Plaxis BV & Saxion University of Applied Sciences

Graduation Report

Developing smart Autocompletion for the PLAXIS command line tool.

Atanas Tsekov
BSc Game Engineering, CMGT
424638@student.saxion.nl
September – February
2019/2020

Plaxis BV
Computerlaan, 14
2628XK Delft
The Netherlands

Saxion
M.H. Tromplaan, 28
7513AB Enschede
The Netherlands

David Khudaverdyan
Company supervisor
Software Engineer II
David.Khudaverdyan@bentley.com

Paul Bonsma
Academic supervisor
Teacher / Researcher
p.s.bonsma @saxion.nl

PLAXIS

TABLE OF CONTENTS

PLAXIS

LIST OF FIGURES

GLOSSARY OF TERMS

CRUCIBLE	Collaborative code review system for software development
git	A distributed version-control system for tracking changes in source code during software development
Intrinsic Properties (IP)	Default PLAXIS object properties. Objects cannot exist without them.
JIRA	Issue tracking system for agile project management
JENKINS	Open source tool used to automate build and test processes
Plaxis	The name of the company
PLAXIS	The software brand name. i.e. PLAXIS 3D: Input, PLAXIS 2D: Output, etc.
Plaxis Object Layer	Core part of the PLAXIS code, responsible for the creation and maintenance of all objects, properties, features and commands.
Thesaurus	This is the object that collects and organizes the data, later used to generate suggestions for the autocomplete popup. In the report it will be often referred to as a “library”.
User Features (UF)	Non-default PLAXIS object features, added manually by the user

PLAXIS

1 INTRODUCTION

1.1 About the company

Plaxis, founded in Delft in 1993, used to be a small privately-owned company, now a part of a much larger family after its acquisition by Bentley Systems in 2018. Under the PLAXIS brand, the company supplies a range of software tools as well as educational and support services, targeted at the world of geotechnics, geomechanics, geo- and civil-engineering. At Plaxis it is all about the soil knowledge – the software is based on the Finite Element Method (Wikipedia, 2019, p. 1) and intended for 2D and 3D engineering and analysis of soil and rock deformation, structure interaction, as well as groundwater and heat flow. It is applied in areas such as excavations, foundations, tunnels, mines and so on.

Plaxis is a leader in the field of geotechnical engineering. Outside of the Netherlands, the company has offices in Singapore and the United States of America, and its customer base includes nearly every country worldwide.

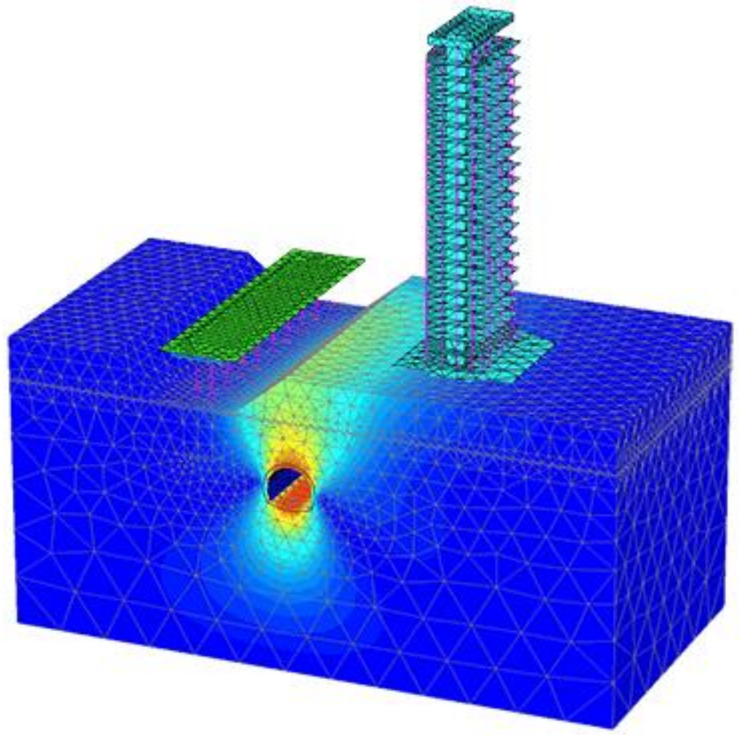


Figure 1.1 **- Example PLAXIS3D model output**

1.2 Company Workflow

With around 65 employees, the Plaxis force is by no means large, however, it is structured in an organized and easily scalable manner. Work is split between several distinct departments:

- **Competence Centre Geo-Engineering** – responsible for the scientific and geotechnical backgrounds of the PLAXIS software. Conducts research and development of the finite element-based calculation kernel, numerical methods and soil constitutive models.
- **Software Engineering** – tasked with the development of the end-user software.
- **Quality Assurance** – ensures no scientific mistakes or software defects are allowed in the final products.
- **Marketing, Sales & Services (Customer Support, Educational Services, Expert Services)** – handle the commercial and customer interaction tasks.
- **General Management and Staff (Finance, IT, HR)** – the Plaxis management team consists of the managers of each department. They are tasked with enabling their respective teams to do the best job possible.

1.2.1 Agile workflow

Each department consists of several self-managing teams, where each team follows the Scrum method of agile development and has a scrum master, product owner and a team manager.

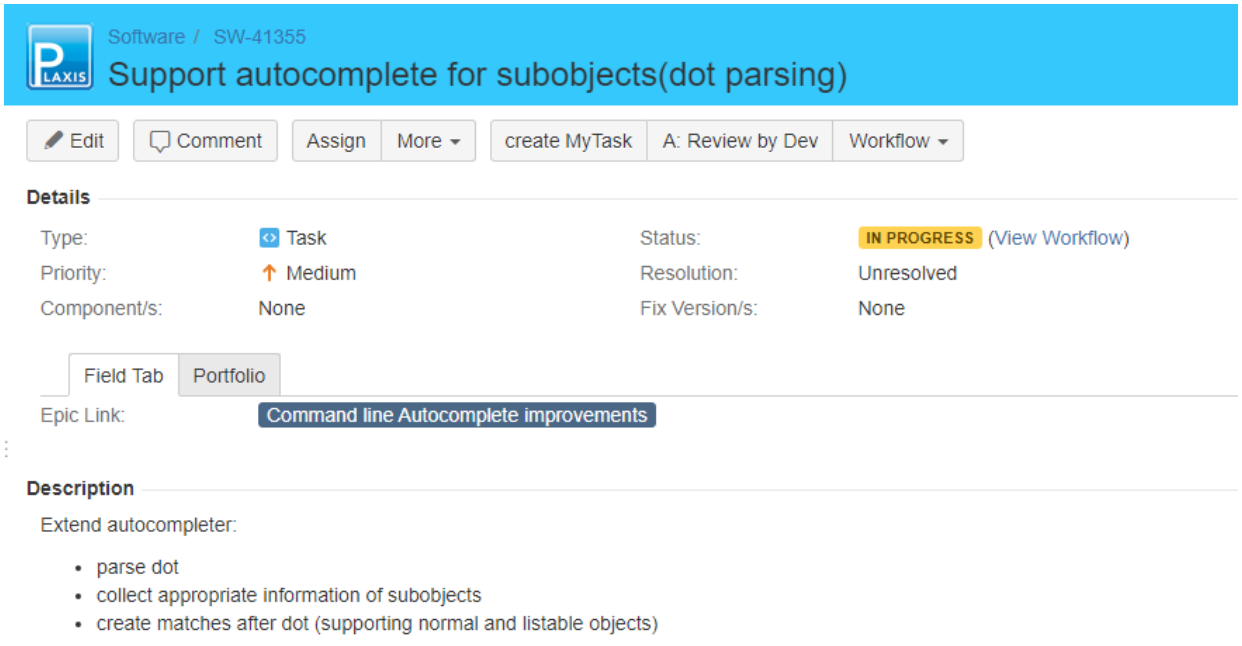
For the graduation project, I am a part of one of the software engineering teams. There are six other developers and two quality assurance engineers.

There are four iterations every year, one for each quarter, and the company releases a new version of the software at the end of each iteration.

Each sprint starts with stakeholder's meeting, where most of the major decisions are made for the coming sprint, followed by a retrospective and sprint planning meeting with the team. Tasks are assigned using Atlassian's JIRA (Figure 1.2).

1.2.2 Software Development

The software engineering teams use git for source control. When a new task is picked up by a developer, they are responsible for working on a new branch named after the task's identification number.


Software / SW-41355

Support autocomplete for subobjects(dot parsing)

Details

Type:	Task	Status:	IN PROGRESS (View Workflow)
Priority:	Medium	Resolution:	Unresolved
Component/s:	None	Fix Version/s:	None

Epic Link: Command line Autocomplete improvements

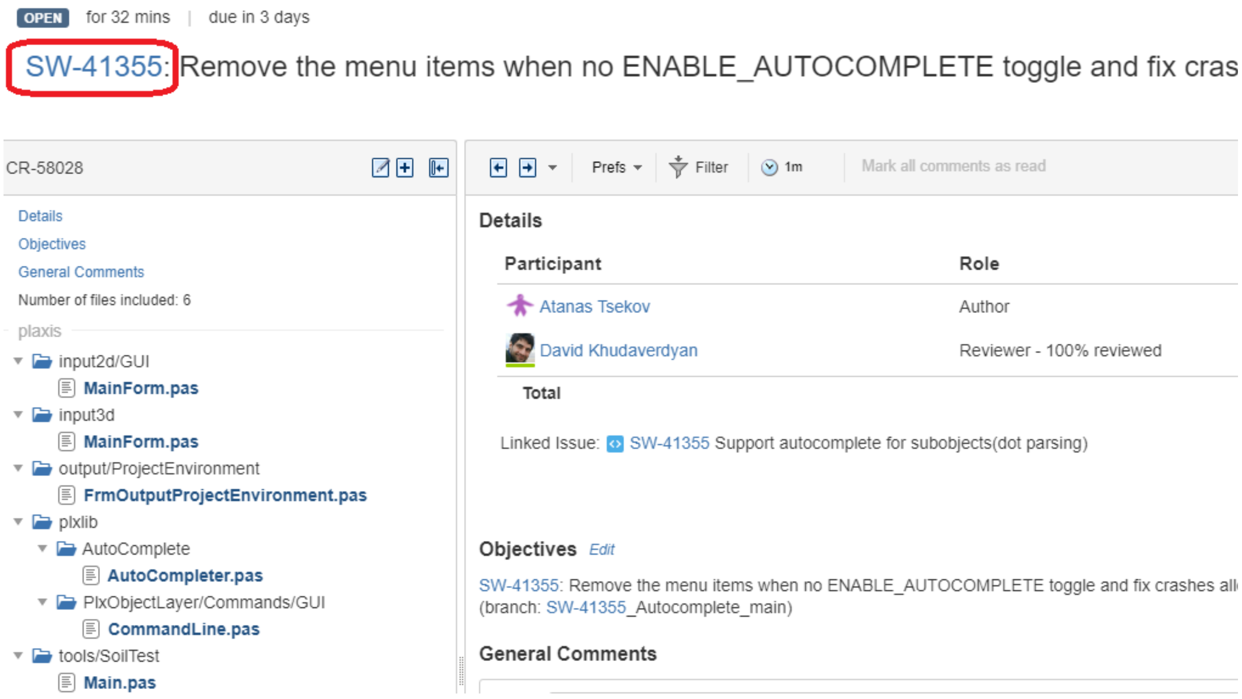
Description

Extend autocompleter:

- parse dot
- collect appropriate information of subobjects
- create matches after dot (supporting normal and listable objects)

Figure 1.2 **Example JIRA task**

This is an important step to ensure that the task tracking (**JIRA -** Figure 1.2), code review (**Crucible -** Figure 1.3), build (**JENKINS -** Figure 1.4) and source control (**git -** Figure 1.5) systems can function in accord.


OPEN for 32 mins | due in 3 days

SW-41355: Remove the menu items when no ENABLE_AUTOCOMPLETE toggle and fix cras

CR-58028

Number of files included: 6

```
plaxis
▼ input2d/GUI
    MainForm.pas
▼ input3d
    MainForm.pas
▼ output/ProjectEnvironment
    FrmOutputProjectEnvironment.pas
▼ plxlib
  ▼ AutoComplete
      AutoCompleter.pas
  ▼ PlxObjectLayer/Commands/GUI
      CommandLine.pas
▼ tools/SoilTest
    Main.pas
```

Details

Participant	Role
Atanas Tsekov	Author
David Khudaverdyan	Reviewer - 100% reviewed
Total	

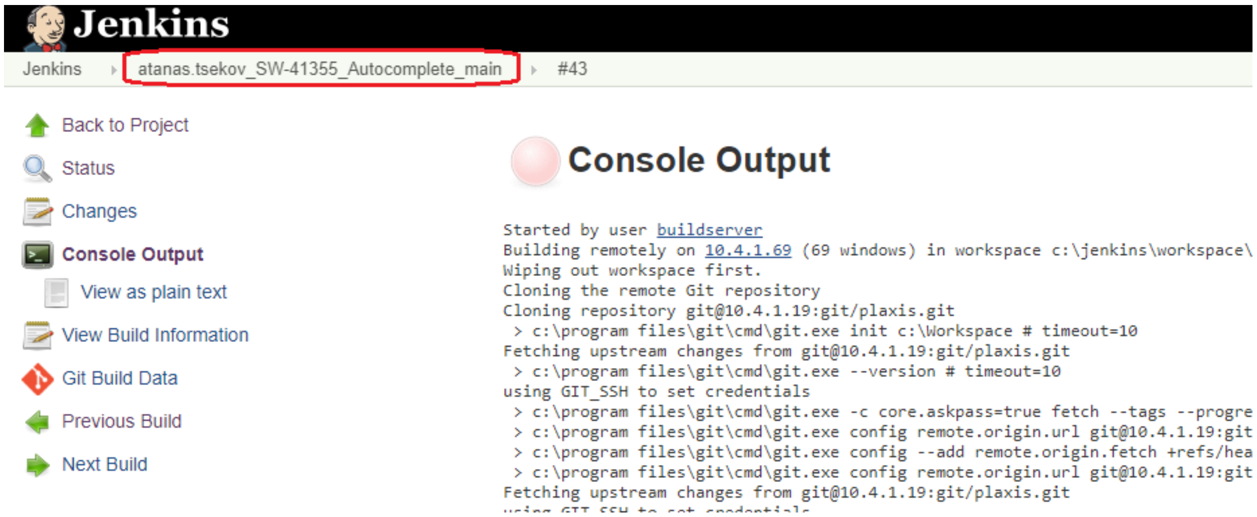
Linked Issue: SW-41355 Support autocomplete for subobjects(dot parsing)

Objectives Edit

SW-41355: Remove the menu items when no ENABLE_AUTOCOMPLETE toggle and fix crashes all
(branch: SW-41355_Autocomplete_main)

General Comments

Figure 1.3 **Example Crucible Code Review**



```
Started by user buildserver
Building remotely on 10.4.1.69 (69 windows) in workspace c:\jenkins\workspace\
Wiping out workspace first.
Cloning the remote Git repository
Cloning repository git@10.4.1.19:git/plaxis.git
 > c:\program files\git\cmd\git.exe init c:\Workspace # timeout=10
Fetching upstream changes from git@10.4.1.19:git/plaxis.git
 > c:\program files\git\cmd\git.exe --version # timeout=10
using GIT_SSH to set credentials
 > c:\program files\git\cmd\git.exe -c core.askpass=true fetch --tags --progre
 > c:\program files\git\cmd\git.exe config remote.origin.url git@10.4.1.19:git
 > c:\program files\git\cmd\git.exe config --add remote.origin.fetch +refs/hea
 > c:\program files\git\cmd\git.exe config remote.origin.url git@10.4.1.19:git
Fetching upstream changes from git@10.4.1.19:git/plaxis.git
```

Figure 1.4 **Example Jenkins building a branch**

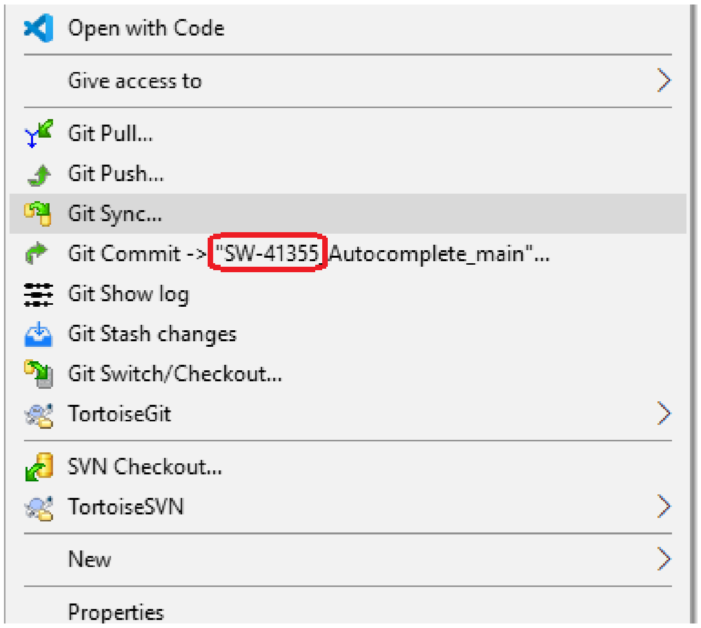


Figure 1.5 **Example GIT branch name**

Once work is finished, the code is pushed to the server where the build process is tested. Alongside that, the code is also submitted for review. When the code reviews are closed and the branch builds successfully on the build server, the task's state is changed to 'Awaiting Merge' and a request is made to merge into Master. Once a merge marshal picks the request, they check whether the build server has successfully built the project and ran all unit tests, all the code reviews have been closed. If everything checks out, the code is merged into the master branch and status changes to 'Resolved'. When the new Master branch builds the Quality Assurance engineers pick it up for testing. If everything checks out the task is finally marked as 'Done'.

2 PROJECT DESCRIPTION

During the graduation period work was mostly[1] focused on implementing a new feature for the PLAXIS software – command line autocompletion.

2.1 Reason for the project

Nowadays, smart autocompletion is something that we, as users of the internet, take for granted. It is in almost every program, app or website we visit. It has become somewhat of a norm when it comes to quality of life features, and thus it is difficult to realize how much it actually helps. It can speed up the typing process as well as correct spelling and grammar mistakes, which is both educational and good for our image.

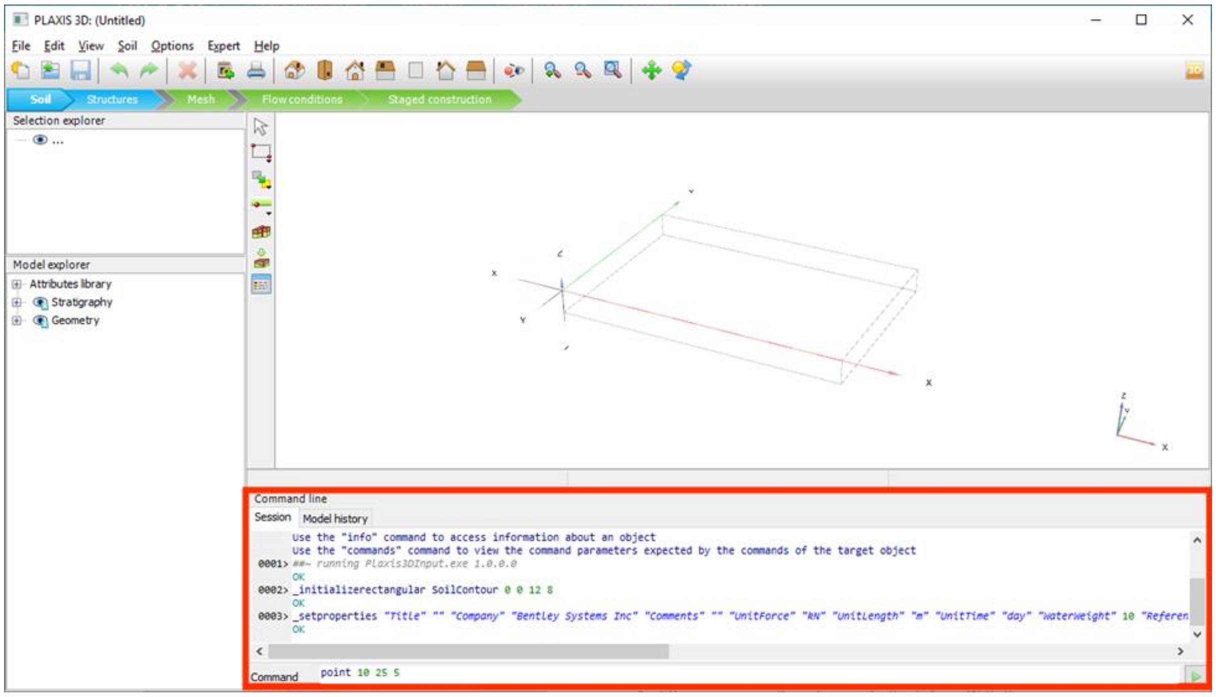
For me, however, the biggest benefit of autocompletion is discoverability. We often find ourselves in situations where we know what we are looking for but don't necessarily know or remember how to find it. So, we go and type something related in Google. Chances are, the word we are looking for is in the top suggestions. It is very similar in the world of tech and software engineering. There are usually giant documentation pages, where the information can be found, however looking through them is a time-consuming task and it heavily relies on their healthy maintenance. A far easier approach is to just start typing and work with the suggestions that an auto-complete feature provides. It can often happen, that these suggestions provide something even better suited for our needs than what we were initially looking for.

Whether it is games, websites, apps or code editors, auto-complete can always improve the user experience in one way or another. That is what makes this project interesting for me to dive into the ins and outs of. Aside from that, I will also be learning and working with a new language, Delphi, and gaining professional experience.

2.2 Problem statement

The PLAXIS software is generally comprised of two main modules – Input and Output. Input has a 2D and a 3D version and is where the user creates a model and specifies its materials. Once the materials' properties have been configured, the Output module is able to provide a detailed presentation of computational results to the user. The modelling process has five different stages: Soil, Structures, Mesh, Flow Conditions and Staged Construction. Each of these stages has its own set of functions and objects it exposes to the user. Collectively, there are hundreds of commands and objects, and far more parameters. This is where the PLAXIS command line editor (Figure 2.1) comes into play.

[1] Several times during the graduation period, work, unrelated to the autocomplete tool, needed to be done. For the sake of keeping the report focused on the main project, no details will be given about the other projects.



```
     Use the "info" command to access information about an object
     Use the "commands" command to view the command parameters expected by the commands of the target object
0001> ##~ running Plaxis3DInput.exe 1.0.0.0
     OK
0002> _initializerectangular SoilContour 0 0 12 8
     OK
0003> _setproperties "Title" "" "Company" "Bentley Systems Inc" "Comments" "" "UnitForce" "kN" "UnitLength" "m" "UnitTime" "day" "WaterWeight" 10 "Referen
     OK
```

```
point 10 25 5
```

Figure 2.1.1 - **PLAXIS Input3D interface**

It allows the more tech savvy users to much more quickly and efficiently define their models, by saving them the time it takes to click each button in the GUI and input each parameter in its corresponding field. However, to do that, the user must already be familiar with all the commands, their arguments, object types and properties. Otherwise, traversing giant documentation pages, trying to find a command and examples of its use, will more-often-than-not take longer than just using the normal GUI.

In other words: the command line editor has the potential to dramatically speed up the modelling process, however there is currently no real incentive to get the user to learn it.

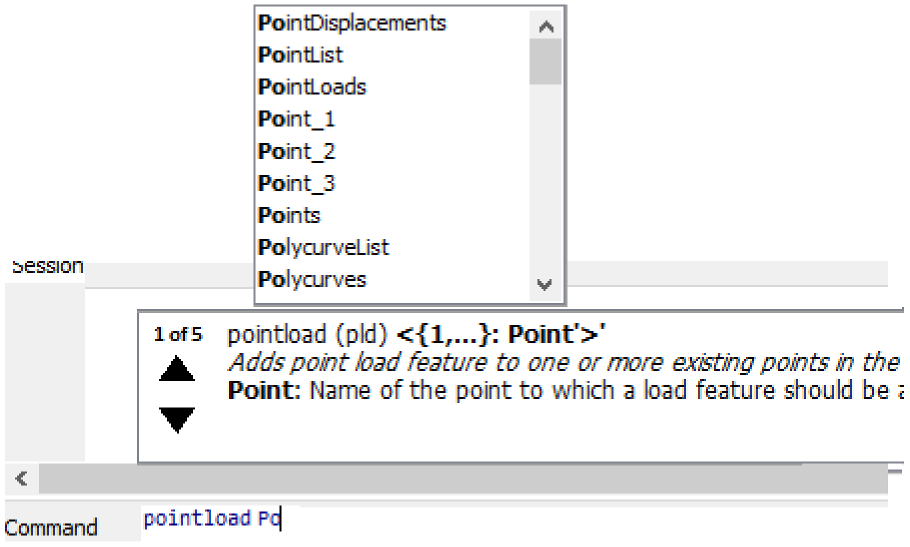
So, the main research questions for the project are:

- How to integrate an autocompletion tool in the current command line editor?
- What are the pieces needed to build an autocompletion tool?
- How to make sure that it is useful and intuitive to the user?

2.3 Resolution

The graduation project aims to resolve this problem and make using the command line editor a more preferred way of working with PLAXIS. Put simply, the autocomplete feature aims to take the information from the documentation pages, combine it with the data from within the project, filter through and provide the user with valuable suggestions. Valuable, meaning, the auto-complete will not suggest commands that cannot be used in the current stage. It will suggest objects of the expected parameter type first. It will also provide the user with the available **Intrinsic Properties** and **User Features** when typing the ‘dot’ character after an object, greatly improving discoverability. Further, the auto-complete will also feature an additional popup (Figure 2.2) displaying quick information for the commands. This information includes:

- Command descriptions
- Command alternatives
- Required arguments
- Highlighting and describing the currently expected parameter



```
pointload (pld) <{1,...}: Point'>'
Adds point load feature to one or more existing points in the
Point: Name of the point to which a load feature should be a
```

```
pointload Po
```

Figure 2.2 - **Mockup visualization of the Quick Info + Autocomplete popups (without type-based filtering of the suggestions)**

2.4 Limitations and requirements

As the end goal of this project is to make the command line editor a more powerful and helpful tool for the user, the major limitations and requirements to keep in mind are:

- The autocomplete must be visually non-intrusive – i.e. not blocking key parts of the main GUI from the user’s view.
- The autocomplete must not block or redirect user input in an unintuitive way – i.e. the user should still be able to use mouse clicks and scroll wheel to modify the viewport of the model while the auto-complete frame is active.
- Performance must be kept in check, so that the end user doesn’t experience any interruptions or hiccups.
- The autocomplete should not rely on the state of the **Plaxis Object Layer** for its functionality – i.e. if commands or objects are changed or new ones are added, the auto-complete should continue to function as expected with no changes required.

3 PROJECT SPECIFICATIONS

The focus in this phase is on doing functional research and writing a specifications document describing the end product. The document aims to explain, in as much detail as possible, how the auto-complete features should look and function, when finished. It includes use cases, sketches, images, implementation strategies, possible limitations and requirements. The goal is to gain feedback from the managers and stakeholders, without diving too deep into technical details, and improve, until it gets accepted, at which point work on implementing the specs can begin.
The investigations done during this stage helped identify most of the major difficulties that will need to be addressed during implementation.

The main questions the document aims to answer are:

1. What information would be useful to know at a glance?
2. How to provide the information visually?
3. When to provide the information?

3.1 Use-cases

Before doing any specific research and without diving deep into technical and advanced autocomplete details, it was a good first step to start with some use-cases:

- *Novice Plaxis user Nova has a strong CAD background and tries to define a model using the command line, the way she does in AutoCAD. She doesn't know the commands but does know what she wants to achieve: put a load on a point. She would expect to be able to type "load" or "point" and Plaxis to offer her "pointload".*
- *Experienced Plaxis user Evan has a large model. He has diligently assigned meaningful names to all his entities, such as LeftWall, FirstFloor, etc. Unfortunately, he has forgotten his naming convention for the foundation plate - he does know (and see in the rendering) that it's a plate. He would like to be able to type "plate" and be offered a list of all Plate type entities to pick from.*
- *Command line guru Clara has a rich model with many objects. The names are however rather long and typing them is very cumbersome, particularly since she needs to do something with a lot of EmbeddedBeam_... entities. She would like the command line to save her time by autocompleting the "beddedbeam_" part when she types "Em", followed by offering her a dropdown to select one of the embedded beams.*

The definition of these use-cases serves well to give a good understanding of what the autocomplete should do and when it should do it. One of the first questions that arose as a response was: *What happens if the user is in the middle of typing sub-object expression "Line_9.EmbeddedBeamRow."?* This made it clear that one of the most important features of the autocomplete is the ability to suggest sub-objects.

3.2 Investigating existing solutions

As the 'specification document' implies, it needs specific, detailed examples and this is where the need to investigate existing solutions kicked in.

It should come as no surprise that the first such solutions worth looking into were Google's search autocomplete and Visual Studio's IntelliSense, as these are the most advanced and widely used autocomplete tools today in general everyday use and code writing, respectively. They are both implemented so seamlessly, that the users hardly ever stop to think of them as additional features, but rather something that is a given, something that just works, while silently improving their experience. There are, however, some major differences in the way they function, and the service that they provide the user.

Google's search autocomplete is designed to work with practically infinite database of suggestions, expanded and updated constantly by the users' inputs. It creates ordered lists of expressions ranked by factors, such as, how often a group of words are used together, the locations from which they are searched for, prior search histories and more. It aims to predict what a user might be searching for as opposed to just listing a number of suggestions (Figure 3.1). This is explained in one of Google's blog posts – 'How Google autocomplete works in Search' (Sullivan, 2018).

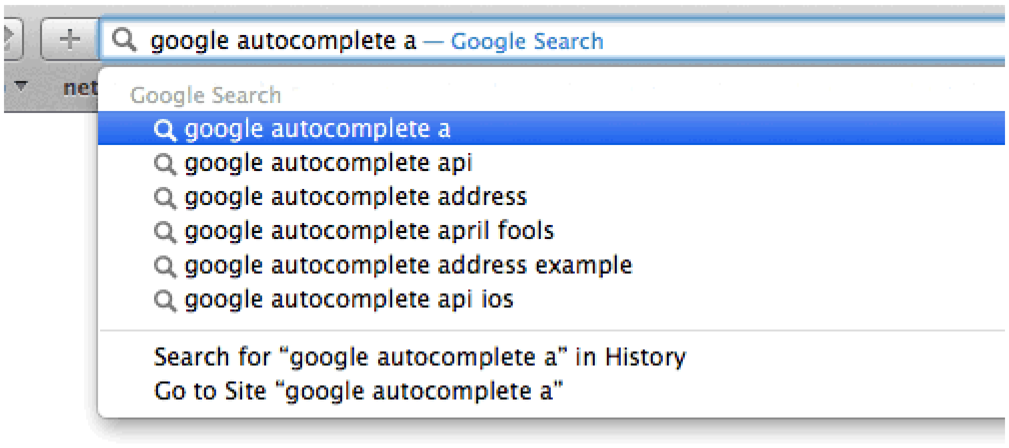
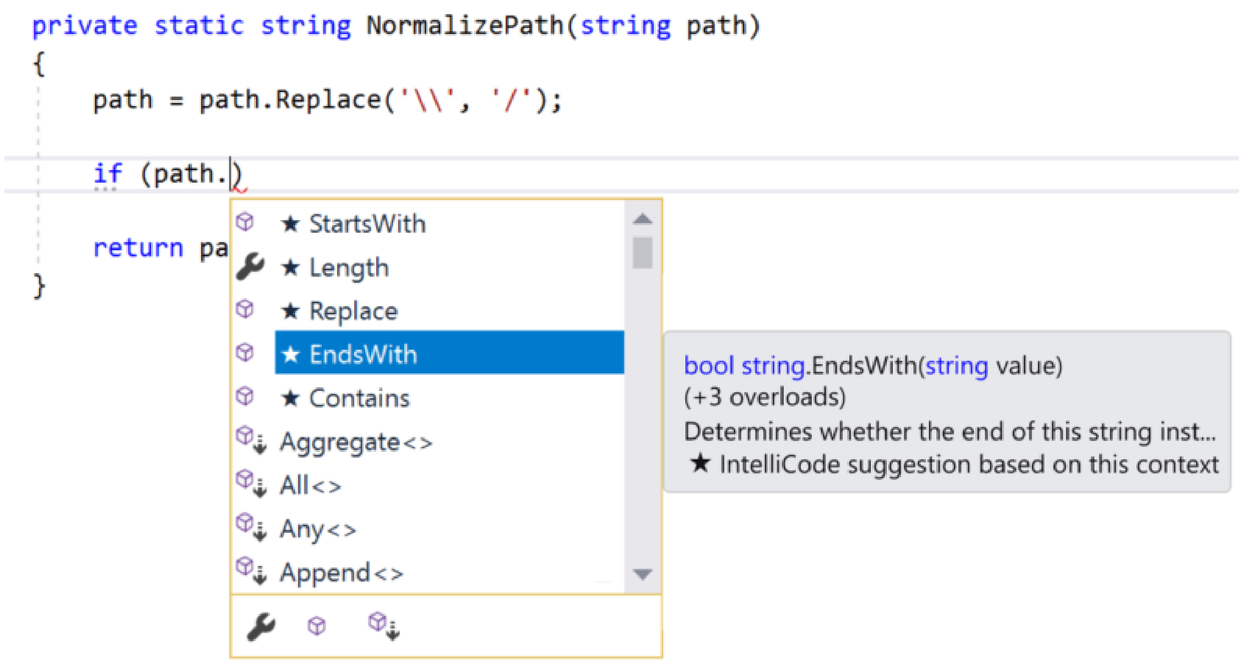

google autocomplete a — Google Search
net
Google Search
google autocomplete a
google autocomplete api
google autocomplete address
google autocomplete april fools
google autocomplete address example
google autocomplete api ios
Search for "google autocomplete a" in History
Go to Site "google autocomplete a"

Figure 3.1 - **Google's search autocomplete predictions**

Visual Studio IntelliSense (Figure 3.2), on the other hand, and all other code autocompletion tools, aim to deliver a list of suggestions to the user, filtered from a larger but limited number of candidates – 'IntelliSense in VisualStudo' (Microsoft, 2018). Although there can be some prediction logic applied in this scenario as well, it was decided to be left out of the scope of the project, as its implementation would require a lot of effort and would yield very little in terms of usefulness to the user in a realistic scenario.



```
private static string NormalizePath(string path)
{
    path = path.Replace('\\', '/');

    if (path.)

    return pa
}
```

bool string.EndsWith(string value)
(+3 overloads)
Determines whether the end of this string inst...
★ IntelliCode suggestion based on this context

Figure 3.2 - **Visual Studio's C# IntelliSense suggestions**

Since PLAXIS was written in Delphi, the Delphi IDE and its autocomplete were also briefly considered, however it quickly fell off the table as it was not robust enough - it required to be force executed with a hotkey[2], it would often cause the IDE to hang for noticeably long periods of time, and it would often not suggest anything useful.

Ultimately, Visual Studio ended up being the biggest inspiration for this project, as for the most part, all the features intended to be added in PLAXIS, were covered by IntelliSense. It served as a goal to aim towards, as well as a benchmark, since throughout the project, any questions and debates that arose could be answered by looking at Visual Studio's autocomplete.

3.3 Concept GUI and user interaction

Since the writing of the specs started prior to any reading or working with the PLAXIS code, it mostly gravitated towards the visual and user interaction aspects.

Concept images were created by editing screenshots of some existing PLAXIS elements.

3.3.1 Suggestions popup

The original idea for the autocomplete popup (Figure 3.3) went through several changes, even after the spec was initially approved.

[2] Code autocompletion tools usually have a trigger hotkey if not activated automatically. Most commonly that hotkey is CTRL Space or some similar combination of a modifier key and Space. This is also true in Visual Studio, even though for the most part it is unnecessary, as it activates automatically whenever it is needed.

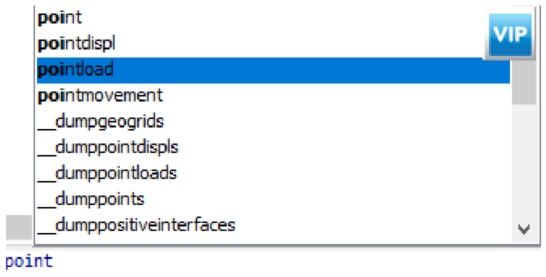


Figure 3.3 – **Original mockup of autocomplete popup for a command with 'point' as input**

Since it was decided that the autocomplete feature would only be available to users with PLAXIS VIP License[3], a VIP icon had to be added to indicate that visually. The problem with that was that the icon was taking valuable real estate in the popup frame. From Figure 3.3 it is noticeable that the icon is hiding the top part of the scroll bar as well as a portion of the top two suggested words. It could be repositioned and made click-through so it doesn't block clicking to select the top few suggestions, however that would still cause some problems in the future, since the popup frame was supposed to be resizable, based on its contents, as well as introduce a second, right edge aligned, column of text displaying the suggested objects' type. After discussing several possible solutions with colleagues and stakeholders it was decided to remove the icon from the popup frame. Instead, introduce it in the new menu item (Figure 3.4) corresponding to the autocomplete.

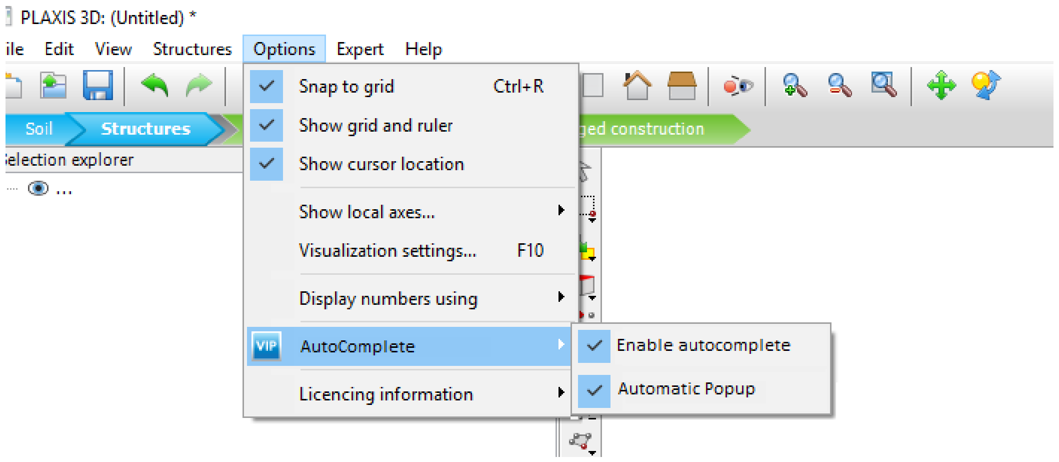


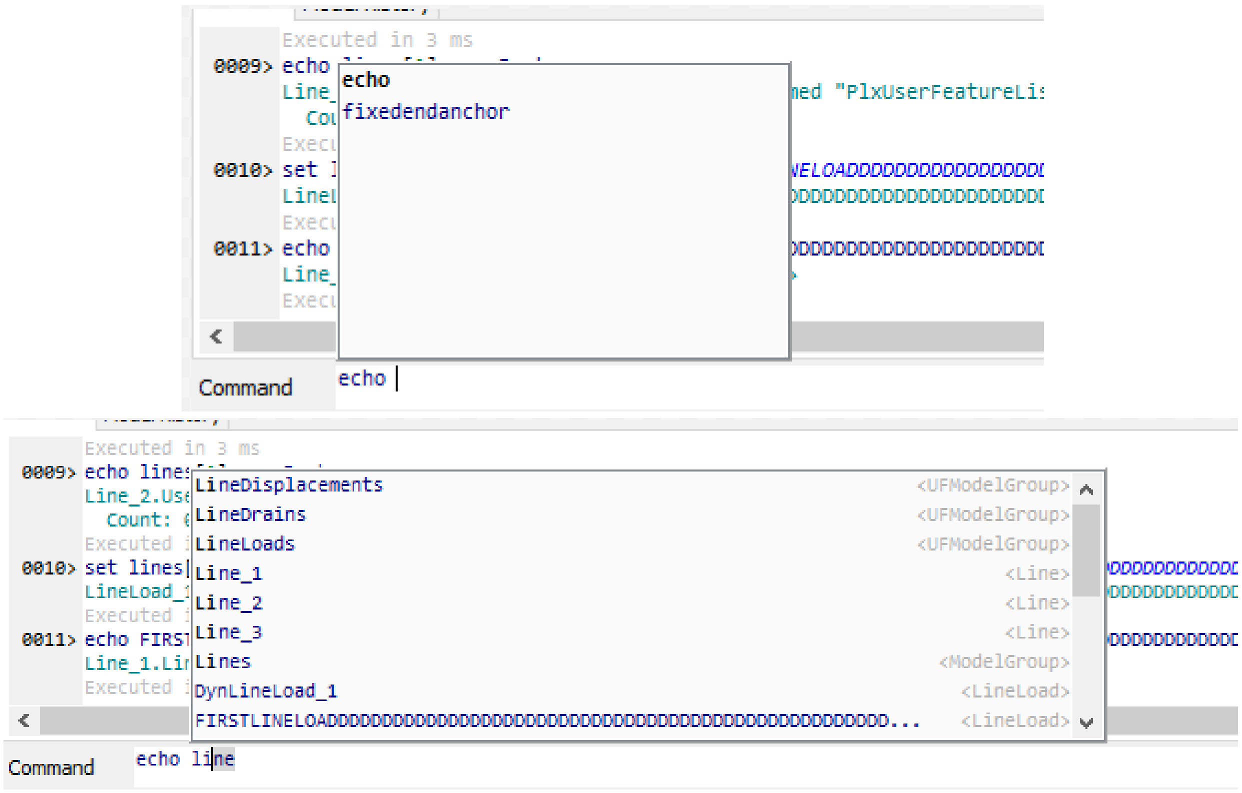
Figure 3.4 – **Original concept for the autocomplete menu items**

As work towards implementing the resizable behavior of the popup began, the need for more adjustments of the spec became apparent. One such change was regarding the font of the text in the autocomplete popup. Originally the default PLAXIS font was used, which is the font seen in all previously shown images, except for the font in the command line editor and history panels. The difference is that the command line uses a monospaced font while the rest of PLAXIS uses a non-monospaced one. A monospaced font, also called a fixed-pitch, fixed-width, or non-proportional

[3] PLAXIS VIP is a subscription model that provides new or extra, easy to learn functionalities, fully compatible with the base product.

font, is a font whose letters and characters each occupy the same amount of horizontal space. (Monospaced font, n.d.)

Switching to the monospaced font used in the command line made correctly calculating the required width and height of the popup frame a lot easier. The minimum and maximum size of the popup frame also went through several iterations. Originally the idea was to have a very dynamic frame that scaled to the exact size required to fit its contents with a limitation only on its width and height maximums, preventing it from exceeding the size the actual command line. However, once a prototype was made and tested it quickly showed its flaws. Because the popup was positioned based on the current caret position, and that position could be very close to the end of the editor, if a very long command was being typed, this left no space for the popup to normally display its contents. That is why it was decided to allow the autocomplete popup to exceed the dimensions of the command line editor and to have constant limitations for both minimum and maximum size. Additionally, the ability the shorten long words with an ellipsis as indicator was added (Figure 3.5).



```
echo
fixedendanchor
```

```
Command   echo |
```

```
LineDisplacements                                                  <UFModelGroup>
LineDrains                                                         <UFModelGroup>
LineLoads                                                          <UFModelGroup>
Line_1                                                                    <Line>
Line_2                                                                    <Line>
Line_3                                                                    <Line>
Lines                                                                <ModelGroup>
DynLineLoad_1                                                         <LineLoad>
FIRSTLINELOADDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD...      <LineLoad>
```

```
Command   echo line
```

Figure 3.5 – **Current minimum(top) and maximum(bottom) size of the popup, with monospaced font and an ellipsis indicating a shortened word**

3.3.2 Inline suggestion

Inline autocompletion is a feature added to the PLAXIS autocomplete spec, but not present in Visual Studio. It gives the command line editor the ability to autofill the input text, when there is one and only one possible way to complete a word. Similar functionality can be observed in Windows 10's Start Menu search (Figure 3.6) and other Windows based forms.

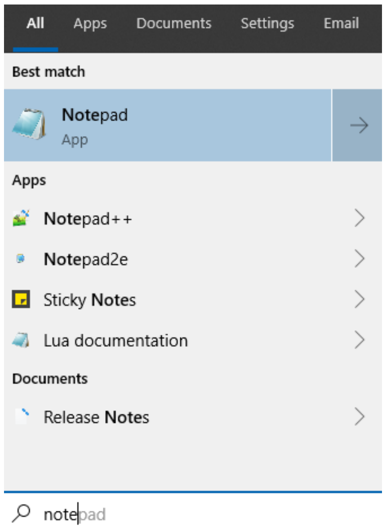
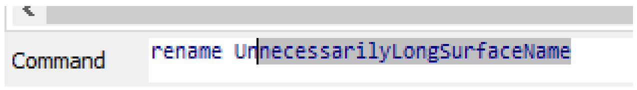


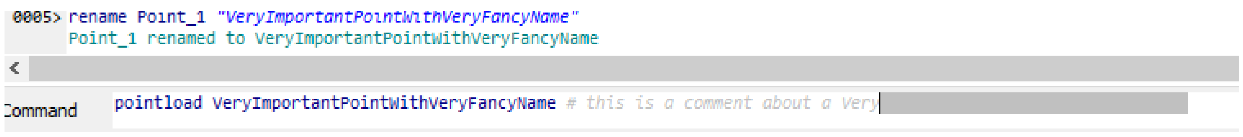
Figure 3.6 - **Example for Inline autocompletion from Win10 Start Menu search**

The PLAXIS command line editor colors the text depending on its type. Some of those types are command and object identifiers (Figure 3.7), numbers, strings (encapsulated in quotes) or comments (Figure 3.8)



```
Command    rename UnnecessarilyLongSurfaceName
```

Figure 3.7 - **Inline autocompletion of object identifier**

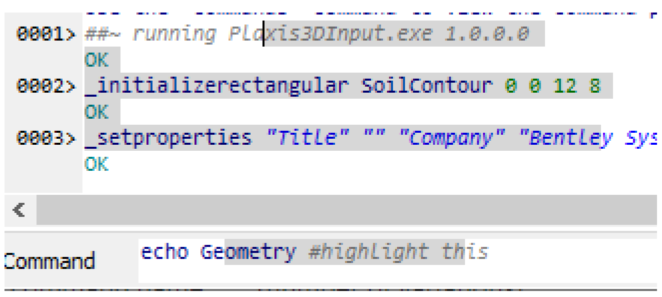


```
0005> rename Point_1 "VeryImportantPointWithVeryFancyName"
      Point_1 renamed to VeryImportantPointWithVeryFancyName

Command    pointload VeryImportantPointWithVeryFancyName # this is a comment about a Very
```

Figure 3.8 - **Inline autocompletion of object identifier in a comment**

Creating concept images for this feature quickly exposed another issue – since the inline suggested text is highlighted using the editor's text selection in the same color as the comment text, it became unreadable. For this problem, it was easier to look for the piece of code responsible for formatting the command line text, create several prototypes with different color schemes and have a discussion with the stakeholders. The solution (Figure 3.9) everyone eventually agreed on was to make the comment color a bit darker and add transparency to the text selection which helped preserve the previous color scheme while making commented text visible under selection.



```
0001> ##~ running Plaxis3DInput.exe 1.0.0.0
OK
0002> _initializerectangular SoilContour 0 0 12 8
OK
0003> _setproperties "Title" "" "Company" "Bentley Sys
OK
```

```
Command     echo Geometry #highlight this
```

Figure 3.9 - **Current color scheme solution**

3.3.3 User Interaction

The original idea for the autocomplete involved an always active inline autocompletion and an only on-demand autocomplete popup. Meaning, that the user would get inline suggestions at all times as long as the autocomplete feature is enabled, but the suggestions popup would only ever be triggered by pressing the CTRL Space hotkey. After discussing it with the company supervisor and later, the stakeholders, it was decided to change the behavior from requiring a hotkey to automatically triggering. This helps achieve several things:

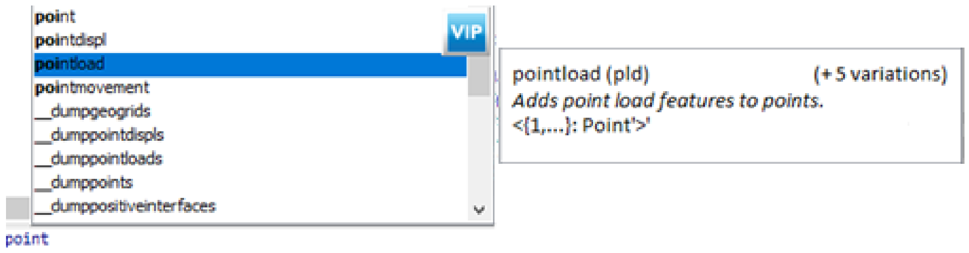
1. Remove the tiresome need for pressing the hotkey for every word the user is interested in.
2. Improve discoverability by automatically suggesting after every input.
3. Having it automatically active forced internal testing within the company, helping with the discovery of crashes and bugs.

3.4 Quick Info and context awareness

The quick info feature is a major reason for the company's interest in the project. Its goal is to provide the user with additional information describing each command and its expected parameters, which is probably the most helpful feature for anyone who is not already a power user. It will further reduce the need to look things up in documentation pages and speed up the learning and work processes. Context awareness can keep track of the current command and the present arguments and use that information to automatically pick the most suitable parameter pattern alternative of the command, as well as help filter out all the unacceptable objects from the autocomplete suggestions.

Since writing of the specs, full context awareness is considered a 'could-have' feature, which means that the Quick Info implementation can be separated in stages.
On a basic level, with no context awareness, the quick info will feature:

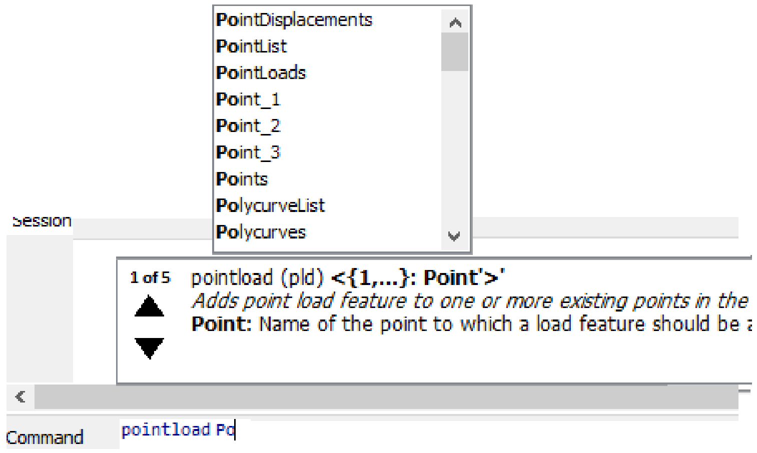
- During command autocompletion (Figure 3.10):
  - Command's name and short abbreviation (if applicable)
  - Number of available alternative parameter patterns
  - Short description of the command
  - List of required parameters for the current command alternative
  - The popup appears to the right of the autocomplete suggestions popup
  - The popup is positioned at the height of the currently selected suggestion



pointload (pld) (+5 variations)
Adds point load features to points.
<{1,...}: Point'>'

Figure 3.10 - Concept of Quick Info popup during command autocompletion

- During target/argument autocompletion (Figure 3.11):
  - Popup positioned directly above the command line editor.
  - Popup is below the autocomplete suggestions popup.
  - Popup will provide the same information as above
  - Popup will also provide the ability to cycle through available alternatives via clickable buttons.



pointload (pld) **<{1,...}: Point'>'**
Adds point load feature to one or more existing points in the
Point: Name of the point to which a load feature should be a

```
pointload Po
```

Figure 3.11 - Concept of Quick Info popup during argument autocompletion

Unfortunately, the time during the graduation period was not enough to reach the point of working on the Quick Info feature and context awareness, however the work done towards designing and describing it in the specifications document still holds its value, and can be picked up and implemented after the graduation project is done.

If context awareness is implemented, depending on its level, additional features can be added to the Quick Info popup, such as:

- Highlighting the currently expected parameter in the commands' pattern list
- Providing description for the currently expected parameter

Additionally, context awareness can be used to further filter out the autocomplete suggestions.

4 TECHNICAL

The majority of technical research was spent with the massive PLAXIS codebase. A simplified diagram of the most important pieces relevant to the autocomplete project can be seen in Figure 4.1.

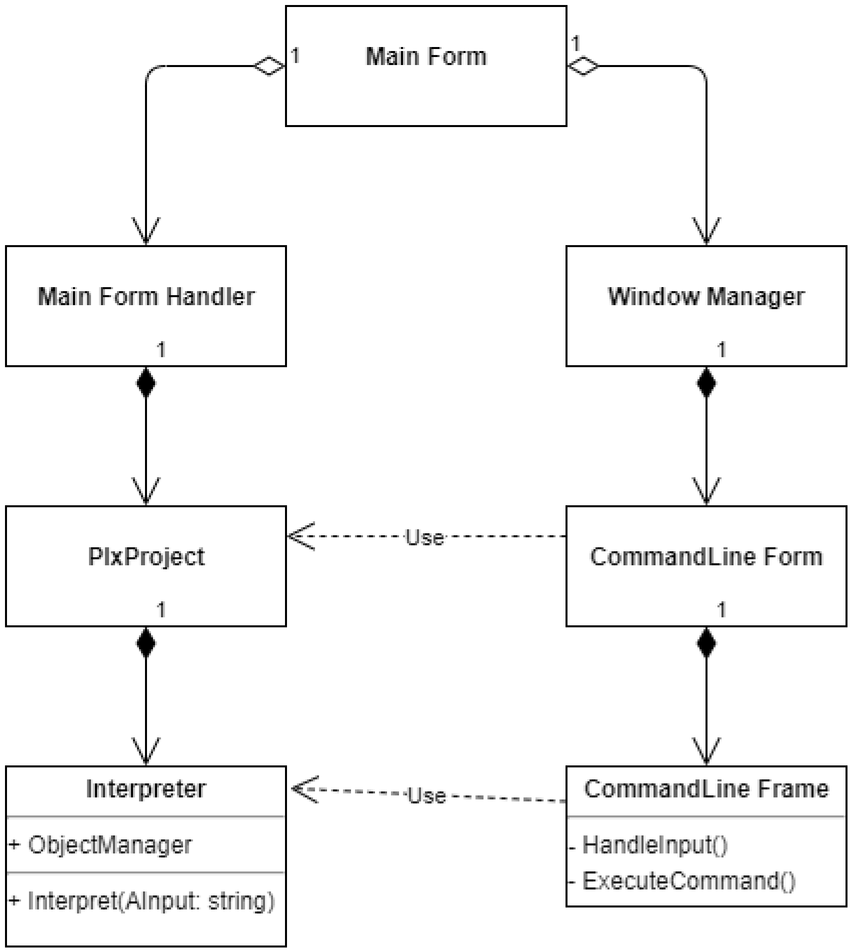


Figure 4.1 - **Simplified structure of a PLAXIS program**

These units are either virtual or completely reused between all the PLAXIS tools, for which the autocomplete project is intended (Input2D, Input3D, Output and SoilTest).

4.1 Plaxis Object Layer

An important part of the PLAXIS code is the so-called PlaxisObjectLayer. As the name suggests, it is responsible for the generation, maintenance and destruction of all objects in the program. Some noteworthy takeaways:

- Nearly everything in a PLAXIS project extends from the PlxObject class.
- Every PlxObject is registered and maintained by an ObjectManager.

- The two noteworthy managers in a PLAXIS project are the so-called Environment and Namespace.
- The Namespace holds the so-called 'Default' objects. These are objects that exist in every project, regardless of its state.
- The Environment holds all the rest of the objects, whether automatically generated helper objects or user created ones.
- The types most relevant to the autocomplete project is the ObservableObject and ObservableObjectList.
- These are mostly objects meant for use and manipulation by the user and are defined by Intrinsic Properties and User Features.
- Each of these objects also keeps track of the commands that can access it through its ObjectAccessor.

A trace of the Plaxis Object Layer can be seen in the Interpreter unit in Figure 4.1 as it contains the project's Object Managers. This is important for the autocomplete project, as it is where the data required for the autocomplete library is stored.

4.2 Commands

A Command in PLAXIS always has the following structure:

<command name> <target object> <...arguments>

Example:
set Point_1.Name 'FirstPoint'

This command will rename the object from **Point_1** to **FirstPoint**. The <target object> in this example is the **Name** of **Point**, which is an Intrinsic Property of the point object. More specifically, it is a Text Intrinsic Property and as such it can be implicitly cast to a string or assigned as one:

Example:
set Point_1.Name 'Point_2'
set Point_1.Name Point_2.Name

The above two commands would yield the same result of renaming **Point_1** to **Point_2**, except in the first command **Point_1** is assigned a string explicitly, while in the second **Point_1** is assigned the value of **Point_2**'s **Name** intrinsic text property.

Example:
set Point_1.x 5
set Point_1.x Point_2.y

The same rules apply for other types of intrinsic properties, such as numbers, as in the above example.

There is, however, one exception, when it comes to the command syntax, which can complicate things a bit. While internally a command must have a target object and an array of parameters, the user is allowed to skip the target object, if it is a Default Object (registered in the project's Namespace). So, there are two acceptable command syntaxes:

<command> <target object> <...arguments>
<command> <...arguments>

This means that the interpreter needs to identify whether the first argument is a valid target object or not, in which case it needs to look for a suitable default object from the Namespace, that matches the provided argument.

Example:
point 1 2 3 – creates a point at x=1, y=2, z=3. The target object is omitted.
point Geometry 1 2 3 – same result as above, but the default target object is not omitted.

This syntax was chosen in favor of

<object>.<command> <...arguments>

which is semantically identical.

It is also worth to note that a command can accept an empty array of arguments, so syntactically a command with no target or arguments provided can still be valid.

Example:
echo Point_1 – prints information about the target object **Point_1,** without arguments.
echo – prints information about the Project object, which is a default object representing the PLAXIS Project.

Due to the complexity of the commands, their syntax and argument patterns it was decided to move the context related parameter type filtering to the back of the backlog. Instead, for the first version of the autocomplete feature, the suggestions include all objects, available in the managers.

5 AUTOCOMPLETE CORE

With a good understanding of the Plaxis Object Layer and the Command details, work on the core autocomplete features could begin. This is the most important and difficult part of the project, from both design and implementation perspectives. It involves:

- Building an ad-hoc library – a collection of context relative data, structured in an easily queryable manner.
- Building an interpreter – a text analyzing tool that splits the command line input into smaller identifiable expressions and tokens.
- Implementing search strategies – the logic to use when filtering and sorting through the data provided by the ad-hoc library, using the interpreter results.

- GUI Visualization – adding menu items, creating popup frames and modifying the command line editor to display hints of the most suitable suggestion.

The structure of the new autocomplete features can be seen in Figure 5.1.

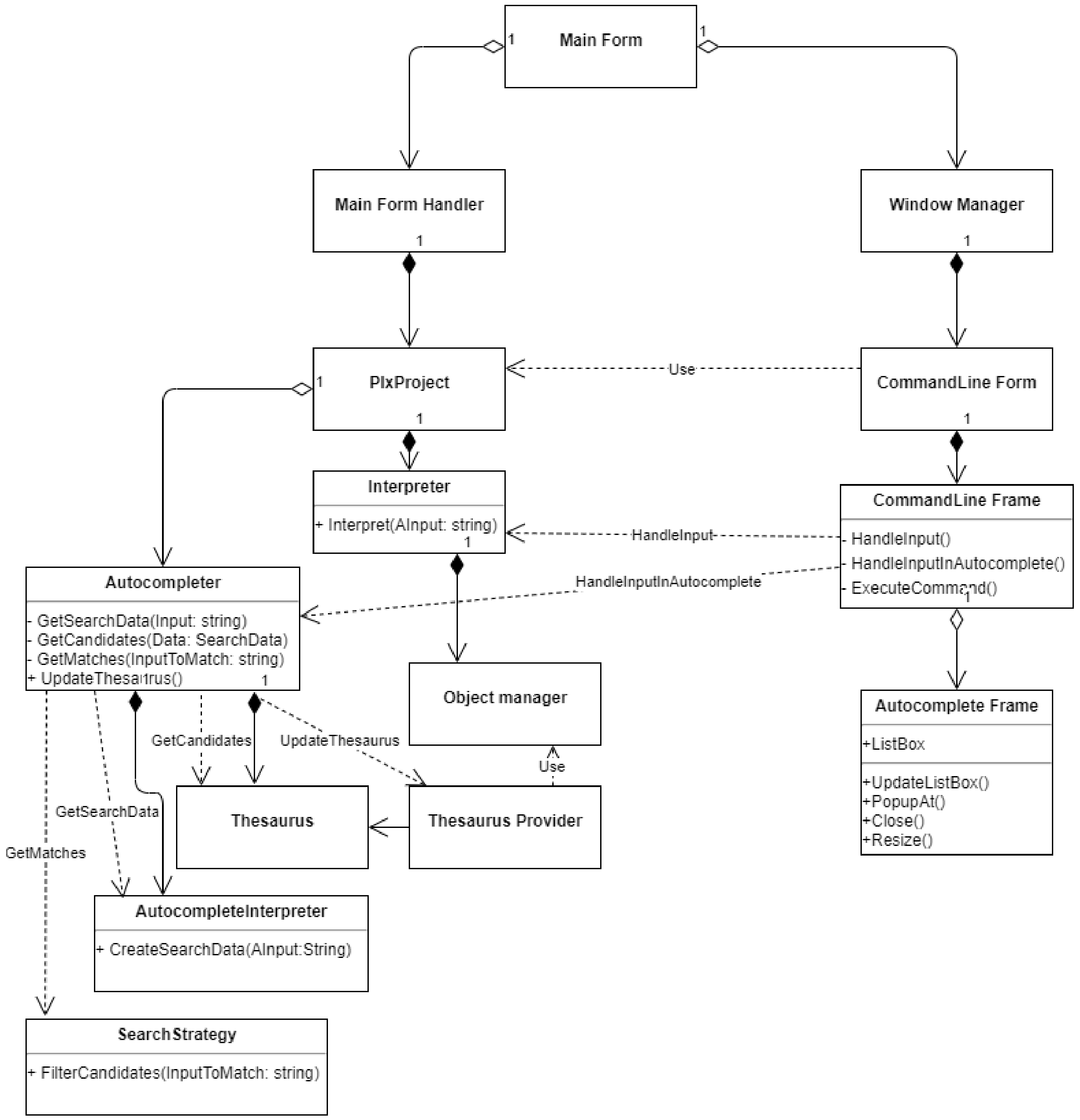


Figure 5.1 - **Project structure with Autocomplete**

5.1 Ad-hoc library (Thesaurus)

As mentioned, the job of the library is to collect all relative information from the project and organize it in a way that is easily accessible. It may be hard to predict what command, target or arguments the user will type in the editor; however, one thing is for sure: it will always start with a

command name. Based on that fact alone, regardless of what structure ends up being used for the library, a collection of all available commands is going to be a major part of it. Acquiring the commands is an interesting task in itself. They are not grouped together somewhere in the project. Instead each command is bound to its target object via an object accessor. In order to collect the commands, the library needs to iterate through every single object, default or not, and ask their accessors for supported commands. It is easy to imagine that this process can be quite slow, depending on the size of the project, so special care needs to be taken to ensure that the library updates as infrequently as possible.
As a solution for the initial prototyping, the library will be updated only on project startup and on every successful command execution, as commands often lead to change in state. Later in the development, more research can be done towards partial updates – instead of invalidating and recreating the library on every update, keep track of the changes in the project's state and only update the modified parts.

The initial implementation of the library will mostly make use of dictionaries and hash sets and it will store references to the commands and objects. While performance might not be the greatest, it will be enough to get up and running and most importantly inexpensive with regards to development time. It should look something like this:

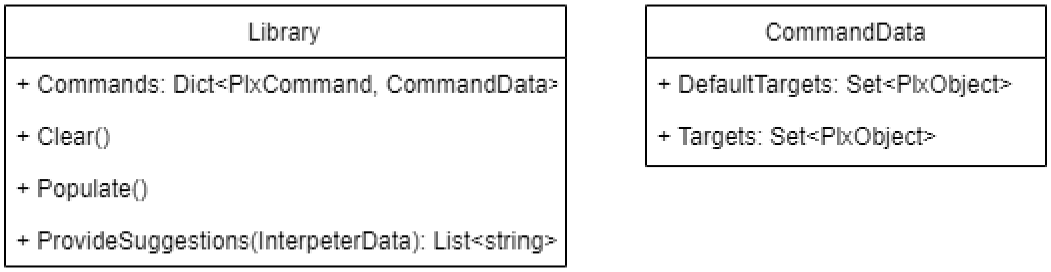

Library
+ Commands: Dict<PlxCommand, CommandData>
+ Clear()
+ Populate()
+ ProvideSuggestions(InterpeterData): List<string>
CommandData
+ DefaultTargets: Set<PlxObject>
+ Targets: Set<PlxObject>

Figure 5.2 - **Simple library setup**

This setup will provide easy access to the available commands and the target objects that support the command, which should be enough for the most basic autocomplete implementation. In the future when work is being done towards implementing context awareness, the CommandData can be extended with a collection of objects matching each parameter pattern.

5.2 Autocomplete Interpreter

The job of the autocomplete interpreter is to take the text from the command line editor, break it into tokens and generate a query object to be passed as an argument for the library's *GetCandidates* method.
Two major actors will be at play here – a tokenizer and a token expression tool.

5.2.1 Tokenizer

The first step after taking the input text from the editor is to split it into individual chunks – tokens. The tokenizer goes through the text, character by character, and creates a token out of every identified word or operator. In this context a 'word' means any collection of characters surrounded by whitespace or operators. Operators are special characters, such as: dot, comma, parenthesis, square brackets, etc. It has no sense of whether a word is valid or not.

Example:
point 1 2 3 – results in 4 tokens: **point**, **1**, **2**, **3**
echo Points[0].x – results in 7 tokens: **echo**, **Points**, **[**, **0**, **]**, ***dot***, **x**

5.2.2 Token Expression Tool

Once the tokenizer's work is done, it provides the data to the Token Expression Tool. The tokens are then grouped, and logical expressions are created. There are two major expression types: Single and Multiple. A single expression contains only one token, while a multiple expression contains more than one. An '*Active token'* or '*Active Expression'* is recognized by the current position of the text caret.

Example:
echo Points[0].x – it starts off as one big multiple expression, containing all tokens.

Every new interpretation starts with a generic multiple expression. It then gets broken down into smaller and smaller expressions recursively. After the first recursion the example above gets split into two expressions: **echo** and **Points[0].x**. The first one is clearly a single expression as it only contains one token: **echo**. The second one is once again a multiple expression, more specifically, a Classifier separated multiple expression. That is because the '*dot'* operator token is called a classifier, which is used to access sub-objects, otherwise known as Intrinsic Properties and User Features. Further, the classifier separated expression **Points [0].x** is split into two new expressions: **Points[0]** and **x**. The **x** is easy – single expression. The other one is an array expression, which is also a special type of multiple expression. It gets further broken into a single expression **Points** and an array indexer expression – **[0]**. Depending on where the caret is currently position, the interpreter now has enough information to create a query object.

Example:
echo Points[0].x| - the caret is right after the **x**. which means that the **Active Expression** is the classifier separated multiple expression **Points[0].x**.

This ultimately translates to the user requesting sub-objects of the first object in the **Points** list, that start with the character **x**. The Interpreter creates 'Sub-object query', sets its input to 'x' and provides it to the library. The first object in the Points list is a point object. The only sub-object a point has that starts with 'x' is its coordinate, so that is the only item in the resulting list of suggestions.

If the caret was positioned right after the '*dot'* classifier, the process would be repeated almost identically except the results would not be filtered by the character **x**, meaning all available sub-objects would be suggested to the user.

Of course, expression can get a lot more complicated than the given example, however if implemented correctly, the recursive interpretation of complex expressions should always end up in a similar fashion.

This implementation should be easily extensible to include required parameter type in the query object, once context awareness is implemented.

This was expected be the most difficult part to implement, as the recursive interpretation can get difficult to follow and debug, however the rest of the autocomplete entirely depends on the interpreter`s proper functionality, so spending extra time on it is acceptable and even encouraged.

5.3 Search Strategies

As described in the spec, the autocomplete supports two search strategies: a basic, exact prefix search and a fuzzy search. This implementation of the fuzzy search differs from the one, typically found online (Tregoat, 2018), as it is a far simplified version – it searches for words that contain the input characters in their corresponding other, at any position. It functions very similarly to an exact prefix search, with a small but substantial difference:

- Exact prefix search compares the characters of an input and a candidate strings at a given index until it finds a mismatch (it's not a match) or runs out of characters to compare (it's a match).
- Our fuzzy search does the same, except it doesn't stop when it finds mismatching characters. It continues until it runs out of characters in the candidate string (it's not a match) or it finds all the input's characters inside the candidate, regardless of their index (it's a match).

This is the place where performance was anticipated to be an issue from the very beginning. The best and worst-case scenarios for a match of the two search strategies' complexities is practically identical, because they can both happen to find all the input characters in the beginning of a candidate, or have a matching input with the same length as the candidate, performing the same amount of comparisons to get a positive result. This made it questionable whether it is of any use to keep exact prefix as a separate search strategy. However, in normal use, the best-case scenarios of the exact prefix search occur far more often than the worst, and vice versa for the fuzzy search. For example, an exact prefix search can exit after the first comparison returns mismatching characters, while fuzzy still has to go through the rest of the candidate's characters. Not only that, but prefix search is also useful for generating inline suggestions.

5.4 Current Performance

Currently the search strategies are implemented in their most basic form. Although no specific performance improvements have been implemented, the autocomplete feature in its current state behaves normally in realistic projects.

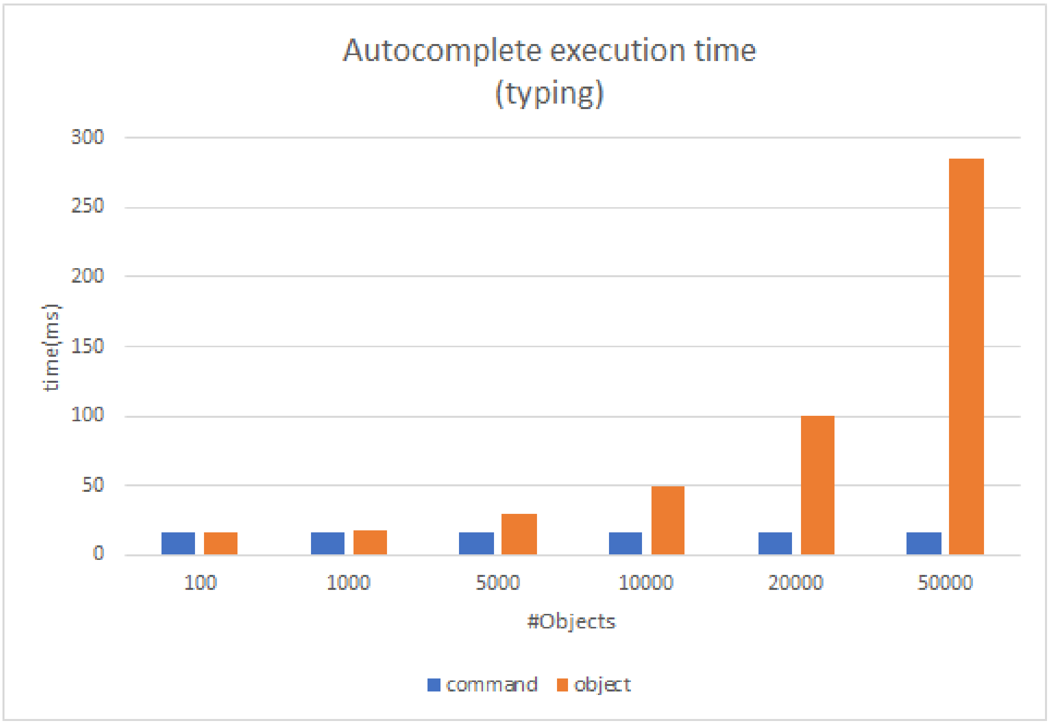


Figure 5.3 - **Autocomplete execution time.**

According to a number of real user projects, provided by the Customer Service colleagues, the number of objects is, on average, in the range of 5,000 – 15,000, depending on the mode. Performance issues become apparent in some artificially created worst-case scenarios, at execution times longer than 150ms (Figure 5.3).

Several possible improvements are being discussed:

- Caching results after every input, potentially reducing the pool of candidates.
- Reducing the number of objects registered as candidates in the autocomplete thesaurus.
- Keeping the UI responsive by introducing asynchronous or parallel autocomplete processing.

Each of these methods has its own pros and cons. Caching can be a useful approach in certain situation but it is far from perfect. Aside from the extra memory allocation, the cache would need to be invalidated after every command execution and at the beginning of every new word. This means that it will yield no positive results at the start of a word, slowly improving with each input. In a worse-case scenario, where the majority of objects have very similar names, distinguished only by last few digits, caching will also have little to no effect. Nevertheless, it is better to have little improvement over none.

Another area with room for improvement is the autocomplete library. As explained in the Technical section, all objects relevant for the autocomplete are of the Observable type. However, not all Observable objects are useful. There are many 'Helper' objects automatically generated in order for other, usually more complex object types, to function properly. These objects are of no interest to the users. The problem is that there is no simple way of differentiating the useful from the useless objects programmatically. After investigating the severity of this issue, the results showed about a 50% increase in thesaurus objects while in a blue mode (Soil/Structure) and a nearly 300% increase in objects while in a green mode (Mesh/Flow/Staged Construction), due to a greater amount of helper objects being created for the complex mesh generation. These numbers clearly show that there is a lot to be gained by more carefully cherry-picking only the needed objects for the autocomplete library and more investigation towards that goal is a worthwhile effort.

Even though adding asynchronous or multithreaded behavior to the autocomplete process might be the most interesting approach, after long discussions, it was decided that it should be left as the final step of chasing smooth autocomplete performance, due to the complexities that it can bring to the table, as well as the fact that it shouldn't be used as a substitute for otherwise unoptimized code.

Another noticeable hot spot of the autocomplete performance is the process of updating the Thesaurus (Figure 5.4). Once again, the execution time starts becoming noticeably slow in the projects with over 20,000 objects. At a first glance, having half a second delay every time a command is executed, should not be a deal breaker, however there is one hidden issue. There are several ways of executing commands. Other than typing it in the command line editor or clicking a GUI item, there is also a remote scripting API as well as a command executor window, both of which allow for large numbers of commands to be rapidly executed. Since the autocomplete Thesaurus is updated using an *OnCommandExecuted* callback, this is a place where having the autocomplete enabled can become quite problematic. However, this potential issue was only brought to light by a stakeholder very recently, and no investigation has been done to confirm or deny its severity or explore for possible solutions.

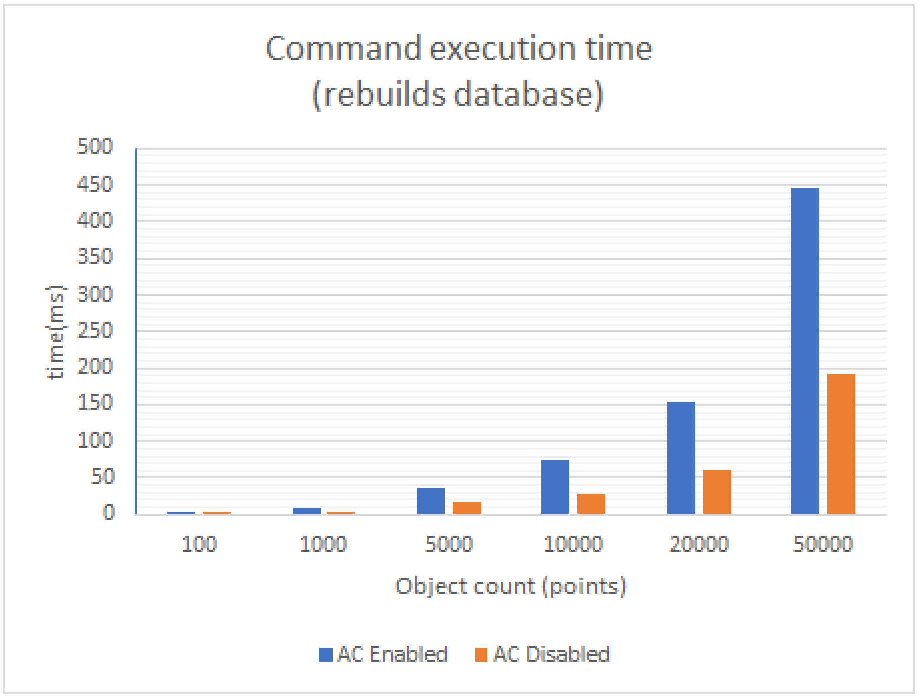


Figure 5.4 - **Command execution time**

Interestingly, while doing the performance investigations, a strange phenomenon occurred in some of the test projects. Even though these projects had fewer objects than the rest, results showed worse performance. This led to a deeper investigation not only logging how many objects we generate from within the project, but rather, how many objects of each specific type are registered from the object managers into the Thesaurus. The results were quite substantial. As it turned out for nearly every object that is generated from within the Plaxis project, several other 'helper' objects were being registered internally. Depending on the mode in which the project is in, these additional objects could increase the total number of registered objects to fifteen times the expected count. Not only that, but after discussing it with several colleagues, we found no reason for these additional objects to be suggested to the user at all. So, needless to say, filtering these objects from the autocomplete database was first order of business when it came to improving performance.

Implementing the filtering itself was a straightforward task, simply introduce a property in the base PlxObject class to determine whether the object should be included in the autocomplete suggestions. More difficult is finding all the object types that should and should not be included – a task that will be handled over a longer testing period. Fortunately, the initial investigations were already enough to gain some substantial improvements in performance, due to filtering many of the unwanted objects.

6 CONDITIONS OF SATISFACTION

The main criteria for satisfaction are set by the specifications document.

All the following features are laid out in detail and accompanied by example images (where applicable) in the specs.

6.1 Must haves

- Produce a specifications document [**done**] [**In winter update release**]
- Implement a basic autocomplete capable of suggesting:
  - Commands [**done**] [**In winter update release**]
  - Objects [**done**] [**In winter update release**]
  - Sub-objects (IPs and UFs) [**done**] [**In winter update release**]
- Menu items:
  - Toggle autocomplete on/off [**done**] [**In winter update release**]
  - Persist the state of the toggle between restarts [**done**] [**In winter update release**]
- Intuitive Mouse and Keyboard controls [**done**] [**In winter update release**]
- Performance allowing for smooth typing experience [**plausible**] [**In winter update release**]
- Unit Tests [**work in progress**]
- No crashes [**done**] [**In winter update release**]

6.2 Should-haves

- Basic Quick Info providing:
  - Command description [**todo**]
  - Number of alternatives [**todo**]
  - List of required parameters [**todo**]
- Resizable autocomplete suggestions popup frame:
  - Tries to fit the name of the suggested object and its type. [**done**] [**In winter update release**]
  - Maintains its size within the min/max dimensions described in the specs [**done**] [**In winter update release**]
  - Maintains its position as described in the specs [**done**] [**In winter update release**]
- Resizable Quick Info popup frame:
  - Tries to fit the command name(s), alternatives count, parameter list and command description [**todo**]
  - Maintains its size within the min/max dimensions described in the specs [**todo**]
  - Maintains its position as described in the specs [**todo**]

6.3 Could-haves

- Context awareness:
  - Quick Info highlights the currently expected parameter [**todo**]

- Quick Info provides description for the currently expected parameter [**todo**]
- AutoComplete prioritizes objects of the expected parameter type. [**todo**]

6.4 Won’t have

- Autocomplete predictions:
    - Indexing successful command executions
    - Attempting to predict words before any user input
    - Attempting to predict whole expressions

7 CONCLUSION

Looking back on the past 5 months of working on developing the autocomplete tool for Plaxis I have undoubtedly gained an enormous amount of knowledge and experience both personally and professionally.
Nearly everything done during the project was a first-time experience.

Starting from the programming language and IDE, feeling completely lost and spending countless of hours trying to tackle the simplest tasks, to now being fluent in Delphi and working smoothly.

Working on a project with a massive code base, millions of lines of code, over thirty years in the making and attempting to integrate my own creation in it, has been both overwhelming and exciting.

Learning the ins and outs of the autocomplete system also proved to be a much bigger challenge than initially anticipated. In all honesty, in the beginning of the project my thoughts were that it would be done before the five months were out, with all the quick info and context awareness features as well. As my company coach kept saying throughout the project – things are always much more difficult than they initially seem, when it comes to software.

Fortunately, the autocomplete feature is in a good state in both functional and user-friendly aspects. The autocomplete core (Thesaurus, Interpreter, Search Strategies) is implemented in a robust and easily extensible manner. The autocomplete feature achieves the goals of improving the command line usability and discoverability, by providing valid, valuable suggestions, while maintaining an intuitive, low profile with minimal overhead when it comes to user input. Most importantly, the new tool is very dynamic, in the sense that, if changes were made to the PlaxisObjectLayer (objects and commands), it would continue to work as intended, taking the changes into account. While improvements can always be done, especially when it comes to performance, this project proved to be a good start up for the company, that will continue to be developed after the graduation period. It is also worth to note that the autocomplete tool was officially released in the latest update of the PLAXIS software as a technical preview, in order to give real users the opportunity to test its robustness and provide feedback.

8 REFERENCES

Microsoft. (2018). *IntelliSense in Visual Studio*. Retrieved from https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2019

Monospaced font. (n.d.). Retrieved 12 22, 2019, from Wikipedia: The Free Encyclopedia: http://en.wikipedia.org/wiki/Monospaced_font

Sullivan, D. (2018, 4 20). *How Google autocomplete works in search*. Retrieved from Google: https://www.blog.google/products/search/how-google-autocomplete-works-search/

Tregoat, J. (2018, 1 9). *An Introduction to Fuzzy String Matching*. Retrieved from medium.com: https://medium.com/@julientregoat/an-introduction-to-fuzzy-string-matching-178805cca2ab

Wikipedia. (2019, 10 3). *Finite element method*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Finite_element_method

APPENDIX

Product demo: https://youtu.be/HzzFyy9OclY