**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CENTRO DE INFORMÁTICA**

**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Breno Rios Ramos**

**Plugin submission for extensible tools - *piStar Tool* case**

**RECIFE**

**2019**

**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CENTRO DE INFORMÁTICA**

**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Breno Rios Ramos**

**Plugin submission for extensible tools - *piStar Tool* case**



**RECIFE**

**2019**

**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CENTRO DE INFORMÁTICA**

**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Breno Rios Ramos**

**Plugin submission for extensible tools - *piStar Tool* case**

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 03 de Julho de 2019.

Banca Examinadora:

Orientador

XXXXXX

Doutor

Examinador

XXXXXXX

Doutor



**AGRADECIMENTOS**

To professors Jaelson Freire Brelaz de Castro and João Henrique Correia Pimentel for the extremely caring guidance.

To all my colleagues, from the ones who became friends to the others, that made this journey a easier to go trough.



**ABSTRACT**

An extensible architecture is the one that allows anyone, beyond the original author of the application, to add functionalities without the necessity of accessing the source code or the original application creators intervention [1]. It is present in all kinds of software applications nowadays and it is a very popular choice in the development of browsers and IDE´s (Integrated Development Environment). In the case of piStar Tool, there is a necessity of expansion for supporting new extensions for the i\* language.

The *piStar Tool* is already a pluggable tool [8]. However it is still necessary the intervention of the original creator in the source code or the intervention of a plugin creator, cloning the source code repository and creating their own *piStar Tool* version. This work aims to define and implement a plugin submission tool for *piStar Tool*. This is going to be achieved through a preliminary analysis of tools that already support this kind of service, as well as tools that are code repository services. As a result we are going to have the documentation and implementation of the submission tool, allowing the installation of new extensions using the *piStar Tool* graphical interface.

**Keywords**: extensible architecture requirements piStarTool

**SUMMARY**

[**Introduction**](#_gjdgxs) **21**

[Motivation](#_2et92p0) 21

[Goals](#_tyjcwt) 22

[**Existing tools**](#_uqxz7hin5zif) **22**

[Visual Studio Code](#_6yhlaz5fk020) 22

[Introduction](#_a389q7d6j6xs) 22

[Plugin Creation Process](#_uj9hs41qaosz) 22

[Plugin Submission Process](#_qzxsgqk63ew1) 23

[Installation and Usage Process](#_eqyq1j3c23pv) 24

[Conclusions](#_gxraya5w0gbi) 26

[Google Docs](#_bxw4k7iadafs) 26

[Introduction](#_crol0qz42ver) 26

[Plugin creation process](#_605bo8ccpnzx) 26

[Plugin submission process](#_3m78xutkhmqd) 28

[Installation and usage process](#_d3rxbs2s22bo) 28

[Conclusions](#_r2ovnovjdzzq) 30

[Sketchup](#_973wdpy6vdl4) 30

[Introduction](#_jodcfs95qn8d) 30

[Plugin Creation Process](#_9cn3wtvufn42) 31

[Plugin Submission Process](#_bi6os5ism8eu) 31

[Installation and Usage Process](#_qplvkk7ceh5j) 31

[Conclusions](#_s53tj3zfwd9h) 31

[StarUML](#_b4czi6je10mn) 31

[Introduction](#_31qbr595eaff) 31

[Plugin Creation Process](#_3xbsy1agtdxb) 31

[Plugin Submission Process](#_bfdkas58ymj9) 31

[Installation and Usage Process](#_g9krurwg2ihe) 31

[Conclusions](#_ybhd9muhaycj) 31

[Atom](#_ufeoe9jto79d) 31

[Introduction](#_26v1kdompk4e) 31

[Plugin Creation Process](#_qyk4n1kqdoah) 31

[Plugin Submission Process](#_m06sw82tu75u) 31

[Installation and Usage Process](#_miix4963ejb8) 31

[Conclusions](#_wic28yc2z32k) 31

[Wordpress](#_8j7876op8wkx) 31

[Introduction](#_un20dckj5y1o) 31

[Plugin Creation Process](#_a9747bjb00cz) 31

[Plugin Submission Process](#_u2voyjt0tkhu) 31

[Installation and Usage Process](#_sf64ecw1krjt) 31

[Conclusions](#_sebjs7l8wqpf) 31

[Eclipse](#_sczlynnc59ry) 32

[Introduction](#_uz8kujehailx) 32

[Plugin Creation Process](#_syfyauho9eu6) 32

[Plugin Submission Process](#_qkj11t79dsa7) 32

[Installation and Usage Process](#_uu1mqcd5sylr) 32

[Conclusions](#_5n04scscfm81) 32

[**piStar Tool**](#_59h6rip07jqh) **32**

[**piStar - Plugin Submission Tool**](#_4s2wmrgsjpe9) **32**

[User Roles](#_abjpa8opatpk) 32

[Administrator](#_he56bnxg2ic) 32

[Non-IT User](#_euhixbhlstse) 32

[IT User](#_rotza746nvmv) 32

[User Stories](#_wsyxjf16sh5u) 32

[Common User Stories](#_hh6d4kmo8xoa) 32

[Register](#_y4vid2pl4w9z) 32

[Login](#_gxykm0o8pdo9) 32

[List of plugins](#_2elwagslg4sq) 33

[Logout](#_udc83kz8nhjl) 33

[Administrator User Stories](#_wfv7asps49g7) 33

[List of all plugins](#_j53dw3hynzto) 33

[Edit submitted plugins data](#_uyeg9hpfoojr) 33

[Delete submitted plugins](#_9wevxxin6eg5) 33

[Search all submitted plugins](#_l89gllhzxirk) 33

[Search all users](#_op8mgvd45myh) 33

[Edit users data](#_8n5h4d8mdwsv) 33

[Delete users](#_10tovkv0xwom) 33

[Novice User Stories](#_1avc7iwyomzv) 33

[Submit plugins](#_kc0r4f3yf2qo) 33

[Expert User Stories](#_9evpp3yboga1) 33

[Submit plugins](#_w4twom3pq0uf) 33

[Architecture](#_hkq4aci43kyp) 34

[MVC Pattern with NodeJS](#_56e65qo65p5z) 34

[Database](#_wpq354e5zz58) 35

[Schema](#_ksz459286dtk) 35

[Plugins](#_ta8nx9yb05h8) 35

[Users](#_iyn11cte171r) 36

[UsersPlugins](#_uziq1diig3f5) 36

[Keywords](#_ebwh8mxc0syy) 37

[PluginsKeywords](#_83qexea7miur) 37

[Redis](#_qtoffya5bwqd) 38

[Postgres](#_2wu5fb7w49yx) 38

[Views](#_5vj6vpi20r4) 38

[Nunjucks](#_tjp44jvzgb6g) 38

[Materialize CSS](#_ifkkspz8kqcg) 38

[**Conclusion**](#_d05og6ysg57m) **38**

[Contributions](#_9icjwigqbkzr) 38

[Future works](#_ujly3kqw7e58) 38

[**References**](#_jiquy3tbxa8r) **39**

[General](#_xwkqr0xwsdun) 39

[MCVEIGH, ANDREW “A Rigorous, Architectural Approach to Extensible Applications.” Imperial College London - Department of Computing](#_mdmpjkxe6eae) 39

[BOLOUR, AZAD “Notes on the Eclipse Plug-in Architecture.” Bolour Computing](#_vqquus5b8in1) 39

[BIRSAN, DORIAN “On Plug-ins and Extensible Architectures.” ACME Queue Volume 3, Issue 2, March 18th, 2005.](#_hq2jzwvkx0b9) 39

[PRIKLADNICKI, RAFAEL “Problemas, Desafios e Abordagens do Processo de Desenvolvimento de Software.” PUCRS.](#_74qv1q6x8040) 39

[RICHARDS, MARK “Software Achitecture Patterns”, O’REILLY.](#_c7h4k4nrg9lw) 39

[GONÇALVES, ENYO “A Systematic Literature Review of iStar Extensions”,](#_3jshyc2dgzpq) 39

[iStar Extension Repository. Available at: <https://istarextensions.cin.ufpe.br/catalogue/publication/list?q=&f=92>. Access date: March 22, 2019.](#_e6ue7jzexcyu) 39

[PIMENTEL, JOÃO “piStar Tool - A Pluggable Online Tool For Goal Modeling.”](#_czcurpjrlrgi) 39

[Visual Studio Code](#_a11qa6nhdla) 39

[Google Docs](#_d2hecqfdzqzg) 40

[Google Docs: Quickstart - Add-on for Google Docs. Available at: <https://developers.google.com/gsuite/add-ons/editors/docs/quickstart/translate>. Access date: May 22, 2019.](#_tr4ukgeidq7) 40

[Google Docs: About. Available at: <https://www.google.com/docs/about//>. Access date: May 22, 2019.](#_us51b8cr3j3b) 40

[Google Docs: Apps Script - Overview. Available at: <https://developers.google.com/apps-script/overview>. Access date: May 23, 2019.](#_66incbdcztq5) 40

[Google Docs: Triggers. Available at: <https://developers.google.com/apps-script/guides/triggers/>. Access date: May 23, 2019.](#_oo533w5gmqqm) 40

[Google Docs Concept: Triggers. Available at: <https://developers.google.com/gsuite/add-ons/concepts/triggers>. Access date: May 23, 2019.](#_fstfxih2fr47) 40

[Google Docs: Tutorial - Google Maps plugin - Overview. Available at: <https://www.labnol.org/internet/write-google-docs-addon/28446/>. Access date: May 23, 2019.](#_vmvw1bf56z15) 40

[Google Docs: Creating a version. Available at: <https://developers.google.com/apps-script/guides/versions#creating\_a\_version>. Access date: May 28, 2019.](#_p4jebhd8e9im) 40

[Google Docs: Publishing editor add-ons - Overview. Available at: <https://developers.google.com/gsuite/add-ons/how-tos/publishing-editor-addons>. Access date: May 23, 2019.](#_z13fo7mlkoq4) 40

[Google Docs: oAuth consent screen. Available at: <https://developers.google.com/apps-script/guides/cloud-platform-projects#completing\_the\_oauth\_consent\_screen>. Access date: May 23, 2019.](#_bndzq9i52cyf) 40

[Sketchup](#_1pq5odlslkw0) 40

[Sketchup Extension Warehouse. Available at: <https://extensions.sketchup.com/>. Access date: May 28, 2019.](#_2fpii6ssod2h) 40

[Atom](#_un4p58mc47q8) 41

[Wordpress](#_ojwzbgazk3he) 41

[Eclipse](#_jgb1wy571zx6) 41

[piStar - Submission Tool](#_ff545ztqbfpf) 41

[About NodeJS. Available at: <https://nodejs.org/en/about/>. Access date: May 26, 2019.](#_xtu651xsavkw) 41

[About NodeJS. Available at: <https://nodejs.org/en/about/>. Access date: May 26, 2019.](#_3yw1sth2hlro) 41

[Free relational database diagram design tool: dbdiagram.io. Available at: <https://dbdiagram.io/home>. Access date: May 26, 2019.](#_j5krn0kh3ybq) 41

[Redis. Available at: <https://redis.io>. Access date: May 27, 2019.](#_7eqalptiti1i) 41

[Redis: connect-redis. Available at: <https://github.com/tj/connect-redis>. Access date: May 27, 2019.](#_7hjtn9nv6obv) 41

[Redis: 15 reasons to use Redis as an Application Cache. Available at: <https://redislabs.com/wp-content/uploads/2016/03/15-Reasons-Caching-is-best-with-Redis-RedisLabs-1.pdf>. Access date: May 27, 2019.](#_6asobw69brbl) 41

**Lista de Figuras**

[Figura 1. Exemplos da estilização *pastel-painting*. Imagem adaptada de [16]. 20](#_30j0zll)

**LISTA DE TABELAS**

[Tabela 1 - Configurações possíveis para filtragem. 21](#_1fob9te)

**INITIALS TABLE**

|  |  |  |  |
| --- | --- | --- | --- |
| **Initials** | **Meaning** | **Context** | **Page** |
| CASE | Computer-Aided Software Engineering | Class of tools that help software engineering activities |  |
| CLI | Command-line Interface | - |  |
| NPM | Node Package Manager | It is two things: an online repository for open-source node projects and it is a *CLI* that interacts with the online repository and helps installing and managing those packages in the users machine or project. |  |
| MVC | Model-View-Controller | Type of software design pattern |  |
| TTL | Time to live | Time to live of a key stored in *Redis* that has a timeout, it has |  |
| GCP | Google Cloud Platform | Cloud computing service from Google |  |
| .rb | *Ruby* file | *Ruby* file extension |  |
| .json |  |  |  |
| API | Application Program Interface | It consists of a set of objects and methods that are used as an access interface between two systems. An *API* allows the user to access only the exposed parts of the system. |  |
| UI | User Interface | Often relates to the interface that users can interact with. |  |
| LADSPA | Linux Audio Developer’s Simple Plugin API | API that allows developers to build plugins for audio applications. |  |
| LV2 | Linux Audio Developer’s Simple Plugin API version 2 | Successor of LADSPA. |  |
| *Nyquist* |  |  |  |
| *VST* | Virtual Studio Technology | Interface that integrates synthesizers and audio effects with digital audio applications. |  |

# Introduction

The Software Engineering has the goal to guide development to guarantee certain qualities as formality, abstraction, decomposition, generalization and flexibility. This last one has become a very desired characteristic due to the high demand of new functionalities that have to be implemented in the minimum possible time [1].

An extensible architecture has become a very popular in all kinds of applications, precisely for the power of mitigating the problem mentioned above. A lot of them are not just capable of adding some plugins, but are completely formed by lots of them connected to a common core [2][3].

Each chapter is going to serve a specific goal. In Chapter 2 is going to be presented a list of tools and a brief analysis of each one. The analysis will consist of five parts: a brief *Introduction* describing what the tool is for, the *Plugin Creation Process*, the *Plugin Submission Process,* the *Installation and Usage Process* and some *Conclusions* about each one of these processes, making observations based on what are the needs of *piStar Tool*. In Chapter 3 is going to be explained a little bit about *piStar Tool*, focusing on the user needs and its plugin structure. In Chapter 4 is going to be introduced *piStar - Plugin Submission Tool* and presented its *User Roles, User Stories*, its project structure and the technologies used. In Chapter 5 is going to be made some considerations about the contributions that this work brings to the community and possibilities of future works to expand the tool.

## Motivation

The *piStar Tool* is a CASE tool that allows the creation of Goal Modelling diagrams using i\* language. It is already pluggable, meaning that it is simple to attach another language extension diagram tools. However it still requires that plugin creators clone the source code and implement their own version of *piStar Tool.*

Commonly, more specific extensions are created to model some areas like Safety, Context, Risks, Sustainability and many more listed in [7]. Besides i\* language extensions , the tool has the possibility of supporting the creation of other modelling languages diagrams [8]. Given the given large amount of i\* language extensions [7] and different languages that use diagrams as tools to modelling, the use of a completely integrated plugin submission tool and the capability of loading these plugins trough *piStarTool* graphical interface would help the growth and the use of the tool.

## Goals

This work aims to define and implement a plugin submission tool to *piStar Tool*, allowing an even faster expansion of its functionalities, adding support of new languages and extensions of the i\* language with the help of the users community. It is going to be achieved through a comparative study of some tools that already use this extensible approach. As a result, the submission tool must allow the submission of new plugins that can be loaded in *piStar Tool* trough its graphical interface, not requiring any intervention in the source code.

# piStar Tool

## Introduction

*piStar Tool* is an open-source goal modelling tool that runs in the browser. It is focused in modelling diagrams using *i\** 2.0 language. It provides visual features like *Actors* auto-resizing and collapsing, error explanation, *SVG* and *PNG* exporting and also supports non-english symbols. The *i\** 2.0 language currently has a great set of extensions that could take advantage of *piStar Tool* modelling [7][8]. To accomplish that, the extension authors should clone *piStar Tool* repository and add the elements and rules of the extension to the *language* directory of their copy of the tool [5][6].

## Plugin Structure

The *language* directory is where all the files related to the language elements and rules are stored. It has the following structure:



* *constraints.js*: object that defines the constraints of the metamodel to be used.
* *metamodel.js*: object that defines the metamodel to be used.
* *shapes.js*: declares the configuration of each shape used in the language.
* *ui.metamodel.js*: function that defines additional information used in the *UI* regarding the elements of the language.
* *images*: directory that stores all the *SVG* files used to represent the elements of the language.

# Existing tools

This chapter is going to describe and analyze some of the most popular tools that use the extensible architecture and have some form of plugin submission system. First there will be a brief introduction about the tool and then there is going to be the description of the plugin creation process and the plugin submission process. After that there is going to be a brief analysis about each of the main features.

*piStar Tool* has a user base with different backgrounds. That is why the tools chosen in this chapter were picked from different areas and with different characteristics. The tools go from code editors, passing to audio and image editors, to complete ecommerce platforms. There are open-source tools and non-open source, as well as web-based and not web-based. Some of them are built for users without any programming background while others have its user base only programmers. The intention was to get a feel of how things are done in different areas. Each section is entirely based on the initial documentation provided in the official websites, so it can reflect how easily a user who wants to build a plugin can get started.

## Visual Studio Code

### Introduction

Visual Studio Code is a cross-platform code editor built by Microsoft that covers the common development cycle of editing, building and debugging. Beyond that it has an extensible architecture model, allowing the community to add more features to the tool. These extensions can be one of the following types: language packs, extension (Typescript), extension (Javascript), color themes, language support, code snippets and keymaps [9].

### Plugin Creation Process

First we have to make sure to have *Git* and *Node.js* installed. To facilitate the process of creating the extensions project, it is used a *NPM* module called *Yeoman.*

Yeoman is a scaffolding tool that provides complete project templates or just useful parts of it. The user must select the extensions type desired and provide additional information depending on the first selection. The generated template provides the folder structure below. [10][11]



The function of each of main directories and files are described below:

* *istar*: is the root folder named after the extension identifier provided to *Yeoman.*
* *.vscode*: contains .json configuration files for launching and debugging the extension as well as building the task that compiles the chosen language, in the example above, Typescript.
* *.src/test*: contains the test scripts.
* *.src/extension.ts*: is the main file, where the plugin functionality is written. It has a function called *activate*, called when the plugin is activated. The function *deactivate* is called when the plugin is deactivated.
* *vsc-extension-quickstart.md*: contains further explanation about the files, instructions to run tests and links to further information.
* *package.json*: contains all the information provided when creating the scaffold with *Yeoman* , basic scripts and information about the dependencies used. It is worth noting that *Visual Studio Code* uses the following extension ID pattern *<publisher>,<name>.*

### Plugin Submission Process

It is possible to test locally the plugin created just copying the root directory to *.vscode/extensions*, that is a directory created in the local system when *Visual Studio Code* is installed. To make the plugin accessible to the community, it can be submitted to *Visual Studio Marketplace* using an specific *CLI* to do it. *Visual Studio Code* uses *Azure DevOps* deal with authentication, hosting and management of the extensions, so the user has to create an *Azure* organization [12]. With the account created, the user must use a *CLI* to submit the plugin.

### Installation and Usage Process

There is a Extensions tab on the left side panel where the user can search the desired plugins available in *Visual Studio Marketplace,* by keywords*.*

**

When a plugin is selected from the list, a description page appears in the right showing the name, authors, download count and details about the usage.

**

To install, the user only must click the green button *Install.*

Plugins that are not only language helpers may need some additional step to use. The Figure 5 shows an example of that.

**

### Conclusions

The creation process requires *Typescript* knowledge, so users without and *IT* background will have a bigger barrier to write a new plugin. Its main user is a programmer, so it is not uncomfortable to them to use a *CLI* to package and submit a new plugin. In *piStar Tool*, the main user is not going to always have an *IT* background and therefore not necessarily comfortable using a *CLI*. The installation process is very easy to use and in most cases don't require the reinitialization of the tool, in order to start using the new plugin. So it is valuable to create a submission process that is going to be comfortable to users without *IT* background.

## Google Docs

### Introduction

*Google Docs* is an online based text editor that is part of Google Drive's office applications suite. The main features are: different users can edit the same document at the same time, automatic saving, compatibility with *Microsoft Word* and the capability of extending the application with new plugins. It is worth noting that a lot of the following overview is valid to all the other Google Drive’s office applications suite, and there are some peculiarities that are specific to just one of them [14].

### Plugin creation process

*Google Docs* has its own script editor to build plugins using *Javascript, HTML* and *CSS*. First is necessary to be logged in a *Google* account and to create a new *Google Doc*. From within the new document the user can open the *Script Editor* and create a new plugin project*.*

**

In this work is going to be used a simple plugin that lets the user search a There are two main parts of a simple plugin project. One of them will use *HTML, CSS* and *Javascript*  to deal with the interface of the plugin and its changes. The other one will deal with the specific *triggers* provided by the tool’s framework, and they will run when the one of the following events occur: a *doc* is opened, the plugin is installed, or an specified time is reached [15][16][17].

The Figure 7 shows an example of a plugin that allows the user to search for a specific map location and add it to a document, generated by the code in Figure 6.

**

### Plugin submission process

All the process is done within the *Apps Script Editor* or in *GCP* console. The first step is to create a new version of the add-on. It consists of a static version of the script and a description provided by the user. The next steps are responsible for configuring an *oAuth* consent screen that is prompted for the plugins user, getting *GCP*’s project key, the plugins version number, the localized assets for each language the user wants to publish, the visibility and other basic information [20][21].

### Installation and usage process

The publicly published plugins are all available at *GSuite Marketplace*. There is a sidebar that displays some plugin categories and a search bar.

**

In the description page is shown some screenshots, reviews the users count and the rating of the selected plugin. To install the plugin it is necessary to press the install button that appears on the top and right side of the page, then a authorization page is prompted asking the user to choose the *Google* account to be used in the installation.

**

Another screen is shown confirming the installation and showing how to open the new plugin that is available in the *Add-ons* list in the top menu of a document.

### Conclusions

The process of creation of a new plugin requires knowledge in *Javascript, HTML* and *CSS* code, turning this feature almost inaccessible for users with no *IT* background to use. The plugin submission process is not very straightforward and simple, it requires reading of some documentation. The installation process is very intuitive and simple, very similar to other extensible tools, providing a page to search and a install button that makes the plugin available for usage right away. The usage will depend on each plugin functionality.

## Sketchup

### Introduction

*Sketchup* is a 3D modelling software owned by *Trimble Navigation*. Although very well known in Architecture community, it is also used to create models for 3D printing and carpentry. The functionalities of *Sketchup* can be extended with one of the 757 plugins available for it in the *Extension Warehouse* [22].

### Plugin Creation Process

A simple *Sketchup* plugin package consists of two *.rb* files. The first one is called the *root* file and . The second is the one that describes the plugin functionality itself. It is called *main.rb* and makes use of the *Sketchup Ruby API* (*Application Program Interface*)[]. The *API* allows the user to interact with *Sketchup* models and with components of the application. The suggested *root* file structure is the following.

|  |
| --- |
| module PublisherName  module ExtensionName  # ...  end  end |

Figura 10 - *root* file structure (Source: [25])

It encapsulates all the plugins from the same author in its own module. This *module* is recommended and enforced to prevent new plugins to affect already installed ones.

There are three main requirements for a plugin to be accepted in *Extensions Warehouse:*

* Inside the *root* file has to carry a register of a *Sketchup Extension* instance.
* The *root* file should only carry a plugin registration and nothing more.
* Each plugin should be confined in its own *module*, therefore the use of global variables, constants or methods are forbidden as well as any *Sketchup API* and *Ruby API* modifications.

The following plugin is an example available at []. It allows the user to add a simple cube to the current *model*. In the *root* file are loaded *sketchup.rb* and *extensions.rb* files, that are needed to make the plugin *registration*. Inside the *Examples* module it is registered only one plugin called *HelloCube*.

|  |
| --- |
| require 'sketchup.rb' require 'extensions.rb'  module Examples  module HelloCube   unless file\_loaded?(\_\_FILE\_\_)   ex = SketchupExtension.new('Hello Cube', 'tut\_hello\_cube/main')  ex.description = 'SketchUp Ruby API example creating a cube.'  ex.version = '1.0.0'  ex.copyright = 'Trimble Navigations (c) 2016'  ex.creator = 'SketchUp'   Sketchup.register\_extension(ex, true)   file\_loaded(\_\_FILE\_\_)  end  end end |

Figura 11 - *root* file of *HelloCube* plugin. (Source: [25])

The first line checks if the plugin is already registered, to prevent it to being registered more than one time. After that, a new *SketchupExtension* instance is created. It takes two arguments: the plugin name and the location of the *main.rb* file. In the following lines some other properties of *SketchupExtension* like *description*, *version*, *copyright* and the plugin's *creator* name. The *registration* itself is made in the following line. The *Sketchup.register\_extension* method takes as argument a *SketchupExtension* and a boolean that tells *Sketchup* to load the plugin by default if this boolean is set to *true*. If not, the user has to load the plugin manually. The last line of the plugin *module* is needed for the load guard to prevent the *registration* of the same plugin more than one time.

|  |
| --- |
| require 'sketchup.rb' # 1  module Examples # 1  module HelloCube # 1   def self.create\_cube # 2   model = Sketchup.active\_model # 3   model.start\_operation('Create Cube', true) # 4   group = model.active\_entities.add\_group # 5  entities = group.entities # 6   # 7  points = [  Geom::Point3d.new(0, 0, 0),  Geom::Point3d.new(1.m, 0, 0),  Geom::Point3d.new(1.m, 1.m, 0),  Geom::Point3d.new(0, 1.m, 0)  ]   face = entities.add\_face(points) # 8   face.pushpull(-1.m) # 9   model.commit\_operation # 10  end  # 11  unless file\_loaded?(\_\_FILE\_\_)  menu = UI.menu('Plugins')  menu.add\_item('Create Cube Example') {  self.create\_cube  }   file\_loaded(\_\_FILE\_\_)  end   end end |

Figura 12 - *main.rb* file of *HelloCube* plugin. (Source: [25])

In the *main.rb* file is declared the function that creates the cube in *#2*. Using the *Sketchup API* it is possible to access the current open *model* in *#3*. Each action performed is between the calling of *start\_operation()* in *#4* and *commit\_operation()* in *#10*. These two functions wrap everything that can be reversed in a single *undo* step. The variable *points* in *#7* defines a set of four points that are the vertices used to build the faces of the desired cube in *#8*. Outside *self.create\_cube* will be created a new menu item that calls the method when clicked in *#11*.

### Plugin Submission Process

The submission requires a *Google ID* and is necessary to be approved to become a developer capable of submitting new plugins. After the approval, the developer must fill out a form with information about the plugin and upload the *.rbz* package. In the form the developer must provide the following information:

* Languages supported by the extension.
* Minimum system requirements.
* Sketchup supported versions.
* Dependencies.
* Documentation.
* Contact for support.

A *.rbz* package is a *.zip* file with a defined structure that contains the *.rb* scripts and possible assets. It is recommended to the documentation to have the following structure:

* A clear guide to get started with menu names and screenshots.
* Usage video.
* Sample files.

A final step before the plugin goes live is the review made by the *Sketchup* review team. They will check if the submission follows the requirements and if it does not, provide some feedback to the developer [26].

### Installation and Usage Process

*Sketchup* provides the *Extension Manager* as tool to install and manage plugins. The plugins available are listed in *Extension Warehouse*. The *Extension Warehouse* homepage displays a list of categories and industries, as Figure 13 shows and only when one of these links are clicked a plugin list is displayed [23].

**

The individual plugin page displays a description and some screenshots. It also displays the number of users, reviews and technical information like the size, compatibility and version. To install the plugin, the user must download the plugin package and use *Extension Manager* to install it [27][28].

**

### Conclusions

The plugin creation requires programming knowledge of *Ruby*. Apart from that the project structure is very simple and easy to understand. The *Sketchup API* and *Ruby API* have dense documentations hence *Sketchup* is a complex graphic software. It is not easy for a user without programming background to start developing plugins for the software. In compensation the plugin submission process is very easy if the plugin package have the right structure and the form have all the required information. The installation and usage process are also very simple and similar to other extensible tools in the market, providing a store to easy navigate through existing plugins and not requiring any programming background to install it. Although the plugin cannot be installed directly from the store, requiring the user to download a file and load it in *Sketchup* using the *Extension Manager*.

## StarUML

### Introduction

*StarUML* is a cross-platform software modelling tool compatible with *UML (Unified Modelling Language)* version 2 standard metamodel and diagrams. Additionally it also supports Entity-Relationship, Data-flow and Flowchart Diagrams [29].

### Plugin Creation Process

First the user must create a folder structure as shown in Figure 15. In the *main.js* is the entry point of the plugin: the *init()* function. *StarUML* provides an open *API* to allow the access to the application components. The plugin example in this section opens an alert window when a command is pressed.



The *API* allows a plugin to be executed from the click on a menu item or from a keyboard shortcut. To call the plugin from anyone of those is necessary to register a command. That is accomplished using the instance *app* of *AppContext* class shown in Figure 16. The function *app.commands.register* receives the name of the command and the code block to be executed.

|  |
| --- |
| function handleShowMessage () { window.alert('Hello, world!') } function init () { app.commands.register('helloworld:show-message', handleShowMessage) } exports.init = init |

Figura 16 - Plugin that opens a message box. (Source: [30])

To add the same plugin to a menu is necessary to create a *JSON* file inside the *menu* folder created in the structure above. The file contains informations like the plugin name, that will appear inside the menu, and the command that it fires when clicked. The same pattern is followed to register a keyboard shortcut inside a *JSON* file in the *keymaps* directory.

|  |
| --- |
| { "menu": [ { "id": "tools", "submenu": [ { "label": "Hello World", "id": "tool.helloworld", "command": "helloworld:show-message" } ] } ] } |

Figura 17 - Declaration of a new menu shortcut that runs a certain command (Source: [30])

The elements in a project can be created, accessed and modified using the instance of the current project through *app.project.getProject()*. The *API* documentation provides an extensive documentation on the functions to access each of the elements present in a project and how to add elements to a project [].

### Plugin Submission Process

There are three ways of distributing a plugin for *StarUML*. It is possible to zip the extension folder and unzip in the extensions folder of the application. Another option is to store the extension in a *Github* repository, so users can install from Extension Manager using the *URL* as shown in Figure 18.

The third option is to register the plugin in *Extension Registry* to allow users to install directly from *Extension Manager*. To do that it is necessary to fill out the *package.json* with the following information:

* Plugin name
* Title
* Description
* Homepage
* Issues
* Keywords
* Version
* Author - name, email and *URL*
* License
* Engines

It is also necessary to upload the plugin to *Github* with a *README.md* file containing any useful information and user manual about the extension. With these steps done, the developer must create a release on *Github* and then send an email to the support team of *StarUML* and the plugin will be available in up to two business days. [30]

### Installation and Usage Process

As seen in section 3.4.3, the installation can be done by unzipping the plugin package inside the extensions folder on the application, using *Extension Manager* with the repository *URL* or directly searching for registered plugins. The user can search and read short descriptions inside the *Extension Manager* window. The usage depends on the plugin installed.



### Conclusions

The plugin creation process requires programming knowledge in *Javascript*. Also, the documentation of the *API* is extensive. These two facts can get away users without any programming background.

The three options of distribution are a good idea to give the developer more options. All of the three ways are easily done, although users without programming background can few a little uneasy in the options that require the use of *Github*.

The *Extension Manager* makes the installation process very easy. Also it allows the user to install plugins that did not get through the *Extension Registry* process. It can cause problems if the user tries to install plugins that are not tested, but at the same time cuts one more barrier to developers to distribute their app, which is a good thing [30][31].

## Google Chrome

### Introduction

*Google Chrome* is a very popular browser launched in 2008 by *Google*. It is also cross-platform and extensible with plugins available in the *Chrome Web Store*.

### Plugin Creation Process

As occurs with *Google Docs* plugins, seen on section 2.2 of this work, a *Google Chrome* plugin consists of a package containing *HTML, CSS, Javascript*, images and other necessary files. A *Google Chrome Plugin* can have the following components:

* Manifest
* Background Script
* UI Elements
* Content Script
* Options Page

The Manifest consists of a *.json* file that holds informations like name, version, permissions and important files that it might have to use. The Background Script contains all the listeners that handle the browser events. The interface is handled by the *UI (User Interface)* Elements. The Content Script is only used in plugins that interact with a loaded page, modifying the *DOM (Document Object Model)*. In the Options Page it is possible to customize the plugin if needed.

A plugin *UI* can have the following elements:

* Popup
* Tooltip
* Omnibox
* Context Menu
* Commands

The *Popup* consists in a *HTML* file that is shows up when the user click in its icon placed in the toolbar . It works like a webpage, accepting *HTML, CSS* and *Javascript* with some limitations.

**

The *Tooltip* is used to display some information about the plugins usage when the user hovers the mouse over the icon in the toolbar as in Figure 20. It is also possible to implement internationalization creating specific directories for each language with the translation of the information in the *\_locale* folder.

**

The *Omnibox* is a way of invoking the extensions functionality searching for a specific keyword registered in *manifest.json*.

**

A plugin can also be invoked in the *Context Menu* that appears when the user clicks with the right button of the mouse in specific areas. It consists in a Menu that displays options based on the context of the place that was clicked, for example, a selection of a text as in Figure 22.

**

It is also possible to invoke a plugin through an specific command specified in *manifest.json*. A command is a set of keys that have to be pressed one after the other.

|  |
| --- |
| {  "name": "Tab Flipper",   "commands": {  "flip-tabs-forward": {  "suggested\_key": {  "default": "Ctrl+Shift+Right",  "mac": "Command+Shift+Right"  },  "description": "Flip tabs forward"  },  "flip-tabs-backwards": {  "suggested\_key": {  "default": "Ctrl+Shift+Left",  "mac": "Command+Shift+Left"  },  "description": "Flip tabs backwards"  }  }  } |

Figure 23 - "commands" field in the *manifest.json.* (Source: [34])

The example below shows a plugin that changes the background of the page in the browser. First there is the *manifest.json* that holds some information about the plugin and sets the *popup.html* file *as the file* where the *UI* is declared. It also specifies the location of icons and the permissions to be used.

|  |
| --- |
| {  "name": "Getting Started Example",  "version": "1.0",  "description": "Build an Extension!",  "permissions": ["declarativeContent","storage"],  "background": {  "scripts": ["background.js"],  "persistent": false  },  "page\_action": {  "default\_popup": "popup.html",  "default\_icon": {  "16": "images/get\_started16.png",  "32": "images/get\_started32.png",  "48": "images/get\_started48.png",  "128": "images/get\_started128.png"  }  },  "icons": {  "16": "images/get\_started16.png",  "32": "images/get\_started32.png",  "48": "images/get\_started48.png",  "128": "images/get\_started128.png"  },  "manifest\_version": 2 } |

Figure 24 - *manifest.json* (Source: [34])

The *popup.html* contains a button and its styling.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <style>  button {  height: 30px;  width: 30px;  outline: none;  }  </style>  </head>  <body>  <button id="changeColor"></button>  </body>  </html> |

Figure 25 - *popup.html* (Source: [34])

After setting up some rules and giving the button a color [], it is time to setup the logic of the plugin. The following code adds an event listener that runs a *Javascript* code that changes the color of the page in the browser [32][33][34].

|  |
| --- |
| let changeColor = document.getElementById('changeColor');  changeColor.onclick = function(element) {  let color = element.target.value;  chrome.tabs.query({active: true, currentWindow: true}, function(tabs) {  chrome.tabs.executeScript(  tabs[0].id,  {code: 'document.body.style.backgroundColor = "' + color + '";'});  }); }; |

Figure 26 - Part of the code that holds the logic of the plugin (Source: [34])

### Plugin Submission Process

All plugins for Google Chrome are placed in *Chrome Web Store*. There are basically eight steps in the submission process [35].

* Create the package: a package for a *Google Chrome's* plugin consists in a *ZIP* file with at least the *manifest.json* file. The presence of other files will depend on the plugin functionality.
* Create a developer account: is suggested the creation of a new account specific for the app.
* Upload the app: the package upload is done through the *Chrome Developer Dashboard*. The system will check the validity of the *ZIP* file and of the *manifest.json* file. If the app needs to use its *app ID* or *OAuth token*, the upload must be done before finishing to app.
* Pick a payments system: only for paid plugins or developers that want to use *Chrome Web Store Payments*.
* Provide store content: the developer must provide a detailed description of the plugin, an icon for the store and at least one screenshot in specific sizes, the category and the language that the plugin uses.
* Pay the signup fee: it is a one time only payment that must be done before publishing the plugin.
* Publish the app: it is possible to publish for test groups first and then publish to the entire community. Once the plugin is published to the world it appears the search result on *Chrome Web Store*. It is also possible to control how fast the app can reach new users setting up a "*max deploy percentage*".

### Installation and Usage Process

To install a plugin it must be found in *Chrome Web Store*. The store is divided in two types: the extension store and theme store. In is possible to see recommended plugins, added recently, featured and browse by category. A search bar is also available as well as some filters like classification and resources.

**

Each plugin has its own page that is loaded when it is selected. In the page is available the plugins description, homepage, update and version information as well as user comments, support contact, suggestions of other related plugins, classification, number of users and its categories.

**

To install it on the browser the user must click in the button *Add* *to Chrome*. Before installation, the browser shows a popup listing the functionality of the plugin that needs permission of the user. If the user accepts and clicks *Add* extension the icon shows up immediately in the tooltip [36].

### Conclusions

The plugin creation process requires the use of *HTML*, *CSS* and *Javascript*, so users without a programming background have a great barrier to overcome. For a user with programming background the process is also not very easy due to the very extensive documentation of the library used to develop for *Google Chrome*.

The submission process is fairly easy compared to *Google* *Doc*s process, and does not require the use of any *CLI* or specific tools.. There is also a required fee for the developers to submit plugins, a fact that can put away some of them. Also the installation and usage are very simple and quick, very similar to other applications with a store and a automatic installation.

## Wordpress

### Introduction

*Wordpress* is an open source and web-based *CMS (Content Management System)* that is embedded in 34% of all websites of the world. It makes easy for anyone to build a website and start blogging. The plugins built for *Wordpress* range from ecommerce platforms to *SEO (Search Engine Optimization)* enhancers [37].

### Plugin Creation Process

A *Wordpress* plugin can consist of one *PHP* file. It can also include some assets like images, *CSS* and *Javascript* files. The example of plugin used in this section lets a user select a post title and retrieves the number of posts with the same title. The HTML file that will be used is shown in Figure 29 and its result in Figure 30.

|  |
| --- |
| </pre> <form id="radioform">  <table>  <tbody>  <tr>  <td><input class="pref" checked="checked" name="book" type="radio" value="Sycamore Row" />Sycamore Row</td>  <td>John Grisham</td>  </tr>  <tr>  <td><input class="pref" name="book" type="radio" value="Dark Witch" />Dark Witch</td>  <td>Nora Roberts</td>  </tr>  </tbody>  </table> </form> <pre> |

Figure 29 - HTML file (Source: [38])

**

To access *Wordpress* elements is used the *Heartbeat API*, that is a server polling *API* that is used to handle frontend updated. It works by setting up a time interval to run and get data to the server and wait the response. In Figure 31 there is declared a *jQuery* function that runs when the page loads. On the second line *jQuery*, now represented by *$* symbol, sets a listener to an event of change inside the class *pref*. That means that everytime any element inside a tag with a class equals to *pref* changes, the given function will run.

|  |
| --- |
| jQuery(document).ready(function($) {   $(".pref").change(function() {   var this2 = this;   $.post(my\_ajax\_obj.ajax\_url, {   \_ajax\_nonce: my\_ajax\_obj.nonce,   action: "my\_tag\_count",   title: this.value   }, function(data) {   this2.nextSibling.remove();   $(this2).after(data);   });  }); }); |

Figure 31 - *frontend part* (Source: [38])

Everytime a change happens inside an element with a class *.pref*, the function *$.post* will be called. This function receives as parameters the *URL* of the request, an object containing a string that represents the action that must be triggered in the server, the data that must be sent and a callback function that runs when the response is retrieved. In this case the data is the selected value passed to the *title* parameter, and the action is “my\_tag\_count” passed to the *action* parameter. If the action that will be triggered in the server changes in anyway the database, it is also needed a parameter *\_ajax\_nonce* with a unique hexadecimal serial number that identifies an instance of a form. Both the request *URL* and and the \_*ajax\_nonce* parameter are set by the *PHP* script and are available in a global object.

On the server side there will be two parts: the one who serves the *jQuery* script in Figure 31 with the needed values, shown in Figure 32, and the one who handles the requests shown in Figure 33. The first part is represented by the function *my\_enqueue*. Inside this function is called *wp\_enqueue\_script* that creates a meta link to the script in the page section. It is not possible to add a meta link without the use of *wp\_enqueue\_script* because the header is not available to direct access. It takes three parameters: an arbitrary tag to refer to the script file, the complete *URL* to the script and an array listing the other scripts that the script depends on, that in this case is *jquery*.

In *wp\_localize\_script* is where the localizing process is done. Localizing is the process of creating the global *jQuery* object that is used in the script of Figure 31. This function takes three parameters: the first is the arbitrary tag that refers to the script file that was set in *my\_enqueue*, the name of the global object and an array with the objects that the object contains. The elements or the array are the url that points to the *PHP* script and the *$title\_nonce* using *wp\_create\_nonce()* that generates the form identifier mentioned earlier.

|  |
| --- |
| <?php add\_action('admin\_enqueue\_scripts', 'my\_enqueue'); function my\_enqueue($hook) {  if( 'myplugin\_settings.php' != $hook) return;  wp\_enqueue\_script( 'ajax-script',  plugins\_url( '/js/myjquery.js', \_\_FILE\_\_ ),  array('jquery')  );  $title\_nonce = wp\_create\_nonce('title\_example');  wp\_localize\_script('ajax-script', 'my\_ajax\_obj', array(  'ajax\_url' => admin\_url( 'admin-ajax.php' ),  'nonce' => $title\_nonce,  )); } |

Figure 32 - *frontend part* (Source: [38])

|  |
| --- |
| add\_action('wp\_ajax\_my\_tag\_count', 'my\_ajax\_handler'); function my\_ajax\_handler() {  check\_ajax\_referer('title\_example');  update\_user\_meta( get\_current\_user\_id(), 'title\_preference', $\_POST['title']);  $args = array(  'tag' => $\_POST['title'],  );  $the\_query = new WP\_Query( $args );  echo $\_POST['title'].' ('.$the\_query->post\_count.') ';  wp\_die(); // all ajax handlers should die when finished } |

Figure 33 - *frontend part* (Source: 38[])

The second part of the server side is going to handle the received requests. In the example of Figure 33, the handler called *my\_ajax\_handler()* first checks the *nonce* using *check\_ajax\_referer()*. If the *nonce* does not check out the function just calls *wp\_die()* that finished the execution. The function *update\_user\_meta()* to save the user selection, represented by *$\_POS[‘title’]*, in user meta. The rest of the handler is going to create the response and send it back, first creating the data that must be sent back in *$args* and then actually sending *$the\_query* back [38].

### Plugin Submission Process

To submit a plugin for the store the developer must have an account in *Wordpress.org*. The developer must also provide a plugin name and a documentation describing what the plugin does, installation instructions and informations about support. Some instructions and a template for this *README.txt* documentation file is provided by *Worpress*. A form must be filled out by the developer with the following information:

* Plugin name
* Plugin description
* Plugin *URL* - a link to the *ZIP* file that contains the complete version of the plugin and includes the *README.txt*.

After the submission of this form, the developer will receive an email by *Wordpress* within 14 business days. These email is going to contain the result of the review. If approved, a link to a *SVN* (*Apache Subversion*) repository will be provided, so that the developer can host the plugin there. After that the plugin will be available in the store [39][40].

### Installation and Usage Process

*Wordpress* provide *Wordpress Plugin Directory* so that users can search for the available plugins.

**

The directory shows a list of plugins divided by category. Each individual plugin page contains the following information:

* Description
* Screenshots
* Authors and contributors
* Technical information (version, last updated, active installations and more)
* Tags
* Rating and Reviews
* Support information

The plugin can be installed directly from the *Wordpress Plugin Directory*. There is a possibility of downloading any plugin *ZIP* file and upload it in *Wordpress* or using *FTP (File Transfer Protocol)* [41].

### Conclusions

The plugin creation process makes it harder to users without programming background to start write plugins, because it requires knowledge of *Javascript*, some of its frameworks and *PHP*. It also requires the reading of an extensive documentation. The plugin submission is fairly simple, requiring the filing out of a small form. It has the downside of having its review process done within 14 business days. Although the process is not necessary to test the plugin.

The installation goes beyond other plugin stores of other applications and offers three different methods. Two of them does not require the submission of the plugin for review, which is good for developers that are just starting out and want to test the plugin.

## Eclipse

### Introduction

*Eclipse* is an *IDE* built by the *Eclipse Foundation*, widely used for *Java* development, but with support for other languages. It is built with focus on extensibility, providing a basic workspace and relying most of its features to plugins [42].

### Plugin Creation Process

The application provides a set of tools for plugin development called *PDE* (*Plug-in Development Environment)*. It helps the developer in from development to deployment phase providing editors, wizards views and a launcher.

A plugin for *Eclipse* consists of a manifest file and, in most cases, the source code written in *Java*. There are plugins that do need any *Java* code such as documentation plugins. This section is going to go through some steps of the development of a Hello World plugin described in []. A new project can be created through *PDE*’s *New Plugin Project* wizard. The developer must create a unique plugin id, the type of project, the directory for the *Java* builder output, the runtime library and the directory of the source folder.

**

In the second part of the wizard, the developer can choose to use a code generator wizard to create a project starting from a template, or to start from a blank one, as shown in Figure 36. In the example analyzed in this section a blank project will be created.

**

**

The *plugin.xml* file describes the plugin content. It contains the following information:

* Basic information like unique identifier, version and name.
* Dependencies section: describes the plugin needs to compile.
* Extensions section: lists the functionalities that the plugin adds to the platform.
* Runtime section: lists the libraries that the plugin code is packaged.
* Extension Points: lists points where other plugins can use to extend its functionality.

In Figure 38 is the code of a plugin that opens a window titled *First plug-in* containing a *Hello, world!* message.

|  |
| --- |
| package com.example.helloworld;  import org.eclipse.jface.action.IAction; import org.eclipse.jface.dialogs.MessageDialog; import org.eclipse.jface.viewers.ISelection; import org.eclipse.ui.IWorkbenchWindow; import org.eclipse.ui.IWorkbenchWindowActivationDelegate;  public class HelloWorldAction implements IWorkbenchWindowActionDelegate {  private IWorkbenchWindow window;   public HelloWorldAction() {   }   public void run(IAction action) {  MessageDialog.openInformation(  window.getShell(),  "First plug-in",  "Hello, world!"  );  }   public void selectionChanged(IAction action, ISelection selection) {   }   public void dispose() {   }   public void init(IWorkbenchWindow window) {  this.window = window;  } } |

Figure 38 - Plugin that opens a window (Source: [43])

**

### Plugin Submission Process

Before the deployment, a package of the plugin must be created. A package of a plugin for *Eclipse* of a *ZIP* file. *PDE* also provides a wizard to guide the packaging process. The process is guided by information about the folders that should be included in the package. Those informations are provided by a configuration file called *build.properties*.

**

To publish a new plugin in *Eclipse Marketplace* is necessary an account. Once logged in in the *Marketplace* the user must create a new listing and fill out a form with the following fields:

* Solution name
* Organization name
* URL
* Logo
* Description
* Development status
* License type
* Support URL
* Categories that the solution fits in.
* Solution type (*Markets*)
* Tags
* Solution version
* Screenshot
* Ownership (authors)

After the submission the solution must pass through a review that should determine if it is eligible to be published or not [45].

### Installation and Usage Process

A user can install a recently developed plugin to test without the submission to *Eclipse Marketplace*. To do this it is necessary to unzip the plugin package directly into the *plugins* folder of the application. To install from *Eclipse Marketplace*, it is necessary to instal the *Marketplace* client using the *Install new software* option in the help menu. The client allows the user to search all available and submitted plugins by name, categories, also listing the recent and most popular submissions [44].

**

### Conclusions

*Eclipse* plugin development still requires knowledge about *Java* programming knowledge, letting out any user without programming background. A very interesting thing about the plugins generated for *Eclipse* are the *Extension Points* that can be provided so that other developers build upon an existing plugin.

To submit a plugin to the *Eclipse Marketplace* the user must fill out a form and provide the generated package provided by the wizard mentioned above. It needs a review period that is around 24 hours. The review can ensure the quality of the plugin, but at the same time creates one more barrier to the user.

The installation of the plugin is very easy done using the *Marketplace* client, although the installation of the client is not usual. The usage will depend on the plugin installed.

## Audacity

### Introduction

*Audacity* is an open-source and cross-platform multi-track audio editor. Among other common audio editor features, it is an extensible application, allowing the use of audio effect plugins with some *API*’ like LADSPA, LV2, *Nyquist* and *VST* [46].

### Plugin Creation Process

As mentioned above, *Audacity* accepts a range of different technologies to build plugins. This section is going to focus on the development of plugins using *Nyquist*, that is a section also present in Audacity Manual.

*Nyquist* is a programming language for audio synthesis and analysis that supports *MIDI*, audio recording and playback, file I/O, object-oriented programming, profiling and debugging. A plugin only consists in a text file with a .*ny* extension and with code written in *Nyquist*.

|  |
| --- |
| ;nyquist plug-in  ;version 4  ;type process  ;name "Fade In"  (mult (ramp) \*track\*) |

Figure 42 - Example of a plugin code in *Nyquist* (Source: [48])

The first and second line of the code are required in this order. They declare that it is a *Nyquist* plugin and its version respectively. The third line declares the type of plugin that can be one of the four: *generate, process, analyze, tool*. The type will define in which menu of the application the plugin will appear. In the list below are the differences between each one.

* *generate*: for plugins that will generate new audio.
* *process*: for plugins that modify existing audio. It is placed in *Effects* menu.
* *analyze*: for plugins that only process audio without modifying it.
* *tool:* plugins that do not fit in any of the categories above.

The setup of a type is important because, for example, for plugins in of the *process* or *analyze* type, *Audacity* let the selected piece of audio selected available for *Nyquist* in a environment variable. The commands in the file are executed in order and must return a value. This value replaces whatever is selected by the user.

|  |
| --- |
| ;control decay "Decay amount" int "dB" 6 0 24 ;control delay "Delay time" float "seconds" 0.5 0.0 5.0 ;control count "Number of echos" int "times" 5 1 30  (defun delays (sig decay delay count)  (if (= count 0)  (cue sig)  (sim (cue sig)  (loud decay (at delay (delays sig decay delay (- count 1)))))))    (stretch-abs 1 (delays \*track\* (- 0 decay) delay count)) |

Figure 43 - Example of a plugin code in *Nyquist* that opens a dialog box and use the user input values to perform its task. (Source: [48])

A *process* or *generate* type plugin can open a dialog box, for example, that allow the user to enter input some parameters to setup the plugin execution. Another example can be a plugin of type *analyze* returning a list of tuples with pairs of time and labels that *Audacity* uses to create a track label in each of these time positions.

**

### Plugin Submission Process

*Audacity* does not have a central repository or store that aggregates all the plugins, therefore the submission process does not exist. In [] there is a list of plugins that already are shipped with *Audacity* and a list of authors that contributed with it. To have a plugin listed its necessary to register on *Audacity Forum* and post the plugin for review. The plugin must follow the recommendations listed in a forum post called *Convention for Nyquist Plugins* []. As it is a discussion, the recommendations are not listed in a formal way. Each user that wants to contribute can just make a comment [48] [49].

### Installation and Usage Process

The installation process will depend on the *API* or language that the plugin was built. In the case of *Nyquist* plugins, the user can add the plugin directly in a specific directory or use a tool called *Nyquist Plugin Installer*. After the installation the plugin will appear in the menu specific for its type ready to use [48].

### Conclusions

The creation of the plugin requires the use of a less common language called *Nyquist*. Although it is possible to write a plugin in under ten lines of code, it would be a barrier for users without a programming background.

A plugin submission can be very easily done, since there is not a central place that gathers all the available plugins. To have a plugin listed in the website, it would have to pass a review done by the community. This decentralization in the distribution makes it a little harder for a user to find a plugin.

The *Nyquist Plugin Installer* turns the installation process very easy, but it is not as easy as other applications that let the user search and install from the store page of the plugin. The usage functions is very similar to other applications, letting the plugin available in a menu that is separated by category, making it easy to be found by the user.

## *GIMP* (GNU Image Manipulation Program)

### Introduction

*GIMP* is a cross-platform image editor. It is possible to get its source code and modify creating unique variations of the application, since it is an open-source tool. The editor delivers common tools available in other applications of the same type. Selection Tools like *Lasso* and *Magic Wand*, Paint tools like *Brushes* and *Gradient* and also basic Transformation tools like *Crop*, *Align* and *Scale*, are features present in other famous editors on the market [50].

### Plugin Creation Process

*GIMP* has a main interface that connect a plugin to the application core called *PDB (Procedural Database)*. It manages all the communication between those two parts. All the code is written in *C*. To build a plugin is necessary to use the *libgimp* library and a *CLI* tool called *gimptool*. A plugin generally serves for one of the following three purposes:

* Modify image data: receive image, modify and return the modified version.
* Generate an image.
* Process image without modifying.

Basically a plugin has four lifecycle functions. The *init()* and *quit()* functions are called respectively when *GIMP* application is open and when it is going to close. The *query()* function is called every time a plugin changes. The *run()* function is where the plugin performs its desired task. It is called when the plugin is asked to run.

The example below demonstrates a simple plugin that only open a window containing a message. In Figure 45, inside the *query()* function, the variable of type *GimpParamDef* holds the type, name and a description of the parameter. After that the function *gimp\_install\_procedure()* is called to declare some information of the procedure like its name and description. In the same function is worth noticing that *GIMP\_PLUGIN* is a constant that declares that the procedure to be external, that means that it will be not executed inside the *GIMP* core, therefore acting as a plugin. The last function called registers the plugin in the top Menu of the application, specifying the path *Filters/Misc* as shown in Figure 46.

|  |
| --- |
| static void query (void)  {  static GimpParamDef args[] = {  {  GIMP\_PDB\_INT32,  "run-mode",  "Run mode"  },  {  GIMP\_PDB\_IMAGE,  "image",  "Input image"  },  {  GIMP\_PDB\_DRAWABLE,  "drawable",  "Input drawable"  }  };   gimp\_install\_procedure (  "plug-in-hello",  "Hello, world!",  "Displays \"Hello, world!\" in a dialog",  "David Neary",  "Copyright David Neary",  "2004",  "\_Hello world...",  "RGB\*, GRAY\*",  GIMP\_PLUGIN,  G\_N\_ELEMENTS (args), 0,  args, NULL);   gimp\_plugin\_menu\_register ("plug-in-hello",  "/Filters/Misc");  } |

Figure 45 - *query()* function (Source: [51])

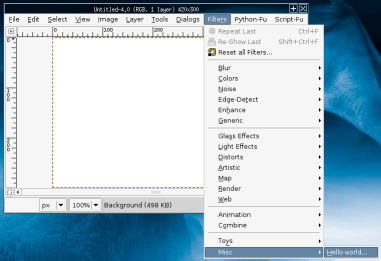


Figure 46 - Plugin path in the application Menu. (Source: [51])

The next necessary function is *run()*. The plugin can run in one of the three available modes, called *run\_mode* declared by the following constants: *GIMP\_RUN\_INTERACTIVE*, *GIMP\_RUN\_NONINTERACTIVE* and *GIMP\_RUN\_WITH\_LAST\_VALS*. The first one let the user use an options dialog opened, when the plugin is selected from the Menu, to configure the necessary parameters before running the plugin effectively. The second and the third do not open any dialog window and run directly from a script or a batch. The result of calling this simple plugin is shown in Figure 48 [51].

|  |
| --- |
| static void run (const gchar \*name,  gint nparams,  const GimpParam \*param,  gint \*nreturn\_vals,  GimpParam \*\*return\_vals)  {  static GimpParam values[1];  GimpPDBStatusType status = GIMP\_PDB\_SUCCESS;  GimpRunMode run\_mode;   \*nreturn\_vals = 1;  \*return\_vals = values;   values[0].type = GIMP\_PDB\_STATUS;  values[0].data.d\_status = status;   run\_mode = param[0].data.d\_int32;   if (run\_mode != GIMP\_RUN\_NONINTERACTIVE)  g\_message("Hello, world!\n");  } |

Figure 47 - *run()* function (Source: [51])



Figure 48 - Plugin that opens a dialog box showing a message. (Source: [51])

### Plugin Submission Process

*GIMP* does not have a repository that centralizes all available plugins. Therefore there is no need of submission. To share a plugin with the community the author just have to make it available for download anywhere on the internet.

### Installation and Usage Process

As *GIMP* does not have a central repository, it is necessary to rely on website lists of some plugins and search engines to find them. Once the desired plugin is found, is necessary to check the form it is distributed. Some plugins come with an installer, making the installing process free of any necessity of knowledge about using a *CLI* or any programming language. Others require the user to copy the downloaded script to the plugins folder of *GIMP*. If the script is in a programming language other than *C*, the user has to make sure that the support for that language is already installed in *GIMP*.

The usage of each plugin depends on its *run\_mode* and where the author chose to register it on the application Menu. Some plugins may open an options dialog to provide some additional configuration to the user before effectively running the script and others run directly when selected in the Menu [51].

### Conclusions

*GIMP* has a great barrier for authors without any programming background. It requires knowledge on *C* programming language. Since it is an image editor, it also requires additional knowledge about computer graphics to navigate through its documentation. The plugin submission process is inexistent, given that there is no central repository. The installation has no standard, it depends on the plugin provider. Some can provide an installer to facilitate the process, but others may require an *IT* background what creates a barrier for the majority of users.

## Shopify

### Introduction

*Shopify* is an online ecommerce platform with more than 800k stores built with it. It delivers a complete solution for building a store online very fast and without prior programming background. *Shopify* also is an extensible application, allowing developers to add features that help every aspect of a store like marketing, finance, design, shipping and many more. It is an interesting tool to analyze because it is aimed to users without any programming background. [53]

### Plugin Creation Process

This section is going to focus on a specific type of plugin that is embedded on the *Shopify Admin* interface. *Shopify Admin* is the interface that store owners use to control all of the operation, see statistics, add products, manage finance among other things.

*Shopify* delivers a product component library called *Polaris* that help developers to make the plugin integrate seamlessly into *Shopify Admin* interface. *Polaris* is a library of components built to use with *React*, a *Javascript* library for user interfaces development. There is no template specific to a *Shopify* plugin provided, only a general scaffolding of a *React* application generated by the use of *Next.js*. A simple page can be created just by adding a *React* component to *index.js* file.

|  |
| --- |
| const Index = () => (  <div>  <p>Sample app using React and Next.js</p>  </div> );  export default Index; |

Figure 49 - *React* component that renders a *HTML* *div* tag with a paragraph containing a message. (Source: [54])

To run this simple plugin inside *Shopify Admin* it is necessary to expose the environment to a public address. The example shows the setup to use a tool called *ngrok*, a tool that is able to expose a local server securely to the public internet.

|  |
| --- |
| ngrok http 3000 |

Figure 50 - *ngrok* command to create a public interface using port 3000 (Source: [54])

To use the plugin exposed by the public interface created, the developer must create a new application inside its *Shopify* developers account. The developer must provide an application name, the public *URL* and whitelisted redirection *URL's* provided by *ngrok*. This creation process will create a *Shopify API key* and a *Shopify API secret key.* It is recommended to store them in the *.env* and *process.env* to make them available.

**

It is also necessary to use an authentication system. In this example will be used *OAuth* setup in a *Node.js* server that is going to run on port 3000. After this everything is setup to authenticate and test the plugin inside *Shopify Admin*.

**

The current example only displays a message on the screen inside the *Shopify Admin*. For the plugin to make use of *Shopify UI* components is used a the *Polaris* library.

|  |
| --- |
| import { EmptyState, Layout, Page, ResourcePicker } from '@shopify/polaris';  const img = 'https://cdn.shopify.com/s/files/1/0757/9955/files/empty-state.svg';  class Index extends React.Component {  state = { open: false };  render() {  return (  <Page  primaryAction={{  content: 'Select products',  }}  >  <ResourcePicker  resourceType="Product"  showVariants={false}  open={this.state.open}  onSelection={(resources) => this.handleSelection(resources)}  onCancel={() => this.setState({ open: false })}  />  <Layout>  <EmptyState  heading="Select products to start"  action={{  content: 'Select products',  onAction: () => this.setState({ open: true }),  }}  image={img}  >  <p>Select products and change their price temporarily</p>  </EmptyState>  </Layout>  </Page >  );  }  handleSelection = (resources) => {  this.setState({ open: false })  console.log(resources)  }; }  export default Index; |

Figure 53 - *index.js* using *UI* components from Polaris. (Source: [54])

The code in Figure 53 uses an *EmptyState* component that displays a view with explanations on how to start using the plugin. This component contains a button that when clicked opens a *Polaris* component called *ResourcePicker*, resulting in the screen on Figure 53.

**

### Plugin Submission Process

The submission is done through *Shopify*'s platform. As shown in section 2.10.4 to test plugin the developer must create it inside its developer account, and this plugin created can be updated and published. The developer must be logged in to create a new *Shopify App Store listing*. To create the listing submission, the developer must fill out a form providing a description highlighting the features of the plugin, chose a category, set pricing information and specify which merchants can use the plugin.

### Installation and Usage Process

The installation is done directly from *Shopify App Store*. The store homepage lists plugins dividing by category and popularity. The user can also search by name, keyword or category using the search bar.

**

The plugin page shows a description and screenshots of the app. It also displays information about pricing for paid plugins. To add a plugin the user must only click the Add button.

**

### Conclusions

The creation process requires the knowledge of the three main areas of software development: front-end using *React*, back-end using *Node.js* to build a server and a little bit of *devops* in the use of *ngrok* to expose the local server to the public internet. Even users with a development background can have difficulties in the development if their programming background does not cover front-end reactive programming and back-end.

The submission process is done through *Shopify's* platform and does not require any specific *CLI* tool, which is good for users without a programming background. The installation is done directly from the *Shopify*'s *App Store* and the usage depends on the type of plugin.

# p*iStar - Plugin Submission Tool*

## Introduction

The project of an plugin submission application for *piStar* *Tool* was planned to provide the features given by state of the art applications like the ones analysed in Chapter 3. pi*Star - Plugin Submission Tool* that has the goal to make *piStar Tool* easier to extend, allowing any *i\** extension developer to make its extension available for all the community in a centralized way.

The developers of an extension must have an account in *piStar - Plugin Submission Tool* to send new plugins. To send a new plugin the developer is required to fill out a form with the following information:

* Plugin name
* Keywords - one of the ways of representing the plugin subject.
* Authors - emails used to register in the platform of each author.
* Short description - the plugin in a sentence.
* Long description - a longer description that describes to the user each functionality of the extension and its rules.
* References - books/articles/websites where the theoretical work present in the plugin can be found.
* Homepage link - link to plugin homepage if it exists.
* Category - category that the plugin fits in [repository].
* Plugin package - *ZIP* file containing the package with a folder structure following the rules described in Chapter 2.

The developers can see a list of submitted plugins that he is cited as an author. An author can also delete a submission to make the plugin unavailable.

The following sessions are going to start describing the application requirements through *User Roles* and *User Stories*. After that, the architecture and technologies used in the development are going to be explained to make it more accessible for any contributors that might want to build more features into the application.

## User Roles

### Administrator

Is the *Administrator* of the system and needs additional privileges. An *Administrator* has the power to *Edit* any information of any plugin submitted or even *Delete* the whole plugin.

### Non-IT User

Is the *User* with an *IT* background and therefore is not comfortable using tools that users from that area are used to. A *Non-IT* user needs the submission process to be as easy and far from common developer specific tools as possible.

### IT User

Is the *User* with an *IT* background and therefore is comfortable with developer specific tools.

## User Stories

As seen in the section 4.1, users were divided in three categories: *Administrator*, *Expert* and *Novice*. The list below is going to describe their *User Stories.* First is going to be listed the common *User Stories* for all the users and then specific ones.

### Common User Stories

#### Register

As an **USER** I want to be able to **create a** **new account** so that I can access the system and perform the actions related to my role.

#### Login

As an **USER** I want to be able to **login** in the system so that I can access the system and perform the actions related to my role.

#### List of plugins

As an **USER** I want to be able to **see a list of plugins** so that I can perform actions related to plugins according to my role.

#### Logout

As an **USER** I want to be able to **logout from my account** so that I can \*\*\*\*.

### Administrator User Stories

#### List of all plugins

As an **ADMINISTRATOR** I want to be able to **see a list of all submitted plugins** so that I can perform the actions according to my role.

#### Edit submitted plugins data

As an **ADMINISTRATOR** I want to be able to ***edit any information of a submitted plugin*** so that I can correct any possible errors or add any missing data.

#### Delete submitted plugins

As an **ADMINISTRATOR** I want to be able to ***delete any submitted plugin*** so that I can remove of possible mal functioning plugins.

#### Search all submitted plugins

As an **ADMINISTRATOR** I want to be able to ***search by name any submitted plugin*** so that I can find the plugin I need to perform any action according to my role.

#### Search all users

As an **ADMINISTRATOR** I want to be able to ***search by email any registered users*** so that I can find the user I need to perform any action according to my role.

#### Edit users data

As an **ADMINISTRATOR** I want to be able to ***edit any information about any user*** so that I can correct any possible errors in registration.

#### Delete users

As an **ADMINISTRATOR** I want to be able to ***delete any user*** so that I can remove any user that asks to be removed or have to removed for violating any *Usage Policy*.

### Novice User Stories

#### Submit plugins

As an **NON-IT User** I want to be able to ***submit plugins easily*** so that I can share my plugins without any contact with developer specific tools that I am not comfortable to use.

### Expert User Stories

#### Submit plugins

As an **IT User** I want to be able to ***submit plugins*** so that I can share my plugins with the community.

## Architecture

### *MVC* Pattern with *NodeJS*

The controller for *piStar - Submission Tool* was built using *NodeJS*. *NodeJS* is a *Javascript* runtime used to build network applications. For a better application structure, *MVC* design pattern was used. One of the main advantages of *MVC* pattern is the separation of concerns, because it makes the application structured with each part of it responsible for an specific task.

The application structure is divided by three main parts. The first one is the *Model*, that handles all code related to the data structure of the application. Generally is where the structures that mirror thedatabase *schema* are declared so they can be used in the code. The second one is the *View,* the part that handles the code that defines what the user sees. Normally, for a web application, this part is written in *HTML* code or, if any *front-end* frameworks or view engines are used, results in *HTML* code. The third and last one is the *Controller,* that is responsible for the communication between the *Model* and the *View*. It basically decides how and when to present or receive data to and from the user. The whole interaction between the user and the three categories are shown in Figure 57.

**

A common problem that happens with *MVC* is to have a bloated *Controller*, that concentrates everything that is not a *Model* or a *View*. Since *piStar - Submission Tool* is a relatively small application, given that it has only one main purpose, that is to submit new plugins, the *MVC* possible problems do not appear. [55]

## Database

In this section is going to be presented some topics about the database used in the tool. First is *schemas* used in the database. Then is going to be presented a brief introduction of the two databases used to build the tool: Redis and Postgres. The image representation of each table of the *schema* was built using the online tool *dbdiagram.io* [56].

### Schema

It is necessary to make a few explanations about the notation used by the tool that generated the diagrams in the following sections. The left column shows the name of the table column and the right column shows the type of the data stored in the respective column. Where it shows *SERIAL* as a type, it means that the data stored will be an integer the increments automatically by one. Where it shows *character* as a type, it means that the data stored will be a *string.* Where it shows *timestamp* as a type, it means that the data stored will be a time in the *ISO 8601* format.

#### Plugins

**

#### Users

**

#### UsersPlugins

**

#### Keywords

**

#### PluginsKeywords

**

### Redis

*Redis* is an in-memory data structure store used as a database. Because it is an in-memory data structure, it has a better performance than other databases. Generally it is used for data that needs high availability and does not require much space. There are two main scenarios that *Redis* is recommended: Cache storage and Session storage. With *Redis* is possible to implement "*sticky sessions*" across many different servers. It allows for a user to preserve its session, while inside the *TTL (Time To Leave)* timebox, even if the connection is made with another server different from the first one. In *piStar - Submission Tool* *Redis* is used for session storage. The setup only require a few lines of code using two main dependencies: *express-session* and *connect-redis.* [57][58][59]

Figure 61 shows the creation of instances of both packages. The object *session* is used to initialize the *RedisStore* object. The configuration of *Redis Store* is done in the *middleware* that is called when the application initializes. Figure 62 shows the configuration of new store with *RedisStore*. It takes three parameters: the host and the port where the *Redis* server is running and optionally takes the Time To Leave parameter that sets a time in milliseconds to expire the session.

|  |
| --- |
| const session = require('express-session') const RedisStore = require('connect-redis')(session) |

Figure 63 - Creation of instances from *express-session* and *connect-redis* packages

|  |
| --- |
| this.express.use(  session({  store: new RedisStore({  host: 'localhost',  port: 6379,  ttl: 1800  }),  name: 'root',  secret: 'piStarPluginSecret',  resave: true,  saveUninitialized: true  })  ) |

Figure 64 - Session configuration

### Postgres

*Postgres* is an open source object-relational database. It is a project with more than 30 years of development, making it a reliable choice for a database solution. *Postgres* extends *SQL* with features that provide safety, reliability, concurrency, extensibility and more. To use *Postgres with Node.js*, *piStar - Plugin Submission Tool* use *Sequelize*, an *ORM (Object-Relational Mapper)* that has the task to map the objects (or models) in *Javascript* to the tables in the database. [60][61]

Before start building any models it is necessary to declare the information to access the database.

|  |
| --- |
| module.exports = {  dialect: 'postgres',  host: '127.0.0.1',  username: 'docker',  password: 'docker',  database: 'piStarPluginSub',  operatorAliases: false,  define: {  timestamps: true,  underscored: true,  underscoredAll: true  } } |

Figure 65 - Database configuration

The first step is to build a object using *Sequelize* like Figure 66 shows. *Sequelize* adds a *createdAt, updatedAt* and an unique *id* automatically, so it is not necessary to declare it explicitly.

|  |
| --- |
| module.exports = (sequelize, DataTypes) => {  const Plugin = sequelize.define('Plugin', {  name: DataTypes.STRING,  short\_description: DataTypes.STRING,  long\_description: DataTypes.STRING,  homepage\_link: DataTypes.STRING,  category: DataTypes.STRING,  reference: DataTypes.STRING,  svgs: DataTypes.STRING,  constraints: DataTypes.STRING,  metamodel: DataTypes.STRING,  shapes: DataTypes.STRING,  uimetamodel: DataTypes.STRING  })   return Plugin } |

Figure 66 - *Plugin* mapping with *Sequelize*

To track changes in the database it is used *migrations*. It helps to revert the changes if needed at any time. Figure 67 show a *migration* file for the *Plugin* table.

|  |
| --- |
| 'use strict'  module.exports = {  up: (queryInterface, Sequelize) => {  return queryInterface.createTable('plugins', {  id: {  allowNull: false,  autoIncrement: true,  primaryKey: true,  type: Sequelize.INTEGER  },  name: {  type: Sequelize.STRING,  allowNull: false  },  short\_description: {  type: Sequelize.STRING,  allowNull: false  },  long\_description: {  type: Sequelize.STRING,  allowNull: false  },  homepage\_link: {  type: Sequelize.STRING,  allowNull: false  },  category: {  allowNull: false,  type: Sequelize.STRING  },  status: {  allowNull: false,  type: Sequelize.STRING,  default: 'Not posted'  },  file\_link: {  type: Sequelize.STRING,  },  created\_at: {  allowNull: false,  type: Sequelize.DATE  },  updated\_at: {  allowNull: false,  type: Sequelize.DATE  }  })  },   down: (queryInterface, Sequelize) => {  return queryInterface.dropTable('plugins')  } } |

Figure 67 - *Plugin* migration file, specifying what is created with the CREATE TABLE and DROP TABLE commands from *SQL*

## Views

### Nunjucks

*Nunjucks* is a *Javascript* templating language that enhances the capabilities of a simple *HTML* file. It add features like the use of variables, filters, template inheritance, conditionals, synchronous and asynchronous loops and more. The most used features used in *piStar - Submission Tool* are exemplified in the figures below. Figure 68 shows an example where an *HTML* paragraph tag displays a variable in the context of a given template.

|  |
| --- |
| <p>{{ user['name'] }}</p> |

Figure 68 - Value of a key *name* inside an *user* object displayed inside a paragraph tag

Template inheritance helps in the organization of code, allowing the developer write code in separate files. In Figure 69 a new *HTML* file is extending another one previous created.

|  |
| --- |
| {% extends "parent.html" %} <div> <h1>Child</h1> </div> |

Figure 69 - *parent.html* is extended by adding a *div* tag with an *h1* tag inside.

To help the developer write less code by reusing it, an *include* block is used. In Figure 68, a *HTML* file starts by using the contents of another file.

|  |
| --- |
| {% include "header.html" %} <body> <div> <h1>My page</h1> </div> </body> |

Figure 70 - *header.html* is included in another file

### Materialize CSS

*Materialize CSS* is a *CSS* framework based on *Material Design*, a visual language built by *Google*. It brings a variety of stylized web elements. Besides the use simplicity, a great advantage is that this visual language is spread all over the web. That brings a important sense of familiarity with the elements of the interface and a intuitive use. Figure 70 shows an example of a submit button that contains an icon. The use of the styles from *Materialize CSS* framework are done through the *class* attribute. The result is shown in Figure 71.

|  |
| --- |
| <button class="btn waves-effect waves-light" type="submit" name="action">Submit  <i class="material-icons right">send</i>  </button> |

Figure 70 - *Button* declaration using *Materialize CSS* styles

**

### Screenshots

**

**

**

**

# Conclusion

## Contributions

An extensive tool can build a very large user base because of this particular feature. It is the case with the tools shown in Chapter 3. Although *piStar Tool* is already extensible, it required some effort to develop, test and share with the community new plugins. With *piStar - Plugin Submission Tool*, testing and sharing are made easier. A developer can make its language extension available to all the users of *piStar Tool*. This helps to get more feedback from users with different backgrounds, since it is not required to have a programming experience or to clone and modify the source code to use a plugin.

## Future works

### Plugin version manager

To update an extension, it is required to delete the submitted version or to make a submission as if it is a new extension. It can lead to the unnecessary display of older versions, what can be confusing to the user. A plugin version manager could avoid that problem and help the developer to keep control of each version.

### Feedback system

A feedback can be provided to the developer by using its email or by visiting the homepage link of the plugin, if a support contact is provided. The necessity to search for a way to send feedback can make the developer loose valuable information that users could provide to help improve the extension. A review and rating system could help that by bringing allowing users to send feedback right on *piStar Tool* interface.

### Plugin development code abstraction

*piStar Tool* has a heterogeneous user base, meaning that not all users have a programming background. Although that with a few changes on the plugin template, an extension can be made available to use with *piStar Tool*, it still requires some code editing. *piStar - Plugin Submission Tool* could allow developers to make those changes in a more abstract way, just filling out a form that could generate the plugin with the specified extension in the end.

# References

[1]PRIKLADNICKI, RAFAEL “Problemas, Desafios e Abordagens do Processo de Desenvolvimento de Software.” PUCRS.

### [2] MCVEIGH, ANDREW “A Rigorous, Architectural Approach to Extensible Applications.” Imperial College London - Department of Computing

[3] BIRSAN, DORIAN “On Plug-ins and Extensible Architectures.” ACME Queue Volume 3, Issue 2, March 18th, 2005.

[4] PRIKLADNICKI, RAFAEL “Problemas, Desafios e Abordagens do Processo de Desenvolvimento de Software.” PUCRS.

[5] PIMENTEL, JOÃO “piStar Tool - A Pluggable Online Tool For Goal Modeling.”

[6] piStar Tool repository. Available at: <<https://github.com/jhcp/piStar>>. Access date: June 19, 2019.

[7] GONÇALVES, ENYO “A Systematic Literature Review of iStar Extensions”,

[8] iStar Extension Repository. Available at: <<https://istarextensions.cin.ufpe.br/catalogue/publication/list?q=&f=92>>. Access date: March 22, 2019.

[9] Visual Studio Code Repository. Available at: <<https://code.visualstudio.com/>>. Access date: May 20, 2019.

[10] Visual Studio Code: Your First Extension. Available at: <<https://code.visualstudio.com/api/get-started/your-first-extension>>. Access date: May 20, 2019.

[11] Visual Studio Code: Yeoman. Available at: <<https://yeoman.io/>>. Access date: May 20, 2019.

[12] Visual Studio Code: Publishing Extension. Available at: <<https://code.visualstudio.com/api/working-with-extensions/publishing-extension>>. Access date: May 20, 2019.

[13] Visual Studio Code: Extension Anatomy. Available at: <<https://code.visualstudio.com/api/get-started/extension-anatomy>>. Access date: May 20, 2019.[14] Google Docs: About. Available at: <<https://www.google.com/docs/about/>[/](https://yeoman.io/)>. Access date: May 22, 2019.

### [15] Google Docs: Quickstart - Add-on for Google Docs. Available at: <<https://developers.google.com/gsuite/add-ons/editors/docs/quickstart/translate>>. Access date: May 22, 2019.

### [16] Google Docs: Apps Script - Overview. Available at: <<https://developers.google.com/apps-script/overview>>. Access date: May 23, 2019.

### [17] Google Docs: Triggers. Available at: <<https://developers.google.com/apps-script/guides/triggers/>>. Access date: May 23, 2019.

### [18] Google Docs: Tutorial - Google Maps plugin - Overview. Available at: <<https://www.labnol.org/internet/write-google-docs-addon/28446/>>. Access date: May 23, 2019.

### [19] Google Docs: Creating a version. Available at: <<https://developers.google.com/apps-script/guides/versions#creating_a_version>>. Access date: May 28, 2019.

### [20] Google Docs: Publishing editor add-ons - Overview. Available at: <<https://developers.google.com/gsuite/add-ons/how-tos/publishing-editor-addons>>. Access date: May 23, 2019.

### [21] Google Docs: oAuth consent screen. Available at: <<https://developers.google.com/apps-script/guides/cloud-platform-projects#completing_the_oauth_consent_screen>>. Access date: May 23, 2019.

### [22] Sketchup. Available at: <<https://www.sketchup.com/>>. Access date: May 28, 2019.

### [23] Sketchup: Extension Warehouse. Available at: <<https://extensions.sketchup.com/>>. Access date: May 28, 2019.

### [24] Sketchup Ruby API. Available at: <<https://ruby.sketchup.com/>>. Access date: June 4, 2019.

### [25] Sketchup: Hello Cube tutorial. Available at: <<https://github.com/SketchUp/sketchup-ruby-api-tutorials/tree/master/tutorials/01_hello_cube>>. Access date: June 5, 2019.

### [26] Sketchup: Developer Center Guidelines. Available at: <<https://extensions.sketchup.com/developer_center/ew_developer>>. Access date: June 5, 2019.

### [27] Sketchup: Installing plugins. Available at: <<https://help.sketchup.com/en/installing-ruby-plugins-extensions>>. Access date: June 4, 2019.

### [28] Sketchup: Managing Extensions. Available at: <<https://help.sketchup.com/en/extension-warehouse/managing-extensions>>. Access date: June 18, 2019.

### [29] StarUML. Available at: <<http://staruml.io/>>. Access date: June 14, 2019.

### [30] StarUML: Getting Started. Available at: <<https://docs.staruml.io/developing-extensions/getting-started>>. Access date: June 14, 2019.

### [31] StarUML: Getting Started: Marketplace Client. Available at: <<https://docs.staruml.io/developing-extensions/getting-started>>. Access date: June 14, 2019.

### [32] Google Chrome: Extensions - Get Started. Available at: <<https://developer.chrome.com/extensions/getstarted>>. Access date: May 26, 2019.

### [33] Google Chrome: Extensions - Overview. Available at: <<https://developer.chrome.com/extensions/overview>>. Access date: May 26, 2019.

### [34] Google Chrome: User Interface. Available at: <<https://developer.chrome.com/extensions/user_interface>>. Access date: June 7, 2019.

### [35] Google Chrome: Publish in the Chrome Web Store. Available at: <<https://developer.chrome.com/webstore/publish>>. Access date: June 10, 2019.

### [36] Google Chrome: Install and manage extensions. Available at: <<https://support.google.com/chrome_webstore/answer/2664769?hl=en>>. Access date: June 10, 2019.

### [37] Wordpress. Available at: <<https://wordpress.org/>>. Access date: May 20, 2019.

### [38] Wordpress: Plugin Handbook. Available at: <<https://developer.wordpress.org/plugins/javascript/>>. Access date: May 20, 2019.

### [39] Wordpress: Add your plugin. Available at: <<https://wordpress.org/plugins/developers/add/>>. Access date: June 12, 2019.

### [40] Wordpress: Planning your Plugin. Available at: <<https://developer.wordpress.org/plugins/wordpress-org/planning-your-plugin/>>. Access date: June 18, 2019.

### [41] Wordpress: Plugin Directory. Available at: <<https://wordpress.org/plugins/>>. Access date: May 20, 2019.

### [42] Eclipse. Available at: <<https://www.eclipse.org/eclipseide/>>. Access date: June 14, 2019.

### [43] Eclipse: *PDE* does Plugins. Available at: <<https://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>>. Access date: June 14, 2019.

### [44] Eclipse: Marketplace Client. Available at: <<https://www.eclipse.org/mpc/>>. Access date: June 14, 2019.

### [45] Eclipse: Marketplace Quickstart. Available at: <<https://marketplace.eclipse.org/quickstart>>. Access date: June 14, 2019.

### [46] Audacity: Features. Available at: <<https://www.audacityteam.org/about/features/>>. Access date: June 10, 2019.

### [47] Audacity: Nyquist - Overview. Available at: <<https://manual.audacityteam.org/man/nyquist.html#program>>. Access date: June 10, 2019.

### [48] Audacity: Creating Nyquist plugins. Available at: <<https://manual.audacityteam.org/man/creating_nyquist_plug_ins.html#Creating_Nyquist_Plug-ins>>. Access date: June 10, 2019.

### [49] Audacity: Conventions for Nyquist Plugins. Available at: <<https://forum.audacityteam.org/viewtopic.php?f=39&t=42106>>. Access date: June 10, 2019.

### [50] GIMP. Available at: <<https://www.gimp.org/>>. Access date: June 6, 2019.

### [51] GIMP: Installing plugins. Available at: <<https://docs.gimp.org/2.10/en/gimp-scripting.html#gimp-plugins-install>>. Access date: June 6, 2019.

### [52] GIMP: Writing a plugin. Available at: <<https://developer.gimp.org/writing-a-plug-in/1/index.html>>. Access date: June 6, 2019.

### [53] Shopify. Available at: <<https://www.shopify.com/>>. Access date: June 6, 2019.

### [54] Shopify: Build a plugin with Node.js and React. Available at: <<https://developers.shopify.com/tutorials/build-a-shopify-app-with-node-and-react>>. Access date: June 6, 2019.

### [55] About NodeJS. Available at: <<https://nodejs.org/en/about/>>. Access date: May 26, 2019.

### [56] dbdiagram.io. Available at: <<https://dbdiagram.io/home>>. Access date: May 26, 2019.

### [57] Redis. Available at: <[https://redis.io](https://dbdiagram.io/home)>. Access date: May 27, 2019.

### [58] Redis: connect-redis. Available at: <<https://github.com/tj/connect-redis>>. Access date: May 27, 2019.

### [59] Redis: 15 reasons to use Redis as an Application Cache. Available at: <<https://redislabs.com/wp-content/uploads/2016/03/15-Reasons-Caching-is-best-with-Redis-RedisLabs-1.pdf>>. Access date: May 27, 2019.

### [60] Postgres. Available at: <<https://www.postgresql.org/>>. Access date: May 27, 2019.

### [61] Sequelize. Available at: <<http://docs.sequelizejs.com/>>. Access date: May 27, 2019.

### [62] Nunjucks. Available at: <<https://mozilla.github.io/nunjucks/>>. Access date: May 27, 2019.

### [63] Materialize CSS. Available at: <<https://materializecss.com/>>. Access date: May 27, 2019.

## 

## 