Universidade do Minho
Escola de Engenharia
Departamento de Informática

Henrique Jorge Caldas Pacheco

A Library of User Interface Widgets Prototypes for Car Dashboards

November 2017

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Henrique Jorge Caldas Pacheco

A Library of User Interface Widgets Prototypes for Car Dashboards

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
José Creissac Campos
Paolo Masci

November 2017

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. José Creissac Campos from Universidade do Minho, for all the guidance and help provided. His contribution was invaluable and an inspiration for the concretization of this dissertation.

I would also like to thank my co-supervisor, Dr. Paolo Masci from INESC TEC, for all the technical support and ideas provided that contributed to the final result.

I would like to express my deepest gratitude to my girlfriend, Mariana Capelo. Words cannot describe the importance of the role she played in this dissertation - she provided intelligent feedback when I needed technical discussions, support when I needed help and motivation when I felt down. Without her, this work would not haven been possible.

I would also like to thank my parents, Henrique and Maria da Conceição, for all the encouragement, and Rui Cruz (CEO of Bsolus and Beevo) for all the comprehension.

ABSTRACT

Ensuring the good usability and user experience of software systems is invaluable, and for that, following a standardized usability engineering process is fundamental. Prototyping plays a crucial role in this process, enabling the proper validation of the usability guidelines before reaching the actual implementation phase. This dissertation focuses on the construction of a JavaScript widgets library to ease the process of prototyping user interfaces. This library will later be incorporated in the prototyping software tool PVSio-Web.

KEYWORDS PVSio-Web, prototyping, user interfaces, usability.

RESUMO

Assegurar a boa usabilidade e experiência por parte dos utilizadores de um sistema de software é algo de valor inestimável, e, para isso, seguir um processo estabilizado de engenharia de usabilidade é fundamental. A prototipagem desempenha um papel crucial neste processo, ao dar a possibilidade de validar a usabilidade de um sistema antes de se iniciar a fase de implementação. Esta dissertação foca-se na construção de uma biblioteca JavaScript de widgets para facilitar o processo de desenvolvimento de prototipagem de interfaces. Esta biblioteca será posteriormente incorporada na ferramenta de prototipagem PVSio-Web.

PALAVRAS-CHAVE PVSio-Web, prototipagem, interface, usabilidade.

CONTENTS

1 INTRODUCTION 1
- 1.1 Context 1
- 1.2 Goals 2
- 1.3 Structure of the document 2

2 STATE OF THE ART 3
- 2.1 Human-Computer Interaction 3
- 2.2 Usability Engineering Process 4
- 2.3 Prototyping 5
- 2.4 UI Prototyping tools 6
- 2.5 Technologies 8
- 2.6 Conclusions 11

3 PVSIO-WEB 13
- 3.1 Concepts and architecture 13
- 3.2 Building prototypes in PVSio-Web 14
- 3.3 Projects in PVSio-Web 16
- 3.4 PVSio-Web and the Open-source initiative 17
- 3.5 Getting started with PVSio-Web 18
- 3.6 Conclusions 22

4 CAR DASHBOARDS ANALYSIS 23
- 4.1 Car dashboards 23
- 4.2 Car dashboards validation 24
- 4.3 Selection and individual analysis of car dashboard models 24
- 4.4 Analysis results 29
- 4.5 Conclusions 30

5 PVSIO-WEB D3-BASED WIDGETS 31
- 5.1 Use of d3.js and d3-gauge-plus.js 31
- 5.2 Implemented widgets 32
  - 5.2.1 Gauge widget 32
  - 5.2.2 CentralPanel widget 35
- 5.3 Implementation analysis 37
- 5.4 Conclusions 38

6 PVSIO-WEB SVG-BASED WIDGETS 39
- 6.1 Use of SVG 39

6.2 Implemented widgets 40
6.2.1 SVGWidget 40
6.2.2 Pointer widget 43
6.2.3 GaugeSport widget 45
6.2.4 Clock widget 51
6.2.5 Gearbox widget 54
6.3 Implementation analysis 56
6.4 Conclusions 57
7 CAR DASHBOARD PROTOTYPES 58
7.1 PVS specification 58
7.2 Construction of Car Dashboard prototypes 60
7.2.1 First car dashboard prototype 62
7.2.2 Second car dashboard prototype 65
7.2.3 Third car dashboard prototype 67
7.2.4 Fourth car dashboard prototype 69
7.3 Conclusions 70
8 CONCLUSIONS AND FUTURE WORK 72
8.1 Comparison of the two adopted methodologies 72
8.1.1 Speed of development 72
8.1.2 Customisation 73
8.1.3 Performance 73
8.1.4 Comparison conclusions 73
8.2 Conclusions 74
8.3 Future work 75

LIST OF FIGURES

Figure 1 The Usability Engineering Process, extracted from [BS EN 62366:2008]. 4
Figure 2 Pyramid chart built with d3.js. 10
Figure 3 An example of scaling a vectorial graphics. 11
Figure 4 An example of scaling a raster graphics. 11
Figure 5 Diagram of the client-server communication mechanism in PVSio-Web, extracted from [Oladimeji et al. (2013)]. 14
Figure 6 Architecture of PVSio-Web GUI, extracted from [Masci et al. (2015)]. 14
Figure 7 Example of the construction of a simple prototype of a microwave, with a time display and two buttons ("plus" button currently selected). 16
Figure 8 AlarisGP prototype built using PVSio-Web. 17
Figure 9 GPCA prototype build using PVSio-Web. 17
Figure 10 BBraun Infusomat Space prototype built using PVSio-Web. 17
Figure 11 Visual representation of the IPhone unlock screen demonstration. 21
Figure 12 Different car dashboards over time 24
Figure 13 Adopted car dashboard models. 25
Figure 14 The first adopted car dashboard model. 25
Figure 15 The second adopted car dashboard model. 27
Figure 16 The third adopted car dashboard model. 28
Figure 17 The fourth adopted car dashboard model. 29
Figure 18 Gauge generated using the d3-gauge-plus library. 32
Figure 19 Documentation of the Gauge widget constructor. 33
Figure 20 Instantiation and rendering of Gauge with default configuration values. 34
Figure 21 Set of provided default style configurations for the Gauge widget. 35
Figure 22 Documentation of the CentralPanel widget constructor. 36
Figure 23 Instantiating and rendering of CentralPanel with default configuration values. 36
Figure 24 Example of the implementation of overlapped arc-shaped gauges using d3-based widgets. 38
Figure 25 Class diagram for the developed widgets 41
Figure 26 Documentation for the changeColor method from SVGWidget prototype. 42

Figure 27 b 42
Figure 28 Example of calling the changeColor method with the "yellow" colour. 43
Figure 29 Documentation of the Pointer widget constructor. 44
Figure 30 b 44
Figure 31 Example of two pointers with different centres of rotation. 45
Figure 32 Documentation of the GaugeSport widget constructor. 46
Figure 33 Discovery of min and max parameters for the rotation of the referred tachometer. 47
Figure 34 GaugeSport instance with example configuration values and rendered value of 2. 48
Figure 35 The set of default variations for the tachometer type. 48
Figure 36 The set of default variations for the speedometer type. 49
Figure 37 The set of default variations for the remaining fuel, thermometer and pressure types. 49
Figure 38 The set of default variations for the compass type. 49
Figure 39 Documentation of the GaugeSport render method. 50
Figure 40 Documentation of the Clock widget constructor. 51
Figure 41 Instantiation and rendering of the Clock widget with example configuration values. 52
Figure 42 The set of default variations for the Clock widget. 53
Figure 43 Documentation of the Gearbox widget constructor. 55
Figure 44 Instantiation and rendering of the Gearbox widget with example configuration values. 56
Figure 45 The possible variations for the Gearbox widget - the first semi-automatic and the other three manuals. 56
Figure 46 The first car dashboard model. 63
Figure 47 First car dashboard prototype in its initial state (top) and after accelarating the car (bottom). 65
Figure 48 The second car dashboard model. 65
Figure 49 Second car dashboard prototype in its initial state (top) and after accelarating the car (bottom). 67
Figure 50 The third car dashboard model. 68
Figure 51 The third car dashboard prototype. 69
Figure 52 The fourth car dashboard model. 69
Figure 53 The fourth car dashboard prototype. 70

1 INTRODUCTION

1.1 CONTEXT

The user interface (UI) is a fundamental factor in the success of a software piece. It is like the facade of a building — it conveys the identity and personality of the system, and plays a crucial role in the first impression it causes to its end users. Other than conveying the correct idea about the system, a good UI is also simple and intuitive, helping users achieving their goals without problems. A badly designed UI may make the system unusable by its end users, possibly causing it to be abandoned — hence, it is vital that UI and user experience (UX) are planned, tested and validated, so that its final result does not become a burden for the system's users.

In this context, prototyping rises as one of the most important steps in software development. It involves planning, testing and validating the UI and UX with the end users before its implementation. This will not only allow a smoother implementation, with less costs both in its development and maintenance, but also a final result that meets the intents of the end users.

There are several tools that provide assistance when it comes to prototyping, for both high and low degrees of fidelity. However, most of the existing tools only take into account the design and navigation of the final system, providing no support for the verification of the UI behaviour. This is even more important in safety critical systems (such as medical devices), where the misuse of the UI can have harmful consequences. These systems require more rigorous forms of verification, and thus the need for prototyping tools that are also able to validate some of the logic represented by the UI.

PVSio-Web [Masci et al. (2015)] is a client side Web application built with JavaScript [Flanagan (2011)] that combines prototype development with the application of formal UI verification. It provides a set of widgets and a drag and drop editor to help creating stateful prototypes. The widgets' interactions are controlled by connecting to a server running PVSio models, which represent the system logic. This tool is currently under development at Universidade do Minho, and constitutes the scenario for the development of this dissertation.

1.2 GOALS

Prototypes built on PVSio-Web have made possible to uncover usability flaws in safety critical systems such as medical devices [Oladimeji et al. (2013)]. It is the author's belief that this is one of the most valuable aspects of UI prototyping and demonstrates one of the main strengths of PVSio-Web, being a solid tool which allows the verification and validation of interactions.

The main goal of this dissertation is to build a widgets library for simulating components found in car dashboards, as tachometers, speedometers and odometers. The widgets built should be aligned with the tool's vision and developed in order to easily be reused in different prototypes. The library will be incorporated in PVSio-Web so that the set of default widgets provided by the tool is extended. This will improve the overall efficiency when building more complex and realistic prototypes.

Additionally, the creation of car dashboard prototypes using PVSio-Web and the implemented library will also be object of study, demonstrating its application in practical examples.

1.3 STRUCTURE OF THE DOCUMENT

The first chapter of the present document exposes the context around the dissertation, and the goals that are meant to be achieved.

The second chapter approaches the current state of the art on the usability engineering field, describing the process followed when developing interactive software applications and devices, and the tools available to create prototypes. A short description of the technologies used in context of this dissertation is also provided.

The third chapter describes PVSio-Web, the prototyping tool that will be used in this dissertation. It provides an overview on prototype construction, the projects developed with PVSio-Web, and an initial demonstration to exemplify the architecture of the tool.

The fourth chapter provides an overview on car dashboards' evolution, its usual components, and the components that should be available in order to create realistic prototypes.

The fifth and sixth chapters describe the contribution of the author to PVSio-Web, firstly with the implementation of widgets using the d3.js-based d3-gauge-plus.js library, and secondly with the complementary set of widgets based on SVG files.

The seventh chapter analyses the construction of car dashboards using the developed widgets and the PVS model that was used to simulate the car engine, and the last chapter assesses the dissertation as a whole, and provides pointers for further improvement.

2

STATE OF THE ART

This chapter discusses HCI as an important research field in software engineering, the relevance of prototyping in the usability engineering process, and some of the tools available to create UI prototypes. A short description of the technologies used later in this dissertation is also provided.

2.1 HUMAN-COMPUTER INTERACTION

Human-Computer Interaction (HCI) is a research field that studies the way users interact with computers. The term was firstly used by Carlisle (1975), but different definitions have been provided since. Baecker and Buxton (1987) defined HCI as “a set of processes, dialogues and actions through which a human user employs and interacts with a computer”. Later, Hewett et al. (1992) defined it as “a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them”. HCI as a subject arose from the need to understand the way users relate with machines and how software could evolve in order to improve this relationship.

Additionally, with the adoption of new technologies such as the mouse or graphical user interface (GUI) operative systems, **usability** started to play a more and more important role in the quality of a software piece. As Jordan (1998) stated, “users begin to see ease of use as central to product quality”.

The rise of HCI as a research field and the growing importance of usability as a product quality lead to the need for an organized and structured **Usability Engineering Process** (UEP). This process would have as its main goal helping software engineers attain excellent usability and ease of use in the software products or devices they manufacture.

UEP is targeted to “making the devices (or software products) safer, more effective and easier to use” [BS EN 62366:2008]. This process is even more important in safety critical systems, where an error may result in harmful consequences. According to [BS EN 62366:2008], devices themselves often contribute to usability errors, most commonly due to **UI design flaws**. The systematic application of usability design principles, reinforced by validation and tests involving end users, is an effective means to discover and resolve such design flaws.

2.2 USABILITY ENGINEERING PROCESS

Usability was defined in [ISO 9241-11:1998] as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". The term *effectiveness* refers to the accuracy and completeness with which users achieve specified goals. *Efficiency* is the amount of resources expended in relation to the accuracy and completeness with which users achieve goals, and *satisfaction* is the freedom from discomfort and the positive attitudes towards the use of the product. In order to achieve these qualities on the development of a device or software piece, it is important to follow a sustainable and verified Usability Engineering Process [BS EN 62366:2008].

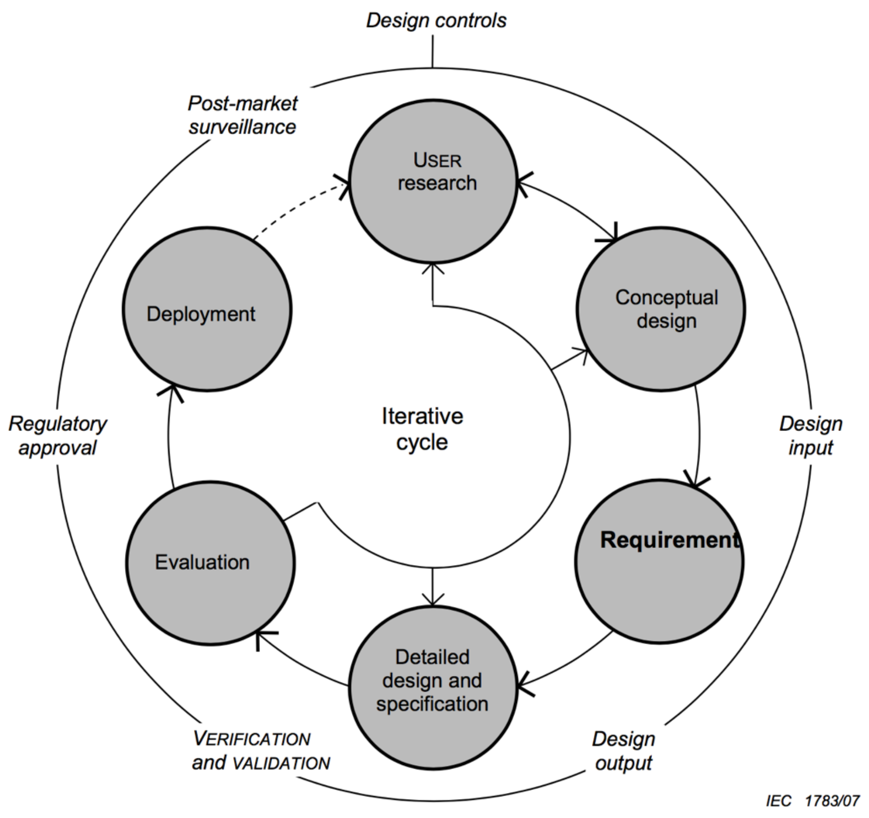


Figure 1: The Usability Engineering Process, extracted from [BS EN 62366:2008].

Description of the Usability Engineering Process (UEP)

As many other processes in systems engineering, the cornerstone of UEP is its **iterative nature** (Figure 1). It is assumed that the product (or prototype) being developed evolves with every iteration of the development cycle, and each iteration provides valuable information and knowledge that can be put to practice in the next one.

UEP starts with a stage of *User Research*. The main goal at this point is trying to understand how users will interact with the product, and involving the end users in the process from the early stages.

The process follows with *Conceptual Design*, where the system's architecture is defined and user needs are validated. It is usual, at this stage, to evaluate how users interacted with precursor systems.

At this point, the development team should have a better understanding of the needs and goals of the end user, and a *Requirements and Design Specifications* step can start. This involves not only specifying the system that will be developed, but also trying to predict potential UI/UX flaws and consequently taking measures to prevent them.

After the requirements have been defined, the actual implementation phase can be started — first, as a prototype, and ultimately, as the final product. The *Evaluation* stage ensures that the prototype (or final product) meets the end user's needs, which can lead to the discovery of new problems and/or misconceptions that bring the team to the beginning of the process, thus the iterative nature [BS EN 62366:2008].

It goes without saying that the faster these cycles are, the more information can be gathered and used to improve the overall results of the process. And this is why prototyping plays such a crucial role — because it allows for these cycles to go faster than they would if the teams would actually be developing the final system. Additionally, prototyping also enables the possibility of getting valuable feedback from the users before the start of the implementation phase, allowing the software team to understand if the proposed system will achieve the end user's expectations and needs. It also reduces time and costs of the project, since it will help improve the quality of the project specification, and prevent further changes to the requirements.

2.3 PROTOTYPING

Prototyping is the activity of creating a prototype of a product. A prototype of a software application is an incomplete version of the system being developed, and typically tries to simulate some aspects of the final product.

Prototyping encourages improved and increased user involvement from the start of the process, which may lead to preventing misconceptions and miscommunications between the intervening parts and increase the end user's satisfaction [Bäumer et al. (1996)].

Types of prototyping

Prototypes are usually grouped in throwaway or evolutionary according to type, and low or high fidelity according to dimension [Nielsen (1993)].

- **Throwaway prototyping** refers to the creation of a model that will eventually be discarded rather than becoming a part of the final delivered product. It involves building a simple working model of the system to visually show the users what their requirements will look like once implemented.
- **Evolutionary prototyping** differs greatly from throwaway prototyping, as the main goal is to build a very robust prototype in a structured manner and constantly refine it. Once built, it becomes the heart of the new system, in which improvements and further requirements will be built.
- **Low fidelity prototypes** try to convey an approximate idea of the final system using drawings or screen shots. These are typically associated with throwaway prototyping.
- **High fidelity prototypes** use more advanced technology (e.g. HTML) to build a more complex, but also more approximate, concept of the final system. Its cost is higher when compared to low fidelity ones, and often high fidelity prototypes evolve to be included in the final system (evolutionary prototyping).

2.4 UI PROTOTYPING TOOLS

There are several tools for the purpose of UI prototyping. Almost every existing tool offers ready-to-use sets of widgets that accelerate the prototype development, as well as multiple export formats.

Of the tools presented, *Pencil*, *Balsamiq* and *PVSio-Web* have as its main goal UI prototyping. In the specific case of *PVSio-Web*, its widgets allow the addition of behaviour to the prototype, while *Pencil* and *Balsamiq*'s prototypes are more static. *Simulink* does not have as its main purpose UI prototyping, but it provides modules or components that enable the user to do so.

Pencil Project

Pencil[1] is an open-source graphical user interface (GUI) tool widely used for the purpose of prototyping. It provides a very extensive set of built-in shapes (or `widgets`), including basic drawing shapes, flowchart elements and shapes specific for prototyping desktop or mobile applications. Developer users can develop their own shapes and share them with the community of Pencil users.

This tool also supports the concept of prototypes as having different *pages*, and the possibility of creating links between them to simulate navigation. This enables the possibility of creating more complex prototypes by composing smaller ones.

Pencil supports the creation of throwaway prototypes, but it provides no support for building evolutionary prototypes, since, despite having multiple export formats, none of them can be reused as a resource for building a software system. Additionally, the navigation model is quite basic, providing only static routes for the built prototype (i.e. no control logic for navigating in the prototype).

Balsamiq

Balsamiq[2] is a tool for the construction of prototypes (or *mockups*). Similarly to Pencil Project, it also provides built-in usable components, and it is useful to build throwaway prototypes. As well as Pencil, it supports the possibility of linking different *mockups*, but the most relevant difference when compared to Pencil is the pricing – Pencil being open-source and free to use, and Balsamiq not so.

PVSio-Web

PVSio-Web[3] is a modelling and prototyping tool, currently being developed at Universidade do Minho. It was built using JavaScript and the d3.js framework, and enables the application of formal validation when building prototypes of a given system [Masci et al. (2015)]. It provides a set of widget that ease the process of creating prototypes, but it does not support the possibility of linking different prototypes. PVSio-Web is the tool the will be used in the present project, and will be described in more detail on Chapter 3.

1 Evolus Pencil Web page: http://pencil.evolus.vn/
2 Balsamiq — https://balsamiq.com/
3 PVSio-Web — http://www.pvsioweb.org/

	Pencil Project	Balsamiq	PVSio-Web	Simulink
Reusable components	Yes	Yes	Yes	Yes
Linking prototypes	Yes	Yes	No	Yes
Open-source / free	Yes	No	Yes	No
Formal verification	No	No	Yes	Yes

Table 1: A comparison between the features of the analysed prototyping tools.

Simulink

Simulink[4] is a block diagram environment for multidomain simulation and model-based design developed by MathWorks[5]. Simulink is often used for modelling complex systems from their internal logic to the UI. It supports UI prototyping, despite not being its focus.

As well as the systems discussed above, Simulink also offers a wide set of predefined blocks (or `widgets`) to help the development process, and enables C/C++ code generation from its models, which makes it very useful for building evolutionary prototypes that can afterwards be incorporated into the actual system being developed.

Comparative analysis

The tools presented above can be grouped in two categories. *Pencil* and *Balsamiq* are low-cost and easy-to-use tools, focused on rapid prototyping. They provide no support for formal validation of UI rules, and their final results are throwaway prototypes.

Simulink and *PVSio-Web* are tools that require a greater level of investment when used — both in cost and in the time required to learn how to work with them. Despite this, more complex forms of validation can be applied to the models, and the final results are frequently used as part of the final system, which represents evolutionary prototyping.

Table 1 displays a tabular analysis of the different features of the described tools.

2.5 TECHNOLOGIES

This dissertation describes the implementation of a widget library to be included in the PVSio-Web prototyping Web application. In this section, the technologies in the base of PVSio-Web — the scripting language JavaScript and the JavaScript framework d3.js — and technologies used to complement it — such as d3-gauge-plus and SVG — will be discussed.

4 Simulink - https://www.mathworks.com/products/simulink.html
5 MathWorks - https://www.mathworks.com/

JavaScript

JavaScript (JS) [Flanagan (2011)] is a high-level, dynamically typed, cross-platform, object-oriented scripting language. It is normally used inside a host environment (e.g., a Web browser), and connected to the objects of its environment to provide programmatic control over them.

JavaScript *per se* is a set of libraries not much different from most programming languages — provides syntactic support, structures to control the flow of execution, array and object handling, libraries for math operations, event handling, etc.

This language can be extended with objects and libraries that serve different purposes. One example is client-side JavaScript, which extends core functionality by supplying objects to control a browser and its Document Object Model (DOM)[6], enabling the usage of the language in Web applications. Another example is server-side JavaScript, that extends it so that it can be used as a fully-functional Web server.

JavaScript has become a dominant language, especially in Web development. In a client side environment, it can be used in small or large scale - from small scripts loaded in a Web page to add interactivity to its UI, to fully structured client side applications including data models, routing and controllers. For the latter, several frameworks have been developed in the language and used in major projects (for instance, Youtube and Gmail use AngularJS framework, Facebook and Instagram are based on ReactJS). Additionally, multiple libraries are available for different purposes, with the goal of providing specific and easy to use features for DOM manipulation (like JQuery, Chart.js and d3.js).

d3.js and d3-gauge-plus.js

d3.js[7] — d3 stands for Data-Driven Documents — is a JavaScript library for manipulating documents based on data and producing dynamic, interactive data visualizations in Web browsers. It makes use of existing standard technologies (such as HTML, SVG and CSS), allowing great control over the final results.

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents a Web page so that programs can change the document structure, style and content. The d3.js library allows the binding of arbitrary data to a DOM object, and the application of data-driven transformations to it. An example of the d3.js usage is the generation of an HTML table or an interactive SVG bar chart from an array of numbers (Figure 2).

6 Mozilla Developer Network Web page on DOM: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`
7 d3.js framework Website: `https://d3js.org/`

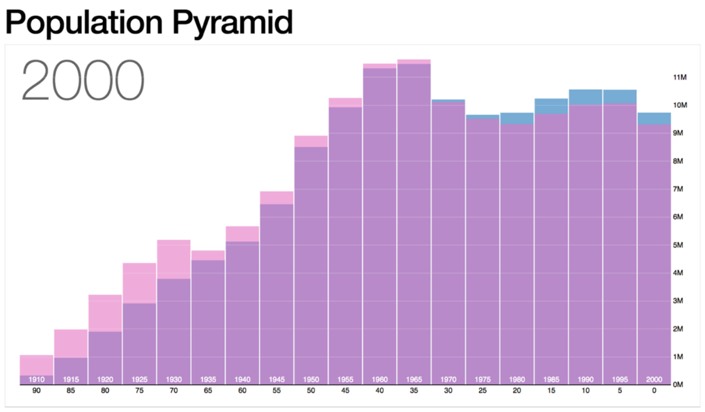


Figure 2: Pyramid chart built with d3.js.

d3-gauge-plus.js[8] is an extension of d3.js originaly developed by Andy Gimblett, focused on the generation of gauges. This library handles the implementation details of a gauge presentation and behaviour, allowing a set of configurations to control both.

SVG

Scalable Vector Graphics[9] (SVG) is an XML-based markup language, with the purpose of describing two-dimensional vector graphics. It is a text-based open Web standard, explicitly designed to work with other Web standards such as CSS or DOM.

Vector graphics (such as SVG files) differ from **raster graphics** (such as PNG or JPEG files) in the way they store the graphics data. Raster graphics use colored pixels arranged in a way to display an image, and vector graphics use paths to tell how each part of the image should be shaped and what color it is bordered with or filled by.

One of the advantages of vector graphics in comparison to raster grpahics is that they do not suffer loss of quality when resized (Figure 3), contrary to raster graphics, which suffer a clear loss of quality on scaling (Figure 4). Another important advantage in the context of this dissertation is that, due to its XML-based nature, vector graphics are treated as DOM elements by browsers. This allows the handling of the SVG element (its creation, positioning, animation or deletion) and changing its properties on runtime.

8 d3-gauge-plus.js Github repository: `https://github.com/gimbo/d3-gauge-plus`
9 Mozilla Developer Network page for SVG: `https://developer.mozilla.org/en-US/docs/Web/SVG`

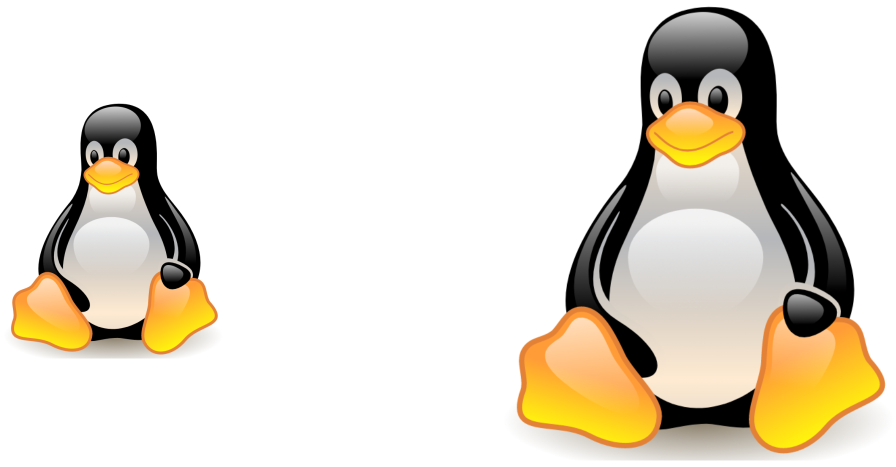


Figure 3: An example of scaling a vectorial graphics.

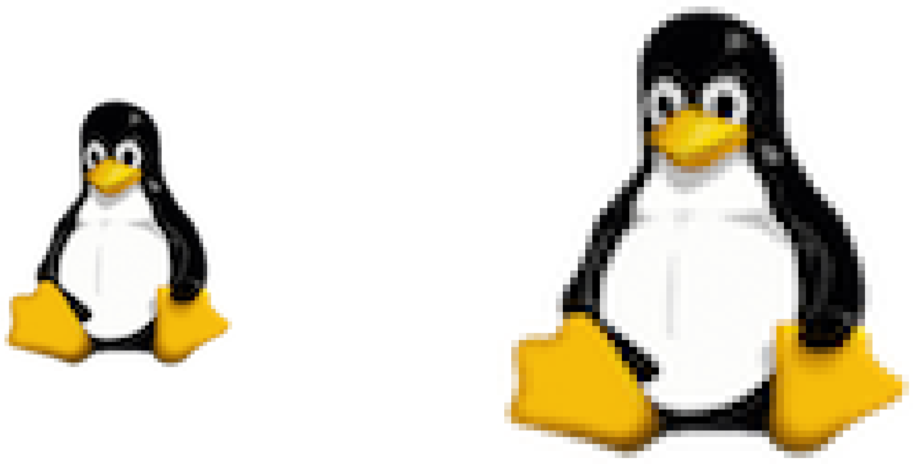


Figure 4: An example of scaling a raster graphics.

2.6 CONCLUSIONS

In this chapter, the relevance of Human-Computer Interaction was briefly discussed. Studying how humans relate with machines has made possible to draw a set of processes that should be followed to optimize the usability and user experience in product development — the Usability Engineering Process.

UEP depends on prototyping as one of its main accelerators. It allows to get feedback from the end users in early stages and also helps preventing miscommunications and UI design flaws in future stages of the development process. This is even more important in the case of safety critical systems, where UI flaws can lead to harmful consequences.

There are multiple tools in the market that are suitable for prototyping, but, with the exception of PVSio-Web, none of them support the application of formal verification into prototypes. The technologies that are on the base of PVSio-Web were described in section 2.5, and the tool itself will be described in detail in the next chapter.

3

PVSIO-WEB

3.1 CONCEPTS AND ARCHITECTURE

PVSio-Web[1] is a formal methods tookit for model-based development of human-machine interfaces. Its initial developments date back to August 2012, and it has maintained a sustainable activity in its GitHub repository since then[2].

PVSio-Web extends the PVSio component [Muñoz (2003)] of PVS [Owre et al. (1996)] with a graphical environment that allows rapid prototyping of device user interfaces based on formal PVS specifications [Oladimeji et al. (2013)]. **Prototype Verification System** (PVS) is a specification language integrated with support tools and a theorem prover, intended to make mechanized formal methods usable for significant applications[3].

In the context of PVSio-Web, a **prototype** consists of a **background image** of the UI to be prototyped, **widgets** (interactive areas over the prototype that can react to clicks or display visual information), and a **PVS specification**; it defines and controls the behaviour of the widgets, by storing the current state and the set of possible state transitions of the prototype.

The interaction between the widgets and the PVS specification happens under a logic of client-server communication (Figure 5). The client-side displays the GUI of PVSio-Web as a Web application, and the Web-server encapsulates the tool's back-end. Actions on the client-side can trigger on-demand executions of the PVS specification on the backend, which in turn responds with the new state of the prototype, and every widget in the client-side is re-rendered, according to the provided state [Oladimeji et al. (2013)].

The client-side GUI of PVSio-Web provides multiple development environments, with different features according to the target user of the tool:

- **Simulator View:** designed for domain specialists and end users, allows the exploration of prototypes generated with PVSio-Web.

1 PVSio-Web Webiste: `http://www.pvsioweb.org/`
2 GitHub repository for PVSio-Web: `https://github.com/thehogfather/pvsio-web`
3 PVS Web page: `http://pvs.csl.sri.com/`

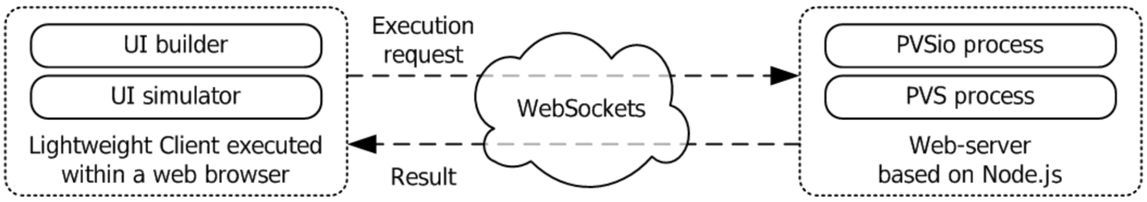


Figure 5: Diagram of the client-server communication mechanism in PVSio-Web, extracted from [Oladimeji et al. (2013)].

- **Prototype Builder:** provides tools for creating the visual appearance of the prototype and editing the PVS model, being targeted for developers and formal methods especialists.
- **Model Editor:** can be used by advanced PVS users for editing and type-checking PVS models.
- **Emucharts Editor:** can be used by developers who are novice PVS users for developing formal models using a graphical notation.

These different environments allow users with different background and expertise to work together with the same underlying formal models (Figure 6) [Masci et al. (2015)].

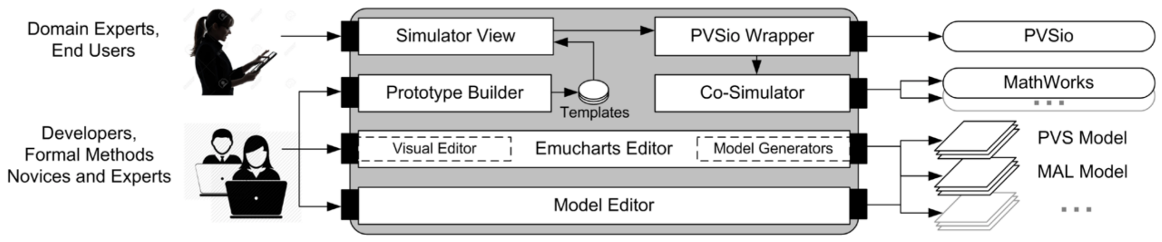


Figure 6: Architecture of PVSio-Web GUI, extracted from [Masci et al. (2015)].

3.2 BUILDING PROTOTYPES IN PVSIO-WEB

Building a prototype in PVSio-Web using the **Prototype Builder** involves, firstly, uploading an image to be used as background. Once uploaded the image, the user has the possibility of defining areas of the image over which widgets will be placed, in order to add interactivity to the prototype. PVSio-Web provides 6 widgets by default:

- **Button:** a clickable widget, which triggers an action that can then be used to provoke changes in the underlying PVS model, and thus in the interface.

- **Basic Display:** displays basic text in the prototype interface.
- **Numeric Display:** displays floating point values in the prototype interface.
- **Touchscreen Display:** a touch enabled display for textual values.
- **Touchscreen Button:** similar to the Button widget, but oriented towards touch enabled devices, adds a clickable widget to the prototype that can be used for clicking and provoking changes in the underlying PVS model.
- **LED:** displays a LED light that can be turned on or off, according to some variable in the underlying PVS model.

While placing the widgets over the background image, the user can associate textual displays to variables provided by the PVS model, or button clicks to actions provided by the API of the PVS model.

If the user wants to display a variable provided by the PVS model (for instance, *time*), he/she can use the **Basic Display** widget, associating the ″Display Name″ property to the variable he/she wants to show (*time*).

If the user wants to trigger an action provided by the PVS model API (for instance, *incTime*), he/she can use the **Button** widget, associating the ″Button Name″ property to the action he/she wants to trigger (*incTime*).

The PVS model associated with the prototype can either be built in-browser using the **Model Editor** or uploaded from a file in the user′s computer.

After adding all the widgets to the prototype, the user can then use the **Simulator View** to interact with the prototype. Figure 7 displays, as an example, the prototype of a microwave built using the **Prototype Builder**. The left panel displays the list of added widgets: a **Basic Display** widget to display the time (″time″), which was previously positioned on the microwave display area, and two **Button** widgets - one to increase the time (″plus″), previously positioned over the ″1″ key of the microwave keyboard, and one to decrease it (″minus″), previously positioned over the ″2″ key of the microwave keyboard.

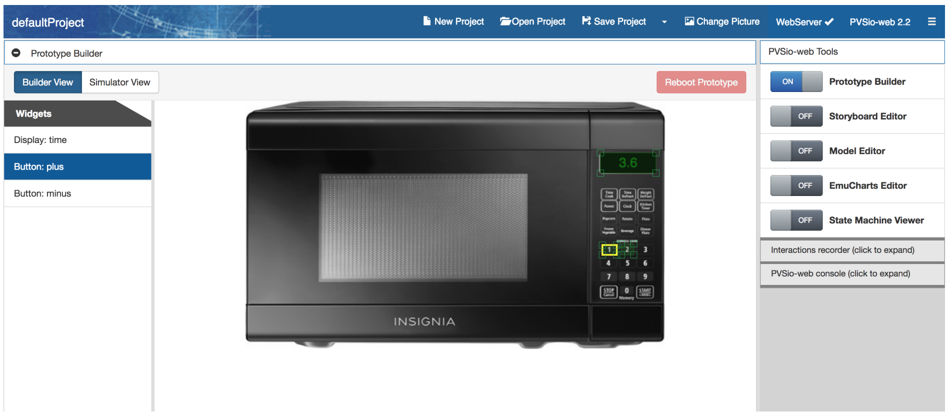


Figure 7: Example of the construction of a simple prototype of a microwave, with a time display and two buttons ("plus" button currently selected).

3.3 PROJECTS IN PVSIO-WEB

PVSio-Web has been successfully used for the implementation of several prototypes, mostly related with the analysis of safety-critical medical devices, such as **infusion pumps**. Infusion pumps are devices that deliver drugs and nutrients into the patient's body at controlled rates and volumes.

These projects have enabled the development of training material and device construction guidelines, in an attempt to raise the awareness about general user interface issues that may put at risk the safety of patients [Masci et al. (2015)].

AlarisGP[4] is a volumetric pump used in infusion therapy. A picture of the online demonstration of its UI modelled in PVSio-Web can be seen in Figure 8. The prototype allows the simulation of several actions, like increasing or decreasing the quantity of substance being pumped by clicking the arrowed buttons.

Generic Patient Controlled Analgesia (GPCA)[5] is a pump that allows the patient to self-control its pain level, by means of increasing or decreasing the amount of analgesia that is injected in his body. Its user interface was also modelled using PVSio-Web, which can be seen in Figure 9.

BBraun Infusomat Space[6] is also an infusion pump modelled in PVSio-Web. Figure 10 shows a snapshot of the live demonstration.

The projects explained are mainly based in textual visualiations as output, allowing simple interactions such as clicking on buttons or textual inputs. This reflects the lack of graphical widgets for prototyping more complex systems in PVSio-Web, which is the moti-

4 AlarisGP online demonstration: `http://www.pvsioweb.org/demos/AlarisGP/`
5 GPCA online demonstration: `http://www.pvsioweb.org/demos/GPCA/`
6 BBraun online demonstration: `http://www.pvsioweb.org/demos/BBraun/`

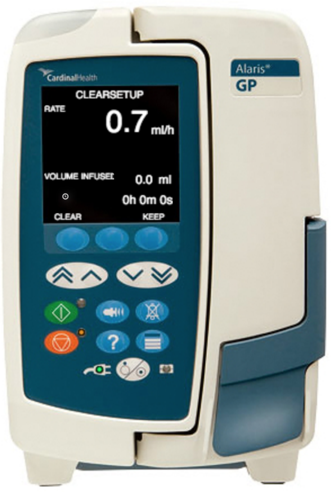


Figure 8: AlarisGP prototype built using PVSio-Web.

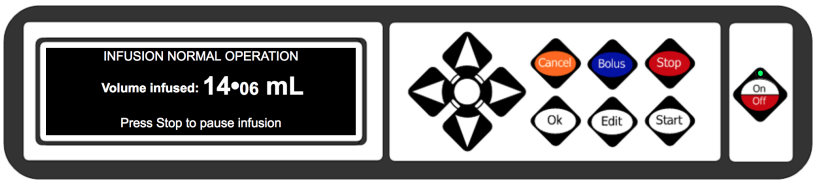


Figure 9: GPCA prototype build using PVSio-Web.

vation behind this dissertation, reflected in its goals — to extend the set of default widgets in PVSio-Web.

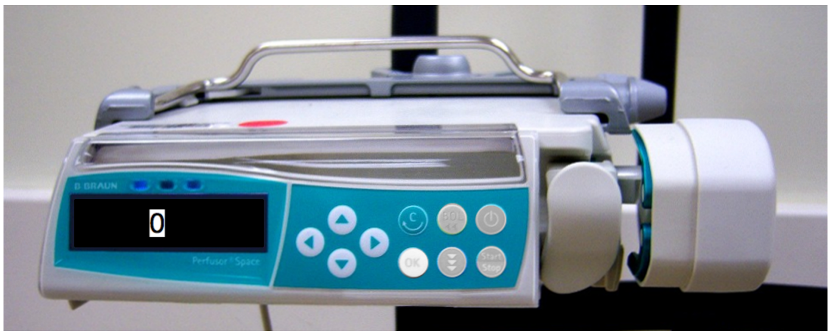


Figure 10: BBraun Infusomat Space prototype built using PVSio-Web.

3.4 PVSIO-WEB AND THE OPEN-SOURCE INITIATIVE

Open-source software[7] (OSS) is computer software where its source code is made available for study, change and distribution for anyone and for any purpose. This software is usually

7 Wikipedia page on open-source software: https://en.wikipedia.org/wiki/Open-source_software

associated with public collaborative development, where large Internet communities of developers contribute to the development process.

OSS was greatly boosted by the Open-Source Initiative (OSI), formed on February 1998 by Eric Raymond and Bruce Perens. This initiative proposed the usage of a label ("open-source") that would be able to eliminate ambiguity and remove the perception of free software as anti-commercial.

OSS empowers developers with the ability of freely contributing for the growth of projects based on what they really want the software to be. Despite this, this software development methodology has inherent advantages and disadvantages.

Pros and cons of open-source software development

Open-source software, being free to obtain and use, tends to easily have more users and increased adoption of standard implementations. The implementation costs are usually much lower when compared with closed source software, due to lower costs of marketing and logistical services.

Additionally, open-source software tends to be more flexible to technological innovation, due to the lack of commercial pressure to satisfy customer requirements, and more reliable, since typically has hundreds or thousands of developers testing and fixing bugs.

However, since the code of an open-source software is open to everyone, it might be more prone to security weaknesses or loopholes than closed source software. Also, it is more difficult to design a sound business model around open-source software, and thus many projects fall in misuse due to the tendency to satisfy mostly technical requirements, and not users or market ones.

PVSio-Web as an open-source software

PVSio-Web has been following the open-source software principles, and counted on 11 main contributors along its development. Its Github repository has also been forked multiple times by users who intend to develop and improve over the software on their own. The present dissertation constitues a contribution to the PVSio-Web project as an open-source project.

3.5 GETTING STARTED WITH PVSIO-WEB

For the purposes of development, PVSio-Web's architecture is best understood with the analysis of a **demonstration**. In the context, a **demonstration** is a prototype where all the steps required for the prototype creation (including the background image, instantiation of widgets and the communication with the PVS back-end) are done programatically, and not recurring to the **Prototype Builder**.

This should represent the unlock process of an IPhone, and is based on two existing PVSio-Web widgets: **BasicDisplay** and **TouchscreenButton** (both described in section 3.2). The goal is to add a *draggable* behaviour to the latter (i.e. make it reactive to drag and drop events), which will allow to simulate the dragging of a button, and, thus, the unlock of the IPhone.

The PVS specification

Listing 3.1 shows the source code of the PVS model used for this demonstrative example. PVS specifications will be described in more detail on section 7.1, so for the purpose of the present demonstration, the simple model shown is enough.

The PVS specification for this demonstrative example is based on two state variables, *current_state* and *previous_state* (lines 8 to 11), which can take one of two values: *locked* or *unlocked*. These variables represent if the IPhone is (or was, respectively) locked, and are described in line 5.

The PVS specification defines that the initial value for both of its variables is *locked* (meaning that the IPhone should initially be locked) – lines 13 to 17, and provides as API the *unlock* method (lines 31-40), which sets the value of *previous_state* to *current_state*, and *current_state* to *locked*.

Listing 3.1: A simple PVS for the IPhone unlock demonstrative example.

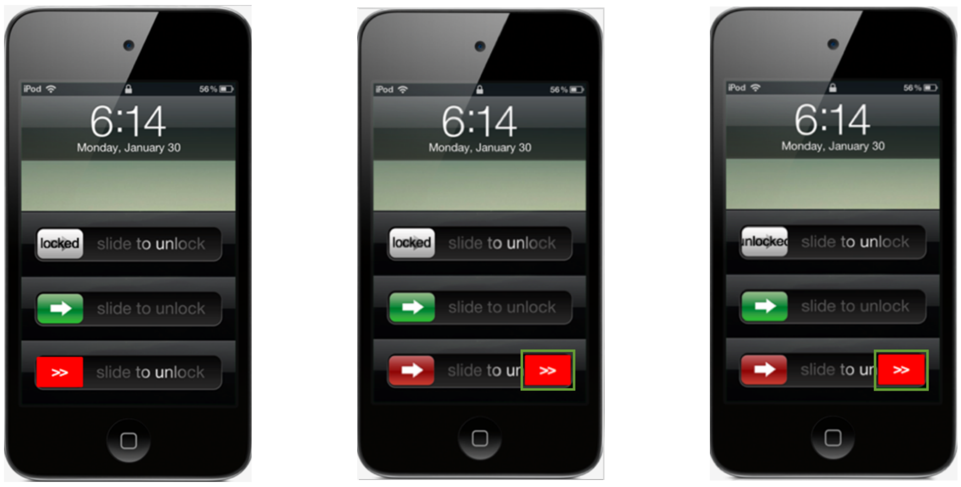
```
1   main: THEORY
2    BEGIN
3
4     %-- Machine states
5     MachineState: TYPE = { locked, unlocked }
6
7     %-- Prototype state
8     State: TYPE = [#
9       current_state: MachineState,
10      previous_state: MachineState
11    #]
12
13    %-- Initial state
14    init(x: real): State = (#
15      current_state := locked,
16      previous_state := locked
17    #)
18
19    %-- utility functions
```

```
    enter_into(ms: MachineState)(st: State): State =
      st WITH [ current_state := ms ]
    leave_state(ms: MachineState)(st: State): State =
      st WITH [ previous_state := ms ]

    %-- transition functions
    per_unlock(st: State): bool =
      ((current_state(st) = locked))
      OR
      ((current_state(st) = unlocked))

      unlock(st: (per_unlock)): State =
      COND
      (current_state(st) = locked)
       -> LET new_st = leave_state(locked)(st)
           IN enter_into(unlocked)(new_st),
      (current_state(st) = unlocked)
       -> LET new_st = leave_state(unlocked)(st)
           IN enter_into(locked)(new_st),
      ELSE -> st
      ENDCOND

    %-- alternative names
    click_unlock_button_red(st: State): State = unlock(st)
    click_unlock_button_green(st: State): State = unlock(st)
    click_unlock_button_silver(st: State): State = unlock(st)
    slide_unlock_button(st: State): State = unlock(st)
  END main
```

Description

The initial screen of this project can be seen on Figure 11a. **BasicDisplay** and **Touchscreen-Button** correspond to the grey and red buttons, respectively. The first is responsible for displaying the state of the *current_state* variable of the PVS model associated to this demonstration, and the second will be responsible for calling the *unlock* method from the PVS model's API.

If the **TouchscreenButton** widget (the red button) is dragged to the right, as one does to unlock an IPhone, a green square signaling the drop zone should appear (Figure 11b). Once the red button is dropped inside the drop zone, the PVS model's method *unlock* is called, and the Iphone should be unlocked. This can be confirmed by inspecting the **BasicDisplay** widget once more, which should display *unlocked* as the value of the *current_state* variable (Figure 11c).



(a) Initial state. (b) Slidding the red button. (c) Final state.

Figure 11: Visual representation of the IPhone unlock screen demonstration.

Implementation

The implementation of this demonstration involved interacting with d3.js **drag** behaviour API[8]. This behaviour (as most of d3.js APIs) works in an asynchronous fashion, allowing to set functions that will be called once a certain event is triggered (these functions are usually referred to as *callbacks*).

Once the dragging of the red button starts, a '*drag*' event is triggered by the browser for every movement that the cursor does. Using the referred d3.js behaviour, a method was set so that the position of the **TouchscreenButton** widget is changed according to the movement of the cursor. This method is also responsible for limiting the movement of the red button along the "slide to unlock" horizontal bar (i.e. the button should not be dragged around the whole page), and for showing the drop zone once the red button is dragged over it.

Once the **TouchscreenButton** widget is dropped, a '*dragend*' event is triggered by the browser. The callback set on this DOM event is responsible for communicating with the PVS model to trigger the IPhone unlock action, if the button was dropped inside the drop zone.

This demonstrative example illustrates the architecture of PVSio-Web. A display widget is used to display the value of one of the variables of the state of the prototype, and a button widget is used to trigger actions on the prototype PVS model.

8 d3.js drag behaviour API: https://github.com/d3/d3-3.x-api-reference/blob/master/Drag-Behavior.md

3.6 CONCLUSIONS

The present chapter approached PVSio-Web, its concepts and architecture, and why it is as a solid toolkit for the development of prototypes of human-machine interface prototypes with formal verification. The method for construction of prototypes using PVSio-Web's **Prototype Builder** was described, and an overview on projects of safety critical devices that were implemented recurring to the tool was also provided.

This chapter also walked through the process of development of a demonstration of a PVSio-Web prototype, using the unlock mechanism of an IPhone as a case of study. This example was intended to put to practice the concepts of the underlying architecture of PVSio-Web, and thus exemplify how they can be applied in real application scenarios.

The following chapters will base themselves on the concepts exposed in this chapter to describe the author's contribution to PVSio-Web as an open-source project.

4 CAR DASHBOARDS ANALYSIS

One of the goals of this dissertation is the construction of car dashboard prototypes. In order to achieve it, car dashboards were studied and a set of models were selected and analyzed in detail.

4.1 CAR DASHBOARDS

A **car dashboard**[1] - also called instrument panel or fascia - is a control panel usually located directly ahead of a vehicle's driver, which displays instrumentation and controls for the vehicle's operation.

Car dashboards have been evolving over time (Figure 12), along with the added features to the car system and with the new display possibilities - as the use of LEDs lights or even touchscreens. The initial car dashboards focused only in basic information as the water temperature, fuel level and speed, while advanced car dashboards may accomodate a broad set of gauges, information about the current shift, fuel consumption, total car mileage, climate control and entertainment systems.

1 Wikipedia page on car dashboards: `https://en.wikipedia.org/wiki/Dashboard`

(a) Ford Model A, 1920's

(b) Ford Crown Victoria, 1990's

(c) BMW i8 Spyder Concep, 2010's

Figure 12: Different car dashboards over time

4.2 CAR DASHBOARDS VALIDATION

As car dashboards become more complex, additional care has to be invested in the interaction between the driver and the car. Car dashboards should be informative, easy to read, and, mainly, attractive but not distracting[2]. Hence, studying the optimal dashboard design and interface, and validating driver's interactions with is a key factor to achieving a more friendly, reliable and, most of all, safer car dashboard.

Prototyping car dashboards using a software tool comes as a less expensive action to gather the necessary data about the dashboard interface and its user interaction. With the right software tool, it is possible to bring into the dashboard prototypes the formal verification used in prototypes of other critical systems.

4.3 SELECTION AND INDIVIDUAL ANALYSIS OF CAR DASHBOARD MODELS

A total of four car dashboard models were adopted in the context of this dissertation. The selected models represent the majority of current car dashboards and are presented in

2 https://medium.com/@dnevozhai/car-dashboard-ui-collection-123ce3ab5303

Figure 13. Each selected car dashboard model is analyzed in its appearence in order to study the involved visual components.

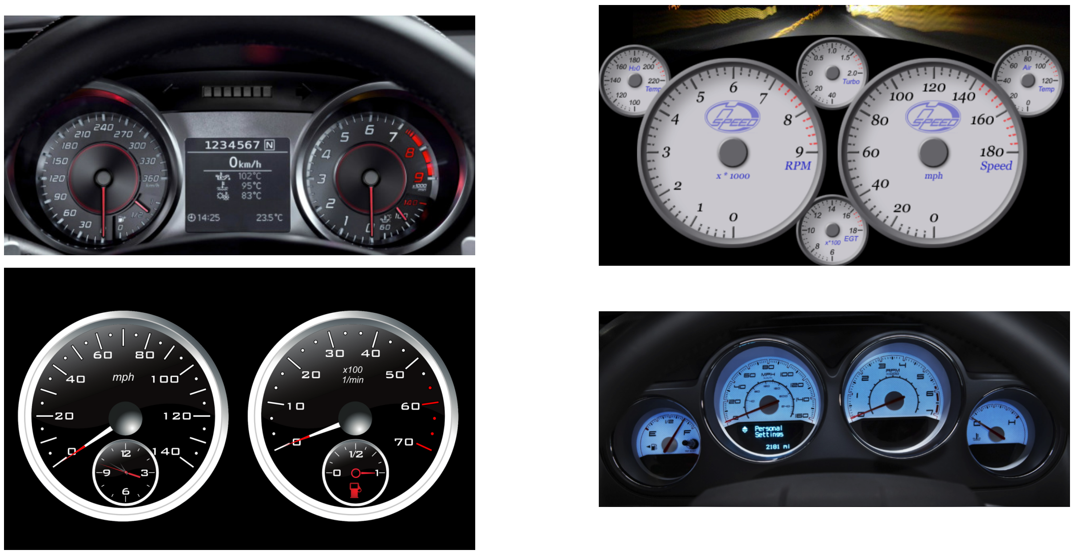


Figure 13: Adopted car dashboard models.

First car dashboard model

The first car dashboard model (Figure 14) is divided into three different areas - left, middle and right.

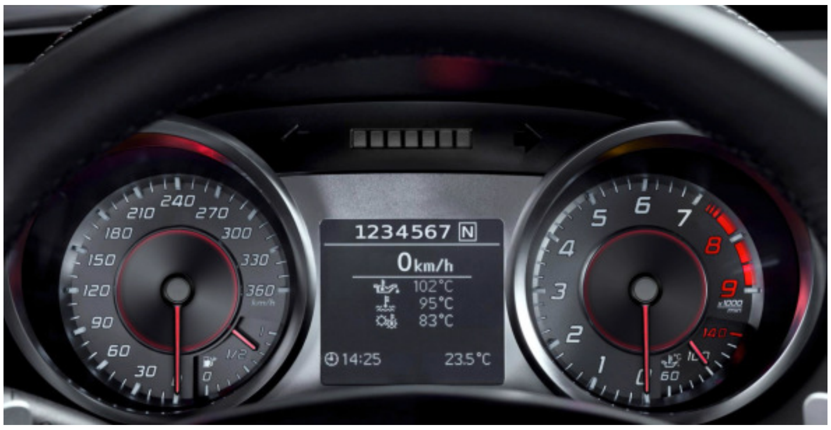


Figure 14: The first adopted car dashboard model.

The left area of the dashboard consists of two overlapped elements:

- **Speedometer gauge:** this gauge graphically shows the speed of the car, in a given moment of time. The configuration includes a maximum value of 360 and a minimum value of 0. The gauge includes 13 major ticks, with an interval of 30 units between each, and two minor ticks evenly separated between each. The gauge pointer rotates clockwise and the metric of the speedometer (kilometres per hour - km/h) is displayed below its maximum value.

- **Remaining fuel gauge:** this gauge graphically shows the remaining fuel of the car. It has a minimum value of 0 and a maximum value of 1, displaying thr remaining fuel as a percentage of the total volume of the tank. The gauge displays 3 major ticks, with 3 minor ones evenly separated between each. The gauge pointer rotates counter-clockwise.

The right area of the dashboard also consists of two elements:

- **Tachometer gauge:** this gauge graphically shows the current speed of the rotation of the car engine. The gauge pointer rotates clockwise between the minimum value of 0 and the maximum value of 9. The range between 7 and 9 is distinguished by being differently colored from the remaining gauge. The metric being used is thousands of rotations per minute (x1000 rpm) and it is displayed below the gauge maximum value.
- **Thermometer gauge:** this gauge displays the current temperature of the car engine. Its values range from 60 to 140, displaying the maximum value in a different color. The gauge contains only 3 major ticks, with 3 minor ones evenly separated between each. The pointer rotates counter-clockwise.

The central area of the dashboard consists of a panel element, which contains several smaller ones:

- **Odometer:** an absolute display of the total distance travelled by the car measured in kilometres.
- **Current shift:** display of the current shift of the car.
- **Speedometer:** a textual display of the speed of the car in a given moment of time, followed by the used metric of kilometres per hour (km/h).
- **Three temperature indicators:** three thermometers that display car element temperatures (engine, oil, etc.) in Celsius degrees. Each element shows a symbol to which the value is referred, followed by the textual display of the value and the used metric (°C).
- **Digital clock:** display of the clock symbol followed by the time in hours and minutes in the format HH:MM.

- **Environment temperature indicator:** thermoter which displays the current environment temperature in Celsius degrees. The textual value is followed by the used metric (ºC).

The presented dashboard is mainly composed of gauges and textual elements. There are several gauge variations reported, with different rotation directions, range values and number of major and minor ticks. The unit used in gauges also varied and it was usually displayed inside them. It can also be report the case of multiple gauges being displayed inside a circular area (overlapping one another). This happens because gauges can have different arcs and so it is possible to display multiple gauges in the same circle without losing information.

Second car dashboard model

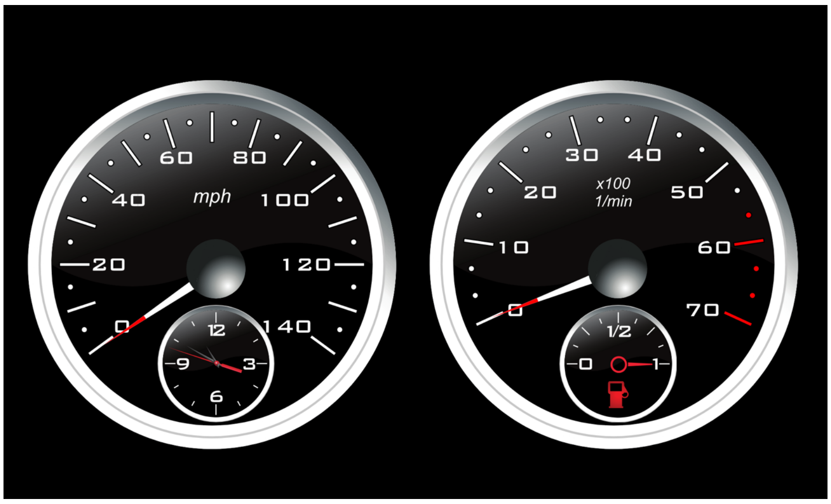


Figure 15: The second adopted car dashboard model.

The second car dashboard model (Figure 15) is divided into two areas - left and right. Similarly to the first analyzed dashboard, the left area of the second model consists of two overlapped elements:

- **Speedometer gauge:** this gauge graphically shows the speed of the car, in a given moment of time. The configuration includes a maximum value of 140 and a minimum value of 0. The gauge includes 8 major ticks, with an interval of 20 units between each, and a single minor tick evenly separated between each. The gauge pointer rotates clockwise and the metric of the speedometer (miles per hour - mph) is displayed below its medium value.

- **Analog clock:** analog clock displays the current time, showing four major ticks labelled with 12, 3, 6 and 9 (intervals of 3 hours) and two minor ticks between each.

The right area of the second dashboard also consists of two overlapped elements:

- **Tachometer gauge:** as seen in the analysis of the first dashboard model, this gauge graphically shows the current speed of the rotation of the car engine. The gauge pointer rotates clockwise between the 0 and 70, distinguishing the range from 60 to 70 by being differently colored from the remaining gauge. The metric being used is hundreds of rotations per minute and it is displayed below the gauge medium value (x100 1/min) .

- **Remaining fuel gauge:** this gauge graphically shows the remaining fuel of the car. It is very similar to the remaining fuel gauge previously analyzed, displaying the remaining fuel as a percentage of the total volume of the tank and using the same number of major and minor ticks. This gauge pointer, however, rotates clockwise.

The selected car dashboard model uses gauge elements and introduces the analog clock element. This element is similar to the gauge, however it has more than one pointer and uses a complete arc, which means that the range of values each pointer takes is circular.

Third car dashboard model

The analysis of the third car dashboard model (Figure 16), reveals that gauge is the main component used. The gauges in the dashboard mainly vary in the range values, number of major and minor ticks and in the labels included (with the gauge metric and additional data - ex: Speed, Turbo). The pointers in the used gauges all rotate clockwise.

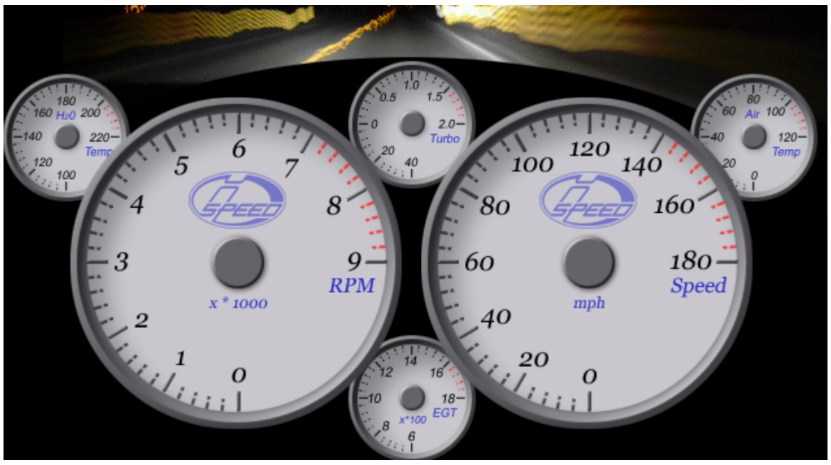


Figure 16: The third adopted car dashboard model.

Fourth car dashboard model

Similarly to the previous model, the fourth selected car dashboard (Figure 17) only uses the gauge component. The dashboard consists on two large gauges in the central area with speed and rotation data, and two smaller ones - one on each side - with remaining fuel and motor temperature data.

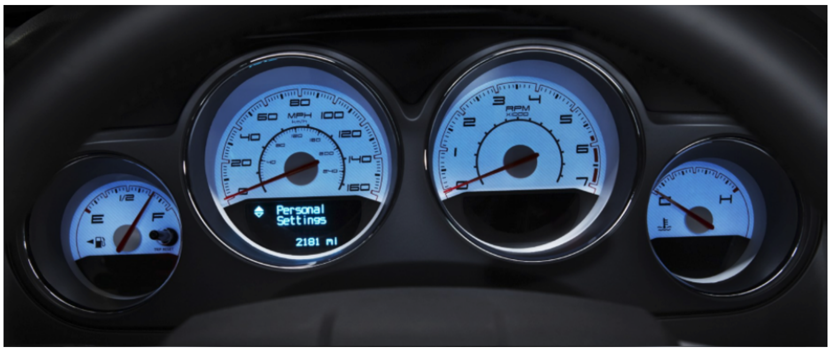


Figure 17: The fourth adopted car dashboard model.

4.4 ANALYSIS RESULTS

After the visual analysis of the selected car dashboard models, it is possible to identify elements with a dominant presence: *speedometers*, *tachometers*, *odometers*, *remaining fuel elements*, and *thermometers*, aside from some less common ones like *clocks*, *dashboard icons* and *current gear elements*.

The identified elements are displayed by a limited set of components: **gauges** (the most used component), **textual and icon display components** and **clock components**. After the realization of the set of components used in car dashboards, it is important to specify which configurations these components should be prepared to handle.

- **Gauge component** The gauge component is used in a variety of contexts. The most important data to which it must adapt includes: the minimum and maximum values, the pointer rotation and the number of major and minor ticks. Additional desired control over the gauge component may involve the ability to distinguish a specific range of values (usually in a different color), identifying the metric in use and displaying it in a certain position (usually below the medium or maximum value), and to define the segment of the arc by which the gauge is drawn, as well as its start and ending points. A usage requirement also involve being able to create nested or overlapped gauges.

- **Textual display component and Icon display component** The textual display component should be prepared for the setting of the text to be displayed, the font size, font color and background color. The icon display component should allow the configuration of the icon to be displayed and its size and color.

- **Clock component** The clock component is used in two scenarios: analog or digital. The choice between analog and digital is the first and most important configuration of the clock component. A digital clock should be configurable in its display format (example: 01:00 or 1:00) and the time convention used (12-hour clock or 24-hour clock). An analog clock should be configurable in the number of major and minor ticks, and support labelling all or specific hours.

At the moment, it is not possible to prototype the selected car dashboards using PVSio-Web since its default widgets library does not provide two of the identified required components - gauges and clocks.

4.5 CONCLUSIONS

Chapter 4 covered car dashboards and the importance of its correct design and user interaction. The use of a software tool to prototype car dashboards was revealed as an unexpensive mean to achieve the validation of car dashboards and the consequent optimal interface.

A collection of car dashboard models were presented and analyzed in order to identify and specify the components that constitute them. Currently, PVSio-Web does not provide the necessary widgets to represent the identified dashboard elements, making it impossible to prototype the selected dashboards. The implementation of new PVSio-Web widgets to represent the identified dashboard components in the PVSio-Web tool will be described in the following chapters.

5 PVSIO-WEB D3-BASED WIDGETS

The first approach to develop PSVio-Web widgets was based on the **d3-gauge-plus** library, generating a set of d3-based widgets. These widget's presentation and behaviour is controlled by a set of configurations provided on the instantiation of the widget.

5.1 USE OF D3.JS AND D3-GAUGE-PLUS.JS

The **d3.js** JavaScript library is part of the basis of the PVSio-Web toolkit, and the widgets can take advantage of its features. A common example of usage of the library is the attachment of specific behaviour to events on the widget's HTML elements (for instance, clicking a button, or dragging events as it was demonstrated in section 3.5), or the handling of DOM elements and properties (e.g. changing an element's class).

Aside from d3.js, other JavaScript libraries can be included to help on the widget's interactions and/or rendering. This is the case of the **d3-gauge-plus**[1], a library specialized in drawing gauges. It is based on a module to generate disk-shaped SVG elements (as circles and arcs), adding the gauge features on top of it (ticks around the drawn circle, text labels, coloured zones and the pointer).

In order to draw a gauge using d3-gauge-plus.js, one must instantiate the Gauge module (providing the id of the HTML element where the gauge will be placed and an object of configurations). The constrcutor of the Gauge module returns a Gauge object, on which the *render()* can be called.

This library offers configurations for changing the values that are displayed around the gauge panel, the number of major and minor ticks, the minimum and maximum gauge values, the gap (in degrees) between the end and the beginning of the gauge or the location of minimum value (also in degrees). It also provides the possibility of changing the style of several of the generated elements (background fill colour, strokes, pointers, coloured zones, text colour and size, etc.).

In Figure 18, an example of a gauge created using the d3-gauge-plus module is displayed.

1 GitHub repository of the d3-gauge-plus library: `https://github.com/gimbo/d3-gauge-plus`

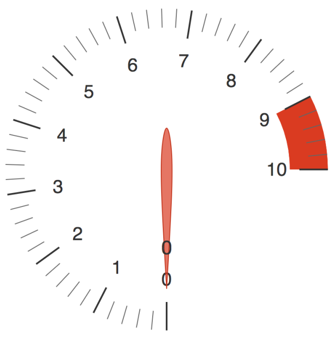


Figure 18: Gauge generated using the d3-gauge-plus library.

5.2 IMPLEMENTED WIDGETS

The implemented widgets include a *Gauge* widget and a *CentralPanel* widget. Both were developed as JavaScript modules, based on the pattern defined by RequireJS[2]. The module defines a JavaScript prototype for the specific widget that is later exported. The implementation using JavaSscript prototype allows the clean definition of the widget constructor, its methods and properties. The packaging of the defined prototype into a module enables that it can easily be required by third parties.

Below follows a description of the implemented d3-based widgets.

5.2.1 *Gauge widget*

The *Gauge* widget allows the easy setup of gauges, a common car dashboard component. It offers a set of configurations to change both the layout as well as the logic of the gauge. Most of these configurations are provided to the d3-gauge-plus library.

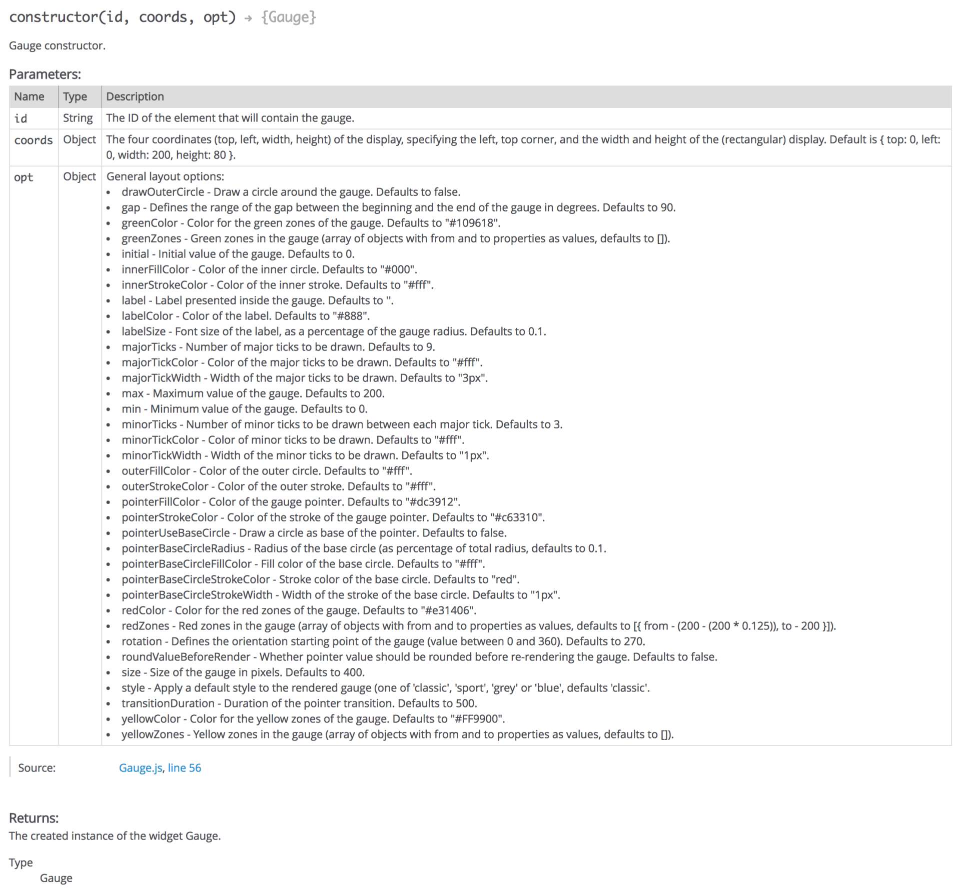
Gauge instantiation and usage

Once the *Gauge* widget is incorporated into the PVSio-Web official widgets library, it will be possible to use it in the PSVio-Web **Prototype Builder**, defining its configurations on a widget instance basis. At the moment, the instantiation of the *Gauge* widget can only be done programatically in the context of PVSio-Web demonstrations, after requiring the *Gauge* module.

The *Gauge* constructor can be called providing a set of arguments. These include the id for the generated HTML element, a coordinates object for positioning the widget on the prototype, and an extensive set of configurations in order to control the gauge and its pointer appearance and logic (for example, the minimum and maximum gauge values).

2 RequireJS online documentation: `http://requirejs.org/`

The full list of the possible configurations can be viewed in Figure 19. The instance obtained from the widget constructor can be rendered by calling the *render()* method with a specific absolute value to be displayed.



```
constructor(id, coords, opt) → {Gauge}
```

Gauge constructor.

Parameters:

Name	Type	Description
id	String	The ID of the element that will contain the gauge.
coords	Object	The four coordinates (top, left, width, height) of the display, specifying the left, top corner, and the width and height of the (rectangular) display. Default is { top: 0, left: 0, width: 200, height: 80 }.
opt	Object	General layout options:
		• drawOuterCircle - Draw a circle around the gauge. Defaults to false.
		• gap - Defines the range of the gap between the beginning and the end of the gauge in degrees. Defaults to 90.
		• greenColor - Color for the green zones of the gauge. Defaults to "#109618".
		• greenZones - Green zones in the gauge (array of objects with from and to properties as values, defaults to []).
		• initial - Initial value of the gauge. Defaults to 0.
		• innerFillColor - Color of the inner circle. Defaults to "#000".
		• innerStrokeColor - Color of the inner stroke. Defaults to "#fff".
		• label - Label presented inside the gauge. Defaults to ''.
		• labelColor - Color of the label. Defaults to "#888".
		• labelSize - Font size of the label, as a percentage of the gauge radius. Defaults to 0.1.
		• majorTicks - Number of major ticks to be drawn. Defaults to 9.
		• majorTickColor - Color of the major ticks to be drawn. Defaults to "#fff".
		• majorTickWidth - Width of the major ticks to be drawn. Defaults to "3px".
		• max - Maximum value of the gauge. Defaults to 200.
		• min - Minimum value of the gauge. Defaults to 0.
		• minorTicks - Number of minor ticks to be drawn between each major tick. Defaults to 3.
		• minorTickColor - Color of minor ticks to be drawn. Defaults to "#fff".
		• minorTickWidth - Width of the minor ticks to be drawn. Defaults to "1px".
		• outerFillColor - Color of the outer circle. Defaults to "#fff".
		• outerStrokeColor - Color of the outer stroke. Defaults to "#fff".
		• pointerFillColor - Color of the gauge pointer. Defaults to "#dc3912".
		• pointerStrokeColor - Color of the stroke of the gauge pointer. Defaults to "#c63310".
		• pointerUseBaseCircle - Draw a circle as base of the pointer. Defaults to false.
		• pointerBaseCircleRadius - Radius of the base circle (as percentage of total radius, defaults to 0.1.
		• pointerBaseCircleFillColor - Fill color of the base circle. Defaults to "#fff".
		• pointerBaseCircleStrokeColor - Stroke color of the base circle. Defaults to "red".
		• pointerBaseCircleStrokeWidth - Width of the stroke of the base circle. Defaults to "1px".
		• redColor - Color for the red zones of the gauge. Defaults to "#e31406".
		• redZones - Red zones in the gauge (array of objects with from and to properties as values, defaults to [{ from - (200 - (200 * 0.125)), to - 200 }]).
		• rotation - Defines the orientation starting point of the gauge (value between 0 and 360). Defaults to 270.
		• roundValueBeforeRender - Whether pointer value should be rounded before re-rendering the gauge. Defaults to false.
		• size - Size of the gauge in pixels. Defaults to 400.
		• style - Apply a default style to the rendered gauge (one of 'classic', 'sport', 'grey' or 'blue', defaults 'classic'.
		• transitionDuration - Duration of the pointer transition. Defaults to 500.
		• yellowColor - Color for the yellow zones of the gauge. Defaults to "#FF9900".
		• yellowZones - Yellow zones in the gauge (array of objects with from and to properties as values, defaults to []).

Source: Gauge.js, line 56

Returns:

The created instance of the widget Gauge.

Type
    Gauge

Figure 19: Documentation of the Gauge widget constructor.

Figure 20 shows the individual instantiating and rendering of a *Gauge* instance without any specifc configurations and its rendered result.

```
define(function (require, exports, module) {
  "use strict";

  // Require the Gauge module
  require("widgets/car/Gauge");

  function main() {
    // After Gauge module was loaded, initialize it
    var gauge = new Gauge(
      // id of the gauge element that will be created
      'speedometer-gauge',
    );

    // Render the Gauge instance
    gauge.render();
  }
});
```

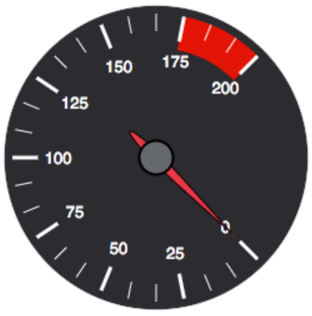


Figure 20: Instantiation and rendering of Gauge with default configuration values.

Widget style variations

The *Gauge* widget provides a set of default styles to speed the bootstrapping of a demonstration. Examples of the provided variations can be seen in Figure 21.

The collection of style variations created target configuring the background colour of the gauge, as well as the pointer appearance. The remaining configurations of the gauge behaviour (e.g. the min and max values, number of ticks, starting and ending angle, etc.) should be provided on the *Gauge* widget constructor call. To use a specific style, the instance should set the "style" parameter on the configurations provided to the constructor with a specific style identifier ("classic", "sport", "grey" or "blue").

Implementation details

The development of the *Gauge* widget followed the **decorator design pattern** [Gamma et al. (1995)]. This pattern was used in this case to decorate the existing d3-gauge-plus library with the PVSio-Web communication behaviour, without affecting the existing library. The responsability of generating the SVG element that visually represents the gauge is delegated to the d3-gauge-plus, and the *Gauge* widget itself controls when the method for setting a new value on the gauge pointer should be called, by re-rendering the d3-gauge-plus gauge object with that value.

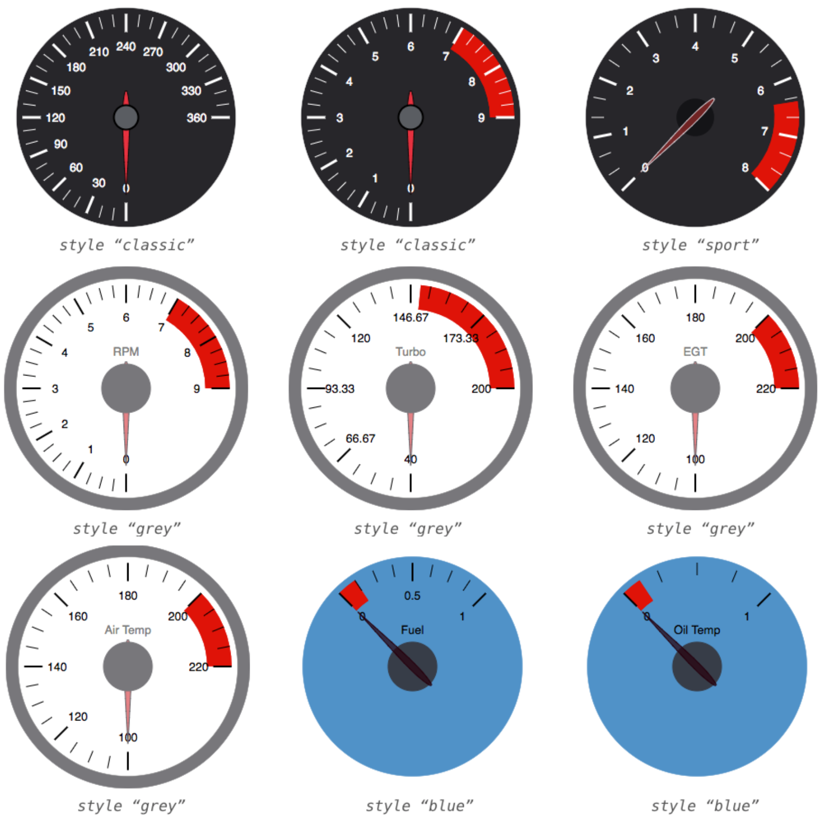


Figure 21: Set of provided default style configurations for the Gauge widget.

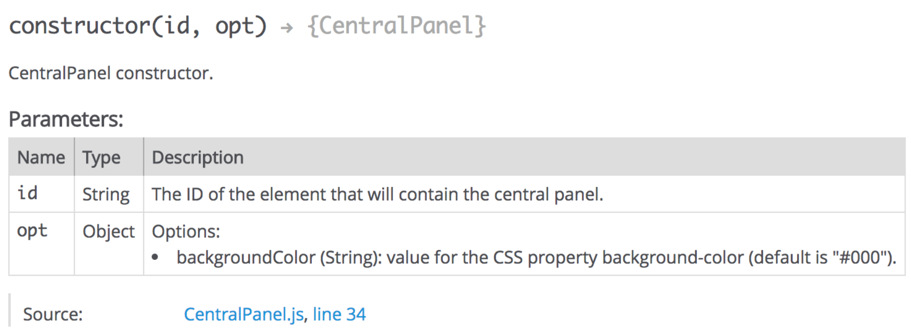
5.2.2 *CentralPanel widget*

The *CentralPanel* widget allows the direct display of a predefined set of components usually found in car dashboards, such as the car absolute speed, odometer, engine and environment temperatures, and the current clock time. It uses the **BasicDisplay** widget provided by PVSio-Web to render the necessary data.

CentralPanel instantiation and usage

As in the case of the *Gauge* widget, once the *CentralPanel* widget is incorporated into the PVSio-Web official widgets library, it will be possible to use it in the PSVio-Web **Prototype Builder**, defining its configurations on a widget instance basis. At the moment, the instantiation of the *CentralPanel* widget is done programatically after requiring the *CentralPanel* module.

An instance of *CentralPanel* is created using its constructor, and rendered with the *render()* method. The constructor call can receive the id of the generated HTML element, a coordinates object for positioning the widget and an object of configurations, which can set the background color for its inner elements (Figure 22).



`constructor(id, opt) → {CentralPanel}`

CentralPanel constructor.

Parameters:

Name	Type	Description
`id`	String	The ID of the element that will contain the central panel.
`opt`	Object	Options: • backgroundColor (String): value for the CSS property background-color (default is "#000").

Source: CentralPanel.js, line 34

Figure 22: Documentation of the CentralPanel widget constructor.

The instantiation and rendering of a *CentralPanel* instance, as well as the obtained result is shown on Figure 23.

Widget style variations

Since the *CentralPanel* widget is a specific car widget, no predefined style variations were implemented.

Implementation details

The development of the *CentralPanel* widget followed the structural **composite pattern** [Gamma et al. (1995)], where several **BasicDisplay** instances are held inside the *CentralPanel* widget. The use of the existing widget to display textual data, complemented by the use of a predefined set of variables to be displayed allowed the fast development of the *CentralPanel* widget.

```
define(function (require, exports, module) {
  "use strict";

  // Require the CentralPanel module
  require("widgets/car/Gauge");

  function main() {
    // After CentralPanel module was loaded, initialize it
    var centralPanel = new CentralPanel("cp");

    // Render the CentralPanel instance
    centralPanel.render();
  }
});
```

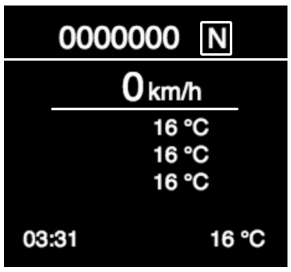


Figure 23: Instantiating and rendering of CentralPanel with default configuration values.

For the purpose of this dissertation, there was no need for further configurations to be added to this widget. However, the concept of aggregating several widgets and creating a single composite one can be of interest in the context of rapid prototyping in the PVSio-Web tool.

5.3 IMPLEMENTATION ANALYSIS

The previous sections describe the implementation of a library of d3-based widgets. The use of existing libraries such as **d3-gauge-plus.js** and **d3.js** was key for the implementation of the *Gauge* widget presented.

The *CentralPanel* widget also emphasizes the possibility of reusing existing PVSio-Web widgets — as it was the case with **BasicDisplay**. This reusage was important in order to maintain the single responsibility principle[3], preventing code duplication and repetition. Additionally, it reveals that building widgets without complex control logic based on JavaScript libraries is fast and lucrative.

The result is a library ready for the development of gauge-like components (such as tachometers and speedometers) and textual displays, which enables achieving functional implementations of some of the components specified in Chapter 4.

However, each dashboard (or even each gauge) has its own peculiatiries, and each difference from the standard would imply a new possible configuration on the *Gauge* widget (or a new widget such as *CentralPanel* to be able to represent it). It is not feasible to represent every component that can be found in a dashboard based only on configurations.

A high number of configurations by itself is not a problem for someone using the widgets to build prototypes. Default values can be set, so that the users only need to change the configurations that they may need. Nonetheless, adding more configurations is an attempt to make the widgets' code more flexible. In the long run, it will be much more difficult to maintain the library, since every non-standard dashboard would imply changing the widgets' source code in order to support new configurations or features.

An example of the lack of flexibility that could not be overcome using configurations is the instantiation of the *Gauge* widget to display two overlapped gauges. The *Gauge* widget could not support two gauges with different rotation directions (clockwise and counter-clockwise) and different pointer positions (Figure 24). In order to be able to support this, it would be necessary to change the source code of the used library (**d3-gauge-plus**).

An alternative path can be searched to overcome the shortcomings of this approach. This alternative should lower the need for extensive layout configurations and focus on the most relevant ones. However, any path taken must take into account the requirements for the components found in car dashboards described in section 4.4.

3 Wikipedia page on SRP: `https://en.wikipedia.org/wiki/Single_responsibility_principle`

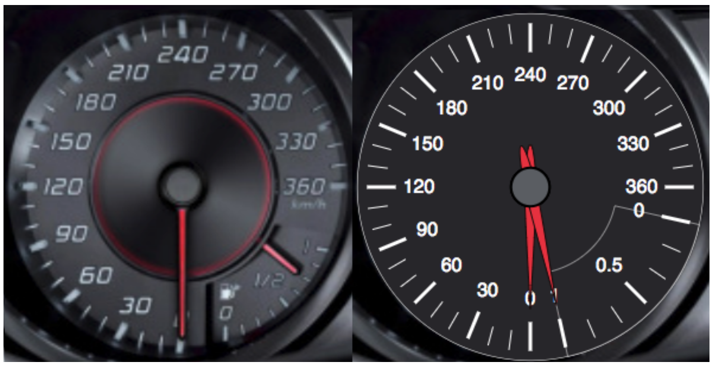


Figure 24: Example of the implementation of overlapped arc-shaped gauges using d3-based widgets.

An approach based on graphics would allow to achieve this kind of behaviour, bringing a higher degree of flexibility to the library. Aside from the increment of flexibility, the development effort could also be significantly decreased, as there are libraries of SVG vector graphics available to jumpstart the development process.

5.4 CONCLUSIONS

This chapter covered the implementation of PVSio-Web d3-based widgets. These widgets are based on existing JavaScript libraries — d3.js and d3-gauge-plus.js. The description of the two widgets implemented (*Gauge* and *CentralPanel*) was provided, as well as examples of instatiation and possible configurations for these widgets.

The widgets developed follow the requirements specified in Chapter 4 for the components that need to be developed for the prototyping of car dashboards. However, section 5.3 indicates some of the disadvantages of this approach, such as the lack of flexibility and the excessive configurations. The alternative path suggested (an approach based on SVG files) will be described in more detail in the next chapter.

6 PVSIO-WEB SVG-BASED WIDGETS

The second approach to develop PSVio-Web widgets was based on the composition of **SVG files** chosen to mime the components being developed. This approach generated SVG-based widgets, where the widgets' layout is no longer configureable, but supported by a SVG file. A configuration object should still be provided, to control the behaviour of the widgets (such as positioning of the elements on the screen and values that control the logic of the widget). However, the principle behind this approach is that most of the customization should be done on the SVG file.

This chapter focuses on providing a set of usable widgets with several default styles (based on different SVG files), as well as a method for easily creating new styles without having to change the widgets' source code.

6.1 USE OF SVG

As referred in section 2.5, SVG files describe vector graphics following a XML-based structure. Vector graphics are treated as DOM elements by browsers, making them especially useful in Web development since they can be created, positioned, animated, changed or deleted on runtime.

Using SVG in the development of PVSio-Web widgets implies the existence of a set of usable SVG files for the widgets elements, on which animations can be applied in order to simulate the dashboard interactivity. In this sense, the widget behaviour is built upon the SVG file.

One of the advantages of using SVG is that it is fairly easy to buy, download, or, for those with graphic design experience, build SVG files. The files used in this dissertation were obtained from free sources on the Internet[1].

1 The online SVG library used: `http://all-free-download.com/`

6.2 IMPLEMENTED WIDGETS

The collection of SVG-based widgets implemented includes *GaugeSport* widget (for the display gauge components), *Clock* widget (for the display of analog clocks), *Pointer* widget (for the display of a rotating pointer, used by *GaugeSport* and *Clock*) and *Gearbox* widget (for the representation of gear boxes).

It should be pointed that since these widgets have not yet been incorporated into the PVSio-Web official widgets library, they can not be used in prototypes using the PSVio-Web **Prototype Builder**.

Each listed widget is defined as a JavaScript module which exports a specific widget prototype defined inside the module. The usage of JavaScript prototypes enables the existence of the widget constructor, methods and properties values, besides allowing to take advantage of inheritage in JavaScript.

All implemented widgets share common features, and so a conceptually abstract ***SVG-Widget*** prototype was implemented. This prototype is extended by every SVG-based widget prototype.

The class diagram on Figure 25 illustrates the conceptual inheritance relationships of the developed widgets.

6.2.1 *SVGWidget*

The *SVGWidget* is a JavaScript prototype that aggregates common features and behaviours shared by the SVG-based widgets. Besides providing the implementation of common methods such as *getId()*, *getValue()*, *isReady()*, *show()* and *hide()*, this prototype aims to establish a set of standard behaviours for all the SVG-based widgets.

A clear example where *SVGWidget* does not provide an actual implementation but encourages a standard behaviour is the *getDefaultStyleConfigs()* method, which receives a style identifier as argument.

Standard behaviour of default styles

All SVG-based widget instances receive a **style identifier** on the configuration object paramenter of the widget constructor (which should be set with a default value if not provided). This identifier points to a set of configuration values that aggregate common aspects of the behaviour and/or appearance of the widget (these values and its meaning may vary from widget to widget).

However, there are two main behaviours underlying this structure:

- The first is that the style identifier's configurations are merged with the configuration received in the widget constructor call, in a way that only missing values are obtained

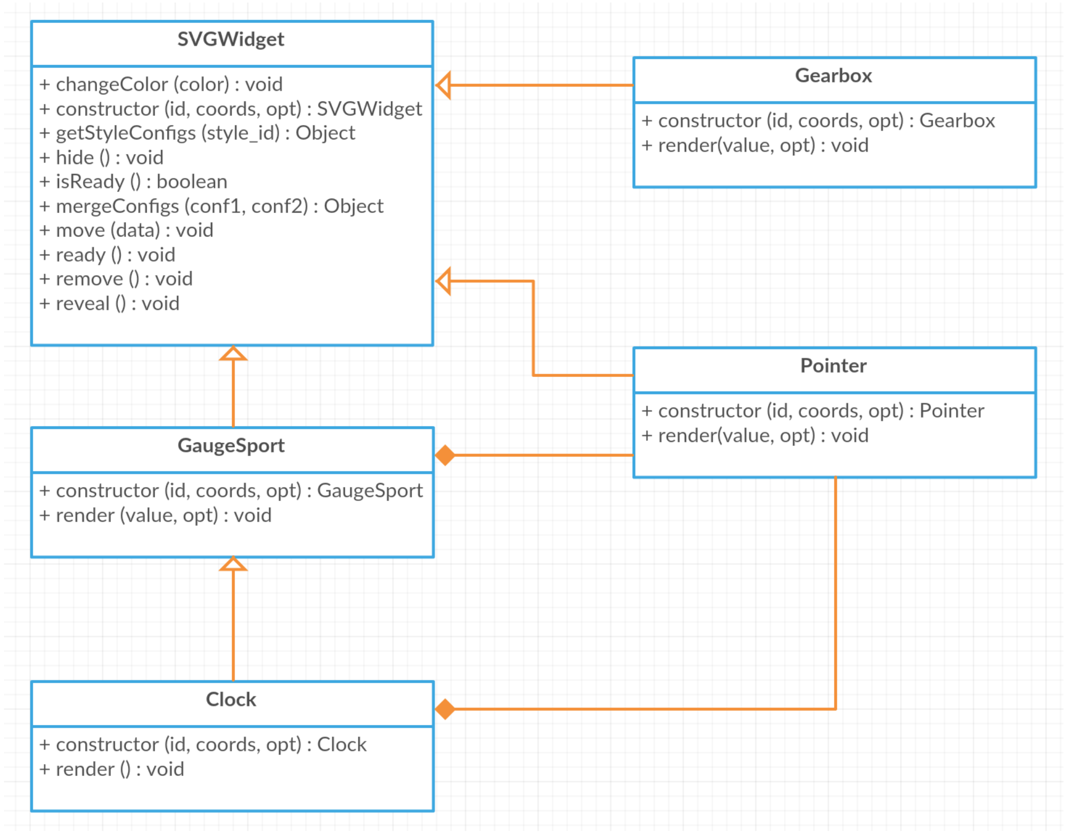

SVGWidget
+ changeColor (color) : void
+ constructor (id, coords, opt) : SVGWidget
+ getStyleConfigs (style_id) : Object
+ hide () : void
+ isReady () : boolean
+ mergeConfigs (conf1, conf2) : Object
+ move (data) : void
+ ready () : void
+ remove () : void
+ reveal () : void
Gearbox
+ constructor (id, coords, opt) : Gearbox
+ render(value, opt) : void
Pointer
+ constructor (id, coords, opt) : Pointer
+ render(value, opt) : void
GaugeSport
+ constructor (id, coords, opt) : GaugeSport
+ render (value, opt) : void
Clock
+ constructor (id, coords, opt) : Clock
+ render () : void

Figure 25: Class diagram for the developed widgets

from the style configurations. This means the style identifier provides the default configurations for the widget instance, but these can be overriden at any time.

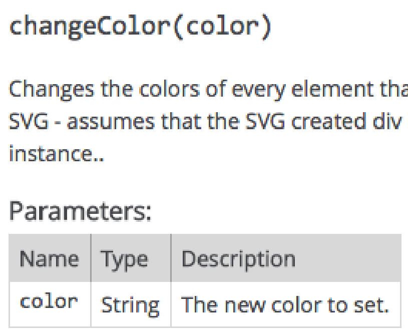
- The second is that any set of configurations that may be repeated in multiple widget instances and form a **pattern or a type of packaged example** of the widget should be included in the *getDefaultStyleConfigs()* method.

The addition of a new default style can be done by adding a new style identifier and its set of configurations to the result of the *getDefaultStyleConfigs()* method. This can be done directly by changing the widget's method implementation, or creating a new widget module, whose internal prototype extends the desired one and implements the *getDefault-StyleConfigs()* method with its own case added. This allows the creation of a **new default style variations** for the widget, which can then be used by any instance which set its style identifier to the identifier of the newly created style.

Changing colour of SVG elements feature

One of the features that is introduced by the *SVGWidget* is dynamically changing the colour of a widget's elements. This is achieved by defining the implementation of the *changeColor()*

method on the *SVGWidget* prototype, which means that the feature is inherited by all SVG-based widgets extending it (see Figure 26 for documentation).



changeColor(color)

Changes the colors of every element that has the "color-change-aware" in the SVG - assumes that the SVG created div is on the "div" property of the instance..

Parameters:

Name	Type	Description
color	String	The new color to set.

Figure 26: Documentation for the changeColor method from SVGWidget prototype.

The *changeColor()* method allows to dynamically change the colours of a widget's generated DOM elements by filtering the ones with the "color-change-aware" class and setting their style *fill* property to a specific value received as argument — SVG elements do not respond to the usual CSS *color* and *background-color* properties, but use specific style properties instead (Figure 27).

In order to take advantage of this feature, the used SVG files must be edited by adding the class to the elements that should be able to change color.

In the example shown in Figure 28, obtained from a *SVGWidget* widget dedicated to drawing gauges, the used SVG file contains the elements corresponding to the four icons in the image (temperature, fuel, battery and engine) identified with the referred class. Unless the *changeColor()* method is directly called over the gauge instance, no colour transformation is applied to the elements and they will be rendered exactly as the original. The image on the left shows the gauge immediately after being initially rendered, and the one on the right shows the same gauge after being called the *changeColor()* method with the parameter "yellow".

```
<svg height="130" width="500">
  <defs>
    <linearGradient id="grad1" x1="0%" y1="0%" x2="100%" y2="0%">
      <stop offset="0%" style="stop-color:rgb(255,255,0);
           stop-opacity:1" />
      <stop offset="100%" style="stop-color:rgb(255,0,0);
           stop-opacity:1" />
    </linearGradient>
  </defs>
  <ellipse cx="100" cy="70" rx="85" ry="55" fill="url(#grad1)" />
  <text fill="#ffffff" font-size="45" font-family="Verdana" x="50" y=
       "86">SVG</text>
</svg>
```

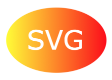


Figure 27: Example of SVG file using specific style and its end result.

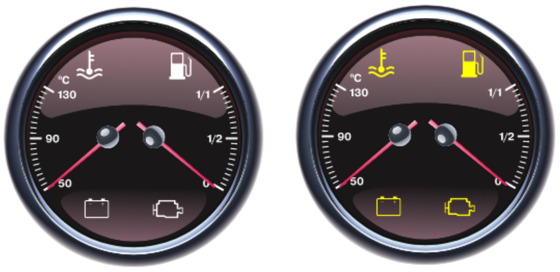


Figure 28: Example of calling the changeColor method with the ”yellow” colour.

6.2.2 *Pointer widget*

The *Pointer* widget displays a rotating pointer. This widget is usually instanciated inside a *GaugeSport* or *Clock* widget.

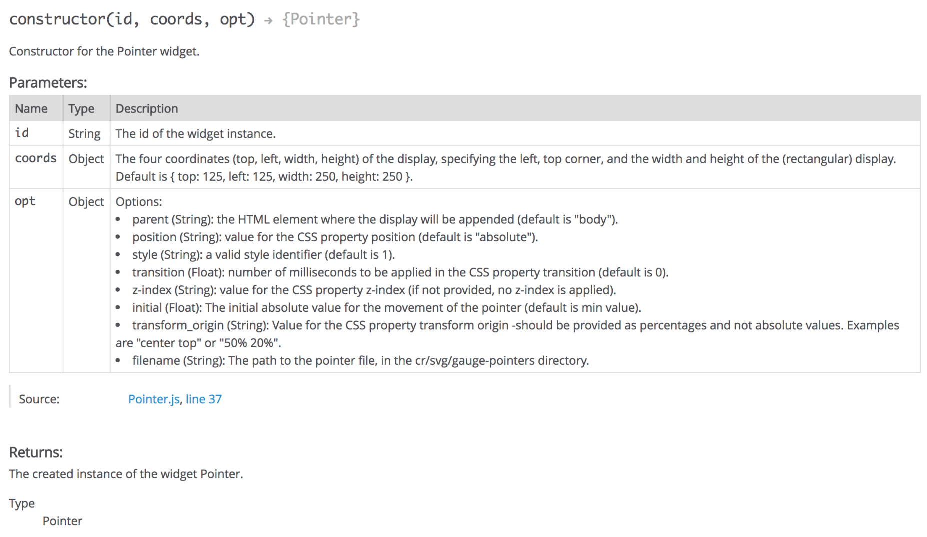
The *Pointer* widget depends solely of a single SVG file with the pointer that will be displayed, which can be specified in the configurations of the widget (if no configuration value is provided, an example pointer will be rendered). The pointer will be rotated through the call of the render() method, receiving the degree of the new angle of rotation. The rotation of the pointer is applied using the CSS property *transform*[2].

Pointer instantiation and usage

The instantiation of the *Pointer* widget can done programatically in a demonstration, after requiring the Pointer module. The *Pointer* constructor is called with the desired set of arguments - including an unique id, set of coordinates to position the widget and an optional configurations object. These configurations include the name of the SVG file containing the pointer to use, the initial value of the pointer rotation, a set of CSS properties values and the parent object identifier. Additionally, following the standard among the *SVGWidget* widgets, one of the possible configurations is the style identifier. The default style identifier configures the origin of the rotation animation to applied on the pointer - setting it at (0,0) coordinates. This can be overridden by specifying a different value for ”transform-origin” configuration (e.g. {`"transform-origin" : "20% 20%"`}). The documentation for the constructor can be seen in Figure 29.

The *Pointer* widget instance obtained by the widget constructor call is rendered by calling its *render()* method. If an argument is provided to the render method, its value will be used as the degree of rotation of the pointer - initially all pointer are pointing down, with an angle of 0º.

2 Mozilla Developer Network guide on the CSS transform property: `https://developer.mozilla.org/en-US/docs/Web/CSS/transform`



constructor(id, coords, opt) → {Pointer}

Constructor for the Pointer widget.

Parameters:

Name	Type	Description
id	String	The id of the widget instance.
coords	Object	The four coordinates (top, left, width, height) of the display, specifying the left, top corner, and the width and height of the (rectangular) display. Default is { top: 125, left: 125, width: 250, height: 250 }.
opt	Object	Options: • parent (String): the HTML element where the display will be appended (default is "body"). • position (String): value for the CSS property position (default is "absolute"). • style (String): a valid style identifier (default is 1). • transition (Float): number of milliseconds to be applied in the CSS property transition (default is 0). • z-index (String): value for the CSS property z-index (if not provided, no z-index is applied). • initial (Float): The initial absolute value for the movement of the pointer (default is min value). • transform_origin (String): Value for the CSS property transform origin -should be provided as percentages and not absolute values. Examples are "center top" or "50% 20%". • filename (String): The path to the pointer file, in the cr/svg/gauge-pointers directory.

Source: Pointer.js, line 37

Returns:

The created instance of the widget Pointer.

Type

Pointer

Figure 29: Documentation of the Pointer widget constructor.

```
define(function (require, exports, module) {
  "use strict";

  // Require the Pointer module
  require("widgets/car/Pointer");

  function main() {
    // After Pointer module was loaded, initialize it
    var pointer = new Pointer(
      // id of the gauge element that will be created
      'pointer',
    );

    // Render Pointer instance
    pointer.render();
  }
});
```

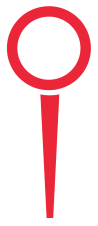


Figure 30: Instantiation and rendering of Pointer with default values.

The individual instantiating and rendering a *Pointer* instance without additional configurations and the obtained result is displayed in Figure 30.

Widget style variations

Since the *Pointer* widget allows the use of different SVG files as pointers, and given the simple nature of its behaviour, it was not found necessary to expand on the widget style variations.

Implementation details

The implementation of the *Pointer* widget is a perfect example of taking advantage of the surrounding technologies to build a simple and reusable widget.

An implementation step to which was given special attention was enabling the correct behaviour of the *Pointer* widget using different SVG files. It is not guaranteed that all pointers will have the same rotation origin, as it is can be seen on Figure 31. In the figure, the left pointer has **50% 20%** as value for the *transform-origin* property (the first value is relative to the X axis, and the second to the Y axis), while the right one has **50% 0%**.

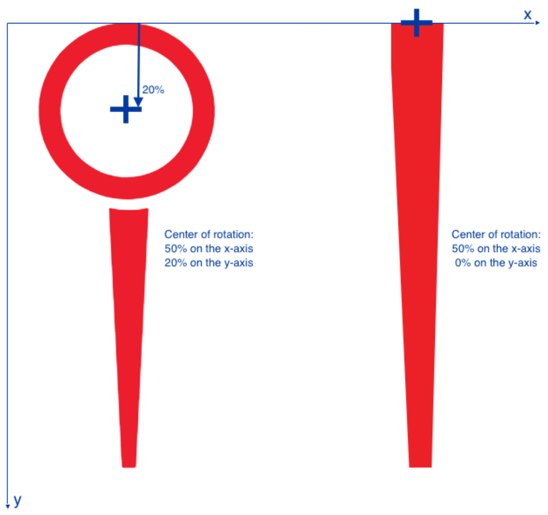


Figure 31: Example of two pointers with different centres of rotation.

As previously stated, the rotation of the pointer is applied using the CSS property *transform*. Taking further advantage of the CSS properties, the widget also allows the control of the *transform-origin*[3] property to adjust the origin of the animation. This is achieved by setting the rotation origin of the pointer in the configurations provided to the widget constructor call.

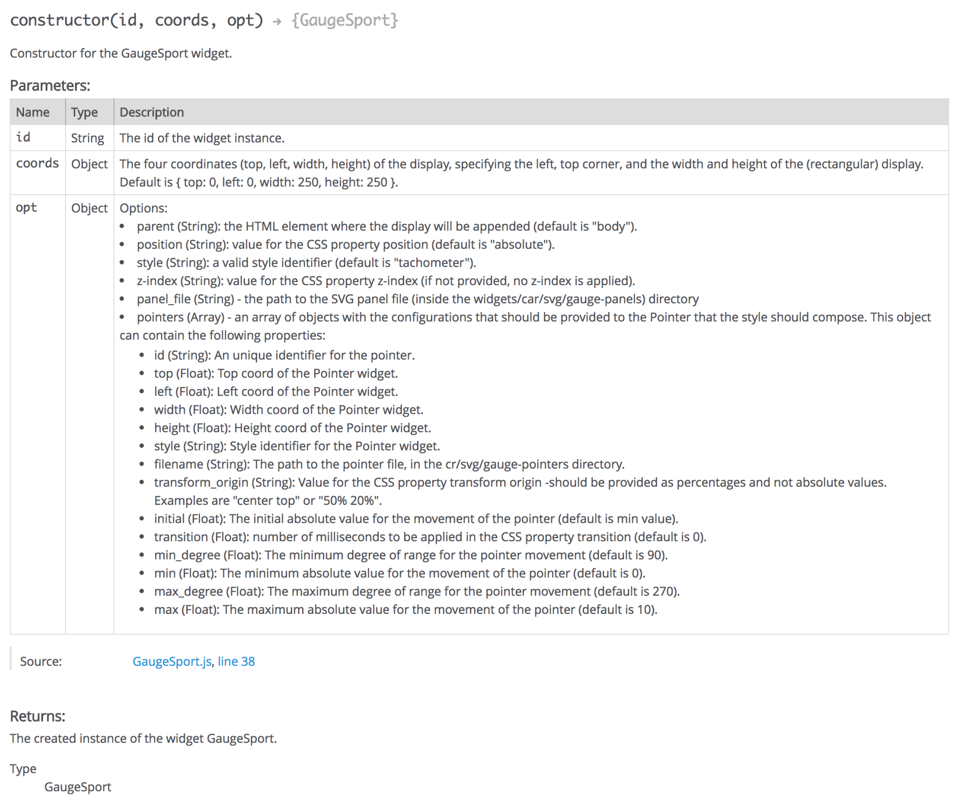
Additionally, in order to achieve a more realistic behaviour, the CSS *transition* property is also (optionally) configurable. The need to control the duration of the animation is more obvious when applying the *Pointer* widget in *GaugeSport* instances representing compasses or remaining fuel gauges.

6.2.3 *GaugeSport widget*

The *GaugeSport* widget displays gauge elements in a flexible way. The widget is composed by a main SVG file (used as the gauge panel element) and a set of *Pointer* widget instances. This widget is mainly used with a single pointer to represent elements as speedometers, tachometers and compasses.

3 Mozilla Developer Network guide on the CSS transform-origin property: `https://developer.mozilla.org/en-US/docs/Web/CSS/transform-origin`

The widget constructor allows the configuration of the SVG used as panel as well as some visual properties, and the configuration of the *Pointer* instances to be used as well as additional control for the movement of each pointer.



```
constructor(id, coords, opt) → {GaugeSport}
```

Constructor for the GaugeSport widget.

Parameters:

Name	Type	Description
id	String	The id of the widget instance.
coords	Object	The four coordinates (top, left, width, height) of the display, specifying the left, top corner, and the width and height of the (rectangular) display. Default is { top: 0, left: 0, width: 250, height: 250 }.
opt	Object	Options: • parent (String): the HTML element where the display will be appended (default is "body"). • position (String): value for the CSS property position (default is "absolute"). • style (String): a valid style identifier (default is "tachometer"). • z-index (String): value for the CSS property z-index (if not provided, no z-index is applied). • panel_file (String) - the path to the SVG panel file (inside the widgets/car/svg/gauge-panels) directory • pointers (Array) - an array of objects with the configurations that should be provided to the Pointer that the style should compose. This object can contain the following properties: • id (String): An unique identifier for the pointer. • top (Float): Top coord of the Pointer widget. • left (Float): Left coord of the Pointer widget. • width (Float): Width coord of the Pointer widget. • height (Float): Height coord of the Pointer widget. • style (String): Style identifier for the Pointer widget. • filename (String): The path to the pointer file, in the cr/svg/gauge-pointers directory. • transform_origin (String): Value for the CSS property transform origin -should be provided as percentages and not absolute values. Examples are "center top" or "50% 20%". • initial (Float): The initial absolute value for the movement of the pointer (default is min value). • transition (Float): number of milliseconds to be applied in the CSS property transition (default is 0). • min_degree (Float): The minimum degree of range for the pointer movement (default is 90). • min (Float): The minimum absolute value for the movement of the pointer (default is 0). • max_degree (Float): The maximum degree of range for the pointer movement (default is 270). • max (Float): The maximum absolute value for the movement of the pointer (default is 10).

Source: GaugeSport.js, line 38

Returns:

The created instance of the widget GaugeSport.

Type

GaugeSport

Figure 32: Documentation of the GaugeSport widget constructor.

GaugeSport instantiation and usage

In order to programatically instantiate the *GaugeSport* widget, its module should be required and its constructor called.

The call to the *GaugeSport* constructor includes an expressive set of configurations. The first level of configurations involves the style of the widget to be applied, the SVG file of the panel to be used, as well as some optional visual properties (like `z-index` and `position` CSS properties) and an array of pointer configurations (see Figure 32 for complete documentation of the constructor method).

The pointer configurations are handled by the *GaugeSport* widget and indicate some basic information about each pointer to be rendered (its filename, style identifier and visual properties values), and the range of its rotations - using min_degree and max_degree parameters. The specification of the rotation range will allow the *GaugeSport* widget to handle the calculations needed in order to render a pointer, simply by providing its new angle of rotation. It also enables the use of a clockwise or counter-clockwise rotation of the pointer (for counter-clockwise rotation, min_degree and max_degree parameters should swap their values).

Once obtained a *GaugeSport* widget instance, it can be rendered by calling its render method. This method receives an object with the pointer(s) identifier(s) and its(their) new absolute values value(s).

As an example, consider the use of the *GaugeSport* instance to display a car's tachometer, with its pointer rotating clockwise and the car's engine speed rotation ranging from 0 to 10 (x1000/min. rotations). In order to instantiate the widget, the range of the rotation angles of the pointer should be discovered - see Figure 33.

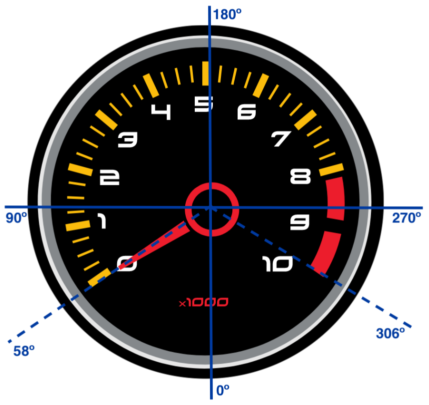


Figure 33: Discovery of min and max parameters for the rotation of the referred tachometer.

The instantiation of the *GaugeSport* widget should configure a set of one pointer, with maximum and minimum values of 0 and 10 and maximum degree and minimum degree of 58º and 306º. This means that the render of the 0 value will match the minimum degree rotation of 58º and the render of the 10 value will match the maximum degree rotation of 306º, hence rotating the pointer clockwise. The *render* method of the *GaugeSport* instance can then be called with an argument specifying the new pointer value (a number between its **min** (0) and **max** (10)) — see **??** for an instantiation example and its rendered result.

Widget style variations

As the standard among the *SVGWidget*, a set of predefined style variations were implemented in order to allow the reuse of configurations as well as providing a collection of out-of-the-box *GaugeSport* variations. These were chosen based on the most common components usually found on car dashboards (Chapter 4).

The styles variations shipped with the *GaugeSport* are separated into five gauge groups: tachometers, speedometers, remaining fuel, engine temperature displays and compasses. The default style identifier used in *GaugeSport* instances is "tachometer" (first gauge dis-

```
define(function (require, exports, module) {
  "use strict";

  // Require the GaugeSport module
  require("widgets/car/GaugeSport");

  function main() {
    // After GaugeSport module was loaded, initialize it
    var tachometer = new Pointer(
      // id of the gauge element that will be created
      'tach',
      // coordinates object
      { top: 100, left: 100, width: 300, height: 300},
      // configuration object
      {
        panel_file: 'gauge-tachometer-panel-1.svg',
        pointers: [
          {
            id: 'tachometer-pointer',
            min: 0,
            max: 10,
            min_degree: 58,
            max_degree: 306,
            width:38,
            top: 133,
            left: 133
          }
        ]
      }
    );

    // Render GaugeSport instance with specific value
    pointer.render(2);
  }
});
```

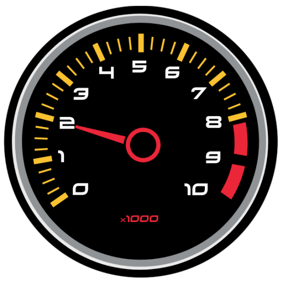


Figure 34: GaugeSport instance with example configuration values and rendered value of 2.

played in Figure 35). The full catalog of the variations is presented in Figure 35, Figure 36, Figure 37 and Figure 38.

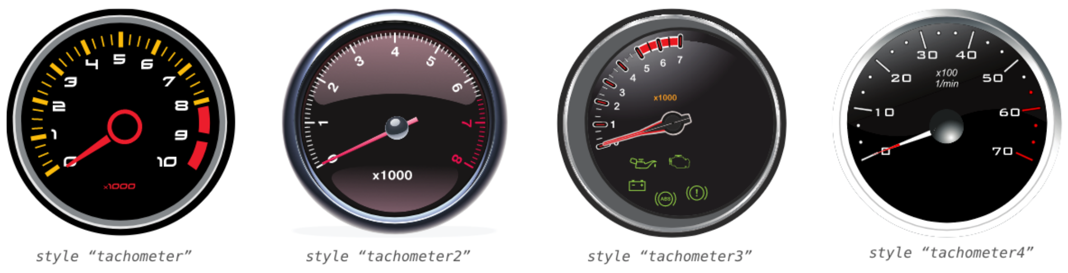


Figure 35: The set of default variations for the tachometer type.

The styles variations were created by specifying a set of style identifiers and a configurations object for each one. The mapping between the two was included in the *getDefaultStyleConfigs()* method of the *GaugeSport* widget. This method is used in the constructor of the widget, using the returned object of configurations to complement any missing configurations of the instance being created.

In order to use a specific style variation of the *GaugeSport*, the configurations provided to the widget constructor when creating a new instance must include the "style" parameter with the desired style identifier as its value. Additional configurations can also be provided in order to override some of the style values.

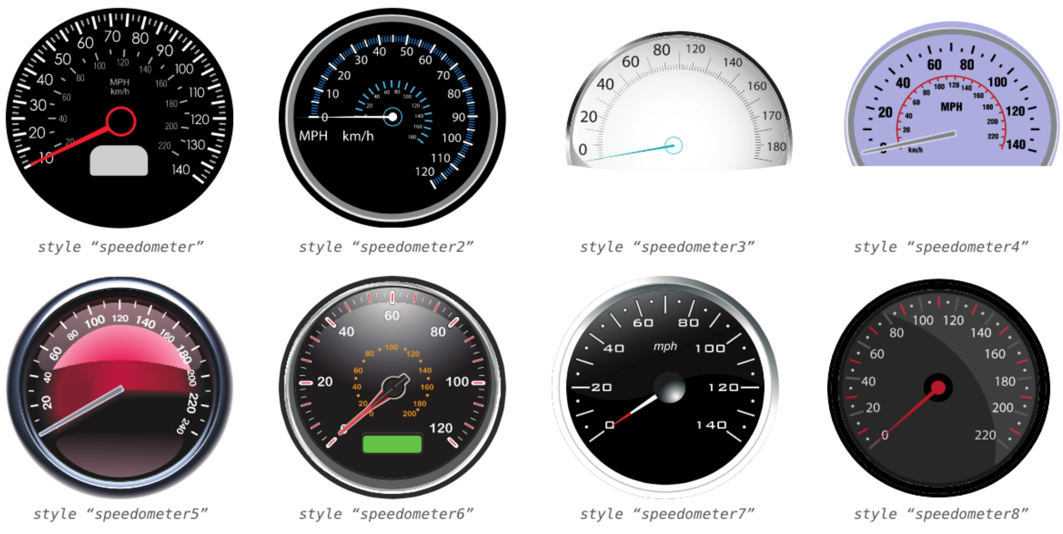


Figure 36: The set of default variations for the speedometer type.

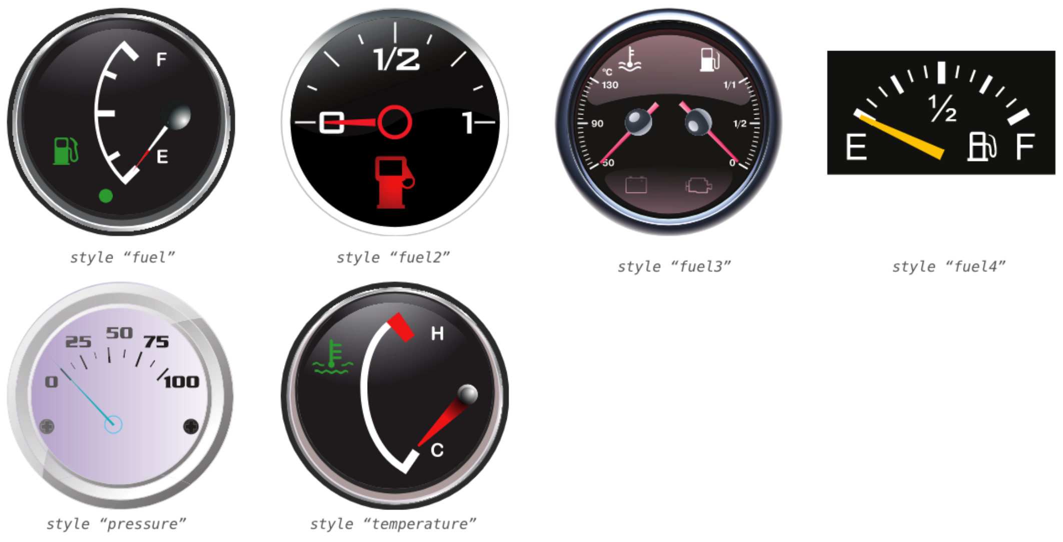


Figure 37: The set of default variations for the remaining fuel, thermometer and pressure types.

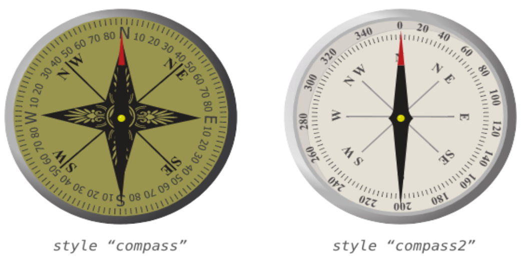


Figure 38: The set of default variations for the compass type.

Implementation details

The *GaugeSport* widget uses the **composition design pattern**. Each *GaugeSport* instance includes one or more *Pointer* instances in order to achieve the widget's purpose. The use of a separate widget to render the gauge's pointers allowed a cleaner design and a more obvious separation of concerns between the two widgets. The *GaugeSport* holds the control of the gauge rendering as well as its inner logic, conveniently using the *Pointer* widget to handle the pointers configurations and rendering.

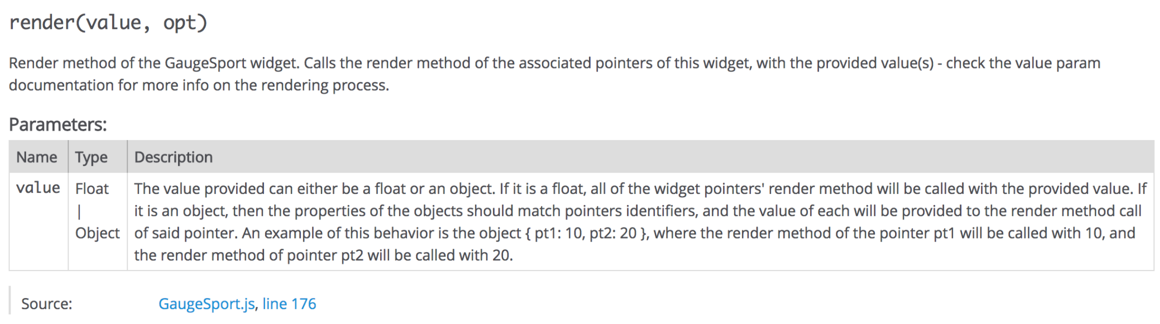
One aspect given special attention when implementing the *GaugeSport* widget was the ease of its usage. Beside allowing the extension of style configurations for a faster ("out-of-the-box") usage, the way the gauge was rendered was also taken into account.

The render method of *GaugeSport* includes the rendering of the gauge pointers, by calling the *Pointer* render method which receives a specific angle of rotation to be applied over the pointer element. In order to avoid sending the rotation angle of each pointer when rendering a *GaugeSport* instance, additional pointer configurations were supported in its constructor method so that the the instance could interpolate the correct angle of rotation of its pointers using the formula:

$$rotationAngle(x) = minDegree + \frac{x - min}{max - min} \times (maxDegree - minDegree)$$

, where x is the absolute value to be displayed, **min** and **max** are the pointer maximum and minimum values (received in the pointer configurations passed to the widget constructor - under **min** and **max**), and **maxDegree** and **minDegree** are the pointer maximum and minimum degree values (also received in the pointer configurations passed to the widget constructor - under **max_degree** and **min_degree**).

Interpolating the pointers angles of rotation allowed the render method of the *GaugeSport* to be cleaner (Figure 39) and its use more comfortable. This is especially relevant when the gauge is being rendered using values provided by the PVS model.



```
render(value, opt)
```

Render method of the GaugeSport widget. Calls the render method of the associated pointers of this widget, with the provided value(s) - check the value param documentation for more info on the rendering process.

Parameters:

Name	Type	Description
`value`	Float | Object	The value provided can either be a float or an object. If it is a float, all of the widget pointers' render method will be called with the provided value. If it is an object, then the properties of the objects should match pointers identifiers, and the value of each will be provided to the render method call of said pointer. An example of this behavior is the object { pt1: 10, pt2: 20 }, where the render method of the pointer pt1 will be called with 10, and the render method of pointer pt2 will be called with 20.

Source: GaugeSport.js, line 176

Figure 39: Documentation of the GaugeSport render method.

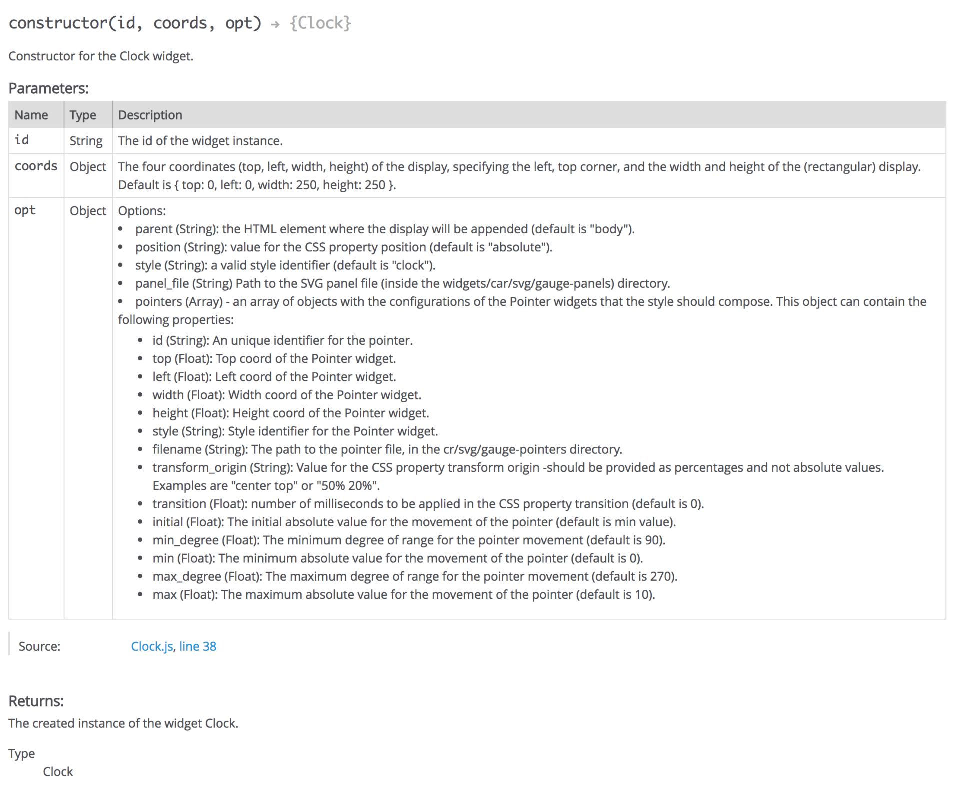
6.2.4 *Clock widget*

The *Clock* widget is a more specific *GaugeSport*, dedicated to the display of analog clocks. Similarly to the *GaugeSport*, this widget is composed by a main SVG file used as the clock background and a set of *Pointer* widget instances. This widget is usually used with two or three pointers (hours, minutes and seconds pointers).

Clock instantiation and usage

The *Clock* widget can be programatically instantiated, using its constructor after its module is required.

The instatiation of a *Clock* widget is very similar to the instantiation of a *GaugeSport* widget, since the constructor also receives configurations involving the style of the widget to be applied, the SVG file to be used, some optional visual properties (like z-index and position CSS properties) and an array of pointer configurations (Figure 40 displays the documentation of the widget constructor).



`constructor(id, coords, opt) → {Clock}`

Constructor for the Clock widget.

Parameters:

Name	Type	Description
`id`	String	The id of the widget instance.
`coords`	Object	The four coordinates (top, left, width, height) of the display, specifying the left, top corner, and the width and height of the (rectangular) display. Default is { top: 0, left: 0, width: 250, height: 250 }.
`opt`	Object	Options: • parent (String): the HTML element where the display will be appended (default is "body"). • position (String): value for the CSS property position (default is "absolute"). • style (String): a valid style identifier (default is "clock"). • panel_file (String) Path to the SVG panel file (inside the widgets/car/svg/gauge-panels) directory. • pointers (Array) - an array of objects with the configurations of the Pointer widgets that the style should compose. This object can contain the following properties: ◦ id (String): An unique identifier for the pointer. ◦ top (Float): Top coord of the Pointer widget. ◦ left (Float): Left coord of the Pointer widget. ◦ width (Float): Width coord of the Pointer widget. ◦ height (Float): Height coord of the Pointer widget. ◦ style (String): Style identifier for the Pointer widget. ◦ filename (String): The path to the pointer file, in the cr/svg/gauge-pointers directory. ◦ transform_origin (String): Value for the CSS property transform origin -should be provided as percentages and not absolute values. Examples are "center top" or "50% 20%". ◦ transition (Float): number of milliseconds to be applied in the CSS property transition (default is 0). ◦ initial (Float): The initial absolute value for the movement of the pointer (default is min value). ◦ min_degree (Float): The minimum degree of range for the pointer movement (default is 90). ◦ min (Float): The minimum absolute value for the movement of the pointer (default is 0). ◦ max_degree (Float): The maximum degree of range for the pointer movement (default is 270). ◦ max (Float): The maximum absolute value for the movement of the pointer (default is 10).

Source: Clock.js, line 38

Returns:
The created instance of the widget Clock.

Type
Clock

Figure 40: Documentation of the Clock widget constructor.

The pointers of the *Clock* widget should be identified with ”hours”, ”minutes” and ”seconds”. These can be freely configured in its appearance but should have a specific config-

```
define(function (require, exports, module) {
  "use strict";

  // Require the Clock module
  require("widgets/car/Clock");

  function main() {
    // After Clock module was loaded, initialize it
    var clock = new Clock(
      // id of the element that will be created
      'clock',
      // coordinates object
      { top: 100, left: 100},
      {
        panel_file: 'gauge-clock-panel-2.svg',
        pointers: [
          {
            id: 'seconds',
            top: 68,
            left: 119,
            width: 12,
            style: 'gauge-pointer-19',
            min: 0,
            max: 59,
            min_degree: 180,
            max_degree: 540
          },
          {
            id: 'minutes',
            top: 109,
            left: 120,
            width: 11,
            style: 'gauge-pointer-18',
            min: 0,
            max: 59,
            min_degree: 180,
            max_degree: 540
          },
          {
            id: 'hours',
            top: 106,
            left: 118,
            width: 14,
            height: 60,
            style: 'gauge-pointer-17',
            min: 0,
            max: 23,
            min_degree: 180,
            max_degree: 900
          }
        ]
      }
    );

    // Render Clock instance
    pointer.render();
  }
});
```

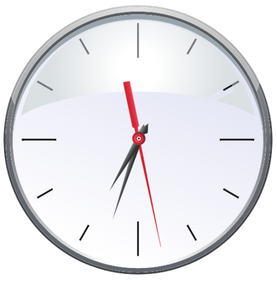


Figure 41: Instantiation and rendering of the Clock widget with example configuration values.

uration for its rotation range. The pointers displaying seconds and minutes should hold a minimum degree of 180º and a maximum degree of 540º (180º+360º). Hours pointers should hold a minimum degree of 180º and a maximum degree of 900º (180º+2*360º).

After the call of the *Clock* constructor and once obtained a widget instance, its *render()* method can be called. This method uses the current time to find the number of hours, minutes and seconds that should be displayed. The *Clock* widget will then interpolate — similarly to the *GaugeSport* widget — the correct rotation angle of each pointer and the resulting value will be used as argument in the call of the *render()* method on each *Pointer* instance.

An example of a complete *Clock* widget instantiation and rendering can be seen in ??.

Widget style variations

In order to provide a collection of out-of-the-box *Clock* variations, a set of different styles were implemented following the standard among the *SVGWidget* widgets. The set of style variations shipped with the *Clock* widget is shown in Figure 42.

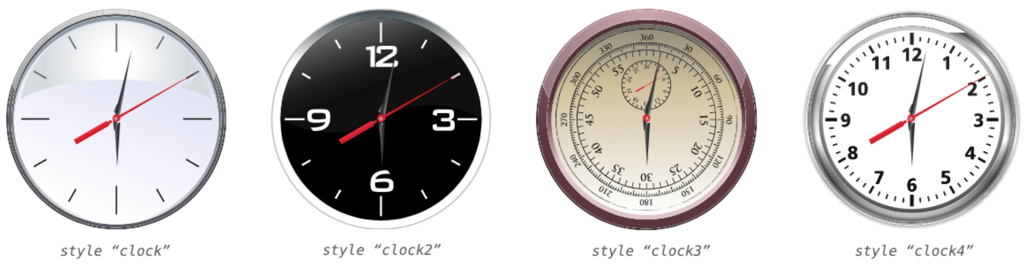


Figure 42: The set of default variations for the Clock widget.

The styles variations were created by selecting a set of *Clock* configuration values - including the name of the SVG file used to display the clock background and the involved pointers. These configurations and the mapping between them and their style identifier are included in the *getDefaultStyleConfigs()* method of the *Clock* widget. The widget constructor will use this method to get default configuration values to be assigned to the instance being created.

Using a specific style variation of the *Clock* widget requires specifying the "style" parameter of the configurations with the desired style identifier. A style variation can be used completly by a instance (without no further specific configuration), partially (providing overridding configuration values in the constructor call) or complemented (providing additional configuration values in the constructor call).

Implementation details

The *Clock* widget was implemented as an extension of the *GaugeSport* widget, simplifying the *render()* method with the implicit logic of a clock behaviour. Besides this difference, a difference among the configurations can also be detected.

A *Clock* widget should only contain three pointers, and their configured rotation angles range and values range take specific values (assumed in *Clock* widget as their default values). The minutes and seconds pointers should rotate between 180º and 540º. This happens because the pointer is originally pointing down, with a rotation angle of 0. The minutes and seconds pointers should consider the starting point of the rotation movement at the top of the clock (at the 12 hours tick), hence the use of the 180º as minimum angle. Both pointer should display 0 to 59 units of time (seconds or minutes), completing a full rotation around the clock (from the starting angle), hence 540º (180º+360º). The same logic is applied to the hours pointer, with the difference that this pointer should display 0 to 23 units (hours).

This pointer should complete two full rotations around the clock (from the 180º starting point), hence the configuration of 900º as maximum rotation angle (180º+2*360º).

6.2.5 *Gearbox widget*

The *Gearbox* widget represents a gearbox car handler. This widget uses two main SVG files - one to be used as the gearbox panel and the other to be used as the stick.

The widget allows the configuration of the SVG files used, understanding the movement of the stick and its meaning using a set coordinates offsets provided as configuration. Since it is possible to define the gears in the gearbox and the movement of the stick, the *Gearbox* widget allows the representation of every type of gearbox — manual, semi-automatic and automatic.

Gearbox instantiation and usage

The constructor of the *Gearbox* receives the identifier of the generated HTML element, a coordinates object to position the element and a configuration objects. The configurations include the name of the SVG file to be used as the gearbox panel, the name of the SVG file to be used as shift representation, two objects with shift coordinates and additional styling configurations (complete documentation of the *Gearbox* constructor in Figure 43. The shift coordinates objects (one with left offset and another with top offset) have the possible shift as properties (and its left or top offset as value), allowing the *Gearbox* instance to know which gear is currently used.

Once a *Gearbox* instance is obtained, the *render()* method can be called receiving the shift value to be displayed (can either be a character for the automatic instances, or an integer for the manual models). The new stick position is obtained from the style configurations and the shift is correctly positioned in the gearbox.

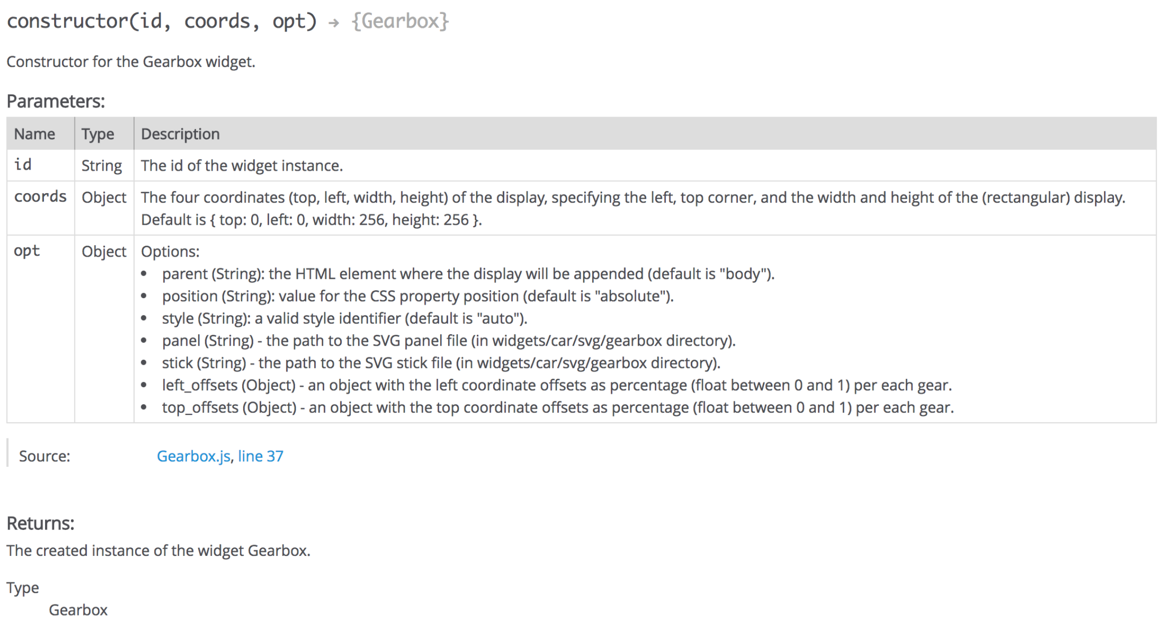
An example of instantiating and rendering of the *Gearbox* and its result is shown in **??**.

Widget style variations

Following the standard of the *SVGWidget*, a set of predefined style variations were implemented in order to examplify the different aspects and behaviours of the *Gearbox* widget.

The collection of styles variations for the *Gearbox* widget display semi-automatic and manual gearboxes and is shown in Figure 45.

The configuration of each style variation was added to the *getDefaultStyleConfigs* method of the *Gearbox* widget, so that it is returned by the method when the corresponding style identifier is received as argument. This method is used in the constructor of the widget to assign to the *Gearbox* instance being created the style configuration as its default.



```
constructor(id, coords, opt) → {Gearbox}
```

Constructor for the Gearbox widget.

Parameters:

Name	Type	Description
`id`	String	The id of the widget instance.
`coords`	Object	The four coordinates (top, left, width, height) of the display, specifying the left, top corner, and the width and height of the (rectangular) display. Default is { top: 0, left: 0, width: 256, height: 256 }.
`opt`	Object	Options: • parent (String): the HTML element where the display will be appended (default is "body"). • position (String): value for the CSS property position (default is "absolute"). • style (String): a valid style identifier (default is "auto"). • panel (String) - the path to the SVG panel file (in widgets/car/svg/gearbox directory). • stick (String) - the path to the SVG stick file (in widgets/car/svg/gearbox directory). • left_offsets (Object) - an object with the left coordinate offsets as percentage (float between 0 and 1) per each gear. • top_offsets (Object) - an object with the top coordinate offsets as percentage (float between 0 and 1) per each gear.

Source: Gearbox.js, line 37

Returns:
The created instance of the widget Gearbox.

Type
Gearbox

Figure 43: Documentation of the Gearbox widget constructor.

Similarly to other *SVGWidget* widgets, to use a specific style variation of the *Gearbox*, the configurations provided to the widget constructor when creating a new instance must include the ”style” parameter with the desired style identifier as its value. The style configuration will be used a set of default values, so any configuration specified in the constructor call of the instance will be respected and will override the style configuration value.

```
define(function (require, exports, module) {
  "use strict";

  // Require the Gearbox module
  require("widgets/car/Gearbox");

  function main() {
    // After Gearbox module was loaded, initialize it
    var gearbox = new Gearbox(
      // id of the element that will be created
      'gear',
      // coordinates object
      { top: 100, left: 100},
      {
        panel_file: 'gear-box-auto.svg',
        stick: 'gear-stick.svg',
        leftOffsets: {
          'D': 0.345,
          'N': 0.345,
          'R': 0.345,
          'P': 0.345
        },
        topOffsets: {
          'D': 0.55,
          'N': 0.4,
          'R': 0.25,
          'P': 0
        }
      }
    );

    // Render Gearbox instance - providing a shift value
    pointer.render('P');
  }
});
```

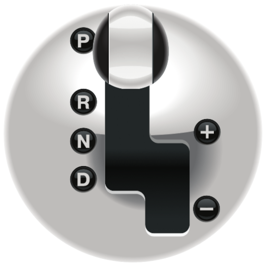


Figure 44: Instantiation and rendering of the Gearbox widget with example configuration values.

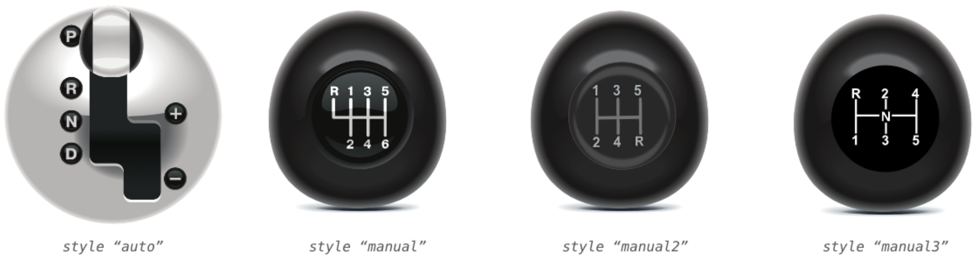


Figure 45: The possible variations for the Gearbox widget - the first semi-automatic and the other three manuals.

6.3 IMPLEMENTATION ANALYSIS

The SVG-based widgets library fully covers the gauge and clock components described in Chapter 4. *GaugeSport* enables the construction of speedometers, tachometers, remaning fuel gauges, thermometers and compasses and *Clock* allows the rendering of analog clocks. A *Gearbox* widget was also implemented, which aims to simulate the behaviour of a car gear box.

Taking an approach based on SVG files has enabled the construction of a flexible widgets library. The implemented widgets take advantage of configurations for the control logic of their behaviour. However, these are not extensive and the possibilities of customisation are endless, due to the delegation of layout concerns to SVG files. The need for changing the source code in order to support every non standard gauge component was eliminated.

Nonetheless, this approach has an inherent disadvantage when compared with the d3-based apporach, which is its dependency on external resources for the correct performance of the widgets. While the widgets described in Chapter 5 were self-suficient (meaning that its correct behaviour depended only on the correctness of the JavaScript libraries and user-input configurations), the SVG-based library depends also on the correctness of SVG files. This dependency can also be problematic if the files have inconsistencies or have an excessive size, which can cause errors on the execution of the widgets or even performance problems in the execution of PVSio-Web as a whole.

However, the additional dependency is to SVG files, which are relatively easy to manage and easy to obtain. These files are broadly used in Web development, and there are a lot of tools and support in the community to handle them. Additionally, the main advantage of this approach is that actually using an external file onto which some of the responsability can be delegated to allows the development of more clean, modular and easy to maintain widgets.

6.4 CONCLUSIONS

This chapter covered the implementation a library of SVG-based widgets for PVSio-Web. These widgets require SVG files for their presentation, providing a higher level of flexibility when compared to the d3-based widgets described on Chapter 5. The description of the four widgets implemented (*Pointer*, *GaugeSport*, *Clock* and *Gearbox*) was provided, as well as examples of instatiation and configurations of these widgets.

The results of the present implementation were analysed, and the next chapter will provide an overview of the construction of car dashboard prototypes using the widgets developed on this dissertation.

7

CAR DASHBOARD PROTOTYPES

With the implementation of a widgets library (which includes the *Gauge*, *CentralPanel*, *Pointer*, *GaugeSport*, *Clock* and *Gearbox* widgets), the construction of car dashboard prototypes in PVSio-Web became possible.

As mentioned in Chapter 3, building PVSio-Web prototypes requires a **PVS Model** representing the system state and interactions, a background image and widgets to build the interface and to connect with the system model.

The car dashboard models that will be prototyped were presented and analyzed in section 4.3. This chapter describes the PVS model used, as well as the construction of each prototype.

7.1 PVS SPECIFICATION

As it was described in Chapter 3, every prototype developed in PVSio-Web requires a formal theory that controls the logic of the prototype, providing variables that can be used by the widgets. Since all car dashboard prototypes are based on the same system (car), a single PVS specification was built and shared among all built prototypes.

A PVS specification consists of a collection of theories. Each theory consists of a signature for the model data type names, constants and the operations associated with the signature. It is also possible to place constraints (assumptions) on the parameters of the defined theories [Owre et al. (2001)].

The PVS model used consists of one single theory, which defines the following data types:

- **Gear** - the current gear of the car, and the list of its possible values.
- **Speed** - an object containing the current car speed and its unit. Its definition can be seen on listing 7.1
- **Rpm** - a non-negative real value for the number of engine rotations (in thousands per minute).
- **Odo** - the mileage of the car (in kilometers).

- **Temp** - an object containing the current environment temperature and its measure units.
- **Time** - an object containing the current hours and minutes.
- **Action** - the set of possible actions on the model (`idle`, `acc` and `brk`).

Listing 7.1: Definition in PVS of the `Speed` data type.

```
MAX_SPEED: real = 220
Speed_Unit: TYPE = { kph, mph }
Speed_Val: TYPE = { x: real | x <= MAX_SPEED }
Speed: TYPE = [#
    val: Speed_Val,
    units: Speed_Unit
#]
```

The state of the car at every instant is also a data type defined on the model, as an object containing concrete values for each of the types previously defined. The initial state for the car model was also specified.

Listing 7.2: The definition of the `State` type.

```
state: TYPE = [#
    speed: Speed, % Km/h
    gear: Gear,
    rpm: Rpm, % x1000/min
    odo: Odo, % Km
    temp: Temp,
    time: Time,
    action: Action
#]

%-- initial state
init(x: real): state = (#
    speed := (# val:= IF x < MAX_SPEED THEN x ELSE MAX_SPEED ENDIF, units := kph
    gear := N,
    rpm := 0,
    odo := 0,
    temp := (# val := ENV_TEMP, units := C #),
    time := get_current_time,
    action := idle
#)
```

The prototype is responsible for interacting with the PVS model. This interaction can happen based on one of five actions exposed by the model:

- **press_accelerate:** Called at the beginning of a continuous acceleration (as pressing the key that simulates the acceleration of the prototype). It changes the current state of the PVS model to `acc`.
- **release_accelerate:** This action should be called once the acceleration process is terminated (as releasing of the key that simulates the acceleration of the prototype). It changes the current state of the PVS model to `idle`.
- **press_brake:** Similarly to the **press_accelerate** action, it changes the current state of the PVS model to `brk` at the start of the braking process (pressing the brake key).
- **release_brake:** Similarly to the **release_accelerate** action, it changes the current state of the PVS model to `idle` at the end of the braking process (releasing the brake key).
- **tick:** This action should be called once every second by the prototype. It intends to simulate the continuous action of the model. This action takes into account the current state of the prototype (one of `idle`, `acc` or `brk`) to change the current state:
  - If the current state is `idle`, the friction process is simulated by slowly changing variables of the model - such as slowly decreasing the speed variable.
  - If the current state is `acc`, the acceleration process is simulated by changing variables of the model - such as incrementing the speed variable.
  - If the current state is `brk`, the braking process is simulated by changing variables of the model - such as rapidly decrementing the speed variable.

The communication of the prototype to the PVS model is done via WebSockets[1]; the new state is returned as a JSON (JavaScript Object Notation)[2] object, and should be handled by the prototype.

7.2 CONSTRUCTION OF CAR DASHBOARD PROTOTYPES

Behaviour of the prototypes

The car dashboard prototypes should connect to the PVS model that represents the car system, and support two main interactions:

- **Pressing the up arrow key:** to accelerate the movement of the car.
- **Pressing the down arrow key:** to brake the movement of the car.

1 MDN Webpage for WebSockets: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`
2 JSON Webpage: `http://www.json.org/`

Usage of demonstrations

In order to prototype car dashboards with gauges, textual and icon display components and clock components, both the official PVSio-Web widgets and the implemented widgets library were used. Since the latter have not yet been incorporated into the PVSio-Web toolkit, they can not be used in the PVSio-Web **Prototype Builder** environment. Instead, in order to build a prototype using all available widgets, **PVSio-Web demonstrations** were created.

Structure of the demonstrations

A **PVSio-Web demonstration** reproduces a behaviour of a prototype by programmatically taking the steps for its creation. The main resources involved are a JavaScript file and an HTML file (which includes the first).

The image to be used as background of the prototype should be included in the HTML file. The JavaScript file is responsible for requiring the widgets, instantiating and rendering them over the image. It is also responsible for enabling the comunication with the PVS model by connecting to the backend server using WebSockets. For the purposes of this dissertation, the communication logic with the PVS model is ommited.

The JavaScript involved in a demonstration starts by requiring all necessary modules (including the PVSio-Web widget modules). Once all the modules are required, a set of simulation bootstrap definitions are made, including the definition of the *onMessageReceived()* method. This method is invoked by PVSio-Web when the back-end sends state updates, and it internally calls the *render()* method, sending the new state as argument. This causes the re-rending of the prototype every time the state is updated.

The actual construction of the prototype interface is done by instantiating the required widgets. The dashboard interaction is also defined, setting the press of the **up key** to trigger the "accelerate" function of the PVS model, and the **down key** to trigger the "brake" function of the PVS model. The *render()* method should be defined with the consequential rendering of the widget instances inside the prototype with the appropriate arguments (read from the received system state).

A representative example of the code structure of a prototype demonstration is shown in Listing 7.3.

Listing 7.3: Structure of thedemonstration code used.

```
require(
  [
    // PVSio-web official widget modules
    "widgets/Button",
    "widgets/TouchscreenButton",
    "widgets/TouchscreenDisplay",
    "widgets/BasicDisplay",
    "widgets/NumericDisplay",
    "widgets/LED",

    // Require the implemented PVSio-Web widget modules
    // ex: "widgets/car/Gauge",

    "widgets/ButtonActionsQueue",
    "stateParser",
    "PVSioWebClient"
  ], function (
    // PVSio-web official widgets
    Button,
    TouchscreenButton,
    TouchscreenDisplay,
    BasicDisplay,
    NumericDisplay,
    LED,

    // Enable the implemented PVSio-Web widgets for use
    // ex: Gauge,

    ButtonActionsQueue,
    stateParser,
    PVSioWebClient
  ) {
    "use strict";
    var client = PVSioWebClient.getInstance();

    // -- SIMULATION BOOTSTRAP PROCESS --
    // (...)

    // -- DASHBOARD COMPONENTS --
    // Instantiate the desired widgets
    // ex: var dashboard = { speedometergauge: new Gauge( (...) ) };

    // -- DASHBOARD INTERACTION --
    // (...)

    // -- DASHBOARD RENDERING --
    function render(res) {
        // Render the instantiated widgets
        // ex: dashboard.speedometergauge.render(evaluate(res.speed.val));
    }

    // -- LISTENING TO MODEL CHANGES --
    // (...)

    client.connectToServer();
  }
);
```

The built prototypes are described in the following sections. These adhere to the presented structure, using the same PVS file and behaviour (reacting to the press of **up and down keys** to simulate accelarating and braking the car). Additionally, the demonstrations created use the image of the model as the background of the prototype in order to achieve more realistic results.

7.2.1 *First car dashboard prototype*

The first car dashboard prototyped is inspired in the model shown in Figure 46. The analysis of the model revealed that the left area consists of speedometer and remaining fuel gauges, and the right area consists of tachometer and thermometer gauges. The central

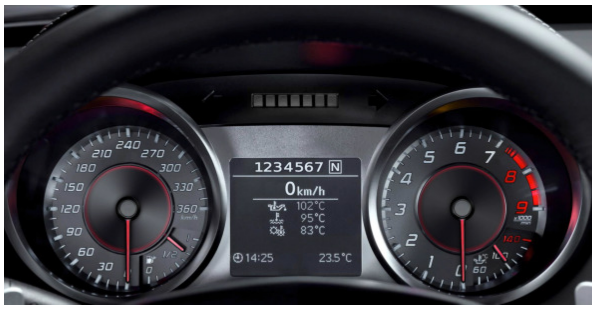


Figure 46: The first car dashboard model.

area is a variated set of textual and icons displays with the current speed and gear, a set of temperature indicators, a digital clock and the mileage of the car.

This model was prototyped using the model image as background and using two widgets - *Gauge* and *CentralPanel*. The *CentralPanel* is an obvious fit to the prototyping of the dashboard central area, since the widget is dedicated to rendering a composition of textual displays in a defined format. The *Gauge* widget was used to represent the gauge components. However, due to its limitations, it was not possible to prototype the two overlapped gauges. Overlapping two gauges would be possible with use of the SVG-based *GaugeSport* widget, but it would require having the necessary SVG files to represent the model. As no such files existed, the SVG-based widget was not used in this prototype.

The created prototype was implemented using the structure presented in the previous section. The dashboard prototype instantiates two *Gauge* instances - a ”speedometer” and a ”tachometer”, using the same ”classic” style but with different configurations (see Listing 7.4 for the instantiation of the *Gauge* widget), and a single *CentralPanel* instance.

Listing 7.4: Two instantiations of Gauge widget in first prototype.

```
var dashboard = {
  speedometerGauge: new Gauge(
    'speedometer-gauge',
    {
      top: 251,
      left: 53,
      width: 360,
      height: 360
    },
    {
      style: 'classic',
      max: 360,
      majorTicks: 13,
      min: 0,
      size: 360,
      redZones: [],
      rotation: -45,
      gap:90,
      roundValueBeforeRender: true,
      parent: 'dashboard-container'
    }
  ),

  tachometerGauge: new Gauge(
    'tachometer-gauge',
    {
      top: 251,
      left: 633,
      width: 360,
      height: 360
    },
    {
      style: 'classic',
      max: 9,
      min: 0,
      size: 360,
      majorTicks: 10,
      minorTicks: 4,
      greenZones: [],
      yellowZones: [],
      redZones: [{ from: 7.01, to: 9 }],
      rotation: -45,
      parent: 'dashboard-container'
    }
  ),
  // (...)
};
```

The *render()* method of the prototype receives a new system state and renders the instanciated widgets with the appropriate value (the "speedometer" instance is rendered with the car's "speed" value, the tachometer instance is rendered with the "rpm" (rotations per minute) value and the *CentralPanel* instance is rendered with the new car state). This guarantees that the dashboard elements correctly react to system updates, displaying the current car state.

The final result is shown in Figure 47, where we can see the prototype in its initial state and the prototype after accelarating the car.

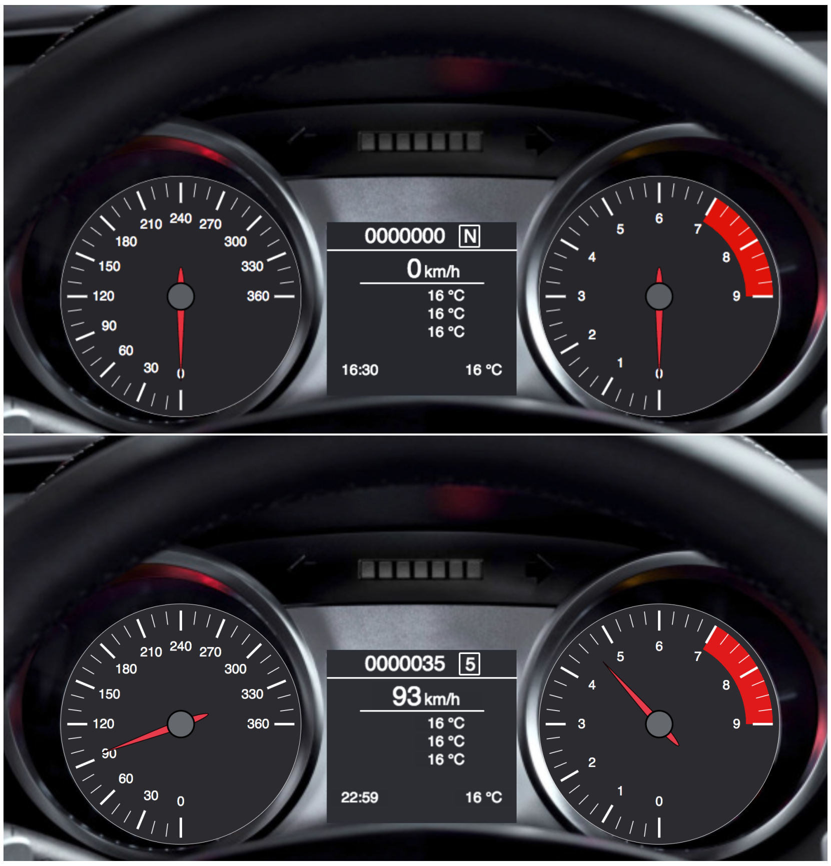


Figure 47: First car dashboard prototype in its initial state (top) and after accelarating the car (bottom).

7.2.2 *Second car dashboard prototype*

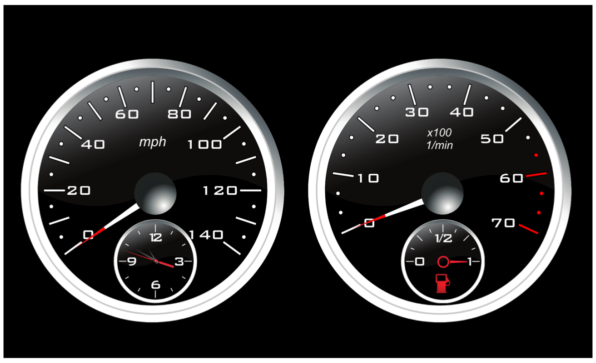


Figure 48: The second car dashboard model.

The second car dashboard prototypes the model shown in Figure 48. The analysis of the model revealed that the left area consists of a speedometer gauge and an overlapped clock, and the right area consists of a tachometer and a remaining fuel overlapped gauges.

To prototype this model a black image was set as background, and two widgets — *GaugeSport* and *Clock* — were used. The use of the *GaugeSport* widget enabled the easy representation of the three gauges in the dashboard, and allowed the overlapping of two of them.

The second prototype was also implemented using a demonstration following the structure presented in the previous section. This prototype instantiates one *Clock* instance and three *GaugeSport* instances.

The *Clock* instance of the prototype uses the ″clock2″ style shipped with the widget, which eliminates the need to provide a specific SVG file for the clock panel and additional configurations for the clock layout, as well as the set of pointers configurations. The only additional configuration provided for this *Clock* instance was the size of the pointers, which were specified in order to adjust them to the case.

The *GaugeSport* instances also take advantages of the style variations shipped with the widget. The remaining fuel gauge uses ″fuel2″ style, the speedometer uses ″speedometer7″ and the tachometer uses ″tachometer4″ style. Since the style variations used satified all the visual requirements of the dashboard gauges, only the positioning of the pointers was specified in the instances' configurations.

To guarantee that the prototype correctly displays the current state of the car, the *render()* method defined in the demonstration - called every time the system state is updated - invokes the *render()* method of all widget instances. The *Clock* rendering updates the time being display using the current time, hence not needed any provided argument. The tachometer and speedmeter *GaugeSport* instances are rendered receiving the new state ″rpm″ and ″speed″ values, correspondingly. The remaining fuel gauge, also a *GaugeSport* instance, receives a specific value of 100, since the data can not be retrieved from the PVS model in use.

The final result of the prototype is displayed in Figure 49 – first in its initial state, and then after accelarating the car.

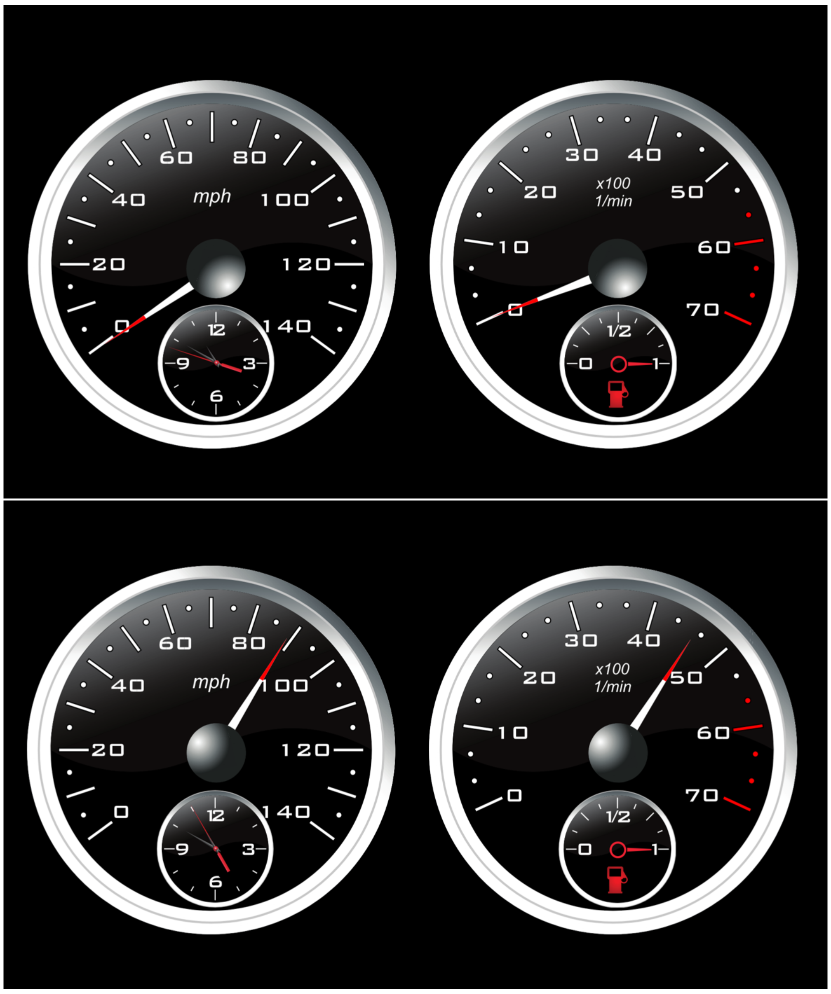


Figure 49: Second car dashboard prototype in its initial state (top) and after accelarating the car (bottom).

7.2.3 *Third car dashboard prototype*

Figure 50 displays the dashboard model used for the third car dashboard prototype. The analysis of the model revealed the existence of two main elements — speedometer and tachometer gauges —, positioned side by side, and a set of smaller gauges surrounding them. These gauges represent the air, water and exhaust gas temperatures (EGT) and the air pressure on the turbo charger (turbo).

In order to prototype the presented model, its image was applied as background and the *Gauge* d3-based widget was used to represent all its elements.

The prototype was implemented using the established demonstration structure. This prototype instantiates six *Gauge* instances - all using the "grey" style but with different configurations and positionings (Listing 7.5 shows an instantiation example).

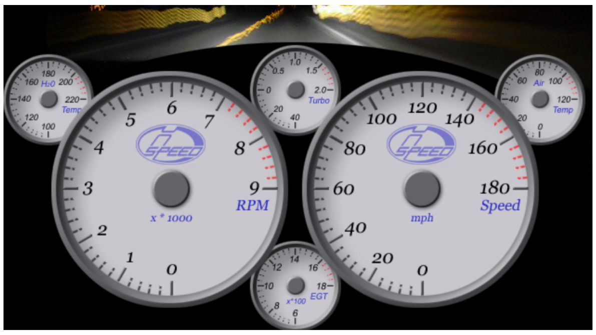


Figure 50: The third car dashboard model.

Listing 7.5: Instantiation of one of the Gauge instance in third prototype.

```
var dashboard = {
  waterTempGauge: new Gauge(
    'water-temp-gauge',
    {
      top: 1369,
      left: 7,
      width: 170,
      height: 170
    },
    {
      style: 'grey',
      size: 170,
      minorTicks: 4,
      max: 220,
      min: 100,
      initial: 100,
      label: 'H20',
      majorTicks: 7,
      greenZones: [],
      yellowZones: [],
      redZones: [{ from: 200.1, to: 219.9 }],
      rotation: -45,
      parent: 'dashboard-container'
    }
  ),
  // (...)
};
```

The dynamic representation of the car state in the prototype is achieved by making the *render()* method of the demonstration re-render the widget instances with the new state values. However, since some of the displayed information is not included in the PVS model — specifically the water temperature, EGT, turbo and air temperature — only the speedometer and tachometer react to the system update. These *Gauge* instances receive as argument on its *render()* method the car's "speed" and "rpm" values, correspondingly.

The obtained prototype in its initial state is shown in Figure 51.

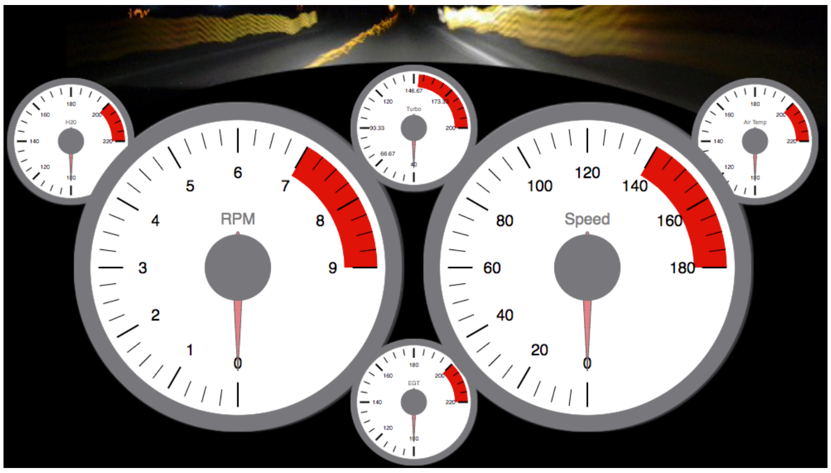


Figure 51: The third car dashboard prototype.

7.2.4 *Fourth car dashboard prototype*

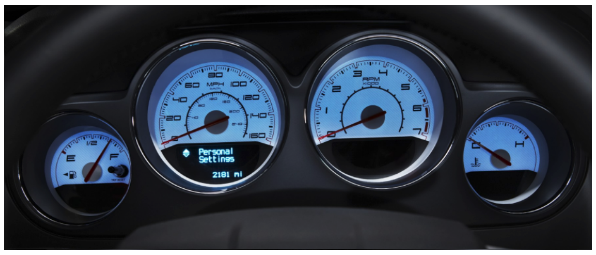


Figure 52: The fourth car dashboard model.

The fourth car dashboard model used is shown in Figure 52. This model consists of four gauge elements - a speedometer and tachometer gauges (larger and positioned at the center of the dashboard) and a remaining fuel and thermometer gauges.

This model was prototyped using the presented model image as background and using the *Gauge* d3-based widget. Similarly to the third dashboard prototype, it was possible to represent all its dashboard elements with a single widget.

The demonstration implemented to create the prototype defines four *Gauge* instances, all using the "blue" style configuration (the instantiation of the "remainingFuelGauge" instance can be seen in Listing 7.6).

Listing 7.6: Instantiation of one of the Gauge instance in fourth prototype.

```
var dashboard = {
  remainingFuelGauge: new Gauge(
    'remaining-fuel-gauge',
    {
      top: 2180,
      left: 87,
      width: 160,
      height: 160
    },
    {
      style: 'blue',
      size: 160,
      gap: 270,
      minorTicks: 4,
      label: "Fuel",
      max: 1,
      min: 0,
      initial: 0,
      majorTicks: 3,
      greenZones: [],
      yellowZones: [],
      redZones: [{ from: 0, to: 0.125 }],
      rotation: 0,
      parent: 'dashboard-container'
    }
  ),
  // (...)
};
```

Similarly to the other prototypes created, the fourth car dashboard prototype renders its widget instances with the appropriate values to represent the system state. The rendering of the prototype is executed every time the system is updated, and calls the *render()* method of its *Gauge* instances. However, since the PVS model does not hold the car fuel and oil temperature information (represented in the remaining fuel gauge and thermometer gauge), only the *Gauge* instances representing the speedometer and tachometer actually receive the new state values (using the state's "speed" and "rpm" values).

The prototype created is displayed in its initial state in Figure 53.

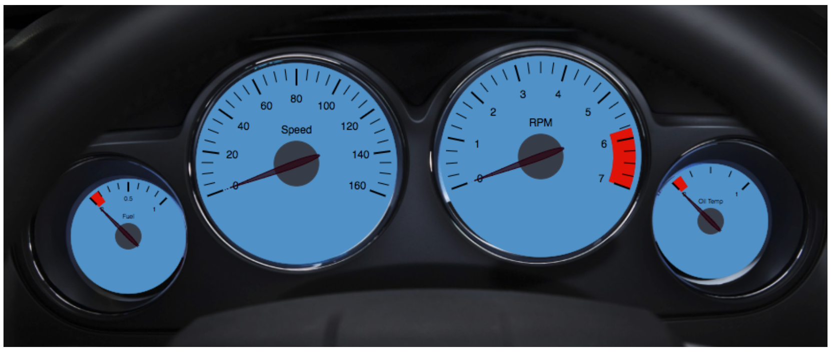


Figure 53: The fourth car dashboard prototype.

7.3 CONCLUSIONS

This chapter elaborated on how the dashboard models analysed in Chapter 4 were implemented by taking advantage of both developed widgets library (d3-based and SVG-based). The PVS model that controls the logic of the car engine was described, followed by the architecture of demonstrations used along the prototype construction. The examples approached help future developers to understand how the widgets built can be put to practice, by providing some visual examples of its instatiation.

The following chapter will discuss both approaches taken, providing an analysis of the lessons learnt from the process of building car dashboard prototypes.

8 CONCLUSIONS AND FUTURE WORK

The present chapter provides a comparative analysis on the adopted methodologies: d3-based and SVG-based. This enables to draw some conclusions on the recommended usage of each type of widget. Section 8.2 provides an assessment of the work done on this dissertation, and the last section provides some pointers for futher improvement on the work done.

8.1 COMPARISON OF THE TWO ADOPTED METHODOLOGIES

The d3-based and SVG-based adopted methodologies for building widgets in PVSio-Web share some similarities and differences. This section focuses on describing the differences between the d3-based approach (which uses the d3-gauge-plus library) and the approach based on SVG files when it comes to actually using the widgets for building gauges and simulating car dashboards. This analysis will follow three criteria: the *speed of development*, the capacity of *customisation* of the newly created variations, and the *performance* of the final solution.

8.1.1 *Speed of development*

The first criteria of analysis adopted is the velocity of development of new widgets (and/or new widget variations), and how quickly can one simulate a given car dashboard using the widgets that were built.

The d3-based widgets allow users to quickly create different variations of widgets, provided that the every feature of the dashboard to be implemented is supported by the widgets. However, if the widgets need to be adapted in order to support every feature in the dashboard to be developed, then the development process using this type of widgets becomes slower.

The SVG-based approach requires the existence of SVG files that are at least similar to the provided dashboard layout. These files can either be bought from commercial sources, obtained from free sources, or developed using tools such as Inkscape[1] or Adobe Illustrator[2].

8.1.2 *Customisation*

The second criteria adopted takes into account how easy it is to create widgets that have differences from previous created variations.

The d3-based approach offers configurations for changing the final layout of the created dashboard elements. However, features that have not yet been supported will imply a change in the widget code. The impact on the code maintainability of these widgets has been discussed in section 5.3.

The SVG-based approach takes this configuration weight from the widget code to the SVG file itself – meaning that every difference from the norm will just imply a new SVG file, and no change to the widget code.

8.1.3 *Performance*

The last criteria adopted takes into account the performance of the final solution.

The SVG-based approach implies more intensive CPU usage when compared to the d3-based approach. This is even more increased if the used SVG files' size is high (due to great detail of customisation in the vectorial graphics). Since they are included in the DOM tree, each DOM manipulation that is performed takes more time, due to the greater size of the DOM tree.

This detected performance problem can be minimized with some simplifications to the SVG files, mainly focused on the reduction of the files' sizes – for example, replacing panels which do not require interactivity with compressed raster graphics. However, a deeper performance analysis would be advised, to get further comprehension of the reasons for the performance deterioration, and what other measures should be applied.

d3-based widgets are less prone to performance issues. Since the gauge representation is controlled by the d3-gauge-plus library, the impact of the generated gauges in the DOM tree is minimal.

8.1.4 *Comparison conclusions*

All in all, each approach has its advantages and disadvantages. And since both methods can be used together without any problem, the general recommendation of which approach

1 Inkscape Webpage: `https://inkscape.org/en/`
2 Adobe Illustrator Webpage: `http://www.adobe.com/products/illustrator.html`

should be taken depends mostly on the complexity of the widgets that needs to be developed.

The d3-based approach is recommended for more standard variations which do not require special customisations, and the SVG-based approach is advised for widgets which present substantial differences in the layout from the configurations that the first referred method allows.

8.2 CONCLUSIONS

The present dissertation approached the importance of prototyping as a way of accelerating the process of development of applications with excellent user interfaces. PVSio-Web enables the application of formal methods when prototyping, providing a set of default widgets for easing that purpose.

The main goal of this dissertation was to provide a valuable contribution to the widget base of PVSio-Web, with a focus on the development of car dashboard prototypes. The end result is a reusable widget library, with a reasonable set of variations ready to be used. It should be mentioned that the widgets developed are not exclusive for the prototyping of car dashboards, with its usage being encouraged on other projects where they might be useful.

The demonstrations presented in Chapter 7 allow to take some insight into prototype construction. The most relevant conclusion is that the implemented library is fully integrated with the architecture of PVSio-Web, which encourages the possibility of being integrated in the tool.

The developed widgets also extend the set of possibilities when prototyping using PVSio-Web. The tool's set of default widgets is very useful for the visualisation and representation of textual information, but there was a lack of components that enabled the construction of prototypes with a greater level of complexity. By giving the possibility of prototyping more complex environments such as car dashboards, the possibility of prototyping different and even more complex systems is also opened.

On the other side, although gauges constitute the majority of elements found in usual car dashboards, there is a set of singular elements that were not considered in this dissertation. Elements such as steering wheels/joysticks, navigation maps, or even board interactive computers were not considered due to their complexity level. The composition of different widgets, as it was shown with the *CentralPanel* widget might play an important role when creating structures that simulate the referred components.

8.3 FUTURE WORK

As any other project, the present dissertation has several points of improvement for someone looking to continue the work done so far.

As it has been referred in section 8.1, the first major point of improvement is the performance of the widgets that use SVG files – *Clock*, *GaugeSport*, *Gearbox* and *Pointer*. It is very important to analyse what is the reason for the impact, and understand what can be done to prevent this situation from happening. This analysis can be done recurring to the Chrome Developer Tools, which provides several tools for auditing the performance of the solution.

Another possible point for improvement would be extending the Pointer widget. This widget is responsible for loading a SVG file and allowing to rotate it, but this behaviour could be useful in other situations, and it would allow to keep the inclusion of SVG files contained to only one widget.

Another possible improvement has to do with the testability of the widgets' source code. Currently, SVG files are required asynchronously – which means that a call-back function is provided and called after the file has been successfully required. However, rendering the widget is only possible once the SVG file is required, which makes testing the rendering of the widget after the constructor is called impossible unless a timeout is set, which is not a viable solution. The usage of promises[3] would make it possible to test the rendering of the widget without having to set timeouts – the constructor of the widgets would return a Promise object on which methods could be chained synchronously.

Lastly, it is also relevant to refer that the inclusion of the libraries in the set of PVSio-Web's default widgets would enable the possibility of using them in the **Prototype Builder** environment. This would allow the easier construction of prototypes through the usage of a GUI, without the need of a programmed demonstration to use the widgets developed.

These are suggestions of points that could be improve the widgets library, aside from actually complementing the library with components that could serve other purposes.

3 Promise/A+ standard: `https://promisesaplus.com/`

BIBLIOGRAPHY

R. M. Baecker and W. A. . San Buxton. *Readings in human-computer interaction: A multidisciplinary approach*. Morgan Kaufmann Publishers, Mateo, CA, USA, 1987.

BS EN 62366:2008. Medical devices — Application of usability engineering to medical devices. Standard, BSI - British Standards, London, UK, 2008.

D. Bäumer, W. Bischofberger, H. Lichter, and H. Züllighoven. *User interface prototyping—concepts, tools, and experience*. IEEE Computer Society, Washington, DC, USA, 1996.

James H. Carlisle. Why human-computer interaction doesn't work like human dialogue. October 1975.

D. Flanagan. *JavaScript: The Definitive Guide, Sixth Edition*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2011.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

T. T. Hewett, R. Baecker, Carey Card, S., J. T., Gasen, M. Mantei, and W. Verplank. *ACM SIGCHI Curricula for Human-Computer Interaction*. ACM, New York, NY, USA, 1992.

ISO 9241-11:1998. Ergonomic requirements for office work with visual display terminals (VDTs). Standard, International Organization for Standardization, 1998.

P. W. Jordan. *Human factors for pleasure in product use (Vol. 29)*. Applied Ergonomics, 1998.

P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimblebly. *PVSio-web 2.0: Joining PVS to HCI*. Proceedings of 27th International Conference on Computer Aided Verification (CAV2015), California, USA, 2015.

César A. Muñoz. Rapid prototyping in pvs. Technical report, National Aeronautics and Space Administration, Langley Research Center Hampton, Virginia 23681-2199, November 2003.

J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 1993.

P. Oladimeji, P. Masci, P. Curzon, and H. Thimblebly. *PVSio-web: a tool for rapid prototyping device user interfaces in PVS*. Proceedings of the 5th International Workshop on Formal Methods for Interactive Systems (FMIS 2013), London, UK, 2013.

S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS Language Reference*. SRI International, 2001.

Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. Pvs: Combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 411–414, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5. URL `http://dl.acm.org/citation.cfm?id=647765.735995`.