DXF Tweeker

Contents

[Getting Started 4](#_Toc185490721)

[System Requirements 4](#_Toc185490722)

[Key Features 4](#_Toc185490723)

[Terminology 4](#_Toc185490724)

[Resources 4](#_Toc185490725)

[1. DXF file format 101 5](#_Toc185490726)

[Group code 5](#_Toc185490727)

[Most common group codes 5](#_Toc185490728)

[Sections 5](#_Toc185490729)

[Example CAD Block 6](#_Toc185490730)

[2. Structure of Application 8](#_Toc185490731)

[Directories 8](#_Toc185490732)

[Execution flow 8](#_Toc185490733)

[3. Package – Windows 9](#_Toc185490734)

[MainMenu Class 9](#_Toc185490735)

[4. Package – Misc 10](#_Toc185490736)

[MessageConsole Class 10](#_Toc185490737)

[5. Package – FileHandlers 11](#_Toc185490738)

[DXFLoad Class 11](#_Toc185490739)

[DXFFileSaver Class 12](#_Toc185490740)

[FileChooser Class 13](#_Toc185490741)

[Settings Class 14](#_Toc185490742)

[6. Package – DXFRead 15](#_Toc185490743)

[DXFDrawing 16](#_Toc185490744)

[SetPolylinesVertexes 16](#_Toc185490745)

[DXFRead 17](#_Toc185490746)

[Tables class/package 18](#_Toc185490747)

[Objects class/package 19](#_Toc185490748)

[Blocks package 20](#_Toc185490749)

[BlockElement class 20](#_Toc185490750)

[Block class 20](#_Toc185490751)

[Blocks class 20](#_Toc185490752)

[Coords class/package 22](#_Toc185490753)

[Methods: 22](#_Toc185490754)

[Entities package 24](#_Toc185490755)

[Entities class 24](#_Toc185490756)

[7. Package - DXFModify 27](#_Toc185490757)

[ModifyHandSeed class 27](#_Toc185490758)

[ChangeAttributes class 28](#_Toc185490759)

[Execution flow 28](#_Toc185490760)

[AddPolyline class 29](#_Toc185490761)

[AddPolyline 29](#_Toc185490762)

[ADDGroups class 30](#_Toc185490763)

[AddPolyline 30](#_Toc185490764)

[8. Package – CADObjects 31](#_Toc185490765)

[CADObjects Class 31](#_Toc185490766)

[ExambleBlock Class 32](#_Toc185490767)

# Getting Started

**DXF Tweaker** is JAVA application with purpose to read DXF files as well as provide a way to modify them. Due to complexity of DXF files, this is not catch-all application and only some parts of DXF format will be present, mainly focusing on 2D drawing. However, brief explanation of DXF file format will be provided to allow for future modification for developer to add new features.

## System Requirements

Application is written into java 8 and Windows OS. There is no need for CAD application to be installed on computer, however, it is recommended to verify the output.

**Language level 8**

**SDK**: 22 Oracle OpenJDK 22.0.1

## Key Features

* Reading DXF files
* Recreating CAD elements in form of JAVA Objects
* Modify CAD elements directly in DXF file
* Adding new CAD elements
* Provide used with text feedback

## Terminology

* CADBlock or Block – represent known in CAD software as block definition, where multiple base elements (i.e. polyline, text, hatch) are combined into single entity under defined name.
* Element – represent single object in DXF file.

## Resources

[AutoDesk – DXF Reference document](https://images.autodesk.com/adsk/files/autocad_2012_pdf_dxf-reference_enu.pdf)

# DXF file format 101

This chapter will provide quick explanation to DXF file format and its overall structure.

DXF format in essence is CAD files represented in text form. Each file is split into sections (Header, Classes, Tables, Blocks, Entities, Objects).

When reading DXF it should read in groups of two lines. First line contains the ***group code*** that is indicated with *integer* number. Second line contain the actual **information/data**.

## Group code

Group code is represented with whole number without decimals and is at minimum 3 characters long. For group codes with characters count less than 3 it is prefixed with whitespace characters (“space”) i.e. group code 5 will be represented as “ 5” instead of “5”. For ease of reading, all whitespace characters in group codes will be represented as “\_”.

Group code represent both “name” of the value its preceding as well as type of the value (string, integer etc.).

## Most common group codes

* “\_ \_0” represent start of new section or element as well as indicates the element type  
  i.e. HEADER, ENTITY, LINE
* “\_ \_5” represent Handle of element, which is unique identifier for that element.
* “ \_ \_8” represent Layer Name the element is located on.
* “\_ 62” represent element Color (out of current palette, for true color 420 code is used)
* “330” represent parentHandle, that will contain Handle of other elements (used to link elements to each other)
* “360” represent softPointer – similar as in code 330
* “\_10” – primary point, this is insertion X coordinate for element (followed by Y and Z value codes 20, 30)
* “ \_11” “\_18” – other points (coordinates) followed by Y values codes 21-28 and Z value codex 31-38

## Sections

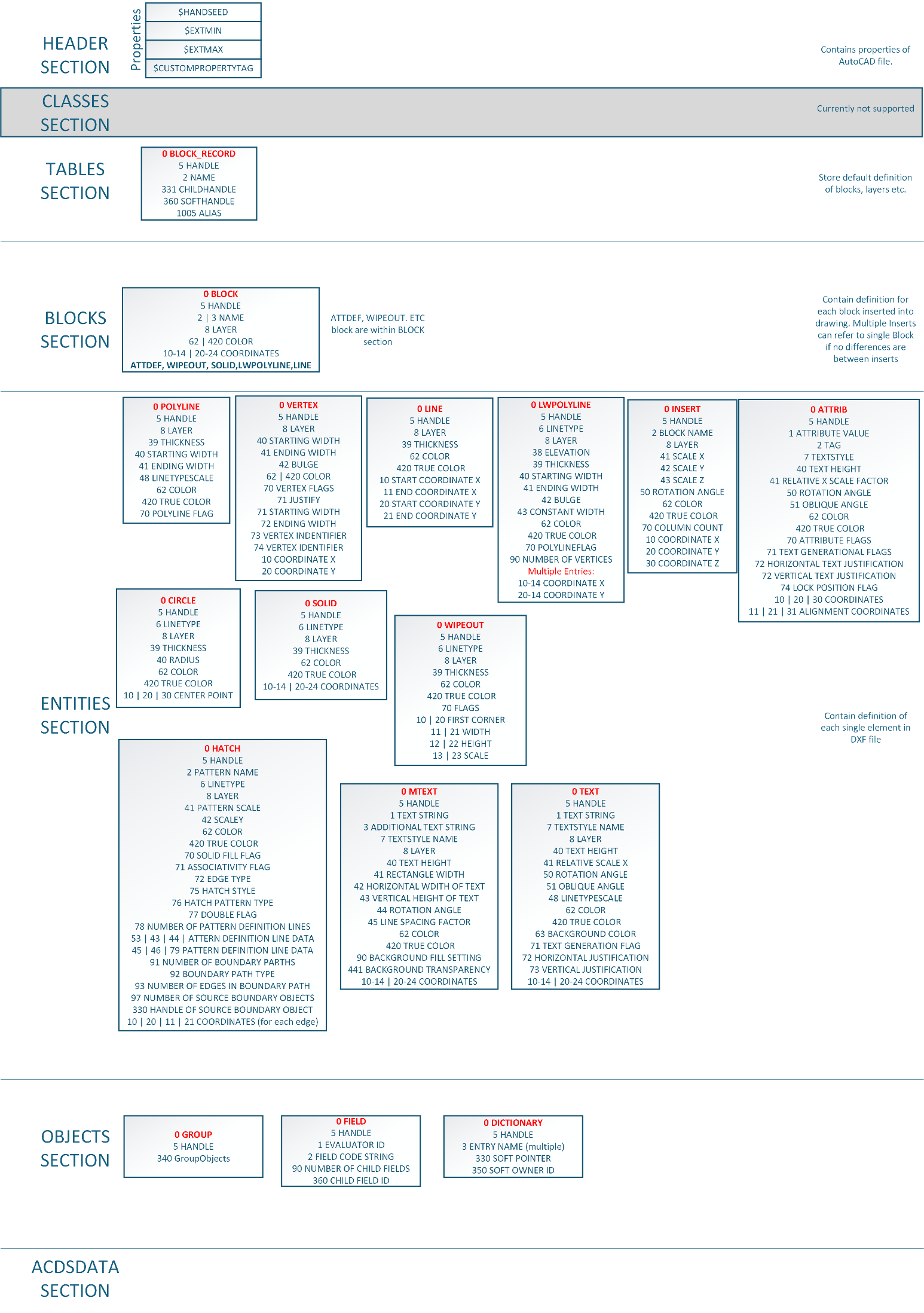
There are 6 distinct sections of DXF file:

* HEADER – contain all variables associated with the drawing, i.e. Last save data, Custom properties, Current Layer, Next Handle etc.
* CLASSES – contain all information about application-defined classes that appear in BLOCKS, ENTITIES, OBJECTS sections. Currently not supported
* TABLES – contain data tables present in drawing, for example names of CAD Blocks.
* BLOCKS - contain definitions of CAD Blocks present in drawing, please note if Block has editable properties, each version will be represented as separate BLOCK
* ENTITIES – largest section. Contain all base elements of the drawing i.e. LINE, POLYLINE, VERTEX, ATTRIB
* OBJECTS – contain elements like “GROUP” where two CAD Blocks are grouped with each other.

For more in depth description of each section, refer to official AutoDesk documentation.

## Example CAD Block

|  |  |
| --- | --- |
| 0 | Start of new element |
| LWPOLYLINE | Name of element |
| 5 | Handle |
| 133A0C2 |
| 330 | ParentHandle |
| 12ABF3D |
| 100 | Subclass marker |
| AcDbEntity |
| 8 | Layer name |
| Layout-Legend |
| 6 | LinyType |
| Continuous |
| 62 | Color |
| 7 |
| 100 | Subclass marker |
| AcDbPolyline |
| 90 | Number of Vertexes |
| 5 |
| 70 | PolylineFlag (1= closed) |
| 128 |
| 43 | Constant width |
| 0.3375 |
| 10 | Coordinate X (of 1st vertex) |
| -124.7301818024072 |
| 20 | Coordinate Y (of 1st vertex) |
| 20.66093563164514 |
| 10 | Coordinate X (of 2nd vertex) |
| -122.6208068024072 |
| 20 | Coordinate Y (of 2nd vertex) |
| 21.29374813164515 |
| 42 | Bulge (not supported) |
| 0.4142135623730951 |
| 10 | Coordinate X (of 3rd vertex) |
| -120.5114318024072 |
| 20 | Coordinate Y (of 3rd vertex) |
| 23.40312313164515 |
| 42 | Bulge (not supported) |
| 2.414213562373094 |
| 10 | Coordinate X (of 4th vertex) |
| -122.6208068024072 |
| 20 | Coordinate Y (of 4th vertex) |
| 21.29374813164515 |
| 0 | Start of new element (and end of current element) |
| MTEXT |



# Structure of Application

## Directories

The application is split into 6 main categories of files/folders.

* **Windows** – contain all UI elements and classes
* **Misc** – contain miscellaneous classes
* **FileHandlers** – contain all classes related to reading, saving files, as well as handling setting file.
* **DXFRead** – contain all classes related to loading and reading DXF file
* **DXFObjects** - contain all classes related formatting all data present in DXF file into form of JAVA objects
* **DXFModifity** – contain all classes related to changing DXF data.

## Execution flow

1. Application start from Main.class in Main package.
2. Class Settings is executed, checking APPData for user setting file, if not present, default settings is loaded.
3. MainWindow class is executed, launching UI.
4. Awaiting for user input.

# Package – Windows

This package contain all classes for UI elements used in application.

## MainMenu Class

The MainMenu class extends JFrame and serves as the main window for the DXFTool application. It provides a user interface for loading DXF files and displaying output messages.

A white rectangular object with a black border

Description automatically generated

Constructor **MainMenu()** set up entire window as above with elements:

1. Window Title
2. Buttom “**DXF Load”** - used by user to load DXF file
3. textPane – that will print all system messages.

Constructor also references class in **Misc** package **MessageConsole** that catches all System.out.Print message and redirect then to **textPane**.

**DXF Load** Button’s ActionListener trigger class DXFRead which start main part of application to load DXF File and subsequently analize the file and build objects based on DXF file elements.

This is loaded into separate Thread to allow printing system messages simultaneously while program is running.

Commented out lines are there to allow for easier debugging when developer want to avoid launching MainMenu at debug stage.

# Package – Misc

This package contain all classes for functions that do not fit into other categories.

## MessageConsole Class

This is class to catch all System.out.Print and redirect it to **TextPane** in **MainWindow**.

It is important for whatever other process is running and printing messages to be on separate threat, otherwise message will not be printed simultaneously.

Class is based on github repo: <https://github.com/tips4java/tips4java/blob/main/source/MessageConsole.java>.

Class is used in **MainWindow** class:

MessageConsole console = new MessageConsole(textPane1);  
console.redirectOut();  
console.redirectErr();

# Package – FileHandlers

## DXFLoad Class

The DXFLoad class is responsible for reading the contents of a DXF file. It provides methods to open the file and read its lines into an array of strings.

Class is initialized within class **FileChooser**.

Returns array of strings. Each string is separate line of loaded file.

## DXFFileSaver Class

Class is used to create and save new DXF file after changes has been made to the loaded one.

Class is initialized with parameters:

* String[] **aryLines** – representing new array of Strings where each string is line of new file
* String **filename** – name of original file.

Execution stepts are as follows:

1. Removes .dxf from original file name

This is to ensure the final file will not have .dxf in middle of its name

1. Get parent directory of original file
2. Get current time and date to later append it at the end of file name.

Saved it in Year-Month-Date-Hour-Minute format. Each part consist of 2 characters.

1. Create new file name consisting of “old file name” + current time.

If file of that name already exists append **(#)** at the end, **#** represent subsequent number depending on number files with that name already existing.

1. Write file to the disk.

## FileChooser Class

Class is responsible for opening prompt window for user to select DXF file to load.

Class is initialized without any parameters.

Initialized window has set filters to only show and allow DXF files.

It calls class settings for variable “**FILECHOOSER**” that contain default location to start in. Additional check if directory under variable **FILECHOOSER** exists.

If not default location is provided, it will default to “user.home” directory.

Return selected file’s absolute path as String

## Settings Class

Class is responsible for loading and/or creating user’s setting file.

Class is initialized as singleton and its variables are declared as static.

List of methods:

* **Constructor Settings**
  + Declares and set variables with default data
* **getInstance**
  + Method used to create singleton version of class. First it ensures settings directory exists. Then load default settings and (if exists) user settings file. At the end it compares these two and If needed update user settings file.
* **ensureDirectoriesExist**
  + Method check if directory with user setting file exists. If not it creates necessary folders.
* **LoadDefaultSettings** 
  + Load default settings file from resources in .jar package. Read the file line by line and put the information into hashmap **defaultSettingsMap** following schema of [variableName]=[variableValue]
* **LoadOrCreateUserSettings**
  + Method populates hashmap **SettingsMap** with data from user’s Settings file, if file does not exist, it populates its with default values from default settings file.
* **Compare Settings**
  + Method compares user setting file with default settings file. In case where user settings has empty variable, it repopulates it from default settings file. If any settings variable is missing, it adds it. Any edits are on hashmaps not on text files.
* **modifySettingsFile**
  + Method modify user’s settings file. It will also change SettingFilePath variables (always) to the correct directory. At the end it save it to the disk.
* **getValue**
  + Method for other classes to call to get value of called variable.

# Package – DXFRead

Package contain all classes and methods required to parse DXF file loaded via **DXFLoad.** It recreates all elements in form of JAVA Objects and store them in corresponding HashMaps.

It contain two main classes

* **DXFDrawing**
* **DXFRead**

The rest of package is split into another set of packages each for different type of DXF elements (although not all of them only major ones):

* **Blocks**
* **Coords**
* **Entities**
* **Objects**
* **Tables**

## DXFDrawing

This class is central Object of **DXFRead** package.

It will contain declaration of main object of DXF File, such as, Tables, Blocks, Objects as well as DrawingProperties hashmap and Indexes for start line of each main section of DXF Files (required for DXF modification methods).

Included methods:

* **updateIndexes**
  + increase indexes where necessary after adding new elements to the DXF file
* constructor **DXFDrawing**
  + Reads through header section of DXF Files and set important variables (i.e EXTMIN, EXTMAN, HANDSEED)
* **getNextSeed**
  + function to get next available Handle when creating new DXF elements.
  + Function will automatically increment HANDSEED value.

**A close-up of a sign

Description automatically generated**

* **SetPolylinesVertexes**
  + Method to assign VERTEX elements to POLYLINE elements. More detailed explanation below.
* **getEndSecIndex**
  + return current value of iterator variable i.

### SetPolylinesVertexes

Polyline elements (do not confuse with LWPolyline) do not contain coordinates in DXF directly (codes 10, 20) and instead there are separate elements VERTEX created and linked to Polyline via ParentHandle code (330).

Method also removes first coordinates if they are 0.0.

## DXFRead

This is main class of package DXFRead.

Execution flow of DXFRead class:

1. **Calls for FileChooser class that prompts user to select DXF Files**

If user does not select file, it return null and cancel further execution.

1. **Initalizes DXFDrawing object**

From class **DXFDrawing**

1. **Loops through entire DXF file (array of Strings)**

Loop iterate through entire file and looks for start of pre-defined sections.

At start of each section, prints message to user to inform which section is currently analized.

1. **Calls for CADObject class**

Start recreating CAD Blocks as JAVA Objects and store them in hashmaps.

Exception is in header section, where is calls constructor of DXFDrawing class instead.

## Tables class/package

This is class to read DXF elements in **TABLES** section. Currently only supported element is **BLOCK RECORD** that will be used to find true name for AutoCAD Dynamic Blocks.

**Fields**:

* static hashmap **BlockRecord**.

#### Constructor Tables():

* arguments:
  + **currentline** – Current iteration index
  + **length –** number of lines of DXF file
  + **file** – DXF File
* **main loop**
  + iterates through DXF file starting from start of TABLES section.
  + Enters second loop to look for DXF codes in DXF file
  + Creates BlockRecord object
  + populates BlockRecord variables based on DXF file (Handle, ChildHandle, Name)
  + Once DXF Code “\_ \_0” is encountered, breaks second loop and input current Block Record into BlockRecord Hashmap
  + Next iteration of first loop
  + Once DXF Code “ENDSEC” is encountered, breaks main loop and closes entire method, returning back to DXFRead class

#### Static class BlockRecord

Contains all necessary variables to build Block\_Record element.

## Objects class/package

This is class to read DXF elements in **OBJECTS** section. Supported elements are **GROUP** **FRIELD** **DICTIONARY.**

This section can be used to look for elements that are grouped and with what elements they are grouped with.

**Fields**:

* static hashmap **Objects**.

#### Constructor Objects ():

* arguments:
  + **currentline** – Current iteration index
  + **length –** number of lines of DXF file
  + **file** – DXF File
* **main loop**
  + iterates through DXF file starting from start of OBJECTS section.
  + Enters second loop and look for GROUP, FIELD or DICTIONARY in DXF file
  + Creates object **dxfObject**
  + populates **dxfobject** variables based on DXF file (Handle, ObjectsValues)
  + if 340 code is encountered, add object handle to GroupObjects map (multiple entries possible)
  + Once DXF Code “\_ \_0” is encountered, breaks second loop and input current **dxfObject** into Objects Hashmap
  + Next iteration of first loop
  + Once DXF Code “ENDSEC” is encountered, breaks main loop and closes entire method, returning back to DXFRead class

#### Static class dxfObject

Contains all necessary variables to build **dxfObject** element.

## Blocks package

This package is to load definitions of CAD Blocks as java objects.

It is important to not misunderstand difference between separate INSERT entities and BLOCK entities.

Block entity is definition of CAD Block definition stored in CAD Template. Block entity includes in its definition all primary DXF elements such as, LWPOLYLINE, ATTDEF etc. All of them share code 330 that is linked to Handle (\_ \_5) of BLOCK\_RECORD entity.

IF in CAD file one CAD Block is inserted two times into the drawing, but with different values of its attributes (i.e. DIAMETER) they will still be present as single BLOCK\_RECORD, however, as two separate INSERT entities (with two different ATTRIB elements)

### BlockElement class

This class is definition of BlockElement that is used for all elements that can be present inside BLOCK definition in DXF file. The final object will be then stored in **BlockElements** hashmap inside Block class.

### Block class

This class is definition of Block that store definition of CAD Block. Inside this class is defined hashmap **BlockElements** that store all primary DXF elements from which Block is built.

### Blocks class

This class is main class of Blocks Package and it purpose is to recreate BLOCKs from DXF file. It consist of two nested loops, first to catch start of definition of BLOCK, second to catch all primary entities belonging to the BLOCK entity, such as LWPOLYLINE, SOLID, ATTDEF. These elements are store in hashmap **BlockElements** of Block class.

Important to remember, each single element of BLOCK element (including BLOCK itself) has its own coordinates. These coordinates are usually in form of offset from main coordinate of block.

**Fields**:

* static hashmap **Blocks**.

#### Constructor Objects ():

* arguments:
  + **currentline** – Current iteration index
  + **length –** number of lines of DXF file
  + **file** – DXF File
* **main loop**
  + iterates through DXF file starting from start of BLOCK section.
  + Before entering second loop creates object **block**
  + populates **block** variables based on DXF file (Handle, ObjectsValues)
  + if any of primary DXF entities names are encountered (ATTDEF, SOLID …) calls CreateBlock method.
  + Once DXF Code “\_ \_0” is encountered, breaks second loop and input current **block** into **Blocks** Hashmap
  + Next iteration of first loop
  + Once DXF Code “ENDSEC” is encountered, breaks main loop and closes entire method, returning back to DXFRead class

#### createBlock method

* Assign corresponding name of **block** object
* Save current DXF line (remnant of old code, might be used in future versions)
* Call **readBlockElement** method
* Add **block** object into the hashmap **BlockElements** of **Block** class

#### readBlockElement method.

This method is called within Block section in DXF. It works similar to all other method gathering information about DXF elements. It loops through entire of Block definition and reads all primary elements (LWPOLYLINE, ATTDEF) and populate them with theirs variables.

## Coords class/package

This class is used to store all coordinates of other DXF elements. As well as all methods used on coordinates through out entire application.

There is only one hashmap **mCoords** that stores **Coordinates** class with incremented Integer as key.

**Coordinates** class is build from two double values of x and y. In case of working on 3D space, it would need to be expanded to z coordinate.

### Methods:

#### AddCoords

Simple method to add new coords to the hashmap.

Two variants:

* accepts two arguments: double x & y
* accepts one argument: as object Coordinate

#### reIndexElements

Method to reindex elements of mCoords Hashmap. Used after other method that might delete some entries (for example duplicates)

#### Size

Return size of mCoords hashmap

#### getThatCoord(int i)

return i-element of mCoords hashmap in form of Coordinate class

#### getNCoords(int i)

same as above

#### getNCoordX(int i)

Return x coordinate of i-element with precision to 1/1000th

#### getNCoordY(int i)

same as **getNCoordX** but for coordinate y.

#### getrawNCoordX(int i)

Returns x coordinate of i-element with any rounding.

#### getrawNCoordY(int i)

Returns y coordinate of i-element with any rounding.

#### replaceCoordsXY(int I, double x, double y)

Replace i-elements of hashmap mCoords with values x,y

#### removeCoords(int i)

Remove i-element of mCoords hashmap.

#### removeZeroCoords()

Iterates through entire hashmap mCoords and remove coordinates that are 0,0

trimAndReturnDuplicates()

Iterates through entire hashmap mCoords and set duplicate values to 0,0. Then call removeZeroCoords to remove these.

RemoveDuplicates()

Iterates through entire hashmap mCoords and removes duplicate values.

## Entities package

Class is used to read all types of primary elements of DXF from section **ENTITIES.**

Every element in CAD file can be split into primary Entities such as LWPOLYLINE, SOLID, HATCH, MTEXT. Entities package support below entities:

* POLYLINE – in simple words, lines in drawing
* LWPOLYLINE – lightweight polyline, it is type of polyline that can only be present as 2D object. They are more efficient when it comes to RAM usage or graphic computation
* LINE – is line that only has start and end point
* VERTEX – is singular point that is usually part of POLYLINE or LWPOLYLINE
* INSERT – as per name it is element that describe insertion point of other elements (i.e. CAD Block)
* ATTRIB – it is element that represent attribute of other objects (is always accompanied with ATTDEF)
* CIRCLE
* HATCH – used to define hatches of some areas in CAD drawing
* SOLID – simpler version of hatch that only support solid fills of area
* WIPEOUT – element used i.e. under text as a background
* TEXT – standard text field, single line
* MTEXT – multi line text field
* POINT
* IMAGE – external image inserted into CAD Drawing

### Entities class

This is core class of this package. It contain constructor **Entities** and static class **Entity**.

#### Static class Entity

This class hold all declarations of variables as well as its setters and getters.

There is one method in this class **addAtribute** which sole purpose is to add new attribute to Attributes hashmap (name as key)

#### Constructor Entities

In similar fashion to other classes in DXFRead main package this constructor loop through DXF File (ENTITIES section) and look for predefinied named of Entities (listed at start of this chapter). Each entity follow execution flow of:

* Create new Entity of class Entity
* Set its name as per DXF name
* Save index of start line of that entity in DXF
* Call method readEntity from Class ReadEntity
* Add finished entity object to EntitiesMap Hashmap

Most entities follow the above pattern, with exception to POLYLINE, LWPOLYLINE, ATTRIB:

* **POLYLINE** – there is additional check for PolylineFlag to check if polyline is closed and if so to duplicate starting coordinate at the end of coordinates hashmap
* **LWPOLYLINE –** same as above
* **ATTRIB –** it will also populate its ATTribs hashmap with the attribute name and value it represent and then add assign it to the object. It also add this attribute to list of all attributes in ATTribEnt hashmap.

#### readEntity Class/Package

This class is called from Entities class and its purpose is to literate through DXF file and populate values of variables of Entity object.

There is only one loop in class that finishes its run once it encounters (\_ \_ 0) code. Only index of current DXF file line is returned as method is working directly on object inherited from Entities Class it has been called from.

Most of switch cases are self explanatory from code i.e.:

case " 6":  
 entity.setLineType(aryLines[i + 1]);  
 break;

The exceptions are codes:

* \_ \_ 2
  + There is additional check for names starting with “\*U” which indicate the current Entity is dynamic block and thus there is need to call method **FindBlockName** to find true name of the block
* \_ \_ 3
  + This code is usually used to store secondary name, with exception to MTEXT entity. String present in MTEXT entity is stored under code \_ \_ 1, however, if its longer than 250 characters, then it will be stored under code \_ \_ 3 (multiple entries possible) so additional check on this code is required.
* \_ 40
  + For entity LWPOLYLINE this code represent Width, for other entities it will represent TextWidth
* \_41
  + Additional check is to store Scale only for objects we support (there are multiple different use cases for 41 code depending on objects : LWPOLYLINE (End Width), HATCH(Pattern Scale), DIMENSION(Text Height).
* \_ 62
  + When true color is used, then code 420 should be read for correct color value. Additional check is needed to ensure 420 code value is not overridden with 62 code value
* \_ 90
  + For MTEXT entity code 90 is used for Background type
* 330
  + Additional check to set ParentHandle only when it is not equal to “1F” to ensure rest of code works correctly (this might be issue only related to my own projects)

**Method FindBlockName**

Purpose of this method is to find true name of dynamic block in AutoCAD.

Dynamic blocks in autocad do not follow standard naming scheme in DXF file. In Insert element their names are stored as “\*U” followed by incremented number i.e. \*U181

To find its true name we need to refer to BLOCK\_RECORD element stored in Tables class.

To do this, class first loops through all BlockRecords and loop for BlockRecord that has same name and our entity name **\*U500**

A screenshot of a computer program

Description automatically generated

Then get its **Allias** value and loop again looking for BlockRecord that has Handle(key) equal to this allias and then from that object export True name of our Entity **Civils-Tee**

A screenshot of a computer program

Description automatically generated

A diagram of a computer

Description automatically generated

# Package - DXFModify

This package contain classes that are responsible for modifying DXF files or creating DXF code for elements created in other classes (i.e. new lines).

There are provided 4 example classes that should be enough as base to create new classes based on project needs:

* **AddGroups**
  + This class create DXF code for GROUP element
* **ADDPolyline**
  + This class create DXF code LWPOLYLINE element
* **ChangeAttributes**
  + This class purpose is to change attributes for already existing CAD Block in drawing
* **ModifyHandSeed**
  + This class purpose is to modify drawing Handseed property. It is required to be run at the end of out DXF file modification, otherwise file will be marked as invalid (required only when adding new elements)

## ModifyHandSeed class

HandSeed property in CAD file is responsible for providing next available Handle number. The number is HEX number.

If any actions have been taken to add new elements to the DXF file, it is important to run this method at the end of these actions, otherwise, CAD file will be marked as invalid and will not open.

When building our methods to add new elements to DXF file, it is important to also call **getNextSeed** from **DXFDrawing** class so HandSeed variable is properly incremented during runtime.

## ChangeAttributes class

This class purpose is to change attributes values for already existing blocks in CAD drawing. It requires to be called with parameters:

* String **attributeName**
* Hashmap<String, String> **Handle of Object(that ATTRIB is part of), newValue**
* String[] **linesOfDXF**
* DXFDrawing **Dxf file**

### Execution flow

1. First it look for element “ATTRIB”
2. Call method **FindAttributeToChange**
3. Check code (330 - ParentHandle) if this is the attribute we want to change
4. Call method **FindAttributeToChangeCheckForAttributeType**
5. Check code (\_ \_ 2 – Name) if we have correct name of ATTRIB
6. Reloop again, since code (\_ \_ 2) for name exist after code( \_ \_ 1) for Value in DXF file
7. Find code (\_ \_ 1 – Value) and change it to desired outcome

A screenshot of a computer

Description automatically generated

## AddPolyline class

Main purpose of this class is to contain all method required to add create POLYLINE elements in form of DXF file code.

This class contain two methods:

### AddPolyline

This method is called with parameters:

* **ArrayList<String>** which is just array of strings of DXF code (example name of DXFCables)
* **String[] aryLines** – which is DXF code itself
* **DXFDrawing CAD** – dxf file

#### Execution flow:

1. First there is check if **ENTITIES** section exists (it is very unlikely it doesn’t and it would most likely be invalid DXF file or corrupted file).
2. Create copy array of **aryLines** from beginning to the **end** of **ENTITIES** section
3. Iterate through ArrayList provided while calling function, and insert each ArrayList element (string) into copied array of **aryLines**
4. To the copied Array of **aryLines** append the rest of DXF code after end of **ENTITIES** section
5. Call method **updateIndexes** from DXFDrawing method (it is to ensure the similar methods like AddPolyline will work properly)
6. Return new **aryLines** (newLines)

#### CreateDXFPolyline

This method is called with parameters:

* **Polyline polyline** which is object of type Polyline with necessary variables
* **ArrayList<String>** which is just array of strings of DXF code (example name of DXFpolyline)

There is just one execution step of this method. Its add to the ArrayList<String> **DXFpolyline** necessary lines to create Polyline element in DXF.

There are only required lines created by the method, any extra lines should be added depending on specification of application.

## ADDGroups class

Main purpose of this class is to contain all method required to add create GROUP elements in form of DXF file code.

This class contain two methods:

### AddPolyline

This method is called with parameters:

* **ArrayList<String>** which is just array of strings of DXF code (example name of DXFCables)
* **String[] aryLines** – which is DXF code itself
* **DXFDrawing CAD** – dxf file

#### Execution flow:

1. First there is check if **OBJECTS** section exists (it is very unlikely it doesn’t and it would most likely be invalid DXF file or corrupted file).
2. Create copy array of **aryLines** from beginning to the **end** of **OBJECTS** section
3. Iterate through ArrayList provided while calling function, and insert each ArrayList element (string) into copied array of **aryLines**
4. To the copied Array of **aryLines** append the rest of DXF code after end of **OBJECTS** section
5. Call method **updateIndexes** from DXFDrawing method (it is to ensure the similar methods like ADDGroups will work properly)
6. Return new **aryLines** (newLines)

#### CreateDXFPolylineGroup

This method is called with parameters:

* **Polyline polyline** which is object of type Polyline with necessary variables
* **ArrayList<String>** which is just array of strings of DXF code (example name of DXFpolyline)

This method will create DXF code for element Group.

Since how this method will be executed depends on structure of application. This method only show example way of doing so.

**Assumptions**

There is hashmap of **Polylines** each Polyline object has hashmap **Grouped** where there are stored Handles of elements it supposed to be grouped with.

Method itself will add elements one by one, required lines of DXF code to the DXFGroups and at the end under code (340) it will add all Handles from Polyline.Grouped hashmap. Each Handle should be accompanied with (340) code.

# Package – CADObjects

This package should contain classes where definitions of all user created CAD objects as well as all necessary methods.

There has been prepared Example package with example class. There is also class Polyline that has basic methods for creating Polyline objects.

Main class of this package is CADObjects, that should be responsible for calling other classes when creating JAVA Objects based on CAD Blocks.

## CADObjects Class

This class only one parameter:

* **DXFDrawing** – the DXF file object (not the file itself)

The easiest way to extract DXF elements is to iterate through hashmaps in DXFDrawing type object we provided to the method.

Depending on DXF element we will need to iterate through different hashmaps.

For user defined CADBlocks it should be iterated through **Inserts** hashmap.

For non-block elements, such as lines, polylines it should be iterated through **Entities** hashmap.

In case we want to get information about grouping of the elements, we should iterate through **Objects** hashmap.

As example there has been provided loop that finds CAD Blocks with specific name **Table**.

for (String key : *Inserts*.keySet()) {  
 Entities.Entity insert = *Inserts*.get(key);  
 if ("Example-Layer".equals(insert.getLayer())) {  
 ExampleBlock exampleBlock = new ExampleBlock();  
 exampleBlock.setBlockName("Table");  
 CreateExampleBlock(insert, exampleBlock, DXF);  
 }  
}

Depending on specification, how this loop is constructed might differ. In example we first look for elements that are inserted on **Example-Layer** and then we check for object **Name**. When we get object we are looking for we can now call **CreateExampleBlock** method that will recreate that object into JAVA object.

## ExambleBlock Class

#### CreateExampleBlock method

This method is populating our variables of **ExampleBlock** object with data from DXF file, such as **Handle, Coords, Scale**.

exampleBlock.setHandle(insert.getHandle());  
exampleBlock.setCoords(insert.getCoords());

Then it declaring hashmap **todelete** that use will be talked explained later.

**method**

CreateExampleBlockGetAttributes (ExampleBlock exampleBlock, String handle, DXFDrawing DXF, Map<String, String> todelete)

This method loops through **Entities** hashmap and look for entities with ParentHandle equal to current **ExambleBlock** Handle.

Then it call **ExampleBlockAttribs** method inside ExampleBlock class examine passes entity for attribute name and if it matches any it assign them to our current **ExambleBlock** object.

exampleBlock.ExampleBlockAttribs(exampleBlock, entity);

**ExampleBlockAttrib method**

This method should include cases for every attribute we would like to read from DXF file.

A screenshot of a computer

Description automatically generated

**CreateExampleBlockFindBlock method**

This method is used to get all primary elements that are part of **ExampleBlock** object (lines, text, attribute definitions).

It first iterates through **BlockRecord** hashmap and look for BlockRecord that has **childHandle** equal to our **ExampleBlock** Handle.

Once match is found, it will then iterate through **Block** hashmap and try to find Blocks which **ParentHandle** is equal to BlockRecord **Handle.** Once found it will assign entire hashmap of BlockElements from Block to ExambleBlock

**A diagram of a company

Description automatically generated**