinPoint,
                  entityY.Extent.MinPoint)) compareResult.GeometryChanged = true;

         if (compareResult.GeometryChanged == false)
            compareResult.GeometryChanged = !CompareEx(entityX, entityY);
```

```
#endregion



#region Color compare
if (Options.CompareColor)
{
   if (entityX.ColorMethod != entityY.ColorMethod)
      compareResult. ColorChanged = true;

   if (!ColorChanged && ColorMethod == ColorMethod.ByLayer
         &&(entityX.Color.RGBColor != entityY.Color.RGBColor))
      compareResult.ColorChanged = true;
}
#endregion

#region Layer compare
if (Options..CompareLayers)
{
   if (entityX.LayerName != entityY.LayerName)
      compareResult.LayerChanged = true;
}
#endregion

#region Linetype compare
if (Options..CompareLineTypes)
{
   if (entityX.Linetype != entityY.Linetype)
      compareResult.LinetypeChanged = true;
```

```
            }
            #endregion


            #region Linewidth compare
            if (entityX.LineWeight != entityY.LineWeight)
                compareResult.LineweightChanged = true;
            #endregion
        }
        public abstract bool CompareEx(Entity x, Entity y);
}
public class EntityComparerFactory
{
    private static Dictionary<string, IEntityComparer> comparers =
                                                new Dictionary<string, IEntityComparer>()
    {
        //ovde treba dodati listu svih klasa za poređenje
    };
    public static bool ComparerExists(string type)
    {
        return comparers.ContainsKey(type);
    }
    public EntityComparerFactory()
    {
    }
    public IEntityComparer GetComparer(string type)
    {
        IEntityComparer comparer = null;
        if (comparers.TryGetValue(type, out comparer))
```

```
            return comparer;
        return null;
    }
}
```

GeometryCompare je pomoćna klasa u kojoj se porede tačke sa tolerancijom koja je predefinisana u klasi Tolerances. Options je klasa koja sadrži već pomenute korisnički predefinisane opcije.

Posle poređenjenja treba pripremiti podatke za prikaz. Svaki entitet se kopira u novu bazu, pri tome se menja sloj u kojem se nalazi. Prave se specijalni slojevi za svaku izmenu. Tako, u ranije navedenom primeru, svi izmenjeni entiteti iz sloja „stolice“ dobijaju ime „changed_stolice“. Slično se i za ostale tipove dodaju prefiksi „unchanged_“, „added_“, „removed_“. Svakom sloju dodeljuje se odgovarajuća boja, a svakom entitetu metoda dobijanja boje postavlja na vrednost „po sloju“ („ByLayer“). Vidljivost se često menja i ona ne treba da učestvuje u poređenju. Svi entiteti i slojevi treba da postave IsVisible na true, kao i IsFrozen na false. Pošto je aplikacija samo za prikaz datoteka, svojstvo IsLocked (za sloj) treba da se postavi na false.

5. Primena algoritama u spajanju datoteka

5.1. Sistemi za organizaciju dokumenata

Ako se u nekom sistemu javlja veliki broj datoteka i ako se često vrše izmene, neophodno je napraviti sistem koji će olakšati organizaciju datoteka i lakše praćenje izmena. Neki od poznatijih sistema za upravljanje datotekama (bilo kakvim datotekama, a ne isključivo Autocad datotekama) su „Team Foundation Server" i "Subversion".

Sistemi za organizaciju dokumenata su tipa klijent – server. Na serveru se nalaze sve datoteke, a klijent ima odgovarajuće kopije u lokalu.

Datoteke su organizovane po revizijama. Revizija u ovim sistemima ima isto značenje kao i reč verzija. Revizija je skup datoteka koje su izmenjene i postavljene na server. Pri postavljanju skupa datoteka na server, svakoj datoteci iz skupa se dodeljuje redni broj revizije. Na primer, ako neka datoteka ima redni broj revizije 5, to znači da je datoteka učestvovala u kreiranju revizije na serveru sa rednim brojem 5, a ne da je to 5. izmena datoteke. Svaka datoteka učestvuje u jednoj ili više revizija. Na serveru se čuvaju informacije o svim revizijama, dok na klijentu uvek postoji samo jedna revizija datoteke, koja se uvek može osvežiti novom revizijom (ako postoji). Postoji tri tipa revizija datoteka, a to su: lokalna (local), poslednja ili serverska (latest) i bazna (workspace).

Bazna revizija je revizija na serveru od koje nastaju ostale revizije. Sveka bazna revizija ima svoj redni broj. Bazna revizija se ne može menjati na serveru. Klijent preuzima reviziju sa servera. Preuzeta revizija je lokalna revizija. Kada korisnik završi sa izmenama, šalje ih na server. Skup poslatih izmena čini jednu reviziju. Redni broj nove revizije se dobija uvećavanjem prethodne revizije za jedan. Poslednja revizija na serveru se često naziva i serverska revizija.

Na primer: na serveru ima 20 revizija. Dva klijenta (A i B) osvežavaju svoje revizije i na lokalu dobijaju datoteku D revizije sa rednim brojem 20. U tom trenutku se lokalna, serverska i bazna verzija podudaraju. Neka A i B naprave izmene na D. Njihove lokalne datoteke će se razlikovati od ostalih, a serverska i bazna će i dalje biti iste. Ako A postavi svoju verziju D na server, serverska verzija, koja dobija redni broj 21, i lokalna verzija

klijenta A će biti iste, dok će bazna revizija (sa rednim brojem 20), serverska (sa rednim brojem 21) i lokalna klijenta B biti različite. Ako klijent B želi da postavi svoju verziju, mora prvo da uporedi svoju lokalnu verziju sa poslednjom, spoji te dve datoteke, i tek onda postavi na server. U protivnom, izmene koje je A uradio bile bi izgubljene.

5.2. Spajanje datoteka

Postoje dva pristupa u spajanju datoteka.

Prvi pristup je poređenje dve datoteke već opisanim algoritmima. Korisnik treba da, na osnovu rezultata poređenja datoteka, izabere koje izmene treba da prihvati, a koje da odbaci. Na osnovu tih informacija, od dve datoteke nastaje nova.

Drugi pristup se koristi mnogo češće do prvog i jedini je pristup u spajanju datoteka u sistemima za organizaciju datoteka. Posmatraćemo spajanje serverske datoteke S i lokalne datoteke L. U ovom pristupu se ne porede direktno S i L, nego se porede S i bazna verzija B, pri čemu se dobija prvi rezultat poređenja, a zatim se porede L i B i dobija se drugi rezultat. Prvi rezultat daje informaciju o izmenama na serveru, a drugi rezultat daje informacije o izmenama na klijentu. Na osnovu ova dva rezultata se vrši spajanje.

Za određene tipove izmena moguće je da softver automatski prihvati izmenu, a za neke je neophodno da korisnik odluči koja izmena će biti prihvaćena.

Sledeća tabela prikazuje za koju vrstu promena na entitetu, iz koje datoteke treba preuzeti izmenu.

server\klijent	ne postoji	isti	izmenjen	dodat	izbrisan
ne postoji				klijentska	
isti		klijentska	klijentska		klijentska
izmenjen		serverska	konflikt		konflikt
dodat	serverska				
izbrisan		serverska	konflikt		klijentska

Posmatrajmo ćeliju u drugom redu i petoj koloni. Naslov kolone „dodat“ označava da je entitet dodat u klijentskoj datoteci (lokalnoj reviziji datoteke), a naslov reda „ne postoji“ označava da entitet ne postoji na serveru (poslednjoj reviziji datoteke na serveru). Vrednost ćelije „klijentska“ označava da će se automatski izvršiti promena koju je napravio klijent (u rezultujuću datoteku će se dodati entitet).

Dosta mesta u tabeli je prazno zato što ti slučajevi nemaju smisla, tj. ne mogu teorijski da se dese.

Ovi sistemi se koriste u ozbiljnim kompanijama tako da ako je neko (projektant, inženjer, menadžer...) dodao neku izmenu u datoteku, smatra se da je to urađeno sa razlogom i zbog toga se sve nekonfliktne izmene izvršavaju automatski. Jedino konfliktne izmene mora čovek da razreši (i one se u praksi rešavaju u dogovoru sa drugim korisnikom koji je učestvovao u izmeni datoteke).

Ceo ovaj koncept važi opšte, a ne samo za Autocad datoteke. Na primer, programeri često koriste ovaj koncept u spajanju koda. Ono što je specifično u Autocad-u je implementacija.

5.3. Implementacija spajanja datoteka.

Cilj spajanja Autocad datoteka je da od dve datoteke, preuzimanjem entiteta iz obe na određeni način, nastane treća datoteka. U praksi je jednostavnije napraviti kopiju datoteke izmene (lokalne datoteke) i na osnovu rezultata poređenjenja zaključiti koje elemente izbaciti iz kopije datoteke, a koje dodati, tj. prekopirati iz originalne datoteke u kopiju. Pošto je obično procenat istih, dodatih i izmenjenih entiteta (bez konflikta) veliki, a ti entiteti se uzimaju iz klijentske datoteke, ovakav koncept je dobar po performanse i jednostavniji je za implementaciju.

Sledeća tabela prikazuje modifikovanu prethodnu tabelu po novom konceptu:

server\klijent	ne postoji	isti	izmenjen	dodat	izbrisan
ne postoji				ostaviti	
isti		ostaviti	ostaviti		izbrisati
izmenjen		izbrisati i preuzeti	konflikt		konflikt
dodat	preuzeti				
izbrisan		izbrisati	konflikt		izbrisati

Algoritam za spajanje entiteta bi bio sledeći:

- Uporediti serversku i neizmenjenu lokalnu verziju i dobiti rezultat SN
- Uporediti lokalnu i neizmenjenu lokalnu verziju i dobiti rezultat LN
- Napraviti kopiju lokalne verzije KL
- Izvršiti algoritam za spajanje slojeva
- Sve entitete iz, SN koji imaju rezultat „dodat“, prekopirati u KL
- Sve entitete iz, SN koji imaju rezultat „izmenjen“, a u LN imaju rezultat isti izbrisati iz KL i preuzeti iz SN
- Sve entitete iz SN, koji imaju rezultat „izbrisan“, a u LN rezultat „isti“, izbrisati iz KL
- Sve entitete iz LN, koji imaju rezultat „izbrisan“, a nisu izmenjeni u SN, izbrisati
- Proći kroz sve entitete iz datoteke SN, koji imaju rezultat „izmenjen“, a u datoteci LN imaju rezultat „izbrisan“ i staviti ih u kolekciju konflikata KK sa informacijama o vrsti konflikta,
- Proći kroz sve entitete iz datoteke LN, koji imaju rezultat „izmenjen“, a u datoteci SN imaju rezultat „izbrisan“, i staviti ih u KK
- Proći kroz sve entitete iz datoteke LN, koji imaju rezultat „izmenjen“, a u datoteci SN imaju rezultat „izmenjen“ i staviti ih u KK
- Kroz interfejs ponuditi korisniku da izabere akciju za svaki entitet

- Ako je korisnik izabrao akcije za svaki konflikt, izvršiti odgovarajuću akciju za svaki entitet iz liste KK
- Završiti spajanje slojeva

Za rešavanje konflikta treba napraviti interfejs koji je intuitivan korisniku. I ovde otpada čisto teksutalni prikaz konflikata. Jedna od ideja je ponuditi korisniku spisak entiteta sa tipovima izmena i prikazati trenutni izgled KL. Pri izboru nekog elementa, grafički naznačiti o kom elementu se radi i u realnom vremenu prividno izvršavati izabrane akcije, a pri rešavanju svih konflikata stvarno i izvršiti te akcije u datoteci. Nije poželjno postavljati akcije na podrazumevane vrednosti, jer onda korisnik ne bi bio siguran šta je on razrešio, a šta je podrazumevana vrednost.

Algoritam za spajanje slojeva nije trivijalan, i mora se razbiti na dva dela. Prvi deo je sličan kao i spajanje entiteta. Postoje promene za koje je moguće automatsko razrešenje i one za koje korisnik mora sam da odluči koja akcija da se izvede.

Entiteti su vezani za slojeve. Može se desiti da je korisnik A izbrisao sloj S1 i sve entitete iz tog sloja, a korisnik B je dodao entitet u sloj S1. Posle automatskog rešavanja konflikata, sloj S1 bi bio izbrisan i entitet koji je dodao B ne bi imao gde da bude dodat. Zbog toga u prvoj fazi ne treba fizički brisati slojeve, nego to ostaviti za poslednju fazu. Novi entitet će biti dodat u sloj S1 i sloj S1 neće biti izbrisan ni u poslednjoj fazi, jer je ipak neophodan. U poslednjoj fazi se brišu slojevi koji, i posle spajanja entiteta ostaju prazni, a imali su rezultat „izbrisan“.

Pri poređenju slojeva treba voditi računa o tome da ime ima prioritet nad hendlom, kao i da je ime jedinstveno.

Primer koda: Kolekcije koje čuvaju vrednosti o entitetima

```
public List<Handle> EntitiesToAddFromSN { get; private set; }
public List<Handle> LayersToAddFromSN { get; private set; }
public List<Handle> EntitiesToRemoveFromSN { get; private set; }
public List<Handle> LayersToRemoveFromSN { get; private set; }
public List<Handle> EntitiesToChangeFromSN { get; private set; }
```

```
public List<Handle> LayersToChangeFromSN { get; private set; }
```

Primer koda: Dodavanje entiteta u odgovarajuću kolekciju u zavisnosti od izabrane akcije i rezultata izmene

```
switch (mergePair.Action)
        {
          case MergeAction.None:
            break;
          case MergeAction.TakeSNChanges:
            switch (mergePair.SNResult)
            {
              case EntityCompareResult.Added:
                EntitiesToAddFromSN.Add(mergePair.Handle);
                break;
              case EntityCompareResult.Changed:
                EntitiesToChangeFromSN.Add(mergePair.Handle);
                break;
              case EntityCompareResult.Removed:
                EntitiesToRemoveFromSN.Add(mergePair.Handle);
                break;
              case EntityCompareResult.Unchanged:
                break;
              case EntityCompareResult.Unmatched:
                break;
              default: throw new ArgumentOutOfRangeException();
            }
            break;
          case MergeAction.UseMasterChanges:
            break;
          default: throw new ArgumentOutOfRangeException();
        }
```

Zaključak

Brz razvoj hardvera računara i potreba industrije, zahtevaju razvoj softvera koji će ubrzati i olakšati rad inženjera i omogućiti im da se efikasnije bave svojim poslom.

Ovde je opisan softver koji treba da pomogne u arhitekturi, građevini, mašinstvu. Projektanti će korišćenjem ovog softvera imati više vremena za projektovanje, a manje vremena će trošiti na ručno upoređivanje crteža. Neće morati da čuvaju veliku količinu crteža, već će sistemi za upravljanje datotekama odrađivati posao čoveka na jednostavan način.

Timski rad je olakšan, jer se spajanje crteža u velikom broju slučajeva vrši automatski, a konflikti su jasni i jednostavno se razrešavaju.

Sve ovo ne bi bilo moguće bez automatskog poređenja datoteka. Automatsko poređenje može korisiti čovek, ali i računar prilikom realizacije nekih algoritama.

Za razvoj gore pomentog softvera korišćen je programski jezik C#. C# je mlad jezik, ali se dosta koristi za razvijanje aplikacija različitog tipa. Njegova glavna prednost, u odnosu na ostale programske jezike, je to što omogućava efikasno razvijanje kvalitetnih aplikacija, koje se lako mogu unapređivati. C# ima podlogu u velikom broju biblioteka napisanih od Microsoft-a, ali i od raznih korisnika. Osim toga, razvijanje grafičkog interfejsa u C#-u zahteva minimalno vreme. Takođe, C# podržava razne programske paradigme. Zbog svih navedenih osobina, C# je dobar izbor za pisanje ovakvih aplikacija.

Autocad je izabran kao najrasprostranjeniji softver za projektovanje. Postoje razne biblioteke za rad sa DWG datotekama, što olakšava njihovo čitanje i pisanje.

Za automatsko poređenje su se pokazali pogodni i algoritam poređenja po hendlu i geometrijski algoritam. Pri samoj implementaciji algoritama, kao i pri prikazu njihovog rezultata, trebalo je donositi neke odluke koje se donose na osnovu toga za šta se primenjuju, ali je algoritme jednostavno prilagoditi potrebama.

Literatura

1. http://www.opendesign.com/files/guestdownloads/OpenDesign_Specification_for_.dwg_files.pdf
2. http://www.wikipedia.org/
3. http://en.wikipedia.org/wiki/C_Sharp_(programming_language)
4. http://en.wikipedia.org/wiki/AutoCAD
5. http://www.microsoft.com
6. http://usa.autodesk.com/