CONTENTS

LIST OF USED ABBREVIATIONS

API - Application Programming Interface

AWS - Amazon Web Services

CI - Continuous Integration

CLSID - Class Identifier

COM - Component Object Model

CSS - Cascading Style Sheets

DTL - Django Template Language

HTML - Hyper Text Markup Language

HTTP - Hypertext Transfer Protocol

IP - Internet Protocol

IPC - Inter Process Communication

JSON - JavaScript Object Notation

MTA - Multi-threaded apartment

NPTS - Nao Presentation Tool Suite

PPTX - PowerPoint Open XML

SDK - Software Development Kit

SSH - Secure Shell

STA - Single-threaded apartment

UX - User Experience

VCS - Version Control System

XML - Extensible Markup Language

XSD - XML Schema Definition

INTRODUCTION

Robotics is a rapidly developing branch of engineering. In order to demonstrate its capabilities, a French robotics company developed Nao. A programmable autonomous humanoid robot. It was created for the purposes of research, education and entertainment. As an extension of this idea, a tool suite was developed with which one can make Nao do presentations.

Nao Presentation Tool Suite (NPTS) consists of mainly 3 modules:

- Nao presentation controller program;

- Nao GUI shell;

- Survey conductor.

Nao presentation controller program is mainly responsible for opening a presentation, making the robot Nao read out the text for each slide and jumping to the next slide. In addition, it is capable of playing embedded audio/video and to hold a survey by interfacing with the Survey conductor. The audible for text each slide is provided in the notes section.

Nao GUI shell is meant to simplify the users' workflow by providing an intuitive user experience through which one may interact with Nao, configure him and launch the presentation controller program.

Survey conductor is a bare web application that can be used to hold surveys. User is meant to provide a PIN through which they are forwarded to a multiple choice survey. Said survey is created and started by the presentation controller program. The interaction between the survey conductor and presentation controller happens through a simple web API.

There exists an available market alternative, known as "NAO Presenter application" which in many ways fulfills the same purpose, aside from some minor differences in the technical implementations and intended use cases. Additionally, "NAO Presenter application" was created as a business-to-business application solution and most definitely not for an individual persons use and according to its creators it was designed for organizations that receive the public, e.g.

- Museums;

- Shopping centers;

- Civic offices;

- Company reception areas.

In all other parameters, there seems to be little difference between "NAO Presenter application" and NPTS.

The development of NPTS was a cooperative effort undertaken by 2 individual students. Work was evenly distributed between them. A concrete work breakdown structure will be provided later on. Since the development phase was a team effort, many measures were put in place in order to ensure a smooth workflow, such as a VCS, Kanban Board, Continuous integration pipeline, automated code linting and quality inspection.

1. DEVELOPMENT OBJECTIVES

From the very beginning, a number of objectives and criteria were established for the end product, so that it will have all the required features to be realistically useful and usable.

1.1. Nao presentation controller

Upon starting the Nao presentation controller, the program will be able to accept the following input arguments:

- Path to a PPTX file;

- IP address for a Nao robot;

- flag to indicate internet usage (internet usage is required in case a survey is held).

The program launches PowerPoint and starts a slide show for the inputted PPTX file. If the program is unable to find the PPTX file or to connect to Nao, an error is raised and the program exits. In case no errors are raised, each slide is sequentially presented to the audience. The text that is read out by Nao is specified in the notes for each slide. In addition to audible words, in the notes there may also be commands that are executed during the reading. These commands may be to:

- Execute an animation;

- Raise/lower voice volume;

- Pause the speech for a certain amount of seconds;

- Trigger the next slide animation;

- Set a reading mode;

- Start playing the embedded media object on the current slide;

- Create/start survey.

A somewhat unified ad-hoc syntax was developed for these commands. The commands will visually look similar to XML tags. However, this visual resemblance is completely superficial. There tags for the most part, are not used to describe or define elements withing the text but rather to signify commands to execute during the speech. Their visual resemblance to XML is mainly used to signify to the reader/writer of the notes that the commands are not part of the speech and are inaudible.

The program goes forward slide by slide, until it reaches the end and then closes the slide show and PowerPoint along with it.

1.2. Nao GUI shell

Nao GUI shell is meant to give a pleasant user experience and enable the user to more comfortable interact with Nao to perform, for example, administrative or maintenance tasks. In addition, it is meant to supplement the presentation controller by being able to validation a presentation and it's notes and to help a user design surveys for the presentations without burdening the user with any XML-like text blobs.

Since this is an application with a graphical interface, some sort of GUI toolkit is necessary. PyQt framework will be used because:

- Comes with a litany of complete and ready UI components;

- Is well-supported and has many available learning resources;

- Is easy to learn and use;

- Enables writing a smoother codebase.

In addition, some SSH related libraries will be used to help establish a secure and stable connection to the Nao robot.

1.3. Survey conductor

The survey conductor is a web application that is meant to be publicly accessible to members of the audience. During the presentation, a survey may be created and started by the presentation conductor. Upon creation, all the questions and answer choices must be provided along with a time limit for each question and also a PIN code must be set by which the audience members will be able to access the survey.

The presentation controller may send a request to start it. The survey conductor then autonomously holds the survey by waiting for the time limit for the question to run out and switching to the next one. After the last question expires the survey is closed and no more interaction is possible with it.

No data processing or aggregation will be conducted. All of the created surveys are essentially stored in memory and not saved or persisted to any storage medium.

1.4. Work breakdown structure

The development of the Nao Presentation Tool Suite was a cooperative endeavor and as such, responsibilities and tasks had to be delegated. Development of the survey conductor and

presentation controller was given to one and the rest to the other. A work breakdown structure diagram was conceived to properly visualize this.
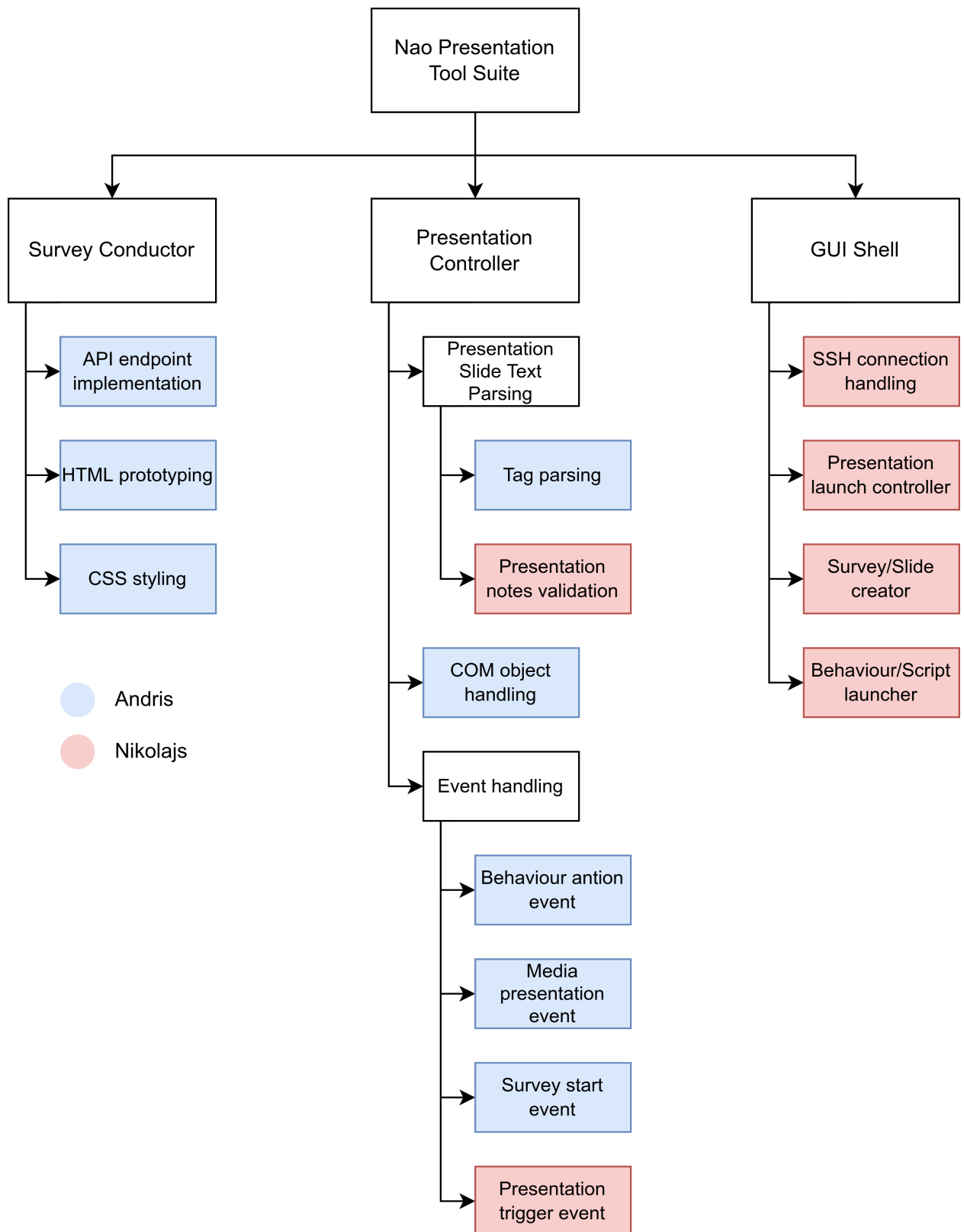
1.1. Figure. **Work breakdown structure for the development**

2. DESIGN ENGINEERING

In order to successfully develop the tool suite as per all the above stated requirements and criteria, many decision and design choices had to be made in advance.

2.1. Nao robot

Nao is an autonomous, humanoid robot that can be programmed with the Qi framework - an easy to use, highly abstracted SDK that enables a novice programmer to write advanced software without getting involved with any low-level hardware concepts like switch debouncing and to quickly and simple utilize all of its motors and sensors, of which it has many.
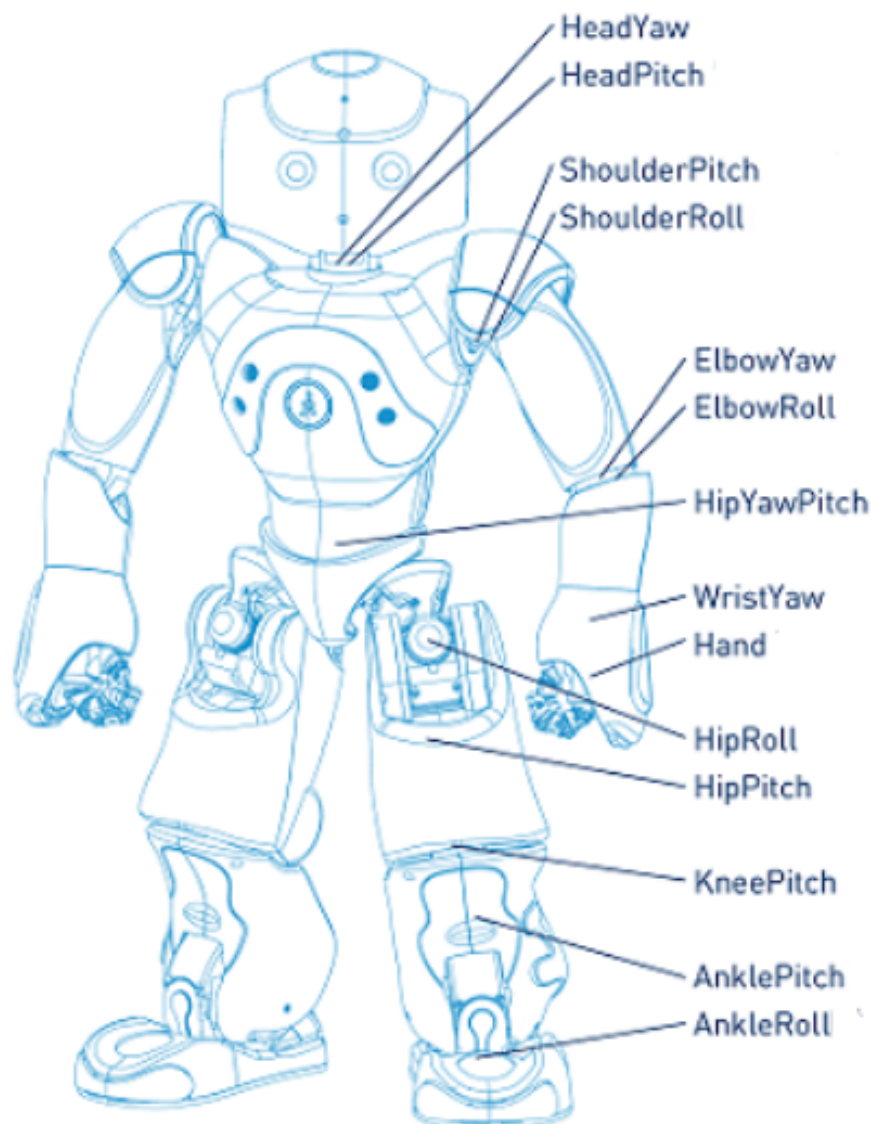
2.1. Figure. **Locations of Nao's motors**

Nao runs a custom Linux distribution. One can connect to it through SSH to debug issues, conduct certain administrative tasks or as a means to study it more thoroughly. An Apache HTTP server instance is also running on Nao, can be reached by using a web browser and used to perform some simple configurations:

- Setting voice;

- Raising/lowering speech volume;

- Configuring Wi-Fi networks.

In the context of the Nao presentation tool suite, Nao is used mainly as a speech synthesizer. The main advantage that Nao has over other text to speech services is its ability to supplement speech with movements and gestures making it much more expressive and effective as a presenter. The following modules of the Qi framework were used during development:

- ALTextToSpeech - used to make Nao talk using the text-to-speech engine.

- ALAnimatedSpeech - makes Nao speak in an expressive manner.

- ALAutonomousLife - module that is used to configure Nao's autonomous behavior.

- ALSpeechRecognition - module used to configure Nao's speech recognitions patterns.

- ALTouch - module used to read state of Nao's sensors.

- ALMemory - allows acceess to Nao's centralized memory, which is used to store Nao's hardware configuration and other vital information.

- ALRobotPosture - module allows to make Nao go to different predefined postures.

- ALBehaviorManager - module to manage behaviors. A behavior being a set of instructions that can be installed on Nao.

2.2. COM objects

In order facilitate the interaction between the presentation controller program and PowerPoint it was decided that COM objects will be used, as it appears to be the most reliable and secure method of IPC for when it comes to Microsoft Office Applications (Stevewhims n.d.). If PowerPoint is installed on a system, it registers itself, and it's CLSID into the system's registry. It is then possible to create a new object instance in a language such as Python and call upon this object various methods to orchestrate a slide show. It should be noted that every office application technically

has its own object model, but due to the efforts of Microsoft the object model has become fairly similar between all the applications, making the learning curve much lower.
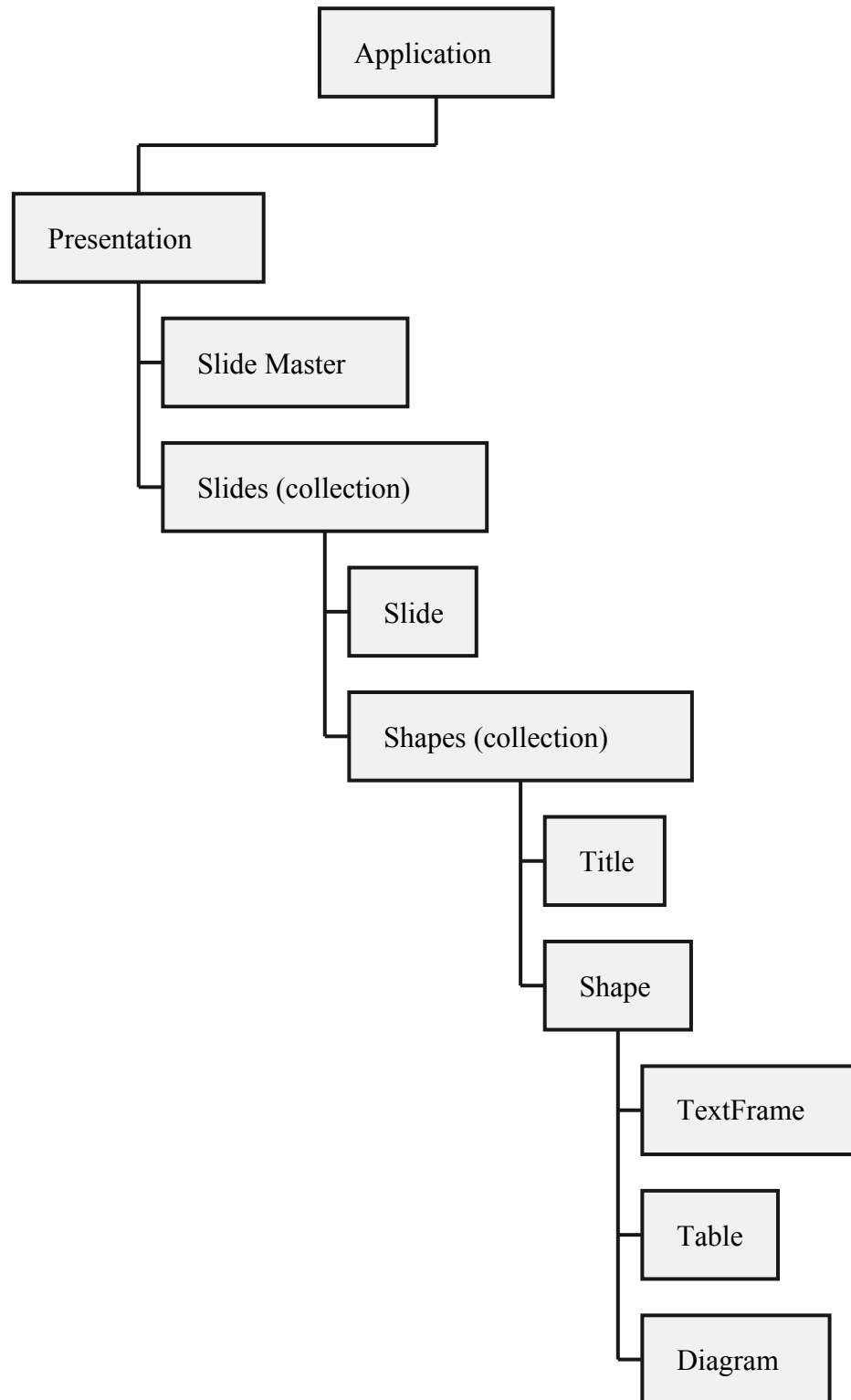
2.2. Figure. **PowerPoint object model (Granor & Martin 2000)**

The API for each COM object type is well documented and described in the official and publicly available Microsoft docs and also some books (Box 1998). For the purposes of conducting a presentation the following objects were used:

- Application object (PowerPoint), utilizing:

  - Method Open - accepts absolute file path to a PPT/PPTX file;

  - Method Quit - exits PowerPoint.

- Presentation object, utilizing:

  - Property SlideShowSettings - object which represents the slide show settings for the opened presentation;

  - Method Close - closes the specified presentation.

- SlideShowSettings object, utilizing:

  - Method Run - runs a slide show of the opened presentation returning a SlideShowWindow object.

- SlideShowWindow object, utilizing:

  - Property View - SlideShowView object;

  - Method Exit - exits the slide show.

- SlideShowView object, utilizing:

  - Method GotoSlide - accepts integer value indicating which slide to go to;

  - Property Slide - object representing the currently selected slide;

  - Method Player - accepts ID for Shape object and returns Player object instance.

- Slide object, utilizing:

  - Property Shapes - a collection representing all elements placed on slide.

- Shape object, utilizing:

  - Property ID - number that uniquely identifies the shape object;

  - Property Type - number indicating type of element (Number 16 stands for embedded media).

- Player object, utilizing:

 - Method Play - starts playback for specified media.

2.2.1. COM in multithreaded applications

The execution of commands (for media playback for example) happen concurrently with the main process thread. However, when it comes to multithreaded usage of COM objects there are some pitfalls that should be avoided.

COM has a very long history in Windows. It existed as far back as 16-bit Windows where each process had only one thread. So much of the component object model was designed and written under the assumption that it would be running in a single-threaded application. When Windows NT introduced multithreading, a limitation was initially imposed upon COM objects. A COM object could only be used on the thread that it was created on. In other words, each thread was treated as a single-threaded pseudo process, later dubbed (STA) "Single-threaded apartment" (Chen 2019).

The only way to effectively use the same COM object on different threads was by marshalling it across, i.e. marshalling the COM object into an ID on the parent thread, relaying that ID to another thread and then unmarshalling the ID back into a COM object. Initially, the presentation controller was designed and developed around this concept.

Some time later, COM introduced the concept of "Multi-threaded apartments" (MTA) in which one can freely use objects among different threads without marshalling. A thread can opt into the MTA by padding `COINIT_MULTITHREAD` flag to function `CoInitializeEx` (Hammond & Robinson 2000). Once this was discovered, presentation controller was redesigned and rewritten to utilize MTA.
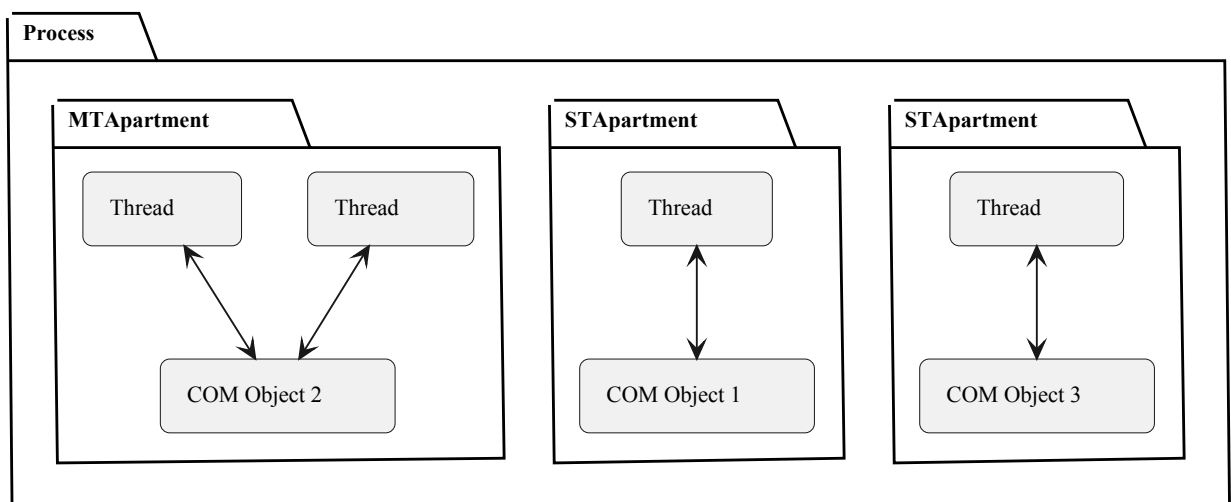
2.3. Figure. **COM threading model**

2.3. Programming Language

In order to rapidly develop applications for Nao, the official SDK was used, which narrows the choice between mostly only 2 programming languages: C++ or Python. Older version of the SDK also had support for `.Net`, Java, JavaScript and Matlab, however these appear to no longer be officially supported.

It was deemed that C++ would be a poor fit for the following reasons:

- Application does not have serious CPU performance requirements. No intense computations will be taking place and as such there is no need for granularity or optimization that C++ permits to the developer.

- Inexperienced use of C++ that involves making use of OOP features like classes and objects, almost always leads to memory leaks. Even more so when development is being done by more than one person.

- Using COM objects in C++ is fairly inconvenient. COM objects are utilization is much more effective with dynamic languages such as Visual Basic or Python.

For these reasons Python was chosen for development. Since the Nao SDK does not support the latest major version of Python (Python 3), Python 2 will be used for developing the Nao presentation controller. In all other cases, Python 3 will be used because it:

- Supports type hinting, which, when utilized, makes the source code more reliable, readable and maintainable.

- Has a slightly more comprehensive and readable syntax.

- Has more new features and operators:

    - Walrus operator;
    - String interpolation (f-strings);
    - Switch/Case expressions;
    - Better and more descriptive error messages;

- Is officially supported and will be patched in cases when vulnerabilities are discovered.

2.3.1. Python IDE

To make the development process smoother, a proper feature-rich integrated development environment can go a long way. For this reason, PyCharm was selected and used. It provides many helpful features such as:

- Live code validation and syntax highlighting;

- support for plugins like sonarlint which help in some cases to immediately identify and resolve semantic code issues;

- allows debugging python code without any external requirements;

- ability to quickly and easily set up and change the default python interpreter for a project.

The only noteworthy disadvantage for PyCharm, being its slow loading time.

2.4. Methodology

Since the development of the Nao presentation tool suite would be a collaborative effort, certain measures, methods and methodologies had to be employed in order to ensure a smooth and frictionless workflow.

2.4.1. Kanban

For the efficient distribution of tasks a Kanban board was created on GitHub. This board is a project management tool meant to help visualize the state of development and maximize efficiency (DeGrandis & DeMaria 2017). The created Kanban board had 3 columns:

- To do;

- Work in Progress;

- Complete.
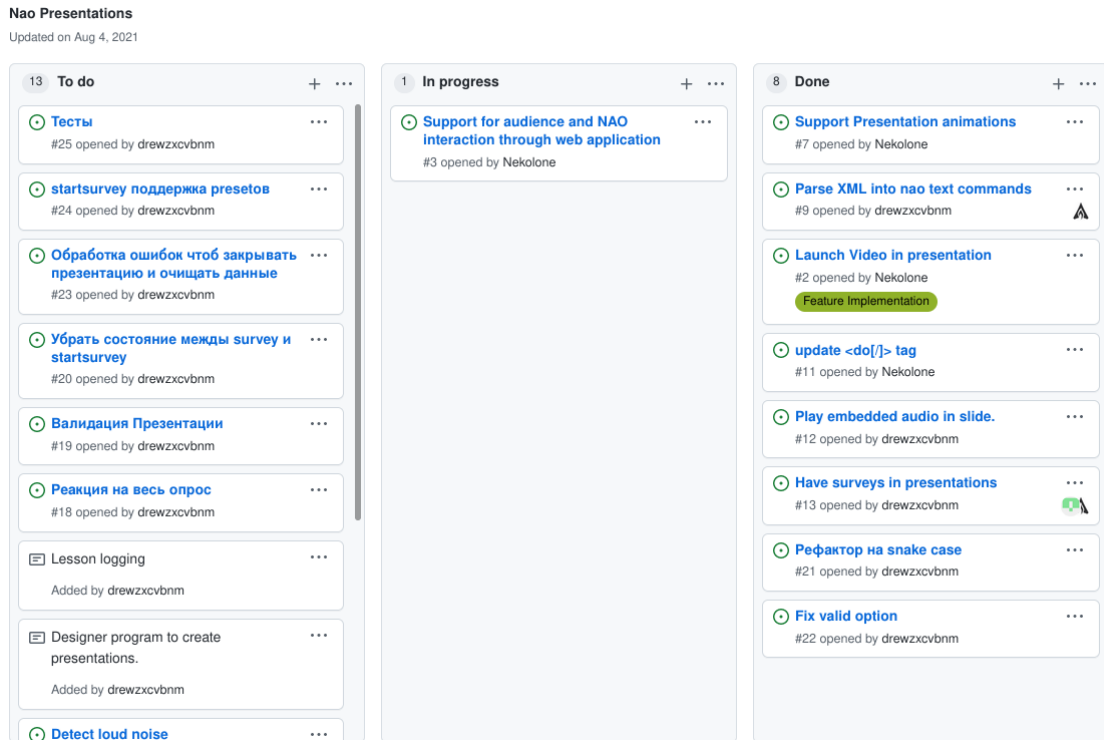
2.4. Figure.   **Snapshot of the kanban board during development**

2.4.2.   Version Control System

A version control system is a necessity for all pretty much all software development projects. It enables developers to work collaboratively and safely. With it developers can synchronize their changes with one another and efficiently resolve conflicting code changes (Loeliger & McCullough 2012).

For the development of the presentation tools suite, Git and GitHub were used. Git can be thought of as a database to store snapshots of the codebase throughout its development phase. At the core of Git is a key-value data store. There are 3 main types of objects that git stores (Chacon 2009):

- Blob: used to store file contents

- Tree: stores file directory structure

- Commit: contains within itself a reference to a Tree object and possibly references to one or two parent commits.

It supports many useful features for development:

- Version branching;

- Great Scalability;

- Distributed development.

In addition, GitHub, git's most popular repository hosting service, also provides many significant advantages:

- Secure remote repository access control;

- Kanban board creation;

- Continuous Integration pipeline support;

- Git branch access control;

- Branch Merge Requests.

By utilizing GitHub's many features, a secure and efficient workflow was established. By default, it is impossible to commit changes to the main repository branch. All changes need to take place on another feature branch. When changes for a feature are finalized, developer must create a Pull Request on GitHub. When a pull request is created the CI pipeline is launched and all unit tests are executed, code linters and analyzers are initiated and if any of them fail or return a negative result, the pipeline fails as a whole making it impossible to merge the feature changes until all the issues are resolved (Hogbin 2015).

In addition, a Pull Request must be reviewed and approved by another developer before it can be merged into the main branch. This helps to maintain code quality and make code changes publicly known to other developers.

2.4.3. Continuous Integration

As mentioned before, whenever a Pull Request is created, the pipeline is launched and number of unit test and code analyzers are executed. To be more specific, the pipeline consists of two concurrently executing jobs:

- Linting, consisting of 2 steps:

    - Set up python interpreter and install project dependencies specified in `requirements.txt`;
    - Run flake8 on source code.

- Testing, consisting of 5 steps:

    - Set up python interpreter and install project dependencies specified in `requirements.txt`;
    - Run unit tests with pytest;
    - Generate coverage report from before executed tests;
    - Run scan for SonarQube.

2.4.4. Code quality control

During active development a lot of code is written. In order to make it more readable and maintainable this code must conform to common standards and conventions (Martin 2018). In python this convention is known as PEP8 (van Rossum et al. 2001, Peters 2004). It introduces many rules by which code must be written, to name a few:

- For an indentation level - 4 spaces are strongly recommended;

    – Tabs and spaces must never be mixed.

- A line of code should not exceed 79 characters

    – For long expressions it is recommended to use Pythons implied line continuation inside parenthesis, brackets and braces

    – Backslashes may be appropriate is case of long `with` or `assert` statements;

- Class and function definitions should be separated with two blank lines;

    – Inside functions, blank lines should be used sparingly to indicate logical sections.

- Import statements should be on separate lines;

- Global imports should be at the top of the file;

- The order of imports should be as so:

    1. Standard library imports;

    2. 3rd party library imports;

    3. Local application imports.

- Relative imports are highly discouraged;

- Binary operators should be surrounded with a whitespace on both sides;

- Using semicolon to fit compound multiple statements onto one line is discouraged.

All of these style rules are checked and enforced with the utility program Flake8 (*Your tool for style guide enforcement* n.d.).

In order to avoid common antipatterns in python SonarQube was also used as a code analyzer. It helps to avoid more subtle code issues:

- Poor variable naming;

- Large cognitive complexity in functions;

- Use of map/filter functions versus list comprehension;

- Use of indices to iterate over an array when unnecessary;

- Using constants in conditions;

- Using deprecated function;

- Defining private class attributes that are never used;

- Importing with wildcards;

- Method overrides that change the method contract (Enforcing Liskov Substitution principle);

- Duplicate string literals.

In addition, SonarQube provides a convenient dashboard where all project information is aggregated and displayed in a user-friendly fashion (*Static code inspection; code analysis tools* n.d.).

2.5. Architecture

The architecture for the presentation tool suite is fairly simple, as there are only 3 components and only 2 of them interact with one another at one time.

2.5.1. Use case diagram

Overall, there are very few use case's, due to the fact that the system was designed to be as autonomous as possible.
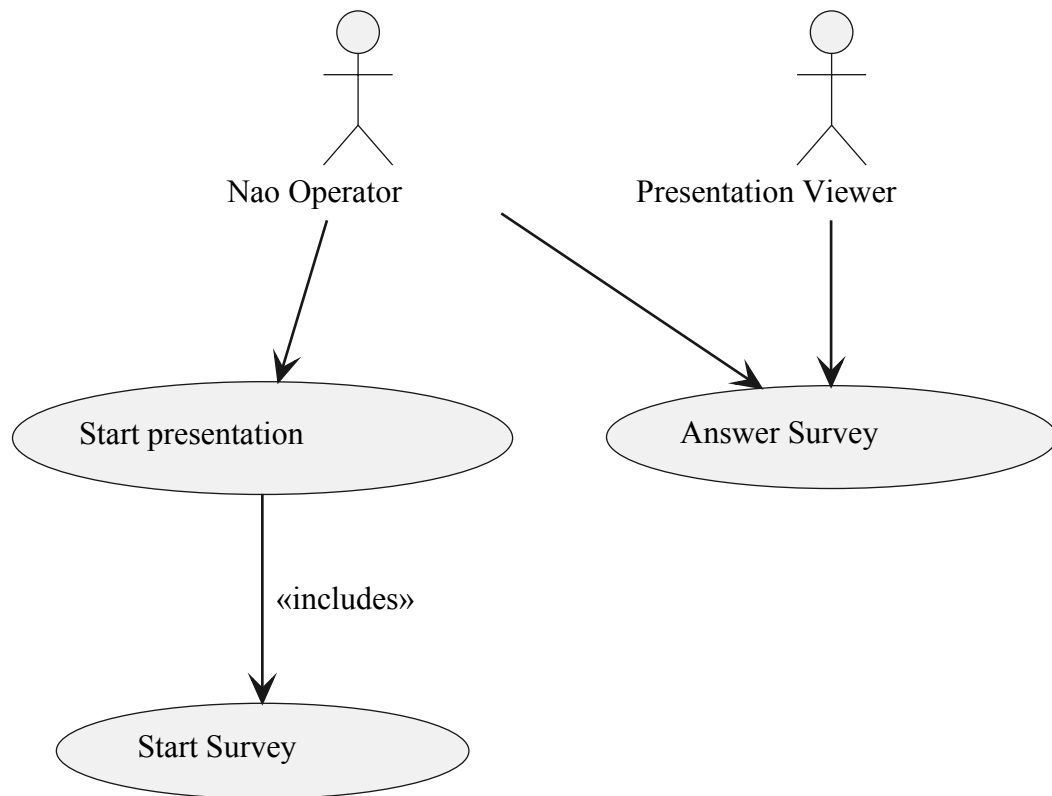
2.5. Figure. **Usecase diagram for presentation controller/survey conductor**

From a user perspective, one can launch a presentation and almost everything else happens on its own. There is minimal user interaction, and as such a minimal use case quantity.

2.5.2. Deployment schemes

While reading a presentation, the presentation controller first establishes a connection with the robot Nao. When the presentation controller is launched an optional flag is passed regarding the availability of an internet connection. If the flag, indicates that there is no internet connection, then the presentation controller does not try to send any messages to the survey conductor, else it does so when starting the presentation.
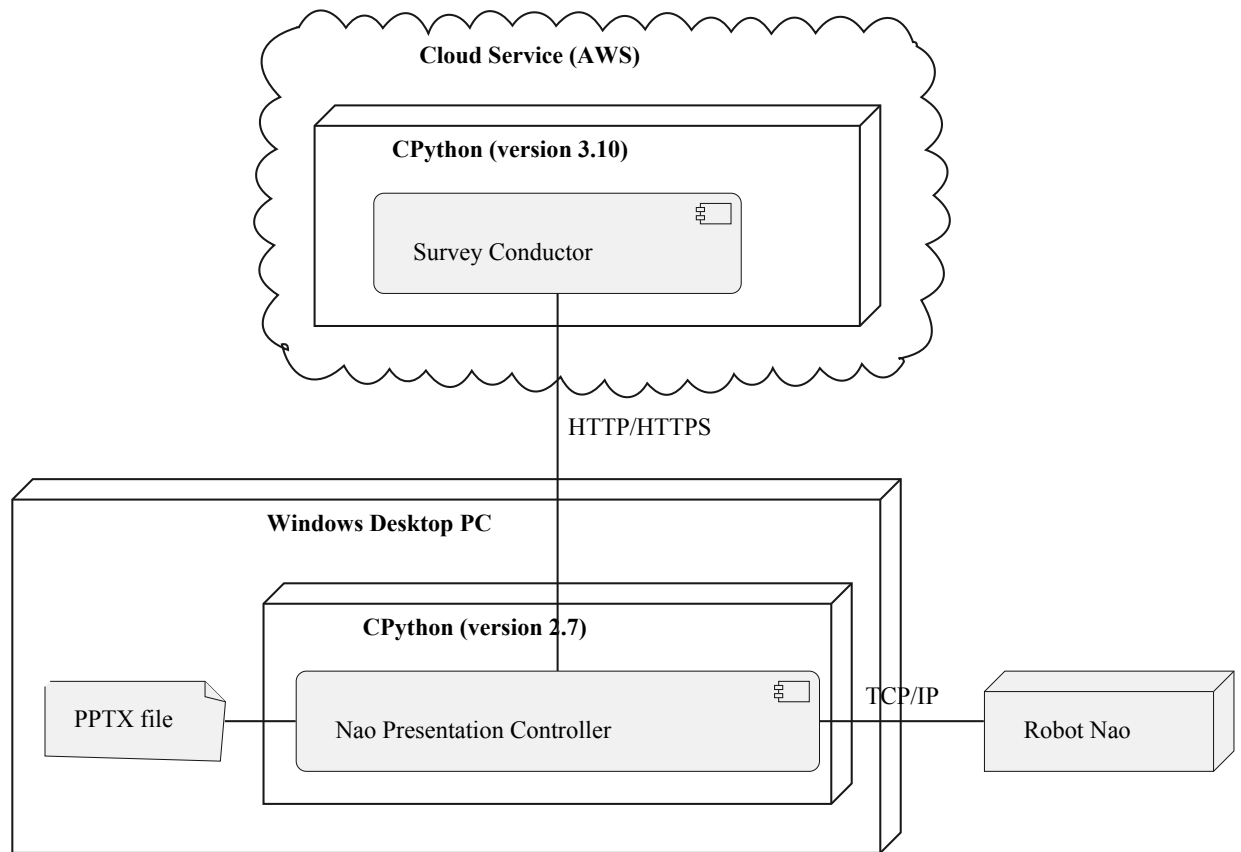
2.6. Figure. **Deployment diagram for reading a presentation**

Information regarding the IP address of the Nao Robot and the location of the PPTX file are provided the application upon startup as command line arguments. However, as an alternative, the Nao GUI shell, aside from allowing the user to interact with Nao, also provides the possibility to launch the presentation controller graphically, instead of using the OS's command prompt.
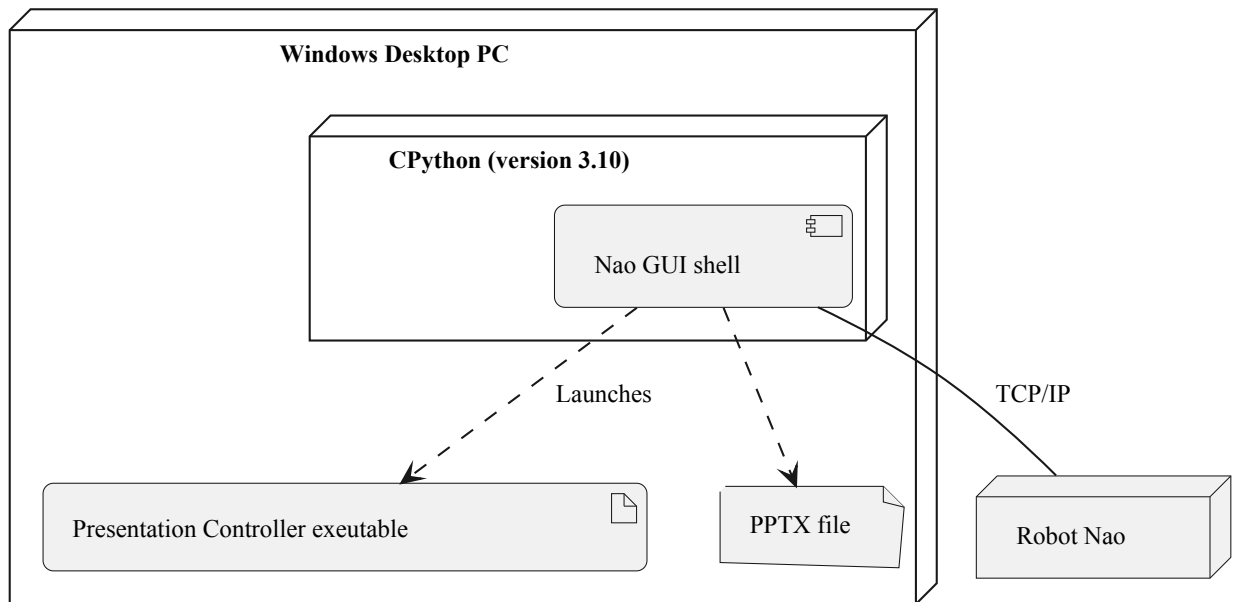
2.7. Figure. **Deployment diagram when launching presentation from GUI shell**

2.5.3. Sequence diagram

The typical workflow with the presentation tool suite is fairly simple. The user must first launch the presentation controller. This can be done using the systems command prompt or by utilizing the GUI shell application. The presentation controller attempts to establish a connection to the robot and to the PowerPoint application object. During the presentation, HTTP requests may be sent to the survey conductor to start a survey. When the last slide is reached, the presentation controller shuts down closing PowerPoint with it in the end.
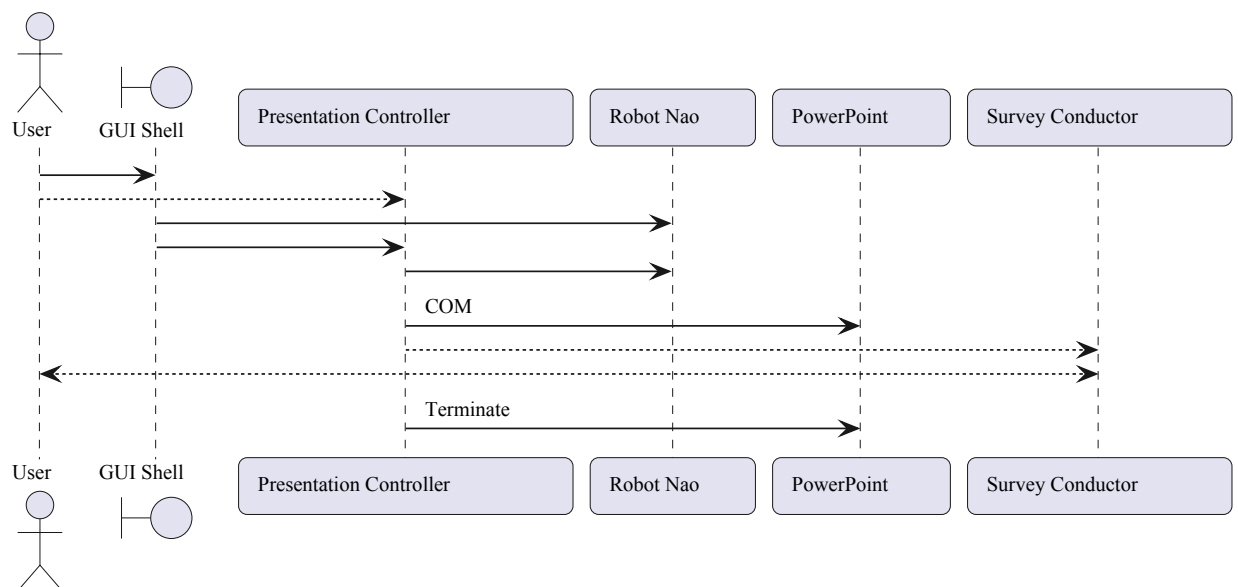
2.8. Figure. **Workflow sequence diagram**

2.5.4. Class diagram

While writing the presentation controller and survey conductor Pythons OOP features were moderately put to use and a simple object model was developed.
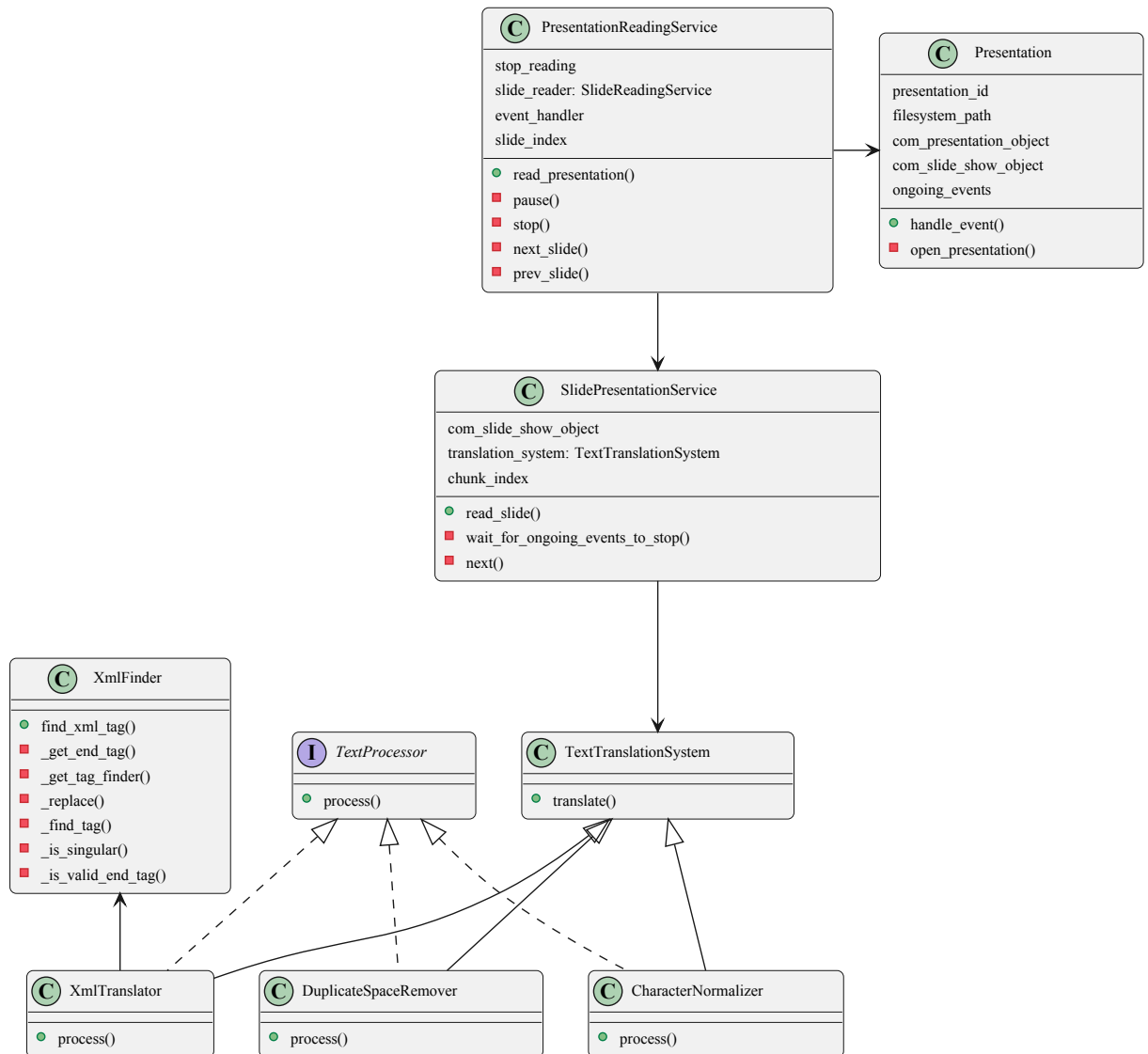
2.9. Figure. **Presentation Controller class diagram**

It should be noted that Python is not strictly speaking an object-oriented programming language, as such much code and many features were not encapsulated inside any class.

3. IMPLEMENTATION

During the development and implementation phase, many unforeseen questions and circumstances had risen and a fairy Agile approach was used when handling them as such some of the solutions may have turned out to be more entropic and less readable/maintainable than what would have been liked.

3.1. Presentation Controller

Presentation controller constitutes the largest and most complicated component in the tool suite. There are many reasons for this, foremost being that it was written with Python 2 where type hints are not present and involves the use of concepts such as COM, multithreading and event handling for which some foreknowledge can go a long way.

3.1.1. Notes Parsing

The text that Nao reads is specified in the notes of each slide, however special tags can be placed that can get perform certain actions. These tags use double backslash as a delimiter, instead of the more widely known angle brackets used in markup languages like XML. As such, to increase and usability and user experience, a parser was implemented to translate XML like tags to the aforementioned double backslash ones. Currently, eleven different tags are supported.

The `<do[/]>` tag is used for executing animations. Can be used as so: "<do animation="path/-to/animation"/>", in which case animation is started and slide reading blocks until the animation ends. Or it can be used like so: "<do animation="path/to/animation"/>next words<do/>", in which case slide reading only blocks and waits for animation to end when it reaches the end tag.

The `<set/>` tag is a general tag used to setting some global values during the execution of a presentation. It can be used to set the animation namespace, so that a complete path does not always have to be specified by the user. It can also be used to set the voice for Nao's speech (neutral/joyful/didactic). This is done by using it's attributes "animation-ns" and "voice".

The `pause` tag is used to pause the speech for variable amount of milliseconds, using the "time" attribute. Time attribute is optional and if not set manually, will be presumed to be set to 100.

During development more text processing was implemented in order to correct symbols that PowerPoint uses instead of the traditional ASCII ones. And also, to fix cases of repeating whitespaces as they cause issues with Nao text synthesizing. Overall, this is all implemented inside the `TextTranslationSystem` class.

```python
class TextTranslationSystem:
    def __init__(self, presentation):
        self.text_processors = [CharacaterNormalizer(),
                                XmlTranslator(presentation),
                                DuplicateSpaceRemover()]


    def translate(self, text):
        for processor in self.text_processors:
            text = processor.process(text)
        return text
```

3.1. Figure. **Event subscribtion example**

3.1.2. Event Handling

Event handling mainly refers to how certain program subroutines are invoked at runtime depending on where Nao is reading in the presentation slide. The general mechanism is fairly complicated.

The most reliable way to create events was with the `ALMemory` module where it is possible to create and subscribe to an event.

```python
mem = session.service("ALMemory")
mem.declareEvent("event")
subscriber = mem.subscriber("event")
subscriber.signal.connect(handle_event_callback)
```

3.2. Figure. **Event subscribtion example**

While reading the notes, it is possible to make changes inside `ALMemory` using tags. When a change is made to the "event" key, the specified callback function is executed. This function retrieves the value that was set to the "event" key and uses that information to launch the required event subroutine.

```python
def handle_event(self, event):

    eventname = event.split(EVENT_ARG_DELIMITER)[0]

    print("HANDLING " + eventname)

    if eventname not in self.event_map.keys():

        print("ERROR: eventmap cannot handle event:" + eventname)

    mem.insertData("event", None)

    self.event_map[eventname].execute_event(event)
```

3.3. Figure. **Event subscribtion example**

Information regarding what subroutine is launched for which event is stored in the `event_map` dictionary which is populated at runtime.

An event may be blocking or non-blocking. Non-blocking events are launched on a new thread and execute in the background. Blocking events are also started on a new thread, but there are mechanisms in place with them that make it possible to wait until the end if this event.

3.1.3. COM Object handling

In order to interface with the PowerPoint office application COM were used for Inter Process Communication (IPC). As mentioned before, in order to easily use COM objects across different threads, all the launched threads had to opt into an MTA and this is done by calling `CoInitializeEx(COINIT_MULTITHREADED)` (Hammond & Robinson 2000). In order to abstract this responsibility away, the `ComThread` class was created.

```python
class ComThread:

    def __init__(self, target):
        self.run = target


    def start(self):
        pythoncom.CoInitializeEx(pythoncom.COINIT_MULTITHREADED)
        self.run()
        pythoncom.CoUninitialize()
```

3.4. Figure. **ComThread class implementation**

When the application exits the appropriate methods are called in order to terminate the running slide show, presentation and PowerPoint application instance and the python equivalent destructor is called to clean up any possible left over resources.

3.1.4. Shutdown system

There are theoretically three circumstances under which the application exits:

- Finished reading the presentation;

- User pressed the escape button;

- An exception was raised somewhere terminating a thread making it, so there are no more foreground threads in the process.

Due to the multithreaded nature of the application, the shutdown process is nondeterministic and may sometimes not happen when expected. If, for example, an event was fired and a foreground thread was created, the application will still continue at this point even if the main thread terminates.

When the proper shutdown process is initiated all running behaviors on Nao are terminated, COM resources are freed, PowerPoint process is terminated, and the process terminated with a zero status code.

```python
def app_exit():
    behman.stopAllBehaviors()
    presentation.__del__()
    reading_service.__del__()
    tts.stopAll()
    kill_process("POWERPNT.exe")
    pythoncom.CoUninitialize()
    sys.exit(0)
```

3.5. Figure. **Application exit function**

3.1.5. Survey Parsing

When a survey must be held during a presentation, then it must first be defined in the slide notes. Defined survey can then be initiated with the "startsurvey" tag later in the notes. The defined survey must define an ID for itself to be able to refer to it later on with the "startsurvey" tag.

```xml
<survey id="2">
        <pin>0A2X</pin>
        <type>auto</type>
        <questions>
                <question>
                        <q>What is the first letter of the english alphabet?</q>
                        <validoption>1</validoption>
                        <options>
                                <o>A</o>
                                <o>B</o>
                                <o>C</o>
                        </options>
                </question>
                <question>
                        <q>What is the second letter?</q>
                        <timelimit>10</timelimit>
                        <validoption>2</validoption>
                        <options>
                                <o>Z</o>
                                <o>B</o>
                                <o>O</o>
                        </options>
                </question>
        </questions>
</survey>
```

3.6. Figure. **Example of a survey definition**

When the survey definition is found, it is processed and mapped to an object and later seri-alized when being sent to the Survey Conductor. The class `Survey` was created to structure and hold the needed survey data. The mapping of XML to object is implemented in the constructor of the class.

```python
class Survey:
    mandatory_fields = ['type', 'questions.question.q',
        'questions.question.options.o', ]


    def __init__(self, survey_xmltag):
        XmlTagValidator.validate(survey_xmltag, self.mandatory_fields)
        self.remote_id = None
        self.local_sid = survey_xmltag.attributes['id']
        self.type = survey_xmltag.get_child_tag_content('type')
        self.pin = survey_xmltag.get_child_tag_content('pin')
        self.questions =
        ↪    self._questions(survey_xmltag.get_child_tag('questions'))
```

3.7. Figure. **Survey class partial implementation**

The survey definition is also validated at first to make sure it has all the mandatory fields before being mapped to an object, though it's fairly superficial and doesn't provide many benefits aside from a slightly more readable exception message in case user incorrectly defined survey in the slide notes. The main validation strategy for surveys would be the usage of the Nao GUI shell application.

3.1.6.   Survey Conductor Interfacing

Interactions with the survey conductor happen through HTTP requests. There are currently four distinct HTTP requests that could be executed:

- POST request to register presentation with survey conductor;

- POST request to create survey instance;

- GET request to retrieve status of the survey (closed/opened);

- DELETE request to delete registered presentation and all its surveys.

By default, it is expected that the survey conductor will be reachable through the domain "www.tsi-nao.com", unless a flag option is relayed to the presentation controller, in which case all internet dependent operations are stubbed. This behavior is achieved using python decorators.

```python
class InetDependent:

    def __init__(self, return_value=None):
        self.return_value = return_value


    def __call__(self, func):
        def decorator(*args, **kwargs):
            if ARGS.no_inet:
                return self.return_value
            return func(*args, **kwargs)


        return decorator
```

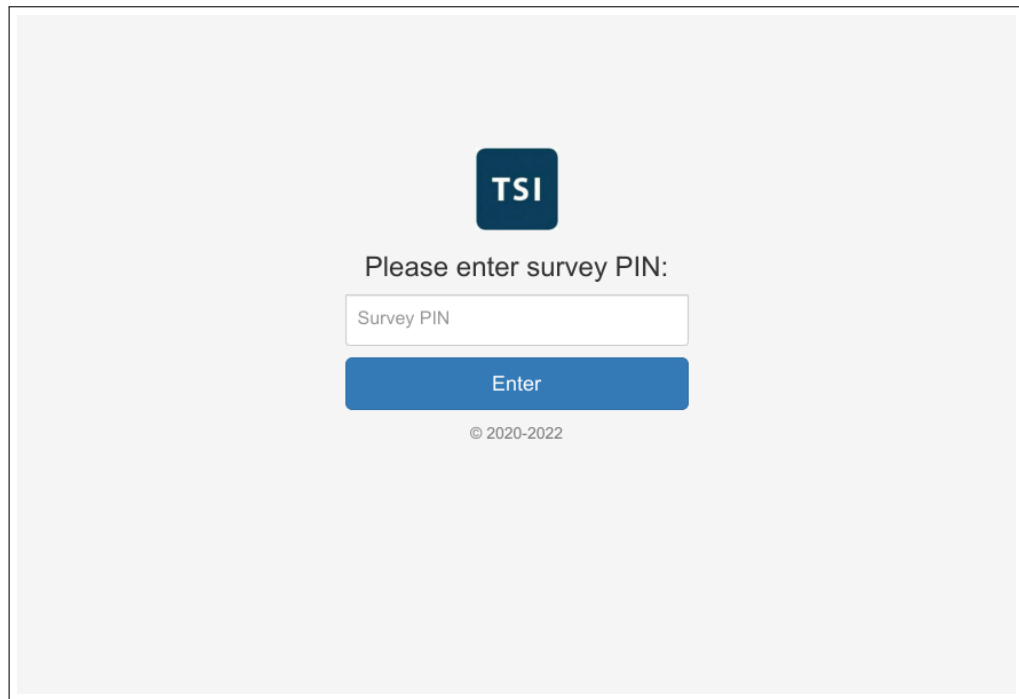3.8. Figure. **InetDepedent decorator implementation**

Decorators are a useful tool that help to isolate different aspects of one's code to separate functions improving the overall cohesion and readability of the code (Anaya 2021). The `InetDependent` decorator works by checking the passed command line arguments, skipping the internet dependent operation and returning a substitute value, if needed.

3.2.   Survey Conductor

In order to hold surveys a separate web application was developed with the python Django framework. The survey conductor is deployed using the 3rd party cloud service AWS. The following AWS services were used:

- Certificate Manager - to create an SSL/TSL certificate;

- Route 53 - to register the domain "tsinao.com";

- Elastic Beanstalk - to orchestrate application deployment;

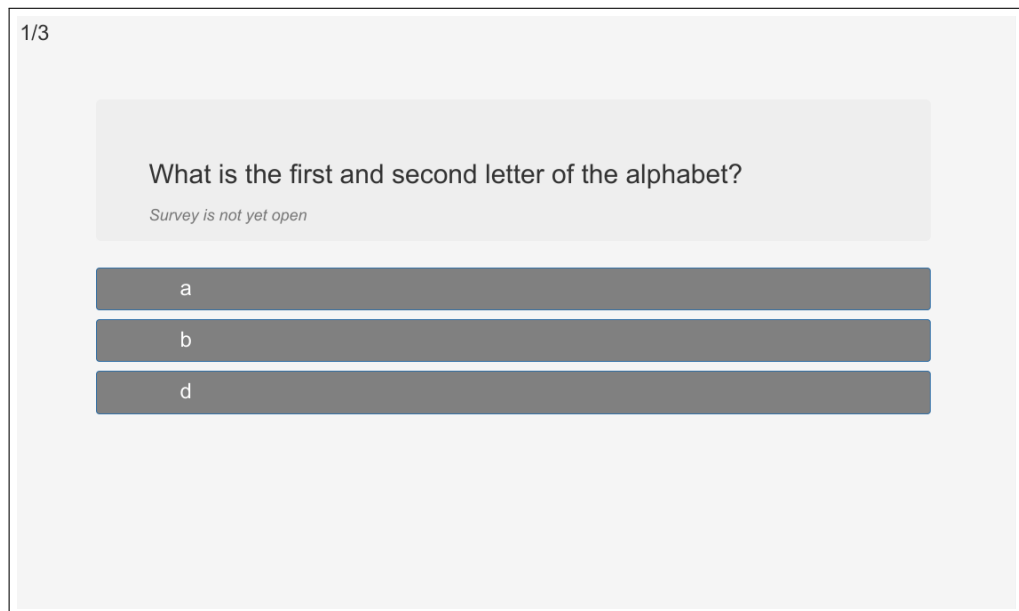- Elastic Compute Cloud - indirectly used by elastic beanstalk.

From a user point of view, a PIN code will be presented a priori which will then be inputted be the user on the main screen when they go there.

3.9. Figure. **Main screen**

Once a PIN is entered, user will be bought to the survey screen. If survey is open then user will be able to press on the different answer options. When the survey is over, it will be possible to go back and forth between the questions and to see the valid answers to them, if there is one.

3.10. Figure. **Survey screen**

3.2.1. Survey entity model

As mentioned before, no data is saved or persisted to any form of database, however to work with the data in a convenient manner it is still structured into some kind of object model. Overall

there are 4 model related classes. The first being the `Item` class.

```python
class Item:

    def __init__(self):
        self.id = next(idGenerator)
        self.creationdatetime = datetime.now(tz.gettz("Europe/Riga"))
        self.viewtime = self.creationdatetime.strftime("%c")
```

3.11. Figure. **Item class implementation**

Item class is used as the parent class for the following entity classes. It defines some simple fields that every entity should have.

```python
class Presentation(Item):

    def __init__(self, name: str):
        super().__init__()
        self.name = name
        self.surveys = []
        presentations[self.id] = self

    def add_survey(self, s):
        self.surveys.append(s)
```

3.12. Figure. **Presentation class implementation**

The presentation entity is used to represent an ongoing presentation, and it encapsulates all of surveys that happen in it. There are also the `Survey` and `SurveyQuestion` entities that store all the information regarding the survey.

3.2.2. Django HTML templating

The Django framework provides the creatively named Django Template Language (DTL) for creating HTML templates. All of the screens were written and implemented with it. DTL syntax is fairly simplistic. It was designed to help express presentation and not program logic,

making it impossible to just directly execute python expressions with it. Though it does provide tags that function similarly to some programming constructs (Shaw 2021) which includes for loops, if statements and file importing.

```
<head>
    <title>Nao Presentations</title>
    {% include "app/header.html" %}
</head>
```

3.13. Figure. **Example of file importing**

All HTML pages import the file "header.html" which contains all the frameworks and libraries used for developing the screens. And also sets the character set to UTF-8 (MacDonald 2014). By doing the importing, the following things were done:

- jQuery library was imported;

- Meta information regarding the viewport and character set was specified;

- Bootstrap stylesheet was imported;

- Common JS file containing common utility functions was imported.

This is a fairly trivial convenience that it afforded with server side rendering, and unfortunately it's impossible to do the same with plain HTML without employing some external JS library.

```
<meta charset="UTF-8">
{% load static %}
<link rel="shortcut icon" type="image/png" href="{% static 'app/favicon.ico'
↪   %}"/>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
↪   href="https://bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
<link rel="stylesheet" href="{% static 'app/style.css' %}">
<script src="{% static 'app/jquery.min.js' %}"></script>
<script type=" text/javascript" src="{% static 'app/app.js' %}"></script>
<script
↪   src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js">
↪   </script>
<script src="https://bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js">
↪   </script>
```

3.14. Figure. **Contents of header.html**

For development of the HTML pages the framework Bootstrap is used and the library jQuery is also utilized. Bootstrap framework mainly being used for stylizing purposes.

3.2.3. CSS styling

CSS is a formal language for describing the appearance of a document (web page) written using a markup language (HTML). It can also be applied to any XML document, such as SVG or XUL, though that does not apply for this case.

Without CSS, web pages do not have any appeal in their appearance making it a prerequisite for any modern web application. However, CSS is also a very extensive language with much history. The result that being that it's properties interact, often in unexpected ways, and so a certain amount of experience is required to make good use of it. In order to side step this barrier the CSS framework Bootstrap was heavily utilized. Bootstrap defines a plethora of CSS classes for all kinds of situations that can be used in the desired HTML pages. For example, the class `jumbotron` for showcasing hero unit style content.

```html
<div class="jumbotron">
    <h1>Ongoing NAO presentations:</h1>
</div>
```

3.15. Figure. **Example use of bootstrap class**

The use of bootstrap enables to develop web pages that work across many different devices, including mobile ones (Jakobus & Marah 2018).

A generally simplistic design was made for all the pages. Blueish colors are used everywhere due it being a generally neutral color and main color for the TTI university. The only exception being when showing correct and incorrect question options, where green is used indicate a valid option, and red for an invalid one.

3.3. Final class diagram

During the development phase, many changes were made to the class diagram resulting in a object model very different from what was initially imagined.
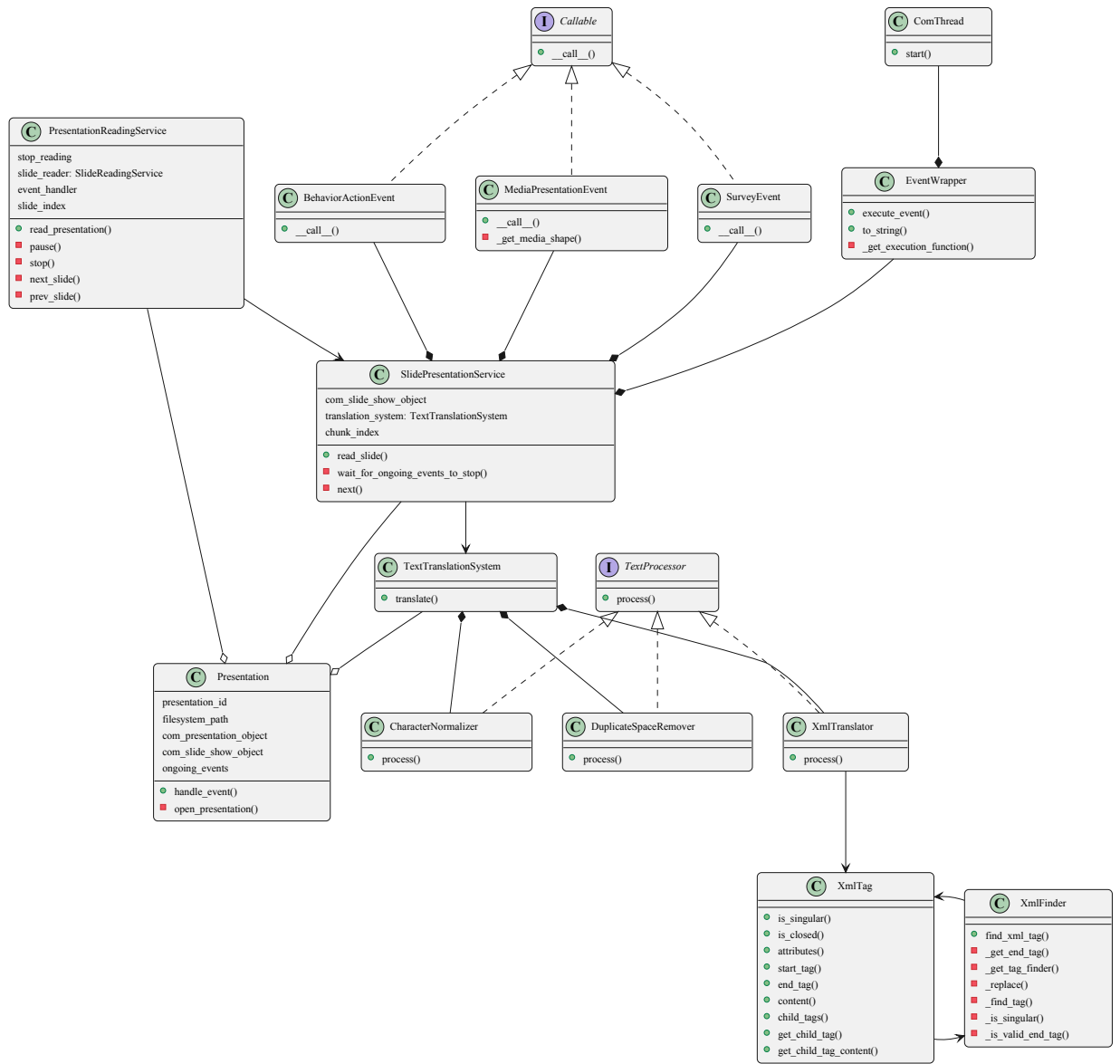
3.16. Figure. **Final class diagram**

There are many reasons for this happening. During development certain technical aspects, for example, relating to COM object usage, forced the introduction of more classes to abstract away some technicalities.

3.4. Code metrics

In order to quantifiably express the quality and maintainability of the code base, some code metrics have been calculated, by utilizing the python utility `radon`.

```
161 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.77639751553)
```

3.17. Figure. **Total average cyclomatic complexity for presentation controller**

As a matter of fact, the cyclomatic complexity does not exceed "5" in the entire presentation controller project. One of most cyclomatically complicated functions being the `_get_end_tag` function inside the `XmlFinder` class.

```python
def _get_end_tag(self, stag, tag_finder):
    stag_name = XmlTag(stag).name
    num_to_find = 1
    l, r, tag = None, None, None
    while num_to_find > 0:
        l, r, tag = next(tag_finder)
        if self._is_valid_end_tag(stag, tag):
            num_to_find -= 1
        elif XmlTag(tag).name == stag_name and not self._is_singular(tag):
            num_to_find += 1
    return l, r, tag
```

3.18. Figure. **Method with highest cyclomatic complexity**

In addition, to guage the maintainability of the code base it's "maintainability index" has been calculated, which shows an overall mixed image.

- args.py - A (100.00);
- getbehaviours.py - A (100.00);
- slidepresentationservice.py - A (53.27);
- constants.py - A (100.00);
- services.py - A (65.75);
- preader.py - A (48.54);
- events/event.py - A (47.31);
- events/mediapresentationevent.py - A (64.46);
- events/surveyevent.py - A (69.72);
- events/behavioractionevent.py - A (100.00);
- translation/texttranslator.py - A (100.00);
- naoxml/xmlfinder.py - A (46.95);
- naoxml/xmltag.py - A (35.63);
- naoxml/xmlexceptions.py - A (100.00);

- naoxml/xmltagvalidator.py - A (76.32);
- naoxml/xmltranslator.py - A (32.94);
- web/_webservice.py - A (100.00);
- web/webinterface.py - A (57.99);
- comthread.py - A (100.00);
- general.py - A (68.20);
- presentation.py - A (61.48);
- survey/survey.py - A (83.43);
- main.py - A (74.31).

The "maintainability index" is a code metric that measures how maintainable (easy to support and change) source code is. The maintainability index is calculated with a factored formula consisting of Cyclomatic Complexity, SLOC (Source Lines Of Code) and Halstead volume. It is used in several automated software metric tools, including the Microsoft Visual Studio integrated development environment, which uses a shifted scale (0 to 100) derivative. (Oman & Hagemeister 1992)

4. TESTING

In terms of testing, not much of it could be done for the development of the presentation tool suite. Since the largest part of the tool suite is the presentation controller, which requires the use of external resources like the survey conductor and Nao robot, it is not possible to write proper integration tests for it. So testing was mainly done through the use of unit tests and manual labor. The only exception being the survey conductor which is not overly dependent of external resources.

4.1. Manual testing

Manual testing was conducted by setting up an environment akin to that of production. Nao robot was set up. Survey conductor was deployed in the cloud and an example presentation was created. This example presentation made use of almost all the created tags, and utilized embedded video/audio.

While testing, there some noted cases of errors/exceptions being raised in some apparently random cases at which point it was deemed that the execution process for the presentation controller is volatile and none deterministic. The cause of this was not clear, but after some investigation was deduced to most likely be a race condition somewhere within the Nao SDK library. So to mitigate potential future issue, most of the manual testing came down to replaying the same presentation over and over again until some error showed up and then tracing that error and finding it's cause.

4.2. Automation testing

Since the Survey Conductor, is not reliant on the presence of Nao or the availability of any external web services, it's testing could be automated to certain extent. For this, Selenium Web-Driver was utilized. Selenium WebDriver is primarily a set of libraries for various programming languages. These libraries are used to send HTTP requests to the driver (hence the name Web-Driver), using the JsonWireProtocol protocol, which indicate the action that the browser should perform within the current session. Examples of such commands can be commands for finding elements by a locator, following links, parsing the text of a page/element, pressing buttons, or following links on a website page (Gundecha 2014, 2012).

There are both official bindings of the library to popular programming languages, and ama-teur ones. For example, the PHP language support library is not official and is being developed by Facebook. But in this case the Python bindings were used. As such, all the automation tests were written in Python.

Whenever any noteworthy changes were made to the survey conductor, the automation tests would be launched and if any errors were encountered, it would be investigated and resolved.

4.3. Unit testing

Unit testing is the process in programming that allows to check the correctness of individual modules of the source code. The idea is to write tests for every non-trivial function or method. This allows to quickly check whether the next change in the code has led to regression, that is, to the appearance of errors in the already tested parts of the program, and also facilitate the detection and elimination of such errors. For example, you can update the library used in the project to the current version at any time by running tests and identifying incompatibilities.

Writing Unit tests in python is a simple affair with PyUnit, a Python port of JUnit. Unit tests were written for all of components that could be executed in some form of isolation:

- Survey XML to Object mapping;

- XML validation;

- Notes text translation.

To create a test case, one simply has to provide a class that inherits from the `unittest.TestCase` class, which provides the test routines and also hooks for making each routine and cleaning up after them.

Most of these unit tests operate in simple black box manner. A certain input is given, and a certain output is expected. Naturally, if an unexpected exception occurs and the returned output does not match what was expected, the unit test as a whole fails.

```python
class MockStartMedia:


    def to_string(self):
        # Event("startmedia", MediaPresentationEvent(), presentation,
        ↪   com_context)
        return " $wait=1 $event=startmedia "



class MockPresentation:
    def __init__(self):
        self.surveys = {}
        self.event_map = {'startmedia': MockStartMedia()}



class TestTranslator(unittest.TestCase):


    def test_translation(self):
        txt = """..."""
        result = TextTranslationSystem(MockPresentation()).translate(txt)
        expected = r"""..."""
        self.assertEquals(result, expected)
```

4.1. Figure. **Unit test for notes text translation**

CONCLUSION

To conclude, all the main components of the Nao presentation tool suite have been implemented to a varying degree of success. The presentation controller successfully opens a presentation, autonomously goes through and sends commands to Nao to speak the required words. The survey conductor establishes a web API by which the presentation controller can create and start surveys. The survey conductor also provides a rather simple and intuitive UX for doing these surveys.

While writing the presentation controller some of the written components could be considered redundant. All parts relating to XML processing were manually implemented. XML parsing could have been achieved with the use of the python library "Beautiful Soup". The process of mapping the XML data to object and then later to JSON could have been skipped, by simply using the library "xmltodict". XML validation could have also been done by using the library "lxml" and writing up an XSD (XML Schema Definition) file.

Regarding COM, it would have been correct to read more on the subject before utilizing them so much. Initially, it was unknown that COM objects could be used across different threads, so a system marshalling and unmarshalling was implemented to make it possible. This system was later entirely thrown out when MTAs were discovered.

While writing the survey conductor many different endpoints were written.

- endpoint to manually go to next/previous question;

- endpoint to update a survey questions;

- endpoint to open/close a survey;

- endpoint to get the answer correctness ratio;

- endpoint to delete all presentations and surveys.

Many of these endpoints are not utilized in integration with the survey conductor because requirements were not properly defined from the beginning leading to a disorganized API design. The endpoints that allow manual manipulation of the survey are not used because all of the operations were offloaded to the survey conductor making the survey execution process autonomous from the presentation controller. In addition, the use of models and class entities is questionable. Since the surveys are not saved anywhere, there is no explicit reason for them to be defined as a class and mapped from JSON to a class instance. Theoretically, it could have been executed differently, by simply validating the received JSON, converting it to a dictionary and get the required fields from it as necessary. Albeit, such an approach would probably leave not much room for future

expansion. Nonetheless, in the resulting work product there are two different class definitions of the survey class, which could have been mitigated by getting rid of one, or perhaps even both.

Most of the above-mentioned inefficiencies came to be for only a couple of reasons, which could have been mitigated at the very start if the appropriate steps were taken.

- Poorly defined initial application requirements;

- Avoidance of 3rd party library solutions;

- Lack of foreknowledge regarding COM.

REFERENCES

1. Anaya, M. (2021), *Using Decorators to Improve Our Code*, Packt.

2. Box, D. (1998), *Essential COM*, Addison-Wesley object technology series, Addison Wesley.
   URL: *https://books.google.lv/books?id=kfRWvKSePmAC*

3. Chacon, S. (2009), *Pro Git*, Apress.

4. Chen, R. (2019), 'A slightly less brief introduction to com apartments (but it's still brief)'.
   URL: *https://devblogs.microsoft.com/oldnewthing/20191125-00/?p=103135*

5. DeGrandis, D. & DeMaria, T. (2017), *Making work visible: Exposing time theft to optimize work; flow*, IT Revolution Press.

6. Granor, T. E. & Martin, D. (2000), *Microsoft Office Automation with visual foxpro*, Hentzenwerke Pub.

7. Gundecha, U. (2012), *Selenium testing tools cookbook over 90 recipes to build, maintain, and improve test automation with selenium WebDriver*, Packt Pub.

8. Gundecha, U. (2014), *Learning selenium testing tools with python: A practical guide on automated web testing with selenium using python*.

9. Hammond, M. & Robinson, A. (2000), *Python programming on win32*, O'Reilly.

10. Hogbin, W. E. J. (2015), *Git for teams*, O'Reilly Media, Inc.

11. Jakobus, B. & Marah, J. (2018), *Mastering bootstrap 4: Master the latest version of Bootstrap 4 to build highly customized Responsive Web Apps*, Packt Publishing.

12. Loeliger, J. & McCullough, M. J. (2012), *Version control with git: Powerful tools and techniques for collaborative software development*, O'Reilly.

13. MacDonald, M. (2014), *HTML5: The missing manual*, O'Reilly.

14. Martin, R. C. (2018), *Clean architecture: A craftsman's guide to software structure and Design*, Prentice Hall.

15. Oman, P. & Hagemeister, J. (1992), Metrics for assessing a software system's maintainability, *in* 'Proceedings Conference on Software Maintenance 1992', pp. 337–344.

16. Peters, T. (2004), The Zen of Python, PEP 20.
    URL: *https://www.python.org/dev/peps/pep-0020/*

17. Shaw, B. (2021), *Web development with django: Learn to build modern web applications with a python-based framework*, Packt Publishing Ltd.

18. *Static code inspection; code analysis tools* (n.d.).
    URL: *https://www.sonarqube.org/features/multi-languages/*

19. Stevewhims, A. (n.d.), 'The component object model - win32 apps'.
    URL: *https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model*

20. van Rossum, G., Warsaw, B. & Coghlan, N. (2001), Style guide for Python code, PEP 8.
    URL: *https://www.python.org/dev/peps/pep-0008/*

21. *Your tool for style guide enforcement* (n.d.).
    URL: *https://flake8.pycqa.org/en/latest/*

APPENDIX

Appendix 1

./slidepresentationservice.py

```python
import sys
import time

import qi
from services import atts, mem, tts
from events.event import Event
from translation.texttranslator import TextTranslationSystem
from events.surveyevent import SurveyEvent
from events.mediapresentationevent import MediaPresentationEvent
from events.behavioractionevent import BehaviorActionEvent


class SlidePresentationService:

    def __init__(self, presentation):
        mem.insertData('wait', 0)
        self.slide_show = presentation.com_slide_show
        self.translation_system = TextTranslationSystem(presentation)
        self.i_chunk = None
        event_map = presentation.event_map
        event_map["next"] = Event("next", self._next, presentation,
            blocking=False)
        event_map["startmedia"] = Event("startmedia",
            MediaPresentationEvent(self.slide_show), presentation)
        event_map["startsurvey"] = Event("startsurvey", SurveyEvent(
            presentation), presentation)
        event_map["behavior"] = Event("behavior", BehaviorActionEvent(),
            presentation)

    def read_slide(self, slide):
        text_note = slide.notes_slide.notes_text_frame.text
        notes = text_note.encode('utf-8')
        print("Before translation:{}".format(str(notes)))
        notes = self.translation_system.translate(notes)
        print("After translation:{}".format(str(notes)))
        chunks = notes.split("<split/>")
        self.i_chunk = 0
        while self.i_chunk < len(chunks):
            chunk = chunks[self.i_chunk]
```

```
            say = qi.async(atts.say, (str(chunk)), delay=100)
            say.wait()
            self._wait_for_ongoing_events_to_stop()
            self.i_chunk += 1

    def _wait_for_ongoing_events_to_stop(self):
        while mem.getData('wait') != 0:
            time.sleep(0.1)


    def _next(self):
        ss = self.slide_show
        ss.View.Next()


    def __del__(self):
        self.i_chunk = sys.maxint
        tts.stopAll()
```

Appendix 2

./preader.py

```python
import sys
import threading
from eventmanager import Eventloop, Event, binaryPredicate
import time
from services import tts, touch
from slidepresentationservice import SlidePresentationService

langs = {
    "rus": "Alyona22Enhanced",
    "eng": "naoenu"
}


class PresentationReadingService:

    def __init__(self, lang="eng"):
        self.lock = threading.Lock()
        self.stop = False
        self.slide_reader = None
```

```python
        tts.setVoice(langs[lang])
        self.event_handler = Eventloop()
        self.event_handler.addEvent(
            Event(self._pause, [], binaryPredicate(lambda: touch.
                getStatus()[8][1], False, True)))
        self.event_handler.addEvent(
            Event(self._next_slide, [], binaryPredicate(lambda: touch.
                getStatus()[7][1], False, True)))
        self.event_handler.addEvent(
            Event(self._prev_slide, [], binaryPredicate(lambda: touch.
                getStatus()[9][1], False, True)))

    def read_presentation(self, presentation):
        self.slide_reader = SlidePresentationService(presentation)
        presentation.com_ppoint.Visible = True
        self.i_slide = 0
        self.event_handler.start()
        while self.i_slide < len(presentation.slides):
            if self.stop:
                break
            self.lock.acquire()
            presentation.com_slide_show.View.GotoSlide(self.i_slide + 1)
            self.slide_reader.read_slide(presentation.slides[self.
                i_slide])
            self.i_slide += 1
            self.lock.release()
            time.sleep(1)

    def __del__(self):
        self.i_slide = sys.maxint
        self.slide_reader.__del__()
        self.event_handler.stop()
        self.event_handler.join()

    def _pause(self):
        print("pause")
        tts.stopAll()
        self.lock.acquire()
        while not touch.getStatus()[8][1]:
            time.sleep(0.1)
```

```python
        print("resume")
        self.lock.release()
        time.sleep(1)

    def _stop(self):
        self.stop = True
        tts.stopAll()

    def _next_slide(self):
        tts.stopAll()

    def _prev_slide(self):
        self.i_slide -= 2
        tts.stopAll()
```

Appendix 3

./events/mediapresentationevent.py

```python
import time


class MediaPresentationEvent:

    def __init__(self, slide_show):
        self.ss = slide_show

    def __call__(self):
        ss = self.ss
        sh = self._get_media_shape(ss.View.Slide)
        time.sleep(ss.View.Slide.SlideShowTransition.Duration)
        pl = ss.View.Player(sh.Id)

        pl.Play()
        while pl.State != 2:
            time.sleep(0.5)

    def _get_media_shape(self, slide):
        for s in slide.Shapes:
            if s.Type == 16:
```

```
            return s
```

Appendix 4

./events/surveyevent.py

```python
import time

from web.webinterface import WebInterface


class SurveyEvent:

    def __init__(self, presentation):
        self.pid = presentation.presentation_id
        self.surveys = presentation.surveys

    def __call__(self, sid):
        survey = self.surveys[sid]
        WebInterface.create_survey(survey, self.pid)
        WebInterface.open_survey(survey)
        st = WebInterface.get_survey_status(survey)
        while st.lower() != 'closed':
            time.sleep(3)
            st = WebInterface.get_survey_status(survey)
```

Appendix 5

./events/behavioractionevent.py

```python
from services import behman


class BehaviorActionEvent:

    def __call__(self, name):
        print("start_custom_behavior ", name)
        try:
            behman.runBehavior(name)
        except RuntimeError as err:
            print ("Behaviour error. Aborting")
```

```python
            print (err)
        print("end_custom_behavior", name)
```

Appendix 6

./translation/texttranslator.py

```python
# coding=utf-8
import re
from naoxml.xmltranslator import XmlTranslator


class CharacaterNormalizer:
    def __init__(self):
        self.not_normalized_dic = {
            '"': '"',
            '"': '"',
            '«': '"',
            '»': '"'
        }

    def process(self, text):
        for k, v in self.not_normalized_dic.items():
            text = text.replace(k, v)
        return text


class DuplicateSpaceRemover:

    def __init__(self):
        pass  # empty init

    @staticmethod
    def process(text):
        return re.sub("□+", "□", text)


class TextTranslationSystem:
    def __init__(self, presentation):
        self.text_processors = [CharacaterNormalizer(),
```

```
                                    XmlTranslator(presentation),
                                    DuplicateSpaceRemover()]


    def translate(self, text):
        for processor in self.text_processors:
            text = processor.process(text)
        return text
```

Appendix 7

./comthread.py

```python
import pythoncom


class ComThread:

    def __init__(self, target):
        self.run = target

    def start(self):
        pythoncom.CoInitializeEx(pythoncom.COINIT_MULTITHREADED)
        self.run()
        pythoncom.CoUninitialize()
```

Appendix 8

./presentation.py

```python
import pythoncom
import win32api
import win32com.client as win32
from web.webinterface import WebInterface
from pptx import Presentation as PPTXPresentation
from constants import EVENT_ARG_DELIMITER
from services import mem


# mem event MUST be declared immediately here
mem.declareEvent("event")
subscriber = mem.subscriber("event")
```

```python
class Presentation :

    def __init__ ( self , path ) :
        name = path . split ( '\\ ' ) [ -1]. split ( '.') [0]
        self . presentation_id = WebInterface . create_presentation (name)
        self . path = path
        self . com_ppoint = win32 . gencache . EnsureDispatch ( 'Powerpoint .
            Application ')
        self . com_presentation = self ._open_presentation ()
        self . com_slide_show = self . com_presentation . SlideShowSettings .
            Run ()
        self . pptx_presentation = PPTXPresentation ( self . path )
        self . slides = self . pptx_presentation . slides
        self . surveys = {}
        self . ongoing_events = []
        self . event_map = {}
        subscriber . signal . connect ( self . handle_event )

    def handle_event ( self , event ) :
        eventname = event . split (EVENT_ARG_DELIMITER) [0]
        print ( "HANDLING " + eventname )
        if eventname not in self . event_map . keys () :
            print ( "ERROR: eventmap cannot handle event : " + eventname )
        mem . insertData ( "event" , None )
        self . event_map [ eventname ]. execute_event ( event )

    def __del__ ( self ) :
        self . com_slide_show . View . Exit ()
        self . com_presentation . Close ()
        self . com_ppoint . Quit ()
        WebInterface . delete_presentation ( self . presentation_id )
        for values in self . event_map . values () :
            del values

    def _open_presentation ( self ) :
        try :
            return self . com_ppoint . Presentations . Open ( self . path )
        except pythoncom . com_error as err :
            msg = win32api . FormatMessage ( err [2][5])
```

```python
raise RuntimeError("Error opening presentation: " + msg)
```