



Norwegian University of  
Science and Technology

# End User Programming for TILES: Methods and Tools

**Jonas Kirkemyr**

Master of Science in Computer Science

Submission date: June 2016

Supervisor: Monica Divitini, IDI

Co-supervisor: Simone Mora, IDI

Norwegian University of Science and Technology  
Department of Computer and Information Science



*To my beautiful sons*



# Abstract

TILES is a toolkit consisting of both hardware and software. A tangible embedded hardware device supporting user interaction, is part of the TILES toolkit, and referred to as a TILES device. Creating applications for TILES devices, which should respond to user-interactions from single and multiple TILES devices, and communicating with both third-party services over the Internet and other TILES devices, is a difficult and complex task to endeavor, requiring great programming experience. This thesis will look at different possibilities of lowering the threshold to develop applications for TILES devices: textually, visually, and physically, for makers and computer science students familiar with programming.

Lowering the threshold of creating applications for TILES, should facilitate makers and computer science students in creating TILES applications, focusing on user-interaction and integration with third-party services. A comprehensive review of state-of-the-art solutions regarding end-user development tools, have helped in creating a development environment for TILES, including a web-browser IDE, and a Domain-Specific Language. The design and implementation of the DSL helps in abstracting the integration with third-party services and handling user-interactions from multiple TILES devices. The web-browser IDE facilitate use of the DSL, both textually and visually, and testing applications for TILES by integrating a compiler for the DSL and emulation of a TILES device directly in the development environment.

Evaluations were conducted on both the DSL, and the development environment as a whole, using focus groups consisting of makers, computer science students, and professional programmers. The results highlight the ease of use to get started creating applications for TILES, and the simplicity of the DSL, which is able to be used for creating simple and complex applications seamlessly.



# Sammendrag

TILES er en kolleksjon av utviklings-verktøy bestående av maskinvare og programvare. En interaktiv innebygd maskinvare som støtter brukerinteraksjon er en del av TILES, kalt for en TILES enhet. Det å lage applikasjoner for TILES enheter som kan respondere på bruker-interaksjoner fra en eller flere TILES enheter, og samtidig kommunisere med både tredje-parts tjenester og andre TILES enheter, er en vanskelig og kompleks oppgave som krever høye programmerings ferdigheter. Denne rapporten vil se på flere muligheter for å senke terskelen med å begynne og utvikle applikasjoner for TILES enheter, både tekstlig, visuelt og fysisk, for "makers" og data-studenter som er kjent med programmering.

Ved å senke terskelen med å lage applikasjoner for TILES, kan det legges til rette for å la "makers" og data-studenter lage applikasjoner for TILES, som fokuserer på bruker-interaksjon og integrering mot tredje-parts tjenester. En omfattende gjennomgang av "state of the art" løsninger av utviklingsverktøy for sluttbrukere har hjulpet med å lage et utviklings miljø for TILES, som inkluderer en IDE kjørende i en nettleser, og et domene-språk, kalt DSL. Designet og implementering av domene-språket hjelper til å abstrahere interaksjonen med tredje-parts løsninger, og for å håndtere bruker interaksjon med flere TILES enheter. IDE'en som kjører i nettleseren legger til rette for å bruke domene-språket, både ved bruk av tekst og visuelt. Utviklingsmiljøet støtter også å teste applikasjoner for TILES ved å integrere en kompilator for språket, og emulere en TILES enhet direkte i fra utviklingsmiljøet.

Det ble gjennomført en evaluering på både domene-språket og hele utviklingsmiljøet, ved å bruke fokus grupper bestående av "makers", data-studenter og profesjonelle utviklere. Resultatet gir lys på hvor enkelt det er å komme i gang med utviklingen av applikasjoner for TILES, og hvor enkelt domene-språket er i bruk, som uten problemer kan brukes for å lage enkle og komplekse applikasjoner.





# Acknowledgments

This paper is written as part of my master thesis conducted at the Norwegian University of Science and Technology. The work is an extension of a specialization project by Varun Sivapalan and myself *Event-driven infrastructure for the Internet of Things supporting rapid development* [31].

I would like to thank my supervisors Monica Divitini and Simone Mora for their support and great feedback throughout the work. I would also like to thank Francesco Valerio Gianni, and those who participated in my user evaluations.

A special thank you goes to my parents for their love and support, helping me get where I am today.

Trondheim, June 10  
Jonas Kirkemyr



# Contents

Abstract	i
Sammendrag	iii
Acknowledgments	v
Contents	x
List of Tables	xi
List of Figures	xiv
Acronyms	xv
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Context . . . . .	2
1.3 Research Questions . . . . .	3
1.4 Research Method . . . . .	3
1.5 Results . . . . .	5
1.6 Outline . . . . .	5
<b>2 Problem definition</b>	<b>7</b>
<b>3 TILES Toolkit</b>	<b>11</b>
3.1 Hardware . . . . .	12
3.1.1 TILES Device . . . . .	12
3.1.2 TILES Mobile . . . . .	13
3.1.3 TILES Cloud . . . . .	13
3.1.4 Infrastructure . . . . .	13
3.2 Software . . . . .	14
3.2.1 TILES Cloud . . . . .	14

3.2.2	TILES Mobile . . . . .	15
3.2.3	Client Libraries . . . . .	15
3.3	Overview . . . . .	16
<b>4</b>	<b>Related work</b>	<b>19</b>
4.1	Visual Programming . . . . .	20
4.1.1	Blocks . . . . .	20
4.1.2	Flowchart . . . . .	22
4.1.3	Data Flow . . . . .	23
4.1.4	Finite-State Machine . . . . .	24
4.1.5	Behaviour Tree . . . . .	25
4.1.6	Event-Based Rules . . . . .	26
4.2	Physical Programming . . . . .	27
4.3	Mini-Language . . . . .	28
4.4	Overview . . . . .	31
<b>5</b>	<b>Requirement Specifications</b>	<b>35</b>
5.1	Language comparison . . . . .	35
5.2	Visual-language Comparison . . . . .	38
5.3	TILES Development Environment . . . . .	39
5.4	Domain-Specific Language . . . . .	42
<b>6</b>	<b>Design</b>	<b>45</b>
6.1	Introduction . . . . .	45
6.2	Use cases . . . . .	47
6.3	DSL . . . . .	52
6.3.1	Modules . . . . .	57
6.3.2	The language . . . . .	58
6.3.3	Example Application - Whack a Mole . . . . .	64
6.4	TDE . . . . .	65
6.5	Emulator . . . . .	65
<b>7</b>	<b>Implementation</b>	<b>67</b>
7.1	DSL . . . . .	68
7.1.1	Modules . . . . .	68
7.1.2	Language . . . . .	77
7.2	Development Environment . . . . .	82
7.3	Emulator . . . . .	86
7.4	CLI . . . . .	89
<b>8</b>	<b>Evaluation</b>	<b>93</b>

8.1	Pre-implementation evaluation . . . . .	93
8.1.1	Discussion . . . . .	95
8.2	Implementation evaluation . . . . .	97
8.2.1	Discussion . . . . .	98
8.3	TDE requirement evaluation . . . . .	99
8.4	DSL requirement evaluation . . . . .	102
8.5	Language evaluation . . . . .	103
8.6	DSL Compilation evaluation . . . . .	105
8.7	Application creation . . . . .	110
<b>9</b>	<b>Conclusion</b>	<b>115</b>
9.1	Results . . . . .	115
9.1.1	MRQ: How to implement a development environment, specialized for makers, and computer science students to easily create applications for TILES? . . . . .	116
9.1.2	PRQ1: Which programming paradigm is best fit for makers, and computer science students to create ap- plications for TILES? . . . . .	116
9.1.3	PRQ2: How should a cross-platform development en- vironment for TILES be designed and implemented for its target users? . . . . .	117
9.2	Future work . . . . .	118
9.2.1	Development environment . . . . .	118
9.2.2	DSL . . . . .	119
9.2.3	Programming paradigms . . . . .	119
	<b>References</b>	<b>121</b>
<b>A</b>	<b>Programming Environments</b>	<b>125</b>
A.1	Visual Blocks . . . . .	125
A.1.1	Scratch . . . . .	125
A.1.2	ScratchX . . . . .	126
A.1.3	Snap! . . . . .	126
A.1.4	Blockly . . . . .	126
A.1.5	App Inventor . . . . .	126
A.1.6	Ardublock . . . . .	127
A.1.7	Gameblox . . . . .	127
A.1.8	Sciptr; . . . . .	127
A.1.9	Zipato Rule Creator . . . . .	127
<b>B</b>	<b>EBNF visual</b>	<b>129</b>

<b>C Focus Group</b>	<b>131</b>
C.1 Online topics . . . . .	131
C.2 Language specification . . . . .	133
C.2.1 Table of contents . . . . .	133
C.2.2 Starting point . . . . .	134
C.2.3 TILES Identifier . . . . .	134
C.2.4 Me variable . . . . .	136
C.2.5 Commands . . . . .	136
C.2.6 Events . . . . .	137
C.2.7 Statements . . . . .	138
C.2.8 Data sources . . . . .	141
C.2.9 Example Applications . . . . .	145
<b>D TILES Toolkit</b>	<b>149</b>
D.1 DSL Grammar Rules . . . . .	149
D.2 Primitives . . . . .	155
D.3 DSL Class-Diagram . . . . .	157

# List of Tables

4.1	Related work overview . . . . .	32
5.1	Requirements for the TDE . . . . .	41
5.2	User Stories for DSL . . . . .	43
5.3	Acceptance criteria for User Stories . . . . .	44
6.1	Use Case #1 - Retrieve TILES Devices . . . . .	48
6.2	Use Case #2 - Update TILES Devices information . . . . .	49
6.3	Use Case #3 - Create TILES application textually . . . . .	50
6.4	Use Case #4 - Create TILES application visually . . . . .	51
6.5	Use Case #5 - Restore TILES application . . . . .	52
7.1	EventEmitter methods . . . . .	70
7.2	DSL types . . . . .	80
8.1	DSL types . . . . .	111





# List of Figures

1.1	The research process [27]	4
3.1	TILES Toolkit Overview	11
3.2	Interaction primitives for TILES. Source: Simone Mora	12
3.3	Three-tier infrastructure for TILES	14
3.4	TILES Connect App layout. Source: Simone Mora	16
3.5	TILES Toolkit outline. Source: Simone Mora	17
4.1	Example of building Blocs for creating logic. Example show the use of Scratch	21
4.2	Example of a Flowchart . Example show the use of Flowgorithm VPL	22
4.3	Example of a Dataflow . Example show the use of NODE-RED	24
4.4	Example of a Finite-State Machine. Example show the use of Unity3D Macanim	25
4.5	Example of a Behaviour Tree . Example show the use of Behaviour3	26
4.6	Example of Event-Based Rules . Example show the use of Kodu	27
4.7	Example of physical programming. Example show the use of PrimoToys	29
4.8	Example of a mini-language. Example show the use of Code-Combat	31
5.1	Language layer, where top layers have access to the underlying layer	37
6.1	Use Case diagram showing each Actors goal	47
6.2	Pipe-and-Filter pattern for events	56
6.3	User Story mapping	56
6.4	Modules for the DSL	58
6.5	TILES DSL - EBNF start	63

6.6	Whack A Mole - VPL example . . . . .	64
6.7	TILES Emulator . . . . .	66
7.1	TILES Software Components . . . . .	67
7.2	Class-diagram TileDSL and UserDSL . . . . .	69
7.3	Class-diagram User module . . . . .	72
7.4	Class-diagram Event module . . . . .	73
7.5	Class-diagram Command module . . . . .	75
7.6	Class-diagram Data-source module . . . . .	76
7.7	Class-diagram DSL Parser . . . . .	83
7.8	Class-diagram Development Environment . . . . .	85
7.9	Development Environment - textual editor layout . . . . .	85
7.10	Development Environment - visual editor layout . . . . .	86
7.11	Development Environment - TILES layout . . . . .	87
7.12	Class-diagram Emulator . . . . .	88
7.13	Emulator layout . . . . .	89
7.14	Emulator debug layout . . . . .	89
7.15	CLI example usage . . . . .	91
8.1	Demo of TILES . . . . .	99
8.2	TDE layout . . . . .	110
8.3	Build options menu . . . . .	111
8.4	Running TILES application . . . . .	112
8.5	Syntax error TILES application . . . . .	112
8.6	WhackAMole using emulators . . . . .	113
B.1	TILES DSL - EBNF If statement . . . . .	129
B.2	TILES DSL - EBNF Repeat statement . . . . .	129
B.3	TILES DSL - EBNF Sync statement . . . . .	129
B.4	TILES DSL - EBNF Events . . . . .	129
B.5	TILES DSL - EBNF Commands . . . . .	130
B.6	TILES DSL - EBNF TileEStatement . . . . .	130
D.1	Interaction primitives for TILES. Source: Simone Mora . . . . .	156
D.2	UML class-diagram for the DSL modules . . . . .	158

# Acronyms

**IDE** Integrated Development Environment

**IoT** Internet of Things

**LAN** Local Area Network

**VP** Visual Programming

**VPL** Visual Programming Language

**EUD** End User Programming

**DSL** Domain-Specific Language

**TUI** Tangible User Interface

**TD** TILES Device

**TC** TILES Cloud

**TDE** TILES Development Environment

**HTTP** Hypertext Transfer Protocol

**LED** Light-Emitting Diode

**CoAP** The Constrained Application Protocol

**JSON** JavaScript Object Notation

**XML** Extensible Markup Language

**MQTT** Message Queuing Telemetry Transport

**FR** Functional Requirements

**US** User Story

**CSS** Cascading Style Sheets

**API** Application Programming Interface

**AST** Abstract Syntax Tree

**SSL** Secure Socket Layer

**PEG** Parsing Expression Grammar

**SPA** Single Page Application

**CLI** Command-Line Interface

# Chapter 1

## Introduction

TILES is a software and hardware development toolkit for Internet of Things applications, focusing on user-interaction with embedded hardware devices. TILES facilitates creating applications using an infrastructure which abstracts the application logic from a hardware layer to a cloud-layer. Three-layers makes up the infrastructure, for sending and receiving data between each layer. A pre-study [31], which have served as a groundwork for this thesis, have implemented an infrastructure abstracting the application logic, and making the application logic more changeable. Work done in this thesis will be building on the already implemented infrastructure, to further facilitate application creation for TILES.

### 1.1 Motivation

Prototyping and deploying applications for IoT systems is a hard task to endeavor. TILES want to simplify this prototyping process for makers and computer science students, by introducing easy-to-use programming paradigms. Makers and computer science students with some software development skills, but not necessary interactive objects competencies, should be able to easily create applications for TILES, and control interaction between multiple devices. TILES consist of an ubiquitous and tangible computing device supporting user interaction. This thesis builds on a pre-study [31] where the motivation was to support application abstraction for TILES. From the pre-study's future work, a development environment for controlling and configuring TILES devices was suggested to be further investigated and implemented.

Makers and computer science students have today multiple tools to help them create code and applications for ubiquitous devices. These tools fails to support controlling and interacting with multiple devices. With this thesis I would like to help lowering the threshold for making TILES applications, by providing a software platform where makers and computer science students, can use their creativity and existing knowledge. I also find it interesting to use my knowledge and state of the art technology, to create a development environment for developing TILES applications.

## 1.2 Context

This thesis is part of the TILES project at the Norwegian University of Science and Technology. The project consist of a tangible ubiquitous computing system, referred to as the TILES toolkit. The infrastructure developed for TILES creates a higher-level of abstraction for the application logic, making it easy to change a TILES device's behavior.

*Makerspaces*, *Hackerspaces*, and *Fablabs* are common communities that focus on using their knowledge to create physical goods (metalwork, woodwork, fabrication, arts and crafts), and technology artifacts with the use of open-source hardware and software [18]. These communities shares a vision of making it easy for people to use computer interfaces that interacts with the real-world, and configure them to their own need. They want to empower people by creating and changing the future with technologies they create.

The pre-study successfully implemented an infrastructure providing a higher abstraction of application logic, that help supporting modifiability as direct access of the TILES devices is not necessary, and the application logic being run from a common source.

Creating a development environment should ease the process of creating applications for TILES, and how these applications should be developed is the focus for this thesis.

## 1.3 Research Questions

**MRQ:** How to implement a development environment, specialized for makers, and computer science students to easily create applications for TILES?

**PRQ1:** Which programming paradigm is best fit for makers, and computer science students to create applications for TILES?

**PRQ2:** How should a cross-platform development environment for TILES be designed and implemented for its target users?

## 1.4 Research Method

The thesis is a continued work from a pre-study [31] completed in the autumn 2015. The pre-study introduced multiple suggestions for the future work, one being a development environment for lowering the threshold for developing applications for TILES. This corresponds with the supervisors vision for the project. The research questions were determined from the suggested future work, along with a literature review, which helped in reviewing the current state of End-User-Development for tangible devices.

The research questions will be answered by building a prototype, and have it evaluated by the target users. To design the artifacts and solve the implementation problems addressed in the research questions, the design and creation research strategy is used [27]. The design and creation research strategy will help to create and use knowledge for implementing a solution that will solve the problem at hand, analyze the implemented solution use and performance, and to build and evaluate the final solution. Because the development environment consist of three independent software components, their implementation and testing can be run independently of each other, before being combined and tested as a whole system.

The final evaluation were conducted through multiple user studies as shown in the table below. Feedback from the evaluation was analyzed in a qualitative manner.

Method	Purpose
Survey	To help retrieve feedback on a proposed design before implementation, using a focus group consisting a combination of the target users. Their feedback should help to do further adjustments on the system of any concerns they may have from previous experience.
Observation	Observe how the target users are able to use the implemented system, and create applications for TILES.
Group interview	Evaluate the created language, to help find strength and weaknesses of the language's usability and expressivity

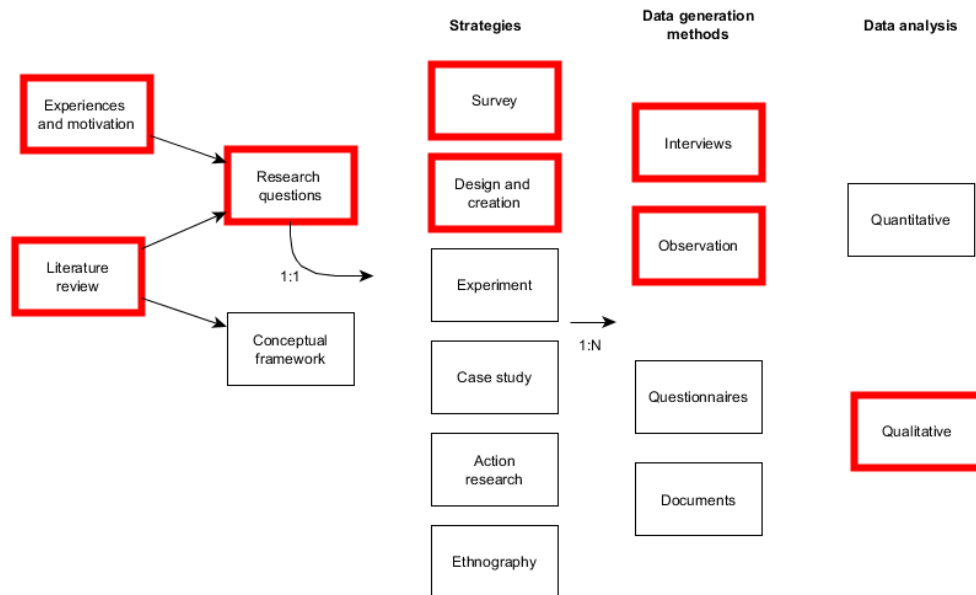


Figure 1.1: The research process [27]



## 1.5 Results

The main result of this research is a computer-based product, consisting of a Domain-Specific Language and a development environment, enabling makers and computer science students to create applications for TILES. The DSL is implemented specifically and customized for TILES, which can be compiled to applications able to run in the TILES infrastructure. The development environment is running in a web-browser, and consist of a text-editor and a visual-editor where the DSL can be written, compiled and published to the TILES infrastructure. The DSL and the development environment can be used by makers and computer science students, to help integrate their TILES devices into applications, and for creating application logic responding to their interaction with a single or multiple TILES devices. This thesis includes descriptive approaches used for implementing the DSL, which in the future can be used by developers who want to extend and introduce more functionality in the language.

There is also created multiple simple example applications using the DSL, showing its simplicity, and the various areas for TILES applications, including multiple TILES devices, reacting to user-interactions, controlling a TILES device output interfaces, and communicating with third-party services. An emulator of the TILES device is implemented and integrated within the development environment, as a complementary to the physical device, which should benefit the development process for TILES applications. By providing all necessary tools in the same development environment, makers and computer science students can both create, verify code, and test applications from their web-browser.

## 1.6 Outline

The motivation and research questions have been presented in this first chapter. The problem to be solved is explained in chapter 2, and an overview of the work which this thesis is based on is explained in chapter 3. The study and state-of-the-art systems are described in chapter 4, which chapter 5 bases its requirements for a DSL and an environment for using the DSL. How the development environment and its tools should be implemented is further explained in chapter 6, before a thoroughly explanation of the implementation process is given in chapter 7. Chapter 8 evaluate the work done, before the evaluation's results and answering the research questions are summarized in

chapter 9, with some suggested future works.

# Chapter 2

## Problem definition

End User Development[23, 33] is concerned about how end-users are able to customize, modify, and create software, to make applications and customized systems meaningful. Makers, and computer science students are the target users for TILES and therefore referred to as TILES' end-users. End-user development for ubiquitous and tangible devices is a crucial element for IoT to succeed [2]. Being able to control and create application for TILES usually requires a special set of skills and interests to do so. Not all end-users have the necessary skills for developing applications, and might find it too high of a threshold starting to create applications. Applications for tangible objects should support to be reconfigured, since its use cannot be envisaged at design time [4]. EUD removes the distinction between programmers and users of software[33], and requires a system to provide a set of methods, techniques and tools for non-professional software developers to do so. Unifying ubiquitous and tangible computing devices opens a whole new specter for interactive applications, applications which end-users are able to interact with, for multiple domains [10], which non-professional developers should be able to influence. EUD is initially done for software, and not IoT. Bringing EUD concepts to TILES should help to ease the process of creating applications for TILES.

There exist multiple solutions today running in a cloud-environment that use simple rules, or actions triggers, to run when a configured condition is met. The condition can be a given state of data from a given source, that the cloud-environment is listening for. Usually these data sources are from fixed hardware devices, configured for a specific task in a given scenario, like home-automation. The existing solutions also provide different environments, based on the users' expertise, for creating rules/applications, both

visually, textually, and physically.

*"Most ubiquitous computing systems are targeted at a wide user population, making it difficult to design a language that meets the needs of all users"*[14]. Steve Hodges [16] believes the maker community will create interesting new ubiquitous products in the years to come, mainly because of their wide variety of skills they possess and how these communities work collaboratively with open-source hardware and software. Modifying and extending example projects, provided and shared within these communities, help others to benefit from the experience of the community, and is argued to be an essential tool for EUD to succeed [12]. Different level of abstraction for programming is used depending on the target users. Lars Grammel [12] describes three-levels of abstractions: *high*, *intermediate*, and *low*. A high level of abstraction requires no programming knowledge, but provides no flexibility by only being able to reuse existing examples created by others. An intermediate level of abstraction requires knowledge about data types and being able to call pre-defined components, but no flexibility in being able to customize the underlying components. A low level of abstraction requires programming knowledge, and gives the most flexibility where components can be created by the users. By providing different levels of abstraction, users are able to choose the appropriate level that fit their skills. Seth Holloway [17] argues that expressive programming, a low abstraction, is needed for users to configure a system to their need. The paper's vision for ubiquitous computing is the need of an extensible software platform where any device is able to communicate with and be configured by. This is not the goal for TILES however, but the description of a system where ubiquitous devices are mapped to conditions and actions is still relevant. The paper also fail to mention the usage of multiple devices affecting each other, which is an important factor for TILES.

Work done in the pre-study [31] was focused on creating an infrastructure supporting real-time communication between TILES devices and client libraries, described in chapter 3. Being able to create applications for TILES, and to configure multiple TILES devices requires a development-environment configured for this purpose. Such an environment should help to reduce the complexity of developing user-interaction applications for TILES. Makers and computer science students should be able to use their creativity and incorporate TILES into multiple environments and use cases, whether thats in games, notification system, or simple actions triggers. Using multiple TILES devices together and incorporating them all into applications can create a game like *'Whack-a-Mole'*, where the player is to 'whack' a mole whenever it shows up on the game board. Translated for TILES usage this could be

tapping on the TILES device whenever its LED lights up. Another application could be created as a notifier, to turn vibration on for a single device whenever a new Twitter<sup>1</sup> message (tweet) is posted with a given hashtag, metadata tag usually found in social networking platforms. Incorporating multiple Internet sources is also possible with applications for TILES. An application checking weather forecast data could post a new twitter message if it's predicted there will be sunny tomorrow.

Users of TILES have multiple ways of interacting with a TILES device. Creating applications for a device that should respond to all user-interactions, communicate with other third-party applications over the Internet, and other available TILES devices is a complex process that includes multiple technologies for communication. Makers and computer science students are not professional-programmers, and should be treated as such by not requiring any knowledge about the underlying technology, to help reduce the complexity of creating such applications. To setup an environment ready to create applications can quickly become cumbersome when dealing with these three different methods of communication and interaction handling. Moving all the complexity behind the scenes and creating a higher-level of abstraction will enhance the creativity for TILES target users when they only need to focus on the application logic [13].

---

<sup>1</sup><http://twitter.com>



# Chapter 3

## TILES Toolkit

TILES [32] is a toolkit consisting of both hardware and software. These set of tools is designed to abstract application logic from an embedded hardware device to a cloud-server, so that physical access to the hardware is not required. The hardware is to transfer any commands and events, while the software is to create and act on these events and commands. The TILES device is designed and created by my supervisor Simone Mora, and the software in the TILES toolkit is developed by Varun Sivapalan and the author.

An overview of the TILES toolkit is shown in figure 3.1.

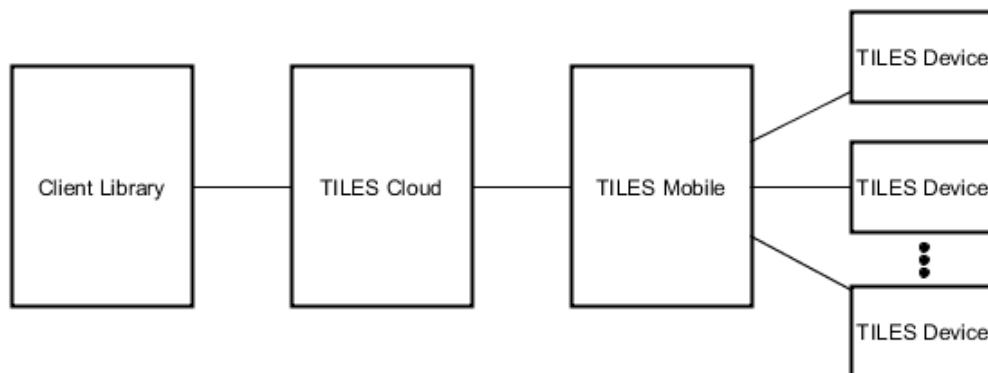


Figure 3.1: TILES Toolkit Overview

## 3.1 Hardware

The TILES toolkit consist of three layers, each consisting of some hardware:

- TILES Device - hardware-layer
- TILES Mobile - gateway-layer
- TILES Cloud - cloud-layer

### 3.1.1 TILES Device

The TILES device (TD) is an embedded hardware device designed for user interaction, with its internal sensors and actuators. An accelerometer sensor is used for detecting whether the devices is moved or shaken. The device also supports touch gestures which can be used for interaction. The different touch gestures supported are: *tap*, *double tap*, *force tap*, *swipe right*, and *swipe left*. Non-audible feedback is provided to the user by a vibration motor and a LED-light. The LED-light is able to provide different light-colors and intervals for how often the LED should be turned on and off. Speaker is another feedback mechanism for the TILES device. The TILES device specification and design is created by my supervisor Simone Mora. An outline of the different inputs and outputs supported by the TILES device is shown in figure 3.2. Inputs are user-events, generated by interacting with the TILES device, while outputs are feedback generated by the device to its users.

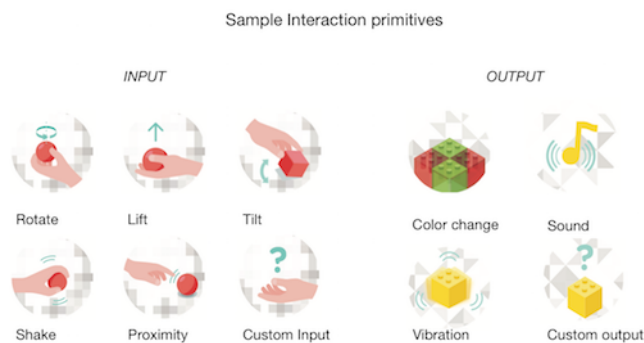


Figure 3.2: Interaction primitives for TILES. Source: Simone Mora

The TILSE device has an internal Bluetooth module, used for wireless communication with the TILES Mobile, a mobile-gateway, like a Smart-Phone.



### 3.1.2 TILES Mobile

As the TILES device do not have a direct Internet connection, the TILES mobile (TM) will provide an indirect connection by propagating all data both from and to the TILES device. As long as the TILES mobile has an Internet connection, whether that's through WiFi<sup>1</sup>, or some mobile telecommunications like 3G or 4G.

For a connection to be established between the TM and TD over Bluetooth, the TM need to search for and discover devices nearby.

### 3.1.3 TILES Cloud

The TILES Cloud (TC) is a server, accessible by anyone over the Internet. The TC is communicating with the TM to receive and send data back and forth.

### 3.1.4 Infrastructure

The three-tier infrastructure implemented during the pre-project [31], is a bidirectional communication channel consisting of three-layers: cloud-layer, gateway-layer, and a device-layer, as shown in figure 3.3. Real-time events triggered on the TILES device, by its user, is sent over Bluetooth to the gateway-layer, which is responsible to propagate this real-time events to the cloud-layer. Communication between the gateway-layer and the cloud-layer is done over MQTT<sup>2</sup>, a lightweight publish/subscribe message protocol. The protocol provides a full-duplex communication channel over TCP, making the cloud-layer able to indirectly control the TILES device's feedback mechanisms by sending commands to the gateway-layer. The gateway-layer is responsible for propagating any commands from the cloud-layer to the TILES device. The TILES devices is connected in a network, where each device is uniquely identified by its MAC address<sup>3</sup>, allowing for data collection and changing a device's behavior. Data collection is done automatic, where data is propagated to the cloud-layer when a user interacts with the TILES device, and semi-automatic/manual for controlling the unique identified TILES devices from the cloud-layer by sending commands to them.

---

<sup>1</sup>Technology for connecting to a Wireless LAN <http://www.wi-fi.org/>

<sup>2</sup><http://mqtt.org/>

<sup>3</sup>Unique identifier of a network interface, like Bluetooth

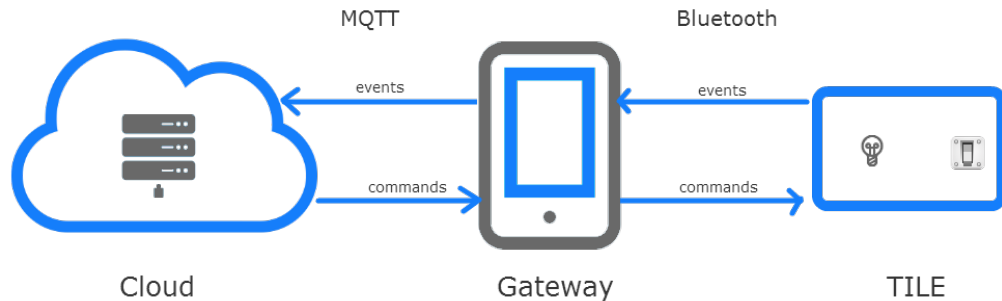


Figure 3.3: Three-tier infrastructure for TILES

The three-tier infrastructure is designed to abstract the application logic from the device-layer to the cloud-layer, where communication between each layer is done by *events* and *commands*. Events are actions triggered on a TILES device (its output), sent to the cloud-layer. Commands are actions sent from the cloud-layer to the TILES device (its input), for controlling the feedback mechanisms. Each event and command is published to a *topic*, used to identify the various connected TILES devices.

## 3.2 Software

For each layer described in section 3.1, software is created to handle data transmitted between each layer.

### 3.2.1 TILES Cloud

The TILES cloud (TC) is written in JavaScript using Node.js<sup>4</sup>. The TC is running Ponte<sup>5</sup>, a library for publishing and receiving data through HTTP, MQTT, and CoAP. Events triggered on a TILES device is received by the TILES cloud, and the TILES cloud is able to send commands back to the TILES devices. Both users and their TILES devices are stored in a database

<sup>4</sup><https://nodejs.org/en/>

<sup>5</sup><http://www.eclipse.org/ponte/>

storage, Mongoose<sup>6</sup>.

Accessing the stored user-data and the last received event from a TILES device is available through the provided API. Accessing the API endpoints is done over HTTP. Commands and latest events on a TILES device can also be retrieved using the API, which is both supported over HTTP and CoAP. This is an in-built feature provided by Ponte.

Real-time data is received by the TILES cloud MQTT broker. MQTT clients (e.g. TILES mobile) can be connected to the broker for both publishing and subscribing to topics. Subscription is used for receiving real-time events, and publish for sending commands to a TILES device.

### 3.2.2 TILES Mobile

An application, TILES Connect app, is running on the TILES mobile (TM), for registering, propagating data, and holding a connection to the TILES cloud. The application is written in JavaScript, using Ionic<sup>7</sup>, a cross-platform tool for developing mobile-applications using web-technology like CSS, HTML5, and JavaScript. Figure 3.4 shows the layout of the application, and steps required to successfully connect to the TC and connecting TILES devices to the TILES mobile.

### 3.2.3 Client Libraries

The TILES client libraries are designed to simplify the process of interacting with the TILES devices connected to the cloud-layer, where developers only need to focus on the application logic. With their wrapped in MQTT client library, the client libraries connect to the cloud-layer with the use of MQTT, like the gateway-layer. The client-libraries expands the infrastructure for TILES further, and are able to subscribe/listen to topics. Any received event to a topic is propagated from the cloud-layer to the designated client. The libraries are implemented in Java, Javascript, Python, and C++, so developers are free to choose the language they are most comfortable with. The client libraries were implemented to simplify the development process, enabling both experts and non-experts to create applications for TILES.

---

<sup>6</sup><http://mongoosejs.com/>

<sup>7</sup><http://ionicframework.com/>

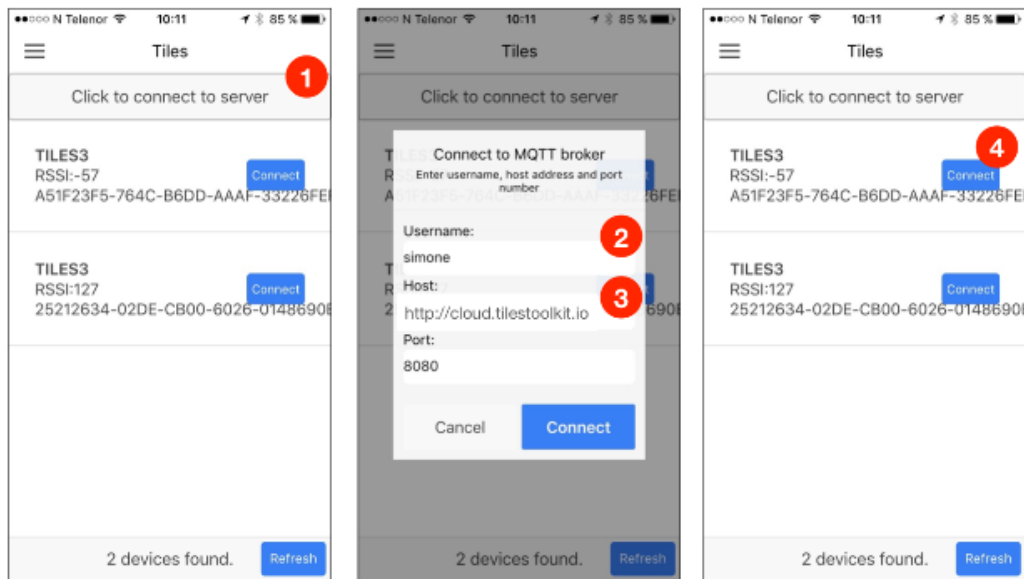


Figure 3.4: TILES Connect App layout. Source: Simone Mora

### 3.3 Overview

Figure 3.5 shows an overview of the whole infrastructure including hardware and software.

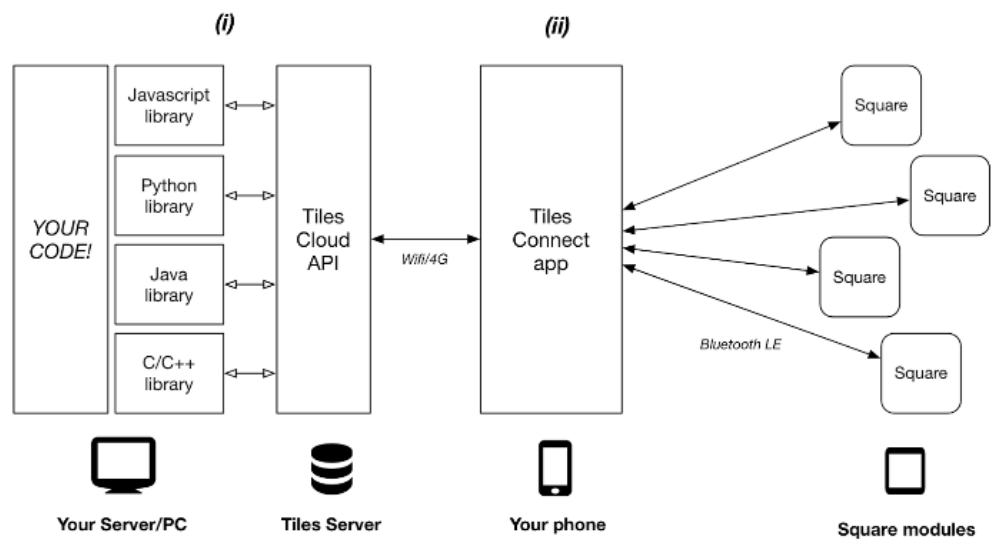


Figure 3.5: TILES Toolkit outline. Source: Simone Mora



# Chapter 4

## Related work

Arduino is an open-source hardware platform popular used in Makerspace, Hackerspace, and Fablabs communities. Creating hardware devices that interacts with the real-world, and help users understand interaction paradigms, like tangible and ubiquitous computing, is one of the goals which Arduino like to accomplish [30, 8]. Creating applications for TILES, where makers and computer science students are able to interact with their own environment and conduct some actions based on these interactions using the application, corresponds with the goal of Arduino and the vision of the Makerspace, Hackerspace and Fablabs communities.

Multiple academic papers address the importance and the need of being able to program ubiquitous device. Making it possible for end-users to create applications for their own ubiquitous devices will help making these devices more personal and to better fit in their own environment. It's difficult for the developers behind these ubiquitous devices to predict and consider all of the possible solutions a device can be used; especially when multiple data-sources is involved, from one or multiple devices and other Internet data-sources, which could be combined in infinite ways. By lowering the threshold for creating applications for TILES, makers and computer science students should be able to customize their TILES device to their liking.

Today, there exists multiple approaches for creating applications [23]: textual, visual, and physical. A textual approach use text for writing instructions to be carried out by an application. A visual approach uses visual elements, which are combined together to form a set of instructions. A physical approach, much like a visual approach, uses physical tangible elements to form a set of instructions. Each approach has their benefit depending

on the use case and the background knowledge for its target users. In this chapter, each approach and the different state of the art solutions are described. The end of the chapter provides an overview and short summary of each solution.

## 4.1 Visual Programming

*Visual Programming Language* (VPL) is a programming language for creating programs/software graphically. The language uses visual expressions used as the syntax for the language. There exist multiple types of visual languages, using different visual expressions [1]: Blocks, Flowchart, Data Flow, Finite-State Machines, Behaviour Trees, and Event-based Rules. They all have in common to ease the development of applications for novice programmers, without the need of any programming experience.

### 4.1.1 Blocks

Block programming is an approach widely used in multiple solutions today, and are found in environments for creating Android<sup>1</sup> applications, games, Arduino<sup>2</sup> software, controlling smart-homes and other tangible objects. The blocks come in different shapes, dictating how they can be connected to each other. Users don't need to learn the syntax of any programming language, but rather look what shapes are able to be connected together.

Scratch<sup>3</sup> [29] is a VPL aimed at children for creating games, interactive stories, and animations. Scratch have created a community around their product, where their users can share and build on others work, to better learn from each other, or just show off what they have been able to create. The system is developed by "Lifelong Kindergarten" group at MIT Media Lab. Applications are created in a visual editor, where blocks are connected together as shown in figure 4.1. A "mini-world" is shown next to the visual editor, representing the state of the running application, and is updated when blocks are removed, added, or updated. This helps its users to better understand what each block accomplish, and what effect it has on the application which the users are creating. The mini-world can control actuators movements,

---

<sup>1</sup><https://www.android.com/>

<sup>2</sup><https://www.arduino.cc/>

<sup>3</sup><https://scratch.mit.edu/>



listen for input from users, output text, and create logic for controlling the mini-world. By using drag and drop, users can easily create a simple application, and modify it as they want. The project aims at appealing to people and help them use their creativity to program simple applications, without needing any programming experience.

Scratch is an open-source solution, and have inspired multiple projects by either building on the same source-code, or use the same concept of visual block programming. Other VPL using blocks is described in appendix A.1. Some solutions is purely used educational, a way of making children adopt a *Programming Thinking* and help in problem solving, while other focus purely on application and game creation. Solutions that exists today are not able to create blocks from already written code, but only transform the created blocks into a programming language (Javascript, PHP, and Python).

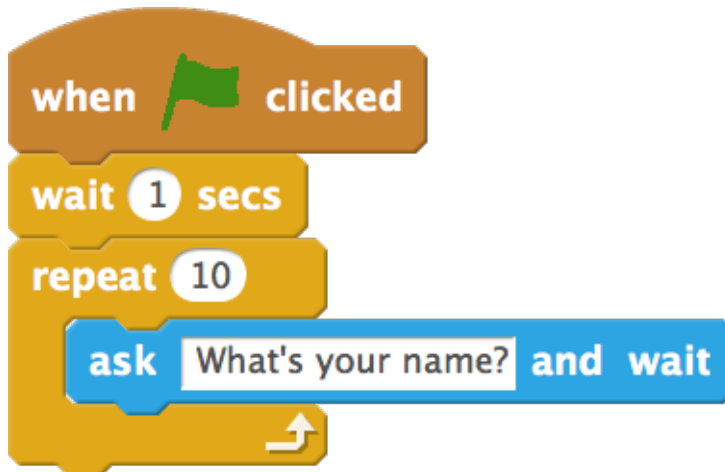


Figure 4.1: Example of building Blocs for creating logic. Example show the use of Scratch

### 4.1.2 Flowchart

Flowchart inspired languages use blocks and arrows to describe and show the control flow within an application. A block of code is executed as created visually, and follows a sequence defined by the connected arrows between the blocks. The focus for a flowchart language is expressing the flow of execution, which help to better identify and understand the different states of an application.

Flowgorithm<sup>4</sup> is a system for creating applications with the use of a flow chart. Flowgorithm uses shapes representing actions for the application, and helps the users to focus more on the application logic, instead of a programming language. The system can export code to more high-level programming languages like C++, C#, Java, Python, and many more. Flowgorithm is a free application, so it's possible for anyone to use and start creating applications using a flow-chart. The system is however only available for Windows.

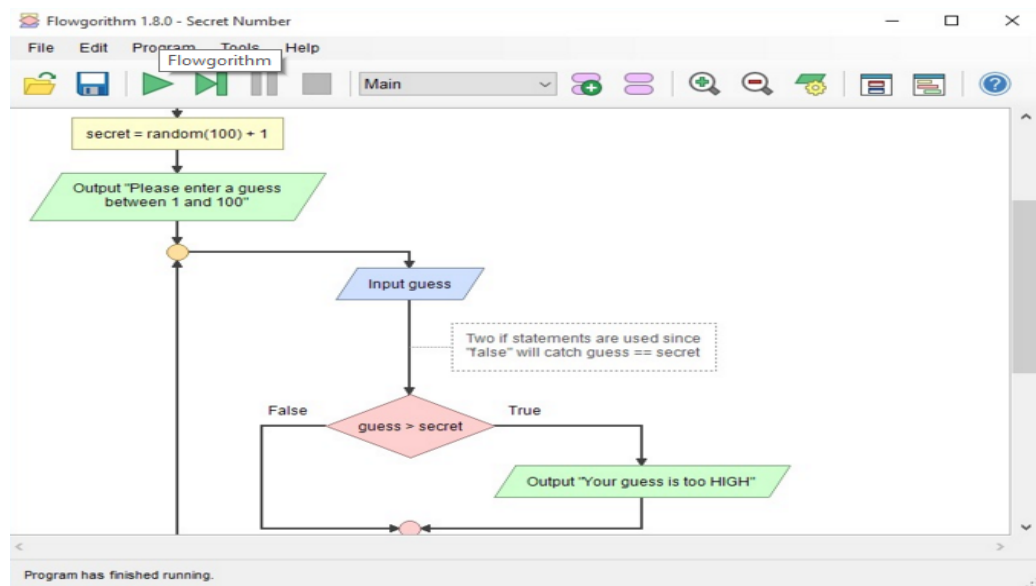


Figure 4.2: Example of a Flowchart . Example show the use of Flowgorithm VPL

<sup>4</sup><http://www.flowgorithm.org/>

### 4.1.3 Data Flow

Data Flow uses blocks representing functions in a programming language. The focus of Data Flow language is coherent with Flowchart inspired languages, to show the flow of execution, and data passed between each block. Each block has an input/output marker, showing what data is passed to a block, and what output the block generates. Most Data Flow languages are aimed at professional designers, with coding knowledge, as each block usually consist of a classical programming language. The main idea with a Data Flow language is to show the structure of a program, and how the data flows within the system, visualized by the blocks.

Node-RED<sup>5</sup> is a data-flow chart editor built on top of Node.js using JavaScript. Nodes are connected together using arrows, which dictates the data-flow of the application. It also incorporates a text-editor, where users are able to write their own JavaScript code which should be run when a node is active. Multiple nodes come pre-built with the system, for common operations like accessing a database, twitter libraries, debug interface, and many more. Users are also able to create and share nodes and full applications with each other, by exporting a project to JSON<sup>6</sup> file-format. Node-RED has a focus on IoT and rapid-prototyping of IoT applications. The system supports different types of distributed IoT devices, and let users specify flows of data, and how to process incoming data.

Juan Haladjian [15] describes TangoHapps, an IDE to create applications for smart garments. While TILES do not focus on garments, both TangoHapps and TILES are dealing with smart-devices and how applications should be created for these devices. TangoHapps provide two tools/approaches for developing applications: *Interactex*, a visual programming environment, and *TextIT*, a textual programming environment. The visual programming environment is provided to lower the threshold for designing, developing, testing and deploying applications to the smart-garments for users with little programming knowledge. The textual environment offers software components ready to be used for developing applications. Both Interactex and TextIT is used to support developers with different backgrounds. The visual programming environment for TangoHapps is using a Data Flow approach, by mapping the input and output interfaces for different components together.

---

<sup>5</sup><http://nodered.org/>

<sup>6</sup><http://www.json.org/>

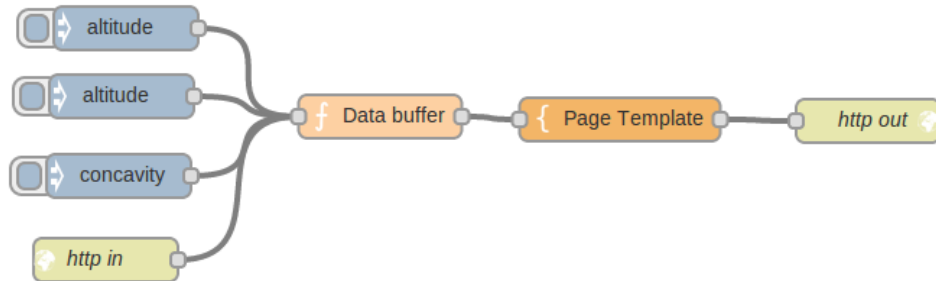


Figure 4.3: Example of a Dataflow . Example show the use of NODE-RED

#### 4.1.4 Finite-State Machine

A Finite-State Machine language consist of states, represented as blocks, and transitions between states, which are triggered by user-defined conditions. As with Flowchart, a flow of execution is created by linking states. Moving between states are controlled through the transitions and conditions between the states, instead of inside them, like in a Flowchart language. Compared to both Data Flow and Flowchart, the Finite-State Machine is harder to understand, because of the use of text expressions needed to manipulate state change conditions. The use of Finite-State Machine is good in viewing the structure of a system, but requires more technical competence to be able to configure it.

Unity3D<sup>7</sup> is a game engine which provides a tool named *Macanim*, a Finite-State machine visual language for creating animations for any visual object in a game created by their engine. Multiple tools are made available additionally to the finite-state machine editor to successfully create and control game animations. The editor itself is used as an overview tool, to show the different animation states and group them together, and can therefore not be used as a tool alone for configuring states. The editor is a complex too to work with and requires deep knowledge about the system.

---

<sup>7</sup><https://unity3d.com/>

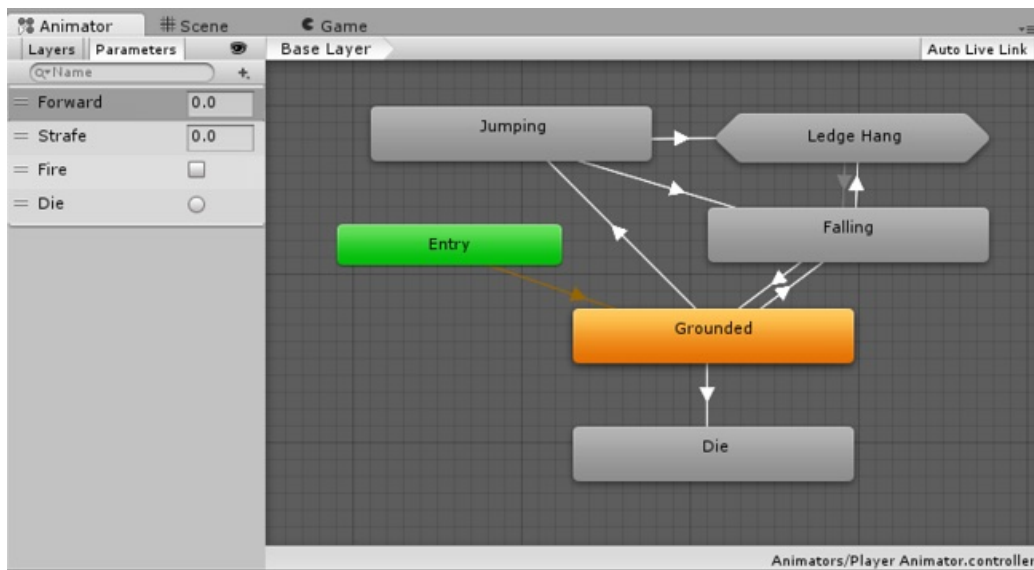


Figure 4.4: Example of a Finite-State Machine. Example show the use of Unity3D Macanim

#### 4.1.5 Behaviour Tree

A Behaviour tree is built on a tree structure consisting of parents and children nodes. Each node within the tree are returning a state, which are made available to its parent. A node can therefore have a behaviour that depends on any state returned by its children, which will control the flow of execution. The flow can be configured by adding other nodes instead of editing a node, removing the need to a broad visual grammar and to edit any code inside nodes.

Behaviour3<sup>8</sup> is a framework which can be used for applications that control agents and/or devices in their environment, called agent-based applications, whether that's in games, simulations, or robotics. The project is open-source, and consist of a visual-editor for combining nodes and creating applications with the use of behavior trees. Libraries are also provided for both JavaScript and Python, which are used by the visual editor for creating and exporting code in a specific programming language. The framework is aimed for both programmers and non-programmers, and provide capabilities for extending the framework by introducing the ability to create customized nodes.

<sup>8</sup><http://behavior3.com/>

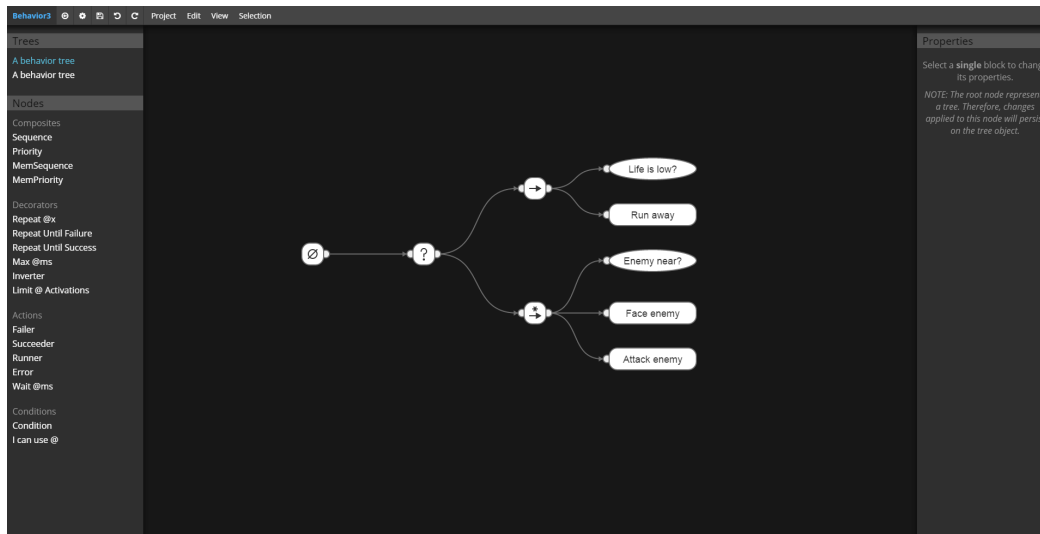


Figure 4.5: Example of a Behaviour Tree . Example show the use of Behaviour3

### 4.1.6 Event-Based Rules

Event-Based Rules are good for simple logic handling, and consist of rules to trigger when a given event occurs. Each rule is defined by a visual block, which can be connected by other blocks, creating a set of rules. The Event-Based Rules are easy to understand because of its simplicity, which also make it good for simple tasks, which in turn can make the language limited.

Kodu is a VPL for creating games for both PC and XBOX, created by Microsoft Research. The system is designed for users with no programming experience, both children and adults. BBC micro:bits<sup>9</sup>, an embedded programmable hardware device, is supported by Kodu and introduces the use of rule blocks to be used for creating simple applications that are able to control and listen to its input and output interfaces. Each rule block consists of an image that represents the capability each block provides.

<sup>9</sup><https://www.microbit.co.uk/>



Figure 4.6: Example of Event-Based Rules . Example show the use of Kodu

## 4.2 Physical Programming

Physical programming is removing the need of a keyboard, mouse, and screen. It involves users engaging with physical tangible elements to create a sequence of instructions, which is compiled into a machine language ready to be run in a computer system. Because of its use of tangible elements, physical programming is usually referred to as tangible programming.

A tangible user interface (TUI) key idea is to give physical forms to digital information [21]. The tangible elements are used to give an abstract representation of some code, and can consist of programming elements, commands and flow-of-control structures [19]. Connecting these representations forms a computer program. Like visual-programming, a tangible interface removes the need of learning a complicated syntax, since users only need to interact with abstract notations of some pre-defined code, and how these tangible elements are implemented is of no concern to its users. Being able to configure and control digitally content using physical elements help to bridge the gap between the virtual and physical world [25], and have been found to be positive for an educational purpose, and as an introduction to programming.

Michal Horn and Robert Jacob [19] describes an approach using tangible

interfaces and the implementation of two tangible languages. Their solution provides interfaces that can be connected together like puzzle pieces. The pieces form instructions, which are carried out by physical or virtual robots. The structure of the pieces is transferred to a computer, where the instructions are compiled to either the Quetzal programming language, which is usually used with LEGO Mindstorms, or the Tern language, used for controlling virtual devices on a computer. Unlike TILES, their solution is aimed at younger children to be used in educational environments, like classrooms. TILES want to ease the process of developing applications for the TILES devices, while Horn and Jacob focus on students learning collaboration and complicated syntax using tangible languages. Multiple concerns are addressed and described for the design and implementation of the two tangible languages because of the target users and the system's area of use. For TILES the implementation of a tangible language is still relevant, even though TILES have a different area of use and target users.

PrimoToys<sup>10</sup> provides a robot that uses tangible elements for controlling its movements. Each tangible element represent a motion which the robot can carry out, making it easy for children to comprehend. The different parts offered by PrimoToys is shown in figure 4.7. The tangible elements are placed on a board, which is compiled and uploaded to the controllable robot.

Little Bits<sup>11</sup> is an approach on physical programing using tangible building blocks, called bits, that can be snapped together using magnets to create a circuit of blocks, where each block provides some input to the next. Each block is pre-set with a feature, from power, input, output and logical operators. The power is blocks which provides power to the whole circuit, while the input is different sensors able to read from its surrounding area, user-interactions or temperature data, outputs can control or notify its environment using LEDs or motors, and the logical block can split up a circuit, corresponding to `if...else` or other logical operators usually found in programming languages.

### 4.3 Mini-Language

A mini-language, also called domain-specific language (DSL), is a programming language with small syntax and simple semantic [5]. Creating a mini-language requires commands, queries and control structures be made avail-

---

<sup>10</sup><http://www.primotoys.com/>

<sup>11</sup><http://littlebits.cc/>



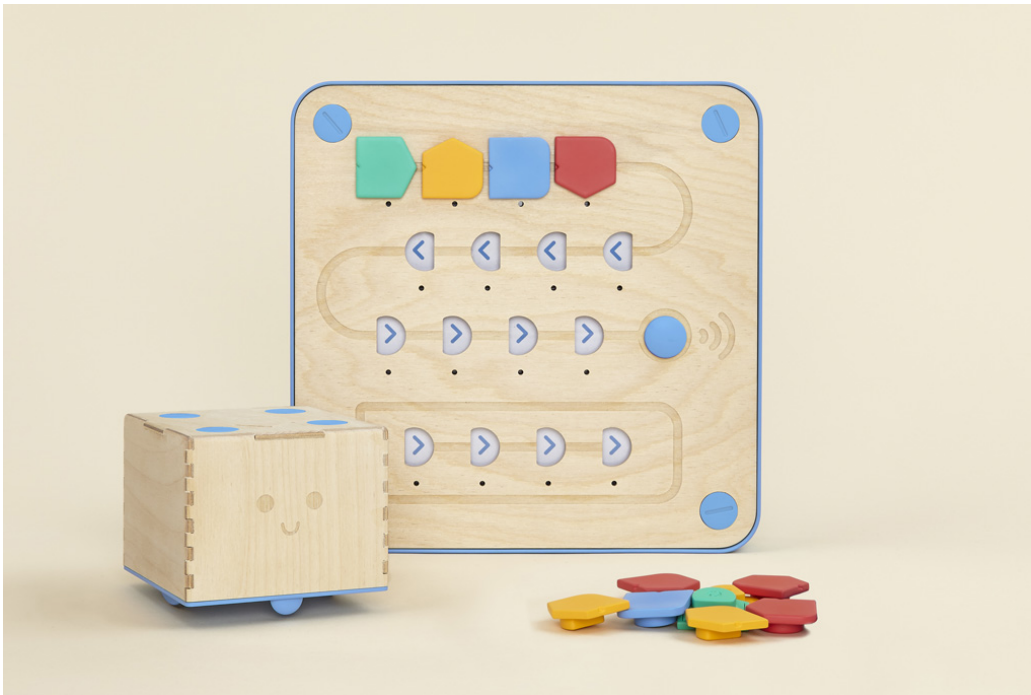


Figure 4.7: Example of physical programming. Example show the use of PrimoToys

able to its users. With a known audience and usage content for the mini-language, the language can be tailored to its needs and the users background knowledge. Control structures, data types and keywords can be provided that are only essential for the use case, and be provided in the users' native language. This together can help in creating an attractive, meaningful and effective environment.

Because of a mini-language simplicity, learning the language does not requires much time, so that the focus can be on developing the actual application logic. Mini-languages are also used to help understand the principles of programming, and more general purpose languages [5] like C, Pascal and Lisp.

A mini-language is used as any other programming language, written in a text-editor. Most IDE's today have an interpreter, an auto-complete feature when writing code, and highlighting of important keywords and variables, which is important for mini-languages as well. Some mini-languages incorporates a mini-world, with an actor which is controllable by the mini-

language. Providing input commands, queries and control structures, the users are able to observe the code-execution and how it affects the actor and the mini-world.

CodeCombat<sup>12</sup> is an online platform created to introduce students to programming by controlling a game, shown in figure 4.8. CodeCombat consist of a text-editor and a visual mini-world. The mini-language is used to control an actuator within the mini-world, and the users should use the provided mini-language to circumvent any obstacles. JavaScript, Python, and Java is available to be used for the mini-language, and multiple game boards and characters are available so that users are able to customize their game experience.

Thomas Kubitzka and Albrecht Schmidt [22] addressed the need for a toolkit to help ease building applications for smart devices in interactive environments, where designers and developers can focus more on the application logic. Configuration of the devices using the proposed toolkit is done in a scripting language, JavaScript, and a web-based IDE where developers are able to write code. The paper argues that the right set of tools for developing applications for smart-devices should empower people with different background to create their own smart-environment, as Arduino has done for physical prototyping. Compared to TILES, this toolkit also provides a client software to remotely control and access a devices actuators and sensors, and a central server which all the devices are connected to, where users are also able to control each device using JavaScript. Unlike TILES, the toolkit is aimed to be a multi-device system, for a heterogeneous set of devices. The paper addresses the need of a visual programming tool (Blockly), that should be built on top of the JavaScript layer.

DemoScript, described in [7], is a system for scripting cross-device wearable applications, and provides a visually storyboard used to illustrate the step-by-step execution of the application. The system consists of a web-IDE which the user interacts with, and a server hosting the scripts developed by the users. The Weave framework [6] is running in the backend of the server, used for creating cross-device interactions with the use of scripting. The framework is built on top of JavaScript, and provides an environment to run the user-generated scripts. DemoScripts has a focus on testing with its provided storyboard, which can be used to revise different aspects of a program by direct manipulation [7]. Unlike TILES, the system is built for cross-device interaction with devices of different variety of capabilities. The paper also addresses the complexity of using multiple interactive devices,

---

<sup>12</sup><https://codecombat.com/>

that are able to influence the application logic, which is a problem TILES want to solve and make more easy to integrate into applications.

Orchestrator.js is a middleware for building applications for multi-devices in heterogeneous environments [24]. This is a web-based tool, where a web-IDE for implementing applications are provided. Users are able to manage their embedded devices from a console, named *Web Console*. From the Web Console, user-created applications can be combined with registered devices, defining observers on actions triggered by devices. The system is also supporting communicating with social applications, which is able to communicate with embedded devices through the Orchestrator.js middleware. The middleware does not know the capabilities a connected device provides however, which need to be stated by the user itself. This is not the case of TILES where all devices connected provides the same capabilities.



Figure 4.8: Example of a mini-language. Example show the use of Code-Combat

## 4.4 Overview

This section summarize the papers and program solutions mentioned previously, and provides an overview of what type of programming they support or mention.

	Visual Programming	Physical Programming	Mini-Language
<b>Demoscript</b>	X		X

<b>TangoHapps</b>	X		X
<b>Quetzal language</b>		X	
<b>Tern language</b>		X	
<b>IoT Toolkit</b>	X		X
<b>Orchestrator.js</b>			X
<b>Tangible programming bricks</b>		X	
<b>Scratch</b>	X		
<b>Flowgorithm</b>	X		
<b>Node-Red</b>	X		
<b>Unity3D</b>	X		
<b>Behaviour3</b>	X		
<b>Kodu</b>	X		
<b>PrimoToys</b>		X	
<b>CodeCombat</b>			X

Table 4.1: Related work overview

As shown in the table, there exists multiple solutions supporting different approaches for creating applications. Each system has in common wanting to ease the development process for its users. For TILES, the different programming paradigms used by the solutions is interesting, to see how state of the art solutions wants non-programming experts to create applications using their creativity. The systems focusing on ubiquitous devices is more interesting for TILES, as these systems focus on how someone is able to use these devices and incorporate them into applications. However, none of the described solutions take into account the usage of multiple tangible devices, that are able to be interacted with at the same time, and act together to influencing the application logic. Other solutions also focus on supporting different types of ubiquitous and tangible devices. TILES only need to focus on supporting one device that provides the same capabilities. Creating a tool where multiple interactive devices can influence the application logic is a hard task to accomplish, which TILES would like to support.

By creating a mini-language, the language could be tailored and better suited for TILES use, enabling TILES target users to easier express their desired functionality through a Domain-Specific Language. Such an approach is considered to improve End-User Development [23], and should provide an intermediate abstraction of functionality, in which TILES target users are able to use and call components within the language, but not create new

components. Using multiple programming approaches would ensure different levels of complexity to be provided, and having a system that is able to adopt to its users. Each abstraction would lead to an increase in complexity and expressivity, ensuring that small changes are simple to accomplish, while more complicated features only require a small increase in complexity [23]. Providing a DSL tailored for TILES usage is therefore argued to give its users great expressivity, helping to accomplish desired functionality than a conventional programming language would.

Many of the state-of-the-art systems are run in a web-environment, making it more accessible to its users. By providing a development-environment as a web-application, creating a community around TILES, in which its target users are able to share knowledge with others, become more easy. Creating a community around a system, is argued to be of great benefit for its users, as is also shown by other state-of-the-art systems, i.e. Scratch, when users are able to learn from others, and become inspired by others work. The architectural aspect of a web-environment also helps in providing up-to-date system without having its users update it manually. A way of integrating a mini-world, as described for multiple state-of-the-art systems, in which TILES target users are able to influence an applications behavior, can be accomplished by implementing a web-based emulator, emulating a TILES device.



# Chapter 5

## Requirement Specifications

Requirements for developing an environment for creating TILES applications is derived from chapter 4, *Related work*. A comparison between the different programming approaches: textual, visual, and physical is done to help choose which approach is the best fit for the different users of TILES: makers and computer science students with some programming experience, but not necessarily interactive objects competencies. To compare the different approaches, each solution is compared in terms of how well they fit into the TILES infrastructure, and how they can be used to create TILES applications.

### 5.1 Language comparison

All three approaches: visual, physical, and mini-language, provide a higher abstraction of programming which could be used to help lowering the threshold of creating applications for TILES.

A visual-programming approach is argued to make the programming task easier for all users, both novice and professional programmers [26]. A visual representation is also argued to be a more natural and efficient representation than a textual approach [26, 28], since the human visual system is optimized for multi-dimensional data, which could help to better generate and understand visual code. By abstracting code using visual notations, much more information of a program state can be presented than a purely textual approach. Information can be hidden away from its users about the underlying implementation, so that syntax, variables, and data structures is stripped

away from the application logic. This can again empower its user to purely focus on the application logic, and help lowering the threshold for creating applications for TILES.

Physical programming introduces the ability to directly influence digital information by engaging with physical tangible objects in the real-world. As with visual-programming, tangible objects that are available to its users represent a higher abstraction of some underlying information which the engaging users do not need to have any knowledge about. Being able to directly influence behavior of an actuator without the need of a computer is more appealing to younger children and supports collaboration and sharing of code [19, 20]. Physical programming using tangible objects is found to be equally easy to understand as a visual-approach [20], but more appealing than both a textual- and visual-approach for users working in groups (i.e classroom) [19]. A physical approach for developing applications for TILES requires pre-built tangible interfaces to be deployed and made available to its users.

A mini-language, DSL, is a language designed for a specific domain and use case. A DSL can be implemented with several approaches [34]. All approaches generate abstract notations that makes up the DSL, to hide the underlying details and give more expressive power to its domain. The difference is whether the general-purpose language should be made accessible alongside with the DSL, to give a broader expressibility. To help lowering the threshold for creating TILES applications, abstract notations should be made available to its users, which does not require any knowledge of the underlying system. A DSL can be implemented as a library, and be used both by a textual-, visual-, and physical-approach for creating applications.

As the target users for TILES are makers and computer science students, they are expected to have basic knowledge about the Internet, and open-source hardware and software. Makers are in communities where sharing of knowledge is important. Being able to store already created code, which can be further re-built at a later stage, and easily shared between users is therefore an important requirement for the development environment for TILES, which a physical approach fails to acquire.

Software updates are more easily deployed and made available to everyone at the same time when done over the Internet. A development environments accessible using a computer is therefore able to always provide up-to-date software to all its users. Users are able to store and share generated code with others over the Internet, both in textual-form and visual-form. Implementing a DSL as an underlying layer which both a textual, visual, and



physical approach are able to use, enhances modifiability, as a modification in the underlying layer will be made accessible to all top-layers, as shown in figure 5.1. A textual and visual approach for developing applications for TILES is prioritized for this thesis, as a physical approach with tangible devices is too big of a task to accomplish with the provided time. Building a physical approach on the common DSL can however be done in future works.

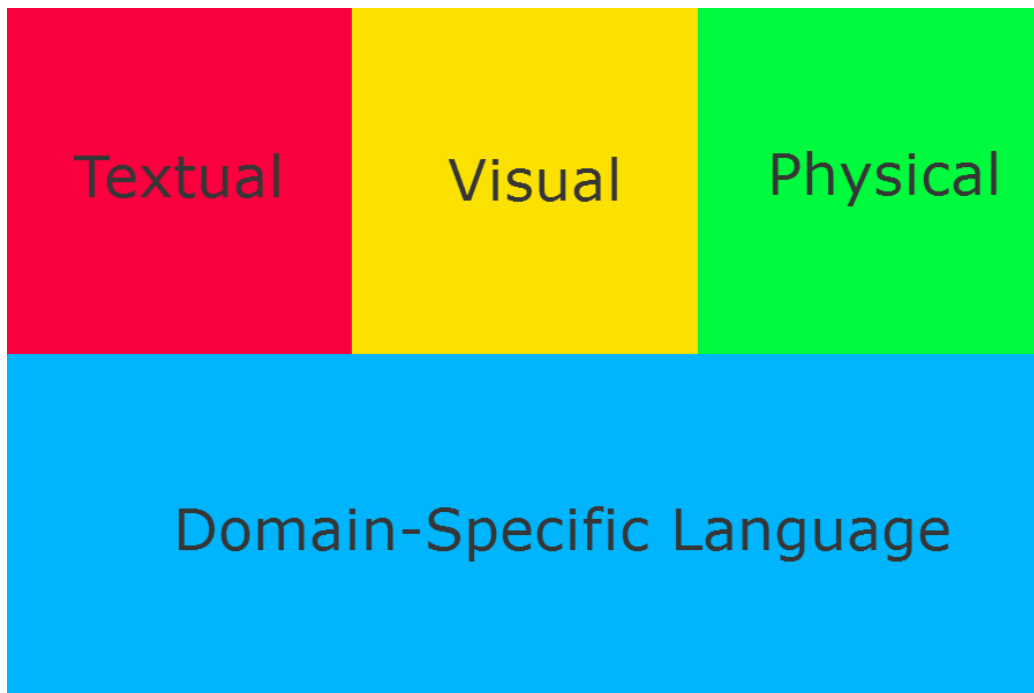


Figure 5.1: Language layer, where top layers have access to the underlying layer

By supporting both textual and visual programming approaches should make TILES available to users with different programming experience and background. Both approaches should give the makers and computer science students a *"low floor"*, *"high ceiling"*, and *"wide walls"*. Approaches with a low floor is solutions that are easy to get started with. For TILES, this means it's easy to get started developing applications, as the target users are able to choose an approach they are more comfortable with: textual or visual. A high ceiling for the language means it provides expressive power for creating applications, and let rather complex applications for TILES to be developed. Using the DSL and the general-purpose language made available through the DSL, makers and computer science students are provided with a wide

range of functionality, which they can make use of within their applications. Wide walls refer to TILES being able to be incorporated into a wide range of areas that is of interest to makers and computer science students, whether that's for games, notification and event-based applications (social network, weather, etc.). With wide-walls, the makers and computer science students are provided the freedom to incorporate TILES into any environment of their choosing.

Alexandre Demeure, Sybille Caffiau, Elena Elias and Camille Roux [9] describes a study for home-automation and different tools for programming for home-automation. The study includes users with different programming experience. Visual programming is found to be used in most houses because of their simplicity. Both block visual programming, event-based rules visual programming, and scripting languages are found to be used for these users. VPLs are used by users with little programming experience, and scripting languages for users with more programming experience. The paper argues that a higher-level of abstraction should be provided for programming ubiquitous devices, to more easily create rules, variables and access devices. Programming abstractions using a DSL for TILES should therefore ease the development process for creating applications.

As syntactic issues is not a problem for users with great programming experience [36], a visual-approach which removes e.g. syntax issues may seem unnecessary. Applying VPL to TILES is however believed to be important for supporting different approaches to create applications for TILES.

## 5.2 Visual-language Comparison

In this section, the visual programming environments described in chapter 4 and appendix A will be compared. For a tool to be successful incorporated into the TILES infrastructure, the system should be extendable, to create custom visual elements that are of relevance to TILES. It should also be possible to generate ready-to-use code of the visual-representation to a programming language supported by the TILES infrastructure, preferably JavaScript, as the solution is built on top of Node.js<sup>1</sup>, and the author has more experience in this programming language.

Visual languages using the block approach for building applications, only Snap! and Blockly are systems implemented in JavaScript, and which sup-

---

<sup>1</sup><https://nodejs.org/en/>

ports generating custom blocks. Snap! is highly inspired by Scratch, and incorporates a mini-world for executing the created applications. These projects can be exported and used in other Snap! environments as well, but for running the created applications stand-alone, the run-time environment of Snap! is required. Exporting the created application from blocks to programming code is only supported by Blockly. Each block represents a combination of some programming code, and the combination of the connected blocks can be exported to a programming language (Python, JavaScript, PHP). This code is then able to be executed without any dependency of Blockly. The use of a mini-world, which Snap! includes, could however help with the development process when generating applications for TILES, but is of no need when the application should be executed. Using Blockly for the development environment for TILES would require a minimal set of dependencies in the run-time environment on the cloud-server, and therefore provide a better performance than with Snap!.

Node-Red is a data-flow visual programming language that is implemented in Node.js (JavaScript), and support generating custom nodes. Node-RED have a visual-editor for generating data-flows, which can be uploaded to and executed in its server-environment. Incorporating Node-Red into the existing infrastructure for TILES, would require both a client and server side application. As the application instance would need to be run alongside the TILES Cloud server, much work would be required to incorporate it into an existing system.

The study of home-automation programming systems [9], also found that most of the users were using visual languages that were using the block approach, because of its simplicity and the expressibility, over the simpler approach with event-rule creators. Because blocks for visual programming provides a high-ceiling and low-floor, incorporating such a system would be a good fit for TILES. Blockly is also found to be the most capable system to implement into an existing infrastructure because of its simplicity, support code-generation from visual-elements, and easily customizable because it's written in JavaScript.

### 5.3 TILES Development Environment

The requirement for creating the TILES Development Environment (TDE) are based on the reviewed literature and related work in chapter 4, and the comparisons in section 5.1 and 5.2. Textual online IDE editors were

compared in the previous study [31], where ACE<sup>2</sup> were found to be the best fit and is incorporated as a part of the TDE requirements as well. The high-level requirements are meant to denote what is needed for creating an environment which should help ease the process for developing applications for TILES, and are shown in table 5.1.

Priorities of the requirements can either be *Low*, *Medium*, or *High*, denoting the importance of a feature for the development environment to be ready for use.

ID	Requirement	Description	Priority	Dependency
FR1	The TDE should be run in a web-environment	Running in a web-environment as a web-application should make the system easy accessible in a web-browser, and not be dependent on any OS	H	
FR2	The TDE should provide a textual-editor, ACE, for writing code	The textual-editor that should be implemented is ACE, a web-IDE that support multiple languages for syntax highlighting	H	FR1
FR3	The TDE should provide a DSL that abstract common operations for accessing other third-party services and other TILES devices	Third-party services are other Internet applications for fetching third-party data, like weather and social network data	H	FR2
FR4	The DSL should be built with JavaScript	This should ensure the language to be easily implemented into the existing TC	H	
FR5	The TDE should provide a visual-editor, Blockly, for generating code visually	The visual-editor that should be implemented is Blockly, a block-based visual editor that supports customized blocks	H	FR1

---

<sup>2</sup><https://ace.c9.io>

FR6	The VPL should be able to generate JavaScript code from the created visual elements	Blockly supports exporting generated blocks to both Python, JavaScript and PHP out of the box	H	FR5
FR 7	The VPL should create visual representation of the abstract DSL functionalities	This should ensure that all functionalities made available with the DPL are accessible visually, and familiarize the users with concepts for TILES to more easily change between both concepts	L	FR3, FR5
FR8	The TDE should support saving both visual and textual code	Users are able to save their work, and not lose any progress	L	
FR9	The TDE should support restoring both visual and textual code	Users are able to restore previous work and further customize it at a later stage	L	
FR10	The TDE should support uploading of any generated JavaScript code to the TC	User-created applications should be run in the TC	M	
FR11	The TC should be able to run user-generated JavaScript code in a sandbox environment	User generated code shouldn't have access to the TC environment and influence resources an application shouldn't have access to	L	
FR12	The users should be able to test their code in a virtual environment	Should help users to write applications with intended features	L	

Table 5.1: Requirements for the TDE

## 5.4 Domain-Specific Language

The requirements for the DSL is described as user stories, a way of formulating requirements from users' perspective. User perspectives for the DSL are formulated from the perspective of TILES target users: makers and computer science students, referred to as *user* in the user stories. The user stories should help denote what features are required of a DSL to create applications for TILES, and what makers and computer science students expect from the TILES DSL. Each user story includes acceptance criteria, to known when a user story is successfully implemented. The user stories could later be used in a backlog, to help developing the features of the DSL.

#	As a/an	I want to...	so that...
US1	user	send commands to a TILES device	I can control its output interfaces
US2	user	be presented will all available commands that I'm able to send to a TILES device	I can easily choose a command, without remembering all of them
US3	user	listen for events triggered on a TILES device	I can act upon an incoming event
US4	user	identify incoming event	I know which event was triggered on the TILES device
US5	user	identify each TILES device registered to me	I know which devices I'm able to create applications for
US6	user	use a device identification name in my application	I can easily refer to one of my registered devices within the application code by name
US7	user	retrieve data from a third-party source	I can use data from other sources within my application
US8	user	send data to a third-party source	I can store/notify/post my data to other Internet applications

<b>US9</b>	user	have application libraries available for common Internet services (Twitter, Facebook, Weather)	I can more simple use services that I'm more likely to use, without the need of copy/paste common code for each application
<b>US10</b>	user	have access to my stored data	I can use TILES devices that I have registered in my applications for communication
<b>US11</b>	user	provide a pattern of interaction consisting of a TILES device and an event	I'm able to trigger an action when a certain pattern of interaction is met

Table 5.2: User Stories for DSL

#	Acceptance Criteria
US1	<ul style="list-style-type: none"> <li>- User is able to control all interfaces of the TILES device</li> <li>- Able to activate an interface</li> <li>- Able to deactivate an interface</li> </ul>
US2	<ul style="list-style-type: none"> <li>- List off all interfaces are shown (e.g. LED, Vibrate, Speaker)</li> </ul>
US3	<ul style="list-style-type: none"> <li>- Application is notified when a user interacts with one of the TILES device interfaces (Button, Shake...)</li> </ul>
US4	<ul style="list-style-type: none"> <li>- Application is able to parse incoming event message</li> <li>- The identified event match the triggered event</li> </ul>
US5	<ul style="list-style-type: none"> <li>- List all devices registered to current user</li> <li>- TILES devices listed are only registered to user</li> </ul>
US6	<ul style="list-style-type: none"> <li>- Using a TILES device's name is interpreted as using the TILES device's MAC identification</li> </ul>
US7	<ul style="list-style-type: none"> <li>- Able to communicate with other HTTP service</li> <li>- Result retrieved</li> </ul>
US8	<ul style="list-style-type: none"> <li>- Able to communicate with other HTTP service</li> <li>- Able to send user-defined data along with the HTTP request</li> </ul>
US9	<ul style="list-style-type: none"> <li>- Libraries created with few user-inputs to start working with its data</li> </ul>
US10	<ul style="list-style-type: none"> <li>- Access to stored username, and TILES registered</li> <li>- Able to refer to TILES by their name</li> </ul>
US11	<ul style="list-style-type: none"> <li>- Able to provide a pattern matcher consisting of multiple events and TILES devices</li> <li>- Action run when given pattern is triggered</li> </ul>

Table 5.3: Acceptance criteria for User Stories



# Chapter 6

## Design

This chapter will formulate an approach to help design the development system for TILES, by first explaining the different actors within the system, and what functionality they expect. A formal definition of a DSL is provided, a language that should provide abstract functionalities to help ease the process of creating applications for TILES

### 6.1 Introduction

Makers and computer science students with some programming knowledge are the intended users for the TILES Development Environment (TDE), and are therefore referred as the *users* of the system. They will be using the TDE for creating applications for TILES, visually or textually. The environment will provide abstract functionalities for TILES specific operations, communication through the TILES infrastructure, and data-retrieval from third-party services. The abstract functionalities, made available through a designed DSL, will help lower the threshold for creating TILES applications. Deep knowledge about the underlying system is not required when dealing with abstract notations of the functionalities, and the users only need to be concerned about how the abstract functions should be used. Using the DSL, the users are able to tell other TILES devices what "to do" and which data-sources the application should retrieve data from. Simple knowledge of the TILES device's output interfaces, and the supported interaction methods, is required, as these are mechanisms that need to be referenced and used in the application logic

Applications created by the users are to be run in the TILES infrastructure, ready to respond to any incoming TILES events from users interacting with their TILES devices. The DSL therefore need to be developed for the run-time environment in the TILES infrastructure, as the created applications are to be executed in this run-time environment. Moving the application from the TDE to the TILES infrastructure requires the TDE to upload the created application logic to the TILES infrastructure.

The TDE will also need to retrieve stored data in the TILES infrastructure about the available TILES devices for its users. Each application is able to communicate and integrate TILES devices into the application logic. Information about the available devices for a user, helps the users to identify which device can be references in the application logic. Each device is denoted with a custom name, along with its MAC address for identification. This identification can be the reference name within the application, as both custom names and MAC addresses should be unique.

Each actors goals are shown in figure 6.1. The user is here defined as the users of the TDE, makers and computer science students, TC as the TILES Cloud, and TD as the TILES Device. This use-case diagram is used to show what goals each actor need to fulfill, to provide the desired functionality for creating applications for TILES from a user perspective.

Creating applications for TILES requires different steps for the TDE to carry out:

1. Connect to TC
2. Discover devices
3. Load textual/visual editor
  - (a) Load DSL

After these tasks are done by the system, the users are ready to start creating applications for TILES:

1. Choose which device the application should be registered to
2. Pick event that should trigger application
3. Implement application logic
4. Upload application code to the infrastructure



Figure 6.1: Use Case diagram showing each Actors goal

## 6.2 Use cases

Use cases that are describing the most critical aspects of the TDE from the maker and computer science student's perspective is shown below, and are referred to as users in each use case. These use cases are used to help denote what actions is required by the user, and how the system should respond to them.

<b># 1 - TDE</b>	Retrieve TILES Device details
Will retrieve all stored information about all TILES devices registered to the current user	
<b>Primary Actor</b>	User
<b>Precondition</b>	TDE connected to TC
<b>Trigger</b>	User click retrieve TILES Device information
<b>User Action</b>	<b>System Action</b>
1. Request TILES Device information	2. Query TC to retrieve all TILES Devices registered to user 3. Check if there are devices registered for current user
	4. Return requested TILES Devices 5. Store information client-side
<b>Exceptional Paths</b>	
	3.1 No devices registered to user 3.2 Returns empty set 3.3 Give error message This ends the use case
	5.1 No storage space free for storing 5.2 Give error message This ends the use case

Table 6.1: Use Case #1 - Retrieve TILES Devices

<b># 2 - TDE</b>	Update TILES Device details
Will update an already registered TILES Device. Able to update its custom name	
<b>Primary Actor</b>	User
<b>Precondition</b>	TDE connected to TC. TD for user retrieved
<b>Trigger</b>	User click update TILES Device information
<b>User Action</b>	<b>System Action</b>
1. Input information to update	
2. Click update	3. Parse input data 4. Send parsed data to TC 5. TC Check if device exist for user 6. Update information
	7. Return success message
<b>Exceptional Paths</b>	
	3.1 No data given to update 3.2 Give error message This ends the use case
	5.1 Device not found for user 5.2 Give error message This ends the use case

Table 6.2: Use Case #2 - Update TILES Devices information

<b># 3 - TDE</b>	Create Textual Application
Describes the process to start creating application for TILES textually. TILES Devices needs to be retrieved to be able to reference them in-code	
<b>Primary Actor</b>	User
<b>Precondition</b>	TD for user retrieved DSL available
<b>Trigger</b>	Open textual editor
<b>User Action</b>	<b>System Action</b>
1. Open textual text-editor	2. Load ACE editor 3. Load DSL library
4. Input program code	5. Highlight DSL syntax 6. Highlight JS syntax
7. Exit text-editor	
<b>Exceptional Paths</b>	
	2.1 DSL library not found 3.2 Give error message This ends the use case

Table 6.3: Use Case #3 - Create TILES application textually

<b># 4 - TDE</b>	Create Visual Application
Describes the process to start creating application for TILES visually. TILES Devices needs to be retrieved to be able to reference them in-code	
<b>Primary Actor</b>	User
<b>Precondition</b>	TD for user retrieved DSL available
<b>Trigger</b>	Open visual editor
<b>User Action</b>	<b>System Action</b>
1. Open visual editor	2. Load Blockly editor 3. Load custom blocks
4. Move and combine visual blocks	
5. Convert blocks to JS code	6. Generated code 7. Check code for endless loop
8. Exit visual editor	
<b>Exceptional Paths</b>	
	6.1 Couldn't generate and verify code 6.2 Give error message This ends the use case
	7.1 Code includes endless loop 7.2 Give error message This ends the use case

Table 6.4: Use Case #4 - Create TILES application visually

<b># 5 - TDE</b>	Restore Code
User want to restore previous work and continue working where they left of, or to add new functionality to already created application	
<b>Primary Actor</b>	User
<b>Precondition</b>	TDE connected to TC
<b>Trigger</b>	Restore button pressed
<b>User Action</b>	<b>System Action</b>
1. Input TILES device that registered with an application that should be restored	2. Query TC to retrieve application 3. Check if device is registered with an application
	4. Return application code 5. Open editor with retrieved code
<b>Exceptional Paths</b>	
	3.1 No application found for device 3.2 Give error message This ends the use case

Table 6.5: Use Case #5 - Restore TILES application

### 6.3 DSL

This section will describe the abstract functionalities that are required by the DSL, to successfully create applications for TILES. The functionalities described are derived from the user stories in table 5.2.

Sending commands to a TILES device requires the user to choose which command to send, and which device that should retrieve the created command. To fulfill *US6*, referring to a device could be done by its stored name. *US2* requires all commands to be public available and shown in the user interface, which could be done in a callable function or Array with its input name. A text-editor could be pre-configured to support intelligent code completion, an auto-completion feature for the developer environment.

A command that can be interpreted by a TILES device consist of two elements:

- *name* - Entry storing the output interface which the command should



control. Output interfaces supported are shown in figure 3.2 "Output" column.

- *properties* - Extra properties to give more description on how the output interface should be controlled, given as an array with maximum of two elements.

The *name* element is a simple text string denoting the output interface which the command should control. Each output interface supports different patterns for controlling the interface, e.g. the LED can be turned on or off, be set to blink, and to light up a specific color. When a command is specified to be received by a TILES device to control its output interfaces, verification of the created command should be done by the DSL before any command is issued to a real TILES device. This is possible as the type of pattern supported by the output interface is given by the TILES device specification, which should be reflected in the DSL. These command structures and the different types supported by each output interface are more difficult to handle and remember by simple typing. Building in support for commands within the DSL, and giving them easier to remember names, makers and computer science students are not exposed to the structure of the commands, as the example below:

```
{name:"led", properties:["on","blue"]}
```

Being able to respond to incoming events created by a user interacting with a TILES device, requires a way to distinguish between the different events that can be triggered. Not only does a listener need to be setup (*US3*), it also needs to parse each incoming event to check that the event type matches the type of event which the maker/student have put down in the code (*US4*). An input event and a TILES device identification need to be given as input to the DSL to successfully check the parsed event. When an event is successfully parsed, the DSL needs to notify any listeners of the newly occurred event. By introducing the publish-subscribe pattern [3], the DSL can publish a message to a given channel, depending both on the parsed event and the TILES device that transmitted the event. Any subscriber for the channel will then be notified of the incoming message. The publisher is not able to know if there exist any subscribers on the channel, nor is it of any concern, so the message will be published either way. This means there's no coupling between the subscribers and publishers.

An event triggered on a TILES device and sent to the infrastructure of TILES

consist of two elements, denoting what kind of event was triggered on the device. The structure is the same as a command:

- *name* - the name of the interaction being triggered on the TILES device. Valid commands is shown in figure 3.2 "Input" column.
- *properties* - Extra properties, providing more description to the event that was triggered. The properties are sent as an array with a maximum size of two elements.

The *name* elements are simple text strings, denoting the interaction being triggered on the TILES device. The main interaction is described using the name element, and the properties array is a more descriptive notation of the interaction. When a TILES device is tapped, the main interaction will be "tap", whereas the properties will contain the string "double" or "single", depending on the type of interaction triggered. This is all handled by the TILES device itself, and the DSL only need to be concerned about parsing the incoming event correctly. Makers and computer science students are not expected to handle the textual form of an event, but rather the parsed data.

An example event received from a TILES device that need to be parsed before it can be used in an application:

```
1 {name: "tap", properties: ["double"]}
```

Using a user-created name as reference to a TILES device, instead of its MAC-address, is much easier to remember and write in code. A problem with using user-generated names as references is that these are changeable names, which could make an already created application stop working if the name of a device should be changed at any time. It's therefore important that the DSL has an inbuilt functionality for parsing from a TILES device's name to its MAC address, and the other way around. This should prevent name-changing having any impact on already created applications. The name of the TILES is only used when the application code is shown to the makers and computer science students through the DSL. In the background, the custom-name is replaced with the device's MAC address, but never shown in the user-interface.

The DSL should support retrieving or sending data to a third-party service over HTTP. When the DSL are to conduct a HTTP-call, this communication should be done asynchronous, to prevent locking down of the system while

waiting for the HTTP-result. This requires a callable function to be run when the result of the HTTP-request is retrieved. The user will most likely want to use the requested data, and should be given the option to input some logic that should be run when the result is retrieved. Using the retrieved data requires the user to correctly implement a custom data-parser on the retrieved result. Incorrect implementation of a data-parser could lead to an unsuccessful call to data that do not exists. As this step requires more deep knowledge about handling third-party-services data, whether the result includes HTTP, JSON or XML-data, providing libraries for services that are more likely to be used, should help the users to prevent making calls that may crash the application.

Accessing the stored data in the TILES infrastructure is required to know which TILES devices are available in the running context. The DSL should be able to retrieve this data, from the maker/student, and use this information for fetching all TILES devices registered. Other stored data may also be of interest for the DSL to incorporate into the TILES applications, and should be made available in a user-context.

An interaction pattern consists of a TILES device identification and an event. The DSL should support interaction patterns, to be able to listen for different events from different devices. In code, there should be an ability to input a set of TILES device identification names, along with an event, and the DSL should tell when the given set of events is triggered by the given devices. The pattern should only be recognized when the exact order of events is triggered on each device. Supporting interaction patterns will give makers and computer science students more opportunities for interacting with their devices, and introduces a more complex usage scenario, when both single and multiple events can be interpreted by the DSL.

Since any incoming event from a TILES device will notify the DSL, following the publish-subscribe pattern, handling a set of events need to be built on top of this pattern. The Pipe-and-Filter pattern [3] is able to transform stream of data to a single output. When an interaction pattern is provided in code, each element need to be setup as a filter mechanism, where an incoming real-time event is transmitting through the "pipe" of filters. At each filter, the event is processed, and if the filter accepts the incoming event, deliver the event to the next filter, and set the status of the filter to "PROCESSED". Any "PROCESSED" filter should just pass an event to the next filter. If the event does not match the filtering process, already "PROCESSED" filters should reset their status, meaning the provided pattern did not match with incoming events. If an event is transmitted through the whole pipe of filters,

the interaction pattern is successfully carried out, and the DSL should be notified. Figure 6.2 shows the outline of a pipe-and-filter mechanisms, where the filter represent the provided interaction pattern, and event as parsed events received from all TILES devices registered to the maker/student.

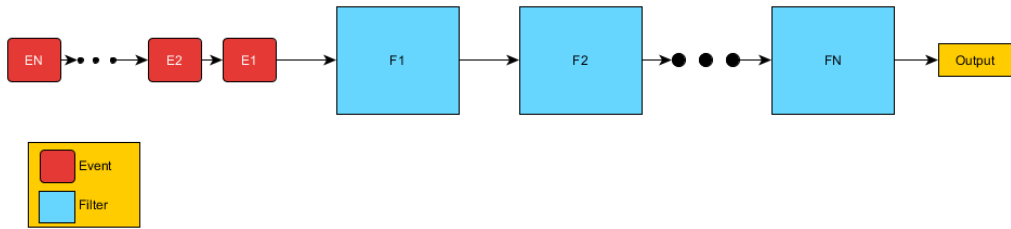


Figure 6.2: Pipe-and-Filter pattern for events

The mapping of the user stories is meant to show how the user stories should be prioritized, under implementation, and sub-tasks that needs to be completed for the main user story to be completed. The mapping is shown in figure 6.3.

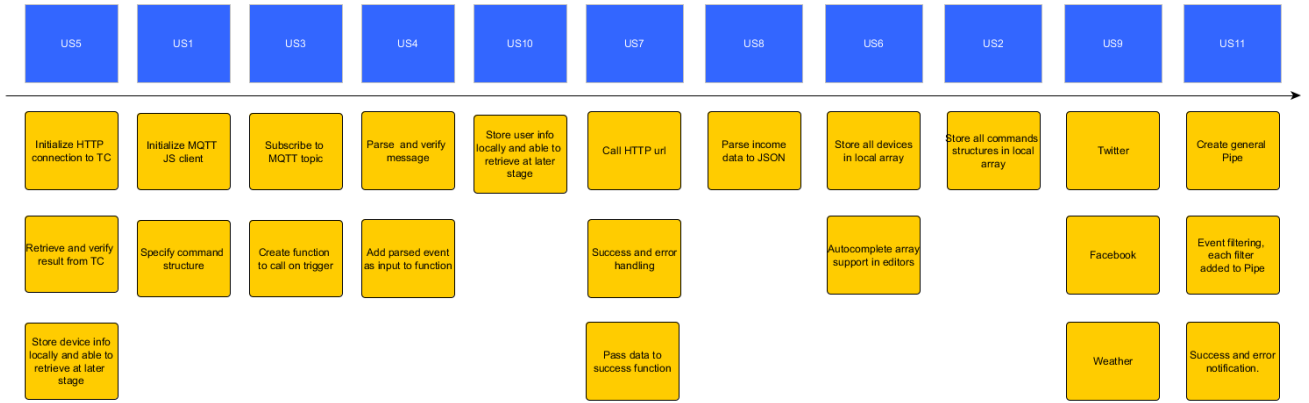


Figure 6.3: User Story mapping

### 6.3.1 Modules

The user stories described in table 5.2 for the DSL outlines important functionality which the DSL should be able to interpret. Modules can be created for the DSL by dividing the user-stories into groups by what services they provide. Following the module design pattern [3] helps to separate functionality provided by the DSL which enhances modifiability, portability and reuse. As each module can be developed and maintained independent of each other, the modules have no dependency of another. Each module can therefore be changed without requiring other modules to be affected, which enhances modifiability. Portability refers to how easily a module can be transferred to and used in other environments. Following the module pattern for the DSL should minimize the module's platform dependency. As the DSL will be part of the TILES toolkit, its modules will also be developed for TILES, but not enclosed to a specific environment. The different modules that the DSL should consist of is:

- *User* - Accessing user data, like TILES devices and username
- *Event* - Parsing incoming events from a TILES device, and able to setup listeners
- *Command* - Creating commands that should be sent to a given TILES
- *Data-source*
  - *Weather* - Retrieving weather data from an Internet API
  - *Facebook* - Rretrieving and posting data using the Facebook API
  - *Twitter* - Rretrieving and posting data using the Twitter API
  - *CustomHTTP* - Communicating with custom third-party services

The DSL can at a later stage be extended with other modules, without affecting already created modules. As the DSL is part of a layer-pattern, as shown in figure 5.1, the DSL also need to implement an interface that combines all the modules for the DSL. This unified interface, called a facade, is made available to the top-layers to help use the underlying modules of the DSL. This interface helps making the DSL modules more readable, as more convenient methods can be created for the underlying modules. Figure 6.4 shows an outline of the different modules which the DSL should support, and how these modules are available to the interface.

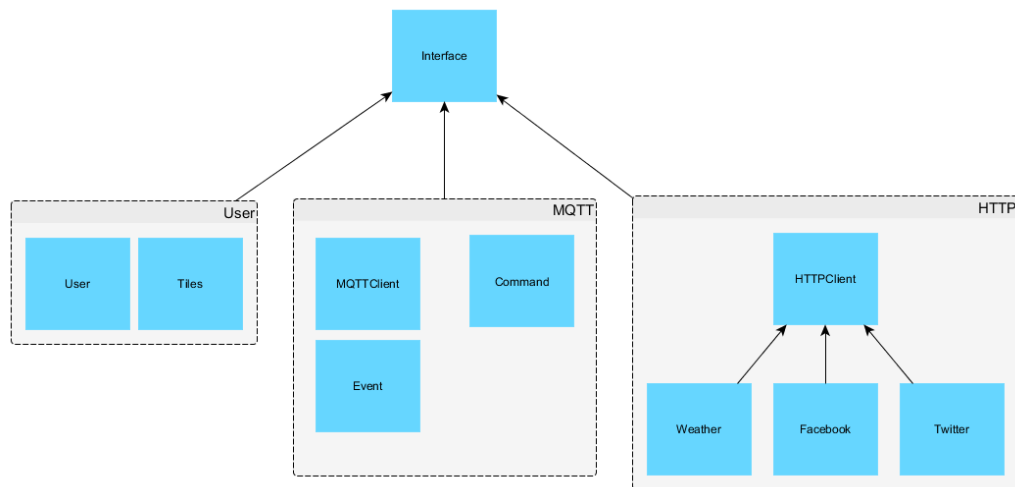


Figure 6.4: Modules for the DSL

A module can consist of several sub-modules, together forming the whole module, e.g. the HTTP module. The HTTP module consist of Internet-service modules. The interface for the DSL need to make all sub-modules accessible.

### 6.3.2 The language

A DSL for TILES should provide multiple abstract functionalities through a simple language to help ease the communication with TILES devices and other third-party services. A DSL should help makers and computer science students to code less, and still do more. Less coding required to accomplish bigger tasks will help ease the process of creating applications for TILES, and provides makers and computer science students the full power over their TILES devices with ease. Using the abstract DSL, quick prototyping using TILES devices is possible, and applications can be written with just a few line of code or visual elements.

The modules described for the DSL are to be made accessible through the facade interface. This is the interface that is callable by the DSL. A language is to be built on top of this interface, in which makers and computer science students will call indirectly through the DSL. The facade interface provide the methods callable by the language used for writing code to create applications for TILES.

The DSL will be built on top of JavaScript, making the DSL a sub-language of JavaScript. Providing the capability of mixing both the DSL and plain JavaScript code in the application code for TILES applications, should help makers and computer science students already familiar with JavaScript to more easily create applications for TILES. The DSL will help in abstracting TILES-specific code, resulting in fewer lines of code required to write, then it would to use a pure JavaScript approach. As the language for the DSL represent a higher level of JavaScript code, the DSL should be compiled down to pure JavaScript code, which the TILES infrastructure is able to run. A compiler is therefore required for this transformation, which takes a sequence of characters and translates them to a lower-level of instructions.

The Visual Programming Language should provide the same features as the DSL. By building the VPL on top of the DSL, the visual language is able to call the abstract functions using a block approach. By customizing Blockly, the blocks can represent the abstract functionalities which the Domain-Specific Language provides.

Designing the DSL requires a formal definition of its syntax. This is also required by a compiler, so that the compiler is able to distinguish between valid and invalid code. The grammar for the DSL consist of the syntax defined for each DSL module (User, Event, Command and Data-Source), and will be defined using the *Extended Backus-Naur Form*, a method of formally defining context-free grammars. The EBNF will help in explaining how the DSL code will be interpreted by a compiler. The compiler will first divide each sequence of characters into tokens, which will be used by a parser for generating an *Abstract Syntax Tree*, and use the AST to generated JavaScript code [35]. All valid tokens are defined in the grammar of the language, as visualized by the EBNF representation. A formal definition of the DSL, along with a set of usage examples, should also help with the implementation of the language. The textual representation of the valid syntax for the language is shown below.

```
1
2 "Name"      = 'TILEDSDL'
3 "Author"    = 'Jonas Kirkemyr'
4 "Version"   = '0.1.0'
5 "About"     = 'TILEDSDL for creating applications for TILES'
6
7 "Start Symbol" = <Program>
8
9 ! DERIVED FROM JS LANGUAGE
10 ! ----- Sets
11
```

```

12 {ID Head}      = {Letter} + [_] + [$]
13 {ID Tail}     = {Alphanumeric} + [_] + [$]
14 {String Chars1} = {Printable} + {HT} - ["\]
15 {String Chars2} = {Printable} + {HT} - [\']
16 {Hex Digit}   = {Digit} + [ABCDEF] + [abcdef]
17 {RegExp Chars} = {Letter}+{Digit}+['^']+['$']+['*']+['+']+['?']
    +['{']+['}']+['|']+['-']+['.']+['.']+['#']+['['']+[']']+
    +['_']+['<']+['>']
18 {Non Terminator} = {String Chars1} - {CR} - {LF}
19 {Non Zero Digits}={Digit}-[0]
20
21 ! ----- Terminals
22 !Rules
23 Identifier     = {ID Head}{ID Tail}*
24 StringLiteral = '"' ( {String Chars1} | '\\\' {Printable} )* '
    "' | '\'' ( {String Chars2} | '\\\' {Printable} )* '\
25
26 HexIntegerLiteral = '0x' {Hex Digit}+
27
28 RegExp        = '/' ({RegExp Chars} | '\\\' {Non Terminator})
    + '/' ( 'g' | 'i' | 'm' ) *
29 DecimalLiteral= {Non Zero Digits}+ '.' {Digit}* ('e' | 'E' )
    {Non Zero Digits}+ {Digit}* | {Non Zero Digits}+ '.' {
    Digit}* | '0' '.' {Digit}+ ('e' | 'E' ) {Non Zero Digits}+
    {Digit}* | {Non Zero Digits}+ {Digit}* | '0' | '0' '.' {
    Digit}+
30
31 Comment Start = '/*'
32 Comment End   = '*/'
33 Comment Line  = '//
34 ! END DERIVED FROM JS LANGUAGE
35
36 OpenBrackets  = ('THEN' | '{' )
37 CloseBrackets = ('END' | '}' )
38
39
40 ! Grammar
41
42 <DsOption> ::= 'APIKEY' '=' StringLiteral
43 <TwitterOption> ::= 'CONSUMER_KEY' '=' StringLiteral
44 | 'CONSUMER_SECRET' '=' StringLiteral
45 | 'TOKEN' '=' StringLiteral
46 | 'TOKEN_SECRET' '=' StringLiteral
47
48
49 <TileId> ::= 'TILE' '.' identifier
50
51 <Equal> ::= Identifier '=' StringLiteral
52 <ArrayAccessor> ::= '[' Number ']' | '[' StringLiteral ']'

```



```

53 <CommandInput> ::= '(' StringLiteral ')'
54
55
56 <Setup> ::= 'SETUP' '(' ')' OpenBrackets <Options>
      CloseBrackets
57 <Main> ::= 'MAIN' '(' ')' OpenBrackets <Code> CloseBrackets
58
59
60 <Statements> ::= <Statement> | <Statements> <Statement>
61 <Statement> ::= <IfStatement> | <RepeatStatement> | <
      SyncStatement> | <IfSimple>
62
63 <Options> ::= <Option> | <Options> <Option>
64 <Option> ::= 'FACEBOOK' '.' 'OPTIONS' '.' <DsOption>
65 | 'TWITTER' '.' 'OPTIONS' '.' <TwitterOption>
66 | 'WEATHER' '.' 'OPTIONS' '.' <DsOption>
67 | 'CUSTOMHTTP' '.' 'OPTIONS' '.' <Equal>
68
69
70 <VariableList> ::= 'FACEBOOK' '.' 'FEED'
71 | 'FACEBOOK' '.' 'PLACES'
72
73 | 'TWITTER' '.' 'FOLLOWERS'
74 | 'TWITTER' '.' 'FOLLOWING'
75
76 | 'WEATHER' '.' 'CURRENT'
77 | 'WEATHER' '.' 'FORECAST'
78 | 'WEATHER' '.' 'HISTORY'
79 | 'CUSTOMHTTP' '.' 'EVENT' '.' <Equal> ! OR CommandInput
80 | 'ME'
81 | 'ME' '.' 'NAME'
82 | 'ME' '.' 'ID'
83 | <TileId>
84 | 'TILES' ! Access stored TILES
85 | 'RANDOM' '(' <VariableList> ')
86
87
88
89 <Events> ::= <Event> | <Events> <Event>
90 <Event> ::= 'TWITTER' '.' 'LIVE' '(' StringLiteral ')'
91 | <TileId> '.' 'TAPPED'
92 | <TileId> '.' 'TAPPED' '.' 'SINGLE'
93 | <TileId> '.' 'TAPPED' '.' 'DOUBLE'
94 | <TileId> '.' 'TAPPED' '.' 'HOLD'
95
96 | <TileId> '.' 'TILTED'
97 | <TileId> '.' 'TILTED' '.' 'LEFT'
98 | <TileId> '.' 'TILTED' '.' 'RIGHT'
99 | <TileId> '.' 'TILTED' '.' 'UPDOWN'

```

```

100
101
102 | <TileId> '.' 'ROTATED'
103 | <TileId> '.' 'ROTATED' '.' 'CLOCK'
104 | <TileId> '.' 'ROTATED' '.' 'COUNTER'
105
106 | <TileId> '.' 'SHAKED'
107 | <TileId> '.' 'SHAKED' '.' 'HORIZONTALLY'
108 | <TileId> '.' 'SHAKED' '.' 'VERTICALLY'
109
110 | <TileId> '.' 'SHIFTED'
111 | <TileId> '.' 'SHIFTED' '.' 'LIFT'
112 | <TileId> '.' 'SHIFTED' '.' 'FREEFALL'
113
114 <Commands> ::= <Command> ';' | <Commands> <Command> ';'
115 <Command> ::= 'FACEBOOK' '.' 'POST' <CommandInput>
116 | 'TWITTER' '.' 'TWEET' <CommandInput>
117 | 'CUSTOMHTTP' '.' 'COMMAND' <CommandInput>
118
119 | <TileId> '.' 'LED'
120 | <TileId> '.' 'LED' '.' Identifier ! RED GREEN BLUE WHITE
    etc
121 | <TileId> '.' 'BLINK'
122 | <TileId> '.' 'BLINK' '.' Identifier
123 | <TileId> '.' 'PLAY'
124 | <TileId> '.' 'STOP'
125 | <TileId> '.' 'PAUSE'
126 | <TileId> '.' 'VIBRATE'
127
128 <Code> ::= <Body> | <Code> <Body>
129 <Body> ::= <Commands> | <Statements> | <VariableStatement> |
    <MemberAccess>
130
131 <IfStatement> ::= 'IF' '(' <Event> ')' OpenBrackets <Code>
    > CloseBrackets
132 <IfSimple> ::= 'IF' <Event> <Command>
133 <RepeatStatement> ::= 'REPEAT' '(' DecimalLiteral ')'
    OpenBrackets <Code> CloseBrackets
134 <SyncStatement> ::= 'SYNC' '(' <Events> ')' OpenBrackets <
    Code> CloseBrackets
135 <VariableStatement> ::= 'VAR' Identifier '=' <VariableList> '
    ;'
136
137 <MemberAccess> ::= Identifier | <MemberAccess> <ArrayAccessor>
    > | <MemberAccess> '.' Identifier
138
139 <Program> ::= <Main> | <Setup> <Main>

```

Listing 6.1: EBNF representation of TILES DSL

By defining all valid functionalities in the DSL using an EBNF notation, which later can be used for creating a compiler, makers and computer science students are able to be notified under the compilation process if any errors are found in the code, syntax wise. From a developer perspective, who is implementing the DSL functionalities, the functionalities need to be defined both in an EBNF notation, along with the actual implementation, and keep both updated. This is a concern for modifiability, as an update requires both modules and compiler to be updated. From a maker and computer science student perspective however, catching errors before any code is run in the TILES-infrastructure is of greater importance, as an error will not be directly visualized, as they don't have access to the run-time environment of the TILES-infrastructure directly.

The EBNF notation defines an initial layout of an application for the TILES DSL to be valid syntax wise. A visual representation of this EBNF notation is shown in figure 6.5.

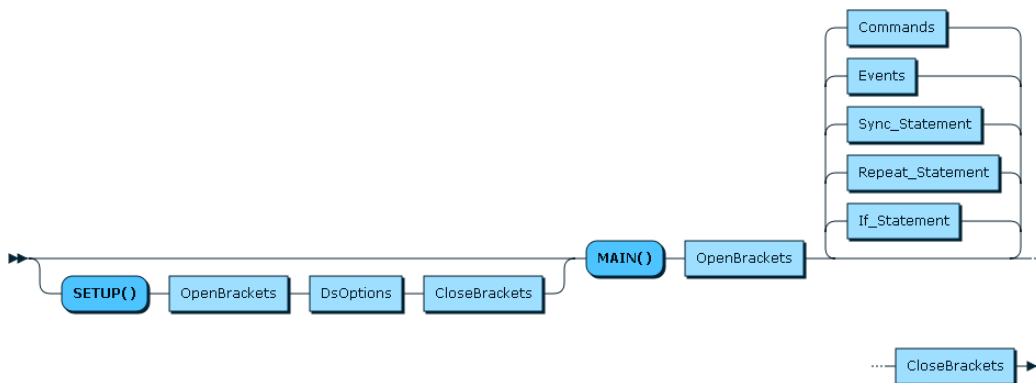


Figure 6.5: TILES DSL - EBNF start

Below is the textual form of the start EBNF notation, where *setup()* is optional

```

1  setup() THEN
2    <options>
3  END
4
5  main() THEN
6    <insert code>
7  END

```

Listing 6.2: TILES DSL - EBNF Start

The visual representation of the EBNF language can be found in appendix B

### 6.3.3 Example Application - Whack a Mole

To help describe how both the DSL and VPL can be used when implemented, an example application for both approaches is given using the DSL as an example. The example application described *Whack A Mole* is a simple game that expect the user to ‘Whack A Mole’ when it appears for it to disappear again. Translated for TILES usage, the user should tap on the TILES device whenever the LED is turned on. The LED should be off, and another random device should light up its LED again.

Here the DSL have made Tiles available in variables called ‘Tiles’, and references to specific TILES devices using its custom-name ‘Alpha’.

```

1 main() THEN
2 var tile=Random(Tiles);
3 tile.led.green;
4 if(tile.tapped.single) then tile.led.off end
5 END

```

Listing 6.3: ‘Whack A Mole’ - textual example

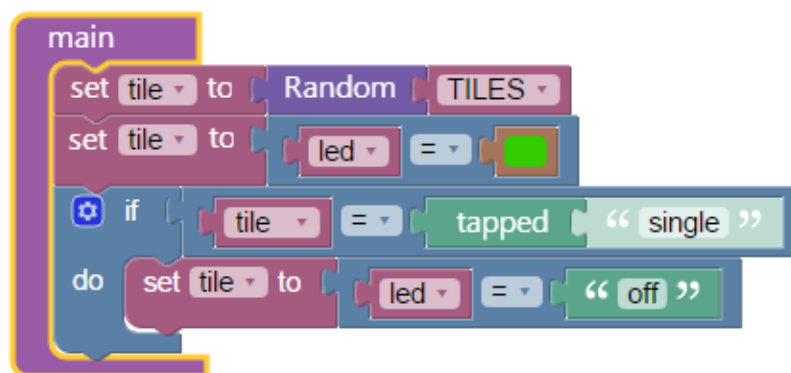


Figure 6.6: Whack A Mole - VPL example

## 6.4 TDE

The structure of the TDE should combine separate components to successfully support the process of developing applications for TILES. The components consist of the DSL, a VPL, an IDE, called ACE, and a TILES infrastructure client for communicating with the TILES infrastructure. The infrastructure client is required for uploading and retrieving applications created by the makers and computer science students. Makers and computer science students using the TDE are able to act on both the textual- and visual-editor from the user-interface of the TDE.

## 6.5 Emulator

Being able to test the created application once it's published to the production environment, makers and computer science students are able to check whether the application work as intended or not. Using an emulator to act as a real TILES device requires communicating with the emulator for both sending events and receiving commands. From a maker and computer science student's perspective, the emulator behaves exactly like a real device. An emulator would make the whole development process available in the same environment, and further benefit the development process. Introducing testing within the environment should help to ease the development process, since the emulator can emulate interactions with multiple emulated devices.

The emulator has to support the same message structures as the TILES infrastructure, as it will act as another connected client to the TILES Cloud. Only communication over MQTT is needed however, and requires to be connected as the current maker/student, with a user-specified name. The developed application code will be running in the TILES infrastructure as well, but instead of using physical TILES devices, an emulator version of a physical TILES device could be used.

An example layout of the emulator is given in figure 6.7.

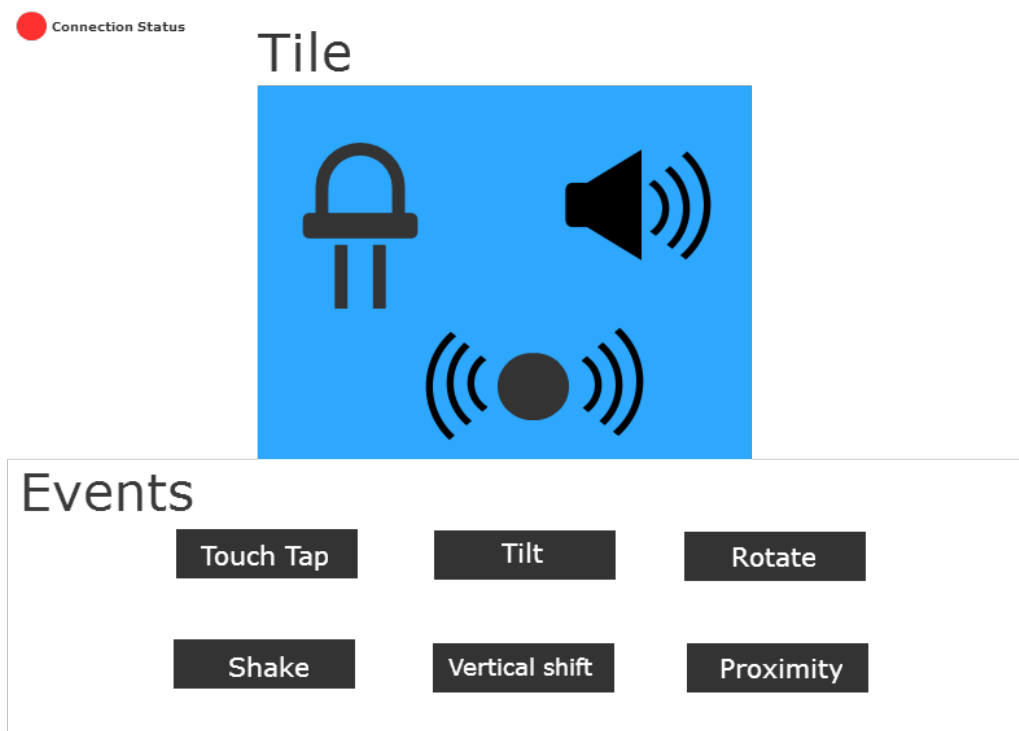


Figure 6.7: TILES Emulator

# Chapter 7

## Implementation

The toolkit to create applications for TILES consist of three software components: The DSL and its modules, a TILES Device emulator, and the TILES Development Environment. How these software components are connected are shown in figure 7.1. This chapter describes the implementation of each component, and a description on how to use these components together to create applications for TILES, from a maker and computer science student’s perspective.

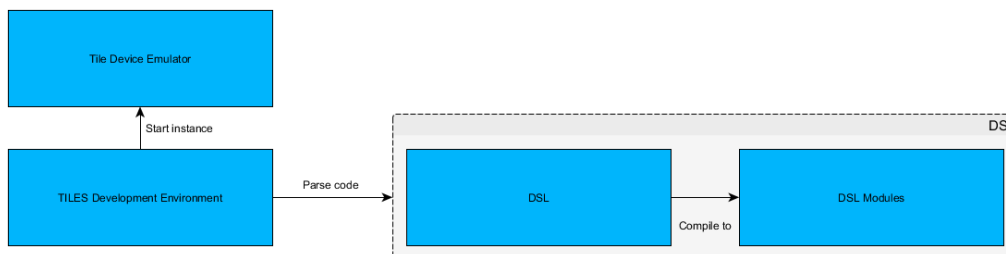


Figure 7.1: TILES Software Components

The toolkit were developed by using an agile approach, by building the toolkit following its requirements, in table 5.1, and the user stories, in table 5.2 incrementally. For the DSL, the user story mapping was used 6.3 to implemented more prioritized features of the language. Each component of the system was implemented independent of each other, and for each requirement and user story implemented, the new feature was tested, along with the component it resided within. As each component was completed, the toolkit was merged and tested as a full-system, making sure the toolkit is able to interact with

each component. By following an agile approach, new features and requirements can easily be added throughout the implementation process. This was an important feature, to provide continuous improvements of the toolkit, and being able to more easily respond to change. Throughout the development process, Trello<sup>1</sup> was used, to organize and keep track of the progress. Each requirement and user story were broken up in smaller tasks for easier implementation, and helped with finding common functionality across the components.

## 7.1 DSL

The DSL component consist of modules, described in section 6.3.1, and a language parser and compiler. Code written in the DSL is compiled to JavaScript, which use the implemented modules for creating the application logic.

### 7.1.1 Modules

The DSL modules are implemented using TypeScript<sup>2</sup>, a superset of JavaScript. The modules are designed to facilitate creation of the application logic for a TILES application, and helps with abstracting the communication with TILES devices, and other third-party services.

The TileDSL class is the main class, which combine and act as a facade for all of the DSL modules. The class stores the current application code to be run, user details and its TILES devices, and provides methods for communicating with third-party services and the user's TILES devices. Upon creation of the class, user details are retrieved from the database storage, and stored in local variables for the current object. A MQTT connection is also required for subscribing to *events* and publishing *commands* to the user's TILES devices. The TileDSL class handles communication with the TILES devices through an implemented MQTT client, which is built on top of the *MQTT.js* library<sup>3</sup>. The MQTT client retrieves events triggered on a TILES device, and publish commands to control the TILES device's output interfaces. The TileDSL class is declared as an *abstract class*, which prevents

---

<sup>1</sup><https://trello.com/>

<sup>2</sup><https://www.typescriptlang.org/>

<sup>3</sup><https://github.com/mqttjs/MQTT.js>



the class to be instantiated, and only be extended by another class through inheritance. The extended class will inherit all the properties of the TileDSL class, except for direct access to its private variables. This is an important property, as the TILES applications will be running in the scope of the inherited class, and won't have access to the parent's private variables or functions. Properties that should not be accessible by the application itself, are all declared as private in the parent class, which from a security perspective will prevent the makers and computer science students of writing code that may alter any configuration that is of importance for the application to run properly (e.g. database access, MQTT and HTTP communication, and user credentials). The parent class have control over the accessible functionality in the application scope, by declaring variables and functions with different access modifiers: *private*, *protected*, or *public*. A UML representation of the TileDSL class, and its extended UserDSL class is shown in figure 7.2.

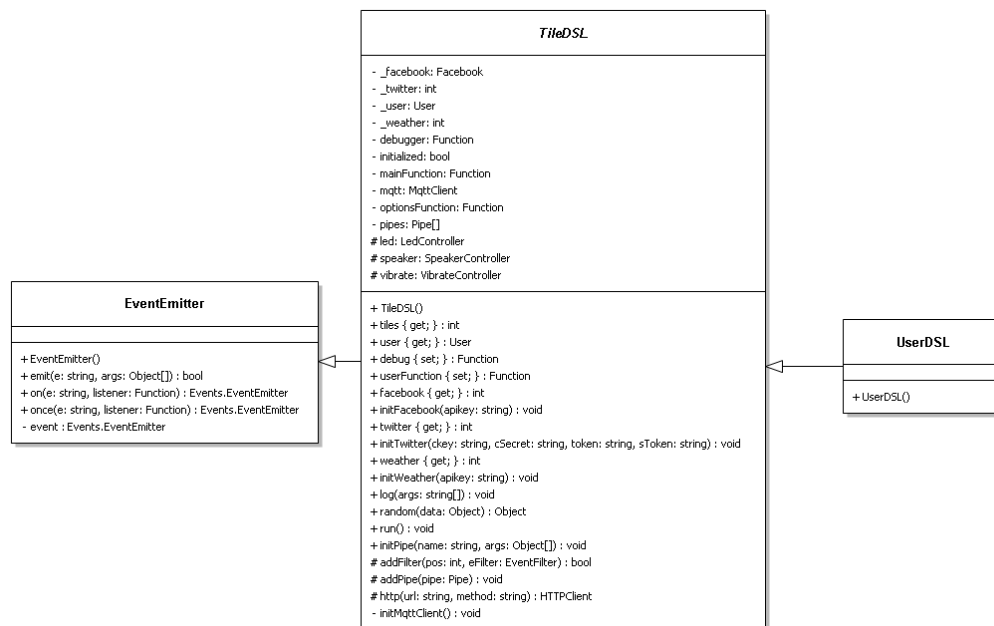


Figure 7.2: Class-diagram TileDSL and UserDSL

The EventEmitter class is an abstract publish/subscribe class, used for emitting messages on class objects. The class supports different callback functions to be run when an internal event is triggered on the object itself. The class consist of three methods used for subscribing and publishing of events, as shown in table 7.1 This class is extended by multiple classes, i.a. the TileDSL

class, to notify any listeners whenever the TILES application is ready to be invoked, after retrieving user details from database, and when the MQTT client is connected to the server.

Method	Description
<code>emit(event:string, data:Object[])</code>	Trigger internal event on object
<code>on(event:string, listener:Function)</code>	Listen for event
<code>once(event:string, listener:Function)</code>	Listen for event once

Table 7.1: EventEmitter methods

The `on` and `once` functions take a callback function as input, which is to be executed when an `emit` is done on the object, whenever the `event` string literal are matching. Using the `once` function, will make sure the listener function is only triggered once, by deleting the subscriber after the first call.

An example for starting a TILES application is shown in a code snippet in Listing 7.1. Line 4 set the application code to run when the `run()` function is called in line 6. `parse.code` is omitted for simplicity, but contains a string of calls to the internal methods of the `UserDSL` class. Line 5 specifies that the callback function should only be called once, whenever the `"ready"` event is triggered on the `UserDSL` class.

```

1 var dsl = new UserDSL();
2 dsl.debug = console.log;
3 try {
4   dsl.userFunction = new Function('output', parse.
      code);
5   dsl.once("ready", function () {
6     dsl.run();
7   });
8 }
9 catch (e) {
10  console.log(e);
11  process.exit(0);
12 }

```

Listing 7.1: Run TILES application

## User Module

The user module consists of the User class and the Tile class. Only an instance of the user class is stored and directly accessible within the TileDSL class, in the `_user` variable. When starting a TILES application, a user is identified by its username. The user class represents the maker/student running the TILES application. Its details and registered TILES devices are retrieved and stored within a user class instantiation. An UML representation of the User module is shown in figure 7.3. The interfaces defined: *iUser* and *iTile*, represent the same structure as the data stored in the database.

The Tile class extends the EventEmitter class, making it possible to emit messages directly on the Tile object, and create listeners on the Tile objects. This could be utilized by the Event module, to emit an event whenever a new event is received from the physical TILES device.

## Event Module

The Event module handle parsing of incoming events received from a TILES device. The parsing make sure only valid events are processed by the different modules. The event parser are built from the TILES documentation, which determines the type of parameters that go along with an event name, as shown in appendix D. The parser is able to determine the type of event, and which TILES device triggered the event. The parsing process is a simple logical comparison, as an event name only supports some fixed parameters. The supported events are defined in a `const` variable, as shown in Listing 7.2.

```
1 const EVENTS = {  
2   TAP: {  
3     SINGLE: "single",  
4     DOUBLE: "double",  
5     TOUCH_HOLD: "touch_hold"  
6   }  
7   ...  
8 }
```

Listing 7.2: Event const used for parsing

A UML class diagram representation of the Event module is shown in figure 7.4.

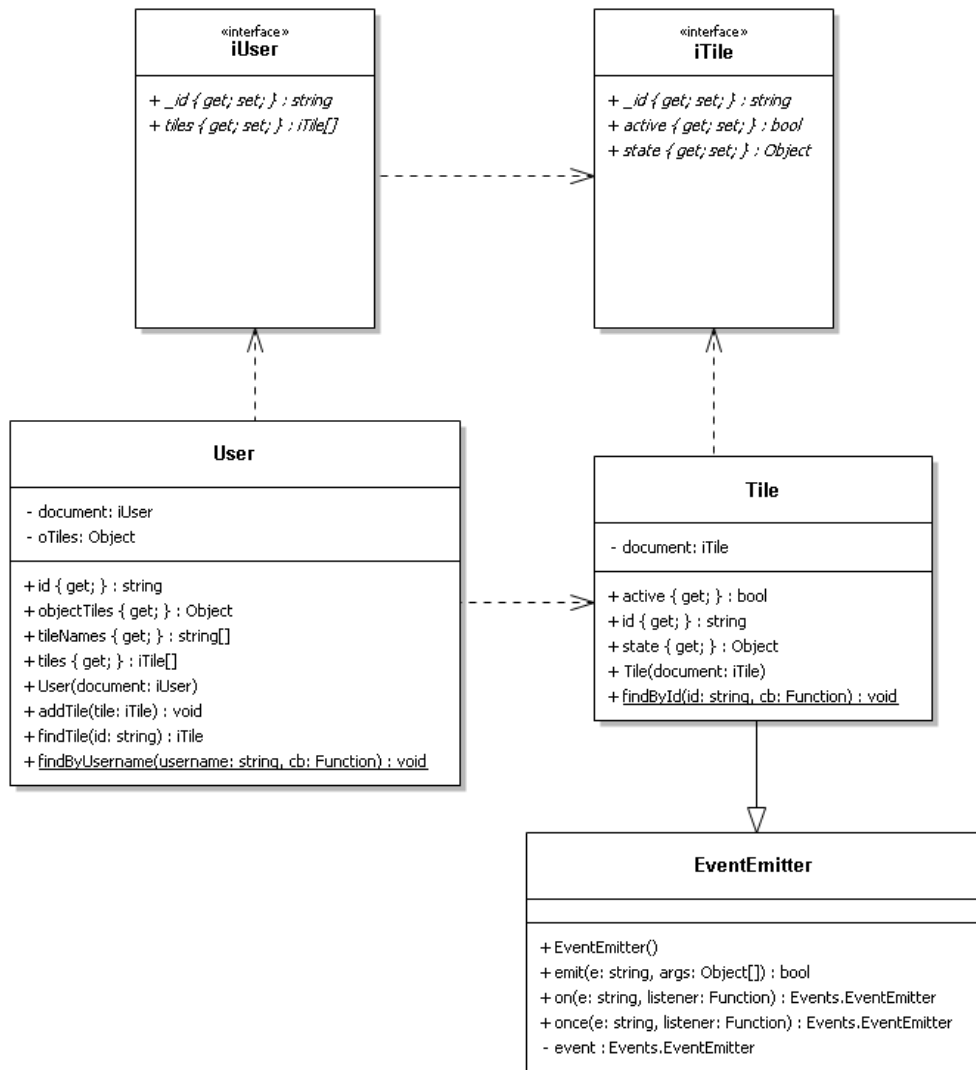


Figure 7.3: Class-diagram User module

The static `parse` function declared in the `Event` class is used to parse an object representation, and check whether that object representation is a valid TILES event. A successful parsing will return a new instance of the `Event` class containing the event data, otherwise a `null` pointer is returned.

The `Pipe` class and `iFilter` interface is implemented following the Pipe-and-Filter pattern, as describe in section 6.3. Multiple filters can be added to the pipe class and stored in an array, using the `add(...)` method. Traversing

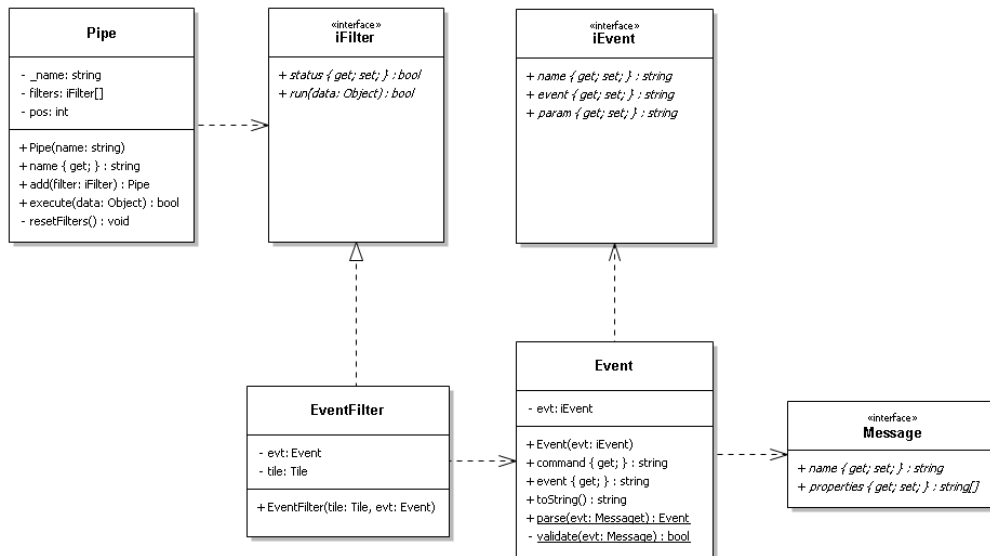


Figure 7.4: Class-diagram Event module

through the added filters is done in the order they were added, and each filter is assumed to process the same data. The `execute(...)` function only processes a filter at a time, and a filter can only be executed as long its prior filters has its status set to `true`. The filters are designed to be independent of each other, meaning they could all be run in parallel and process the input data. However, the pipe class requires all prior filters to be successfully processed before the next can be run, preventing unnecessary computing time if any prior filters should fail. If one filter should fail, all prior filters are reset and changes it's status to `false`, and need to be processed once again.

The `EventFilter` class is a specialized filter class, implementing the `iFilter` interface. The `EventFilter` takes an input `Tile` object and parsed event, which is used for checking whether a given TILES device have executed a given Event. Adding an `EventFilter` to the `Pipe` class is possible by *polymorphism*. The `Pipe` class is only concerned whether the input object of the `add(...)` function implements the `iFilter` interface. The `Pipe` class is therefore not restricted for a given use case, but can easily be extended by creating other filter classes that implements the `iFilter` interface as well.

## Command Module

The Command module is the module which generates commands to control a TILES device's output interface. Pre-defined `const` variables of valid commands that can be interpreted by a TILES device is included. These variables are accessible to the command controllers, and the `TileDSL` class itself. This helps makers and computer science students to control their device, by removing the burden of defining a command's structure from scratch. A pre-defined command which will turn the led on for a given TILES device is shown in listing 7.3.

```
1 const turnOn = new Command({  
2 type: "led",  
3 properties: ["on"]  
4 });
```

Listing 7.3: Led command

A UML class diagram representation of the Command module is shown in figure 7.5. Each output interface on the TILES device has its own controller, as shown in the class diagram. Each function in a controller generates a command, and are given a descriptive name, making it more easy to understand what type of command is generated by each function. The controllers act as a facade for the Command class, direct usage of the Command class would therefore never be required. A controller's task is to generate and send commands to a TILES device. Sending a command to a TILES device is done by using the `MQTTClient` class `sendCommand` function:

```
sendCommand(tile:Tile, command:Command);
```

The variable `tile` refer to the object instance of a TILES device which should receive the `command` instance.

The command module contains a parser, same as the Event module, built on the TILES documentation in appendix D. The different output interfaces have pre-defined valid parameters that can be used, which is built within the parser using simple logic validation. Parsing of commands is however less of a concern compared to the parsing of incoming events, as the commands are only available through the command controllers. Only the developer will have access to the command class, and is responsible for creating valid commands.

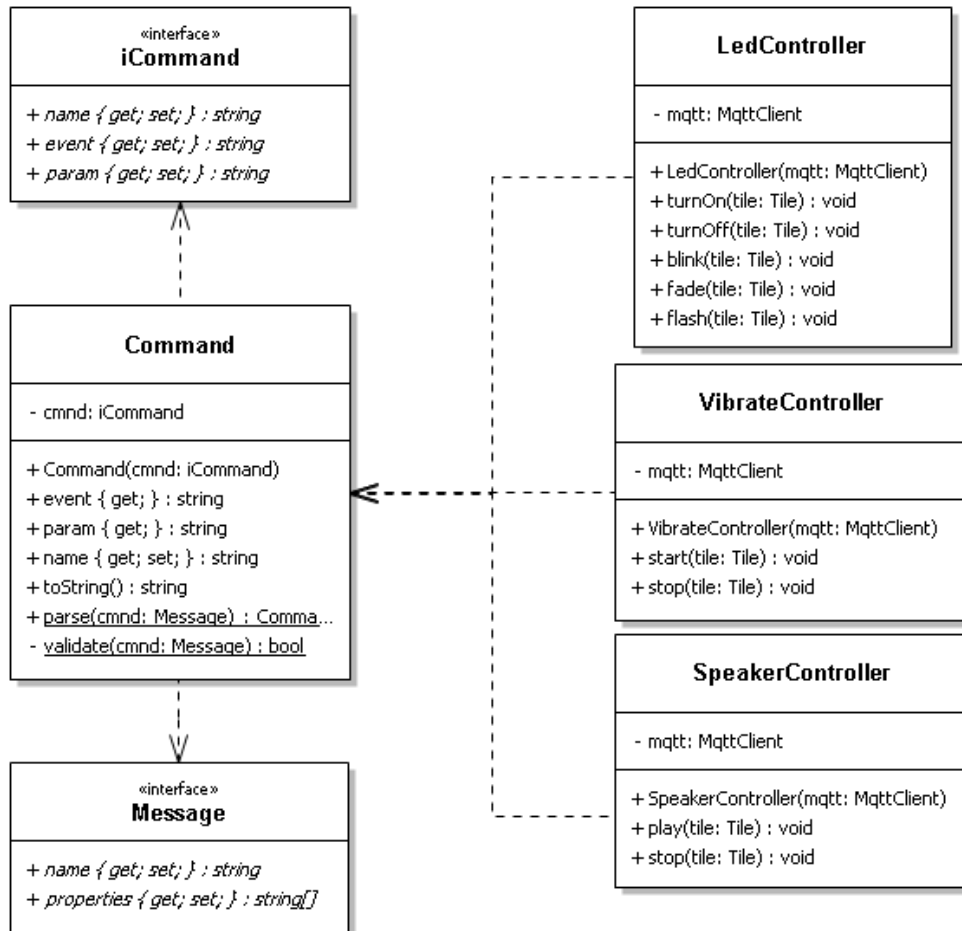


Figure 7.5: Class-diagram Command module

## Data-source Module

The Data-source module contains a HTTP-client for accessing third-party services. It includes a general HTTP-client, along with specialized classes for accessing more common third-party services, like weather, Facebook, and Twitter. The UML representation of the module is shown in figure 7.6. Each specialized HTTP class includes interfaces, representing the data-structure retrieved from its data-source. In the class diagram, the interfaces for Facebook and Twitter is abstracted away for simplicity, as they follow the same principle as shown by the interfaces connected to the Weather class. The





Calling a third-party service can be done through the standard HTTP methods: *POST*, *PUT*, *DELETE*, and *GET*. A simple example for storing some data to a third-party source is shown in listing 7.4. The `HTTPClient` constructor take two parameters, `url` and type of HTTP method. In this example, the HTTP request has a `username` header set, along with request data stored within the `data` key, as shown in line 2 and 3. Line 5 triggers the actual requests, and requires a callback function as an input to its method call, which will be invoked when the HTTP-request has got a response from its source.

```

1 var http = new HTTPClient("https://myurl", "post");
2 http.addHeader("username", "myusername");
3 http.addData("data", "store my data");
4
5 http.exec(function(data: JSON){
6     if(data.status==true){
7         //data stored successfully
8     }
9     else{
10        //couldn't store data
11    }
12 });

```

Listing 7.4: HTTP-Client example

### 7.1.2 Language

From the EBNF notation in listing 6.1, a parser is created, which will be generating AST's from textual code. The parser is implemented using the package *PEG.js*<sup>8</sup>. As the name indicates, the Parsing Expression Grammar, described by Bryan Ford [11], is the implemented parsing method in the PEG.js package. The PEG.js package is written in JavaScript, and is used, like EBNF, to describe a set of valid grammar rules in a language. Listing 7.5 and 7.6 show a comparison of an EBNF rule and the corresponding PEG.js rule respectively.

```

1 <IfStatement> ::= 'IF' '(' <Event> ')'
   OpenBrackets <Code> CloseBrackets

```

Listing 7.5: DSL IF grammar - EBNF

<sup>8</sup><http://pegjs.org/>

```

1 DSLIfStatement = 'IF' i '(' __ event:TileEvent __ ')',
   OpenBrackets body:SourceElements? CloseBrackets
2 {return {type:'DslIf', event:event, body:
   optionalList(body)}};

```

Listing 7.6: DSL IF grammar - PEG.js

A rule consists of a reference name, defined before the equality sign, and a rule definition, a combination of strings and references to other rules. The rule definition is assigned to the rule reference. PEG.js task is to create an object representation, called an AST, from some given programming code. A rule in PEG.js includes some JavaScript code, which specifies the structure of the output object. The rule itself is specified before the open bracket: {, as shown in listing 7.6 line 1. This is the rule which is compared to the input programming code. The JavaScript code within the brackets, shown in line 2, will be executed when a match for the rule is found. The rule can also consist of key-value pairs, denoted as `key:value`, where the `value` refers to a rule reference, and the `key` act as a variable which stores the object output of a rule reference. This `key` variable can be referenced in the JavaScript code defined within the brackets. For the PEG.js example given in listing 7.6 line 1, the key `event` will store the object output of the `TileEvent` rule, which follows the same pattern as the `DSLIfStatement` rule.

The following string will make the rule shown in listing 7.6 to be triggered:

```

1 if(TILE.alpha.tapped) {
2 }

```

Listing 7.7: DSL IF string

Which will generate the following object representation:

```

1 {
2   "type": "DslIf",
3   "event": {
4     "type": "TileEvent",
5     "event": "tap",
6     "tile": {
7       "type": "tile",
8       "name": "alpha"
9     },
10  "param": null

```

```

11   },
12   "body": []
13 }

```

Listing 7.8: DSL IF object representation

The implemented DSL language for TILES extends the JavaScript language. The DSL grammar is built on top of the JavaScript grammar, which is provided as an example in the PEG.js source code<sup>9</sup>. By building the DSL grammar on top of the JavaScript grammar, it's possible to parse plain JavaScript code along with the specified TILES DSL. Makers and computer science students will have the full power of the JavaScript language, along with the expressive DSL because of this extension. The PEG.js rule in listing 7.6 consist of more rule elements than the EBNF notation in listing 7.5. This is mainly because the JavaScript grammar follows a specification, called the ESTree Spec<sup>10</sup>, for how the structure of an output AST should be after parsing some given JavaScript code, and which object types and rules are supported. The rule reference `SourceElements`, shown in listing 7.6 line 1, refer to an ESTree specification, which denotes what code may be placed within the body of an `DSLIfStatement`. The PEG.js grammar rules for the DSL is shown in listing D.1 in appendix D. The JavaScript grammar rules are left out for simplicity.

The parsing process generates an AST, which needs to be compiled to a programming language. As the AST follows the ESTree spec, which is defined by the created grammar, the library *astring*<sup>11</sup> is used for code generation. The astring library is a library which is able to generate JavaScript code from an ESTree-compliant AST. As the TILES DSL specifies its own grammar rules, the astring library is extended, to support each AST type of the DSL. A list of the different DSL types is listed in table 7.2. The description includes a line number, referring to the PEG.js grammar definition in listing D.1.

Type	Description	Line
<code>MainStatement</code>	The application logic is implemented within this statement.	#167
<code>OptionStatement</code>	The setup block for setting up third-party services.	#160
<code>ConsumerSecret</code>	Option for Twitter	#66

<sup>9</sup><https://github.com/pegjs/pegjs>

<sup>10</sup><https://github.com/estree/estree>

<sup>11</sup><https://github.com/davidbonnet/astree>

<code>ConsumerKey</code>	Option for Twitter	#64
<code>Token</code>	Option for Twitter	#68
<code>TokenSecret</code>	Option for Twitter	#70
<code>FacebookAPI</code>	Option for Facebook	#74
<code>CustomHTTP</code>	Option for CustomHTTP	#80
<code>WeatherAPI</code>	Option for Weather	#78
<code>facebookvar</code>	Facebook variable	#39
<code>twittervar</code>	Twitter variable	#41
<code>weathervar</code>	Weather variable	#43
<code>mevar</code>	Current user variable	#45
<code>tile</code>	TILES variable for accessing TILES objects	#47
<code>tiles</code>	Variable holding an object representation of all TILES objects	#49
<code>DslIf</code>	If statement for a TILES event, triggering a block of code when given event is triggered	#135
<code>DslIfSimple</code>	If statement for TILES event, triggering a TILES command when given event is triggered	#137
<code>Repeat</code>	Repeat function to support simple looping	#139
<code>Sync</code>	Listening for a sequence of TILES events triggered in the order given. Executes a block of code when the sequence is matched	#141
<code>Random</code>	Returns a random value from a given input	#143
<code>Reset</code>	Re-start the application from the <code>MainStatement</code>	#145
<code>TileEvent</code>	Supported events for a TILES device used for listening	#99
<code>TileColor</code>	Supported LED-colors for the TILES device	#108
<code>TileCommand</code>	Supported commands for controlling the TILES device's output interfaces	#121
<code>DatasourceCommand</code>	Supported methods for accessing third-party services	#125

Table 7.2: DSL types

Each type listed in the table above, requires its own method in the code-generator, which defines what output to generate from an AST. The output generated will be complainant with the DSL modules described in section 7.1.1. The object in listing 7.8 refers to the `DslIf` type, and its generator method is shown in listing 7.9. The method takes two arguments: `node` and `state`. The node parameter is the part of the AST which matches the type of the generator: `DslIf` with the input object in 7.8. The state parameter is a data-stream class, handling the output string generation. Whenever `state.output.write` is called, the parameter string is appended to the data-stream. Within the generator method, a `this` reference is used, for calling other generator methods, as shown in line 4 and 10 in the listing below. These generators are not shown for simplicity, but follows the same pattern as the `DslIf` generator. The corresponding code output after running through the generator method for the `DslIf` type is shown in listing 7.10.

```

1 this._generator.DslIf = function (node, state) {
2   if (node.event !== null) {
3     if (node.event.tile !== null) {
4       this[node.event.tile.type](node.event.tile,
5       state);
6       state.output.write(".once(\"" + node.event.
7       event + "\", function(event){");
8       if (node.event.param !== null)
9         state.output.write("if(event==\"" + node.
10        event.param + "\\")")
11      }
12      if (node.body !== null) {
13        this["BlockStatement"](node, state);
14        Parser.computeLastBlockPosition(state);
15      }
16      state.output.write("});");
17    }
18  }
19 }

```

Listing 7.9: DSL IF generator method

```

1 this.tiles.alpha.once("tap", function(event){
2 });

```

Listing 7.10: DSL IF generator output

The code-generator for the DSL is written in TypeScript, and extends the astring library. An UML representation of the code-generator is shown in

figure 7.7, and referred to as the `Parser` class, as it parses an `ESTree` compliant and generates JavaScript from the tree-structure. The code-generator implements methods for each type described in table 7.2. Each type described is implemented as an interface, describing the AST structure of each type. Each interface extends the general `ESTree.Node`, as described in the `ESTree` specification<sup>12</sup>. The `Parser` class is storing the generated JavaScript in a local variable, and provides a method for storing the code-string to a local file. The logic of creating a infinite looping application is implemented within this class. The `Reset` type is automatically appended into the AST, within the last block of a DSL type, as long as the `Reset` type is not found in the parsed DSL code.

## 7.2 Development Environment

The development environment for TILES is a web-environment for developing applications for TILES. The environment supports creating applications for TILES by writing code, using the implemented DSL, in a text-editor ACE<sup>13</sup>. Providing the development-environment as a web-environment makes it possible for makers and computer science students to use the provided text-editor to write DSL code in their own web-browser. The parser of the DSL language is created using PEG.js, as described in section 7.1.2. This parser is implemented in JavaScript, which makes it possible to load the parser directly within the browser. This helps with checking if the written code is syntactically correct directly in the browser as its written, since the compilation process can be run within the development environment. The compiler will notify the environment if any syntax errors are found, with a descriptive message, and the location of the syntax error. A syntactically correct DSL code may be chosen to be uploaded to the TILES infrastructure, where it's compiled once more and executed immediately. Compilation on the client-side helps with the development process, by validating the code and provide with the opportunity to fix any syntax errors before its run in the TILES infrastructure. As a compilation of the DSL is done once more when it's received in the TILES infrastructure, the development environment is uploading the DSL code, and not the production-ready compiled code. Because of the DSL's expressivity, less data needs to be uploaded to the infrastructure, as the compliant compiled code of the written DSL code, contains more text and therefore more data. Storage and data-consumption wise, this is better

---

<sup>12</sup><https://github.com/estree/estree>

<sup>13</sup><https://ace.c9.io/>



later be retrieved and altered from any browser environment, which would not be possible by storing the written DSL locally. Makers and computer science students would not be bound to a fixed development environment, but able to continue with their work from any browser running the development environment.

The development environment is built using multiple tools and libraries. The development environment is a web-application, consisting of JavaScript, HTML, and CSS, built as a Single Page Application. A SPA loads all necessary resources required for running an application, and creates the content of the application dynamically as it's needed. The SPA support is done through a library<sup>14</sup> created by the author, written in TypeScript. The static HTML-files is built using Assemble<sup>15</sup>, supporting encapsulation of different elements of a web-page, like its layout and pages. Handlebars.js<sup>16</sup>, a semantic templating system, is used as the template engine for Assemble, for supporting the use of variables, looping, and other helper functions within HTML code. Assemble's task is to build these templates into static HTML files, ready to be served by a web-browser. Handlebars.js is also used for creating dynamic layouts for the development environment. Each dynamic layout is compiled to JavaScript, which can be loaded by the application logic. The application logic of the development environment is implemented using TypeScript, and a UML class-diagram of the application is shown in figure 7.8.

Three routes are implemented for the development environment, as seen in the UML diagram: Tile, Textual, and Visual. Each route loads its own controller for handling the application logic for its route. The textual route creates an instance of the `TextualEditorController`, which handles all logic of the textual-editor page. Figure 7.9 shows the dynamic created layout for the textual route. The ACE editor is loaded, and the `TextualEditorController` handles compilation, and communication with the TILES infrastructure through the implemented `TileApi` class.

Figure 7.10 shows the dynamic created layout for the visual route. The `VisualEditorController` creates a new Blockly instance, to control the visual editor. The controller is able to generate textual code from the visual representation residing within the Blockly instance.

Generating custom blocks requires a block-layout and a code-output to be defined according to the Blockly specification<sup>17</sup>. These definitions are stored

---

<sup>14</sup><https://github.com/jonaskirkemyr/spa>

<sup>15</sup><http://assemble.io/>

<sup>16</sup><http://handlebarsjs.com/>

<sup>17</sup><https://developers.google.com/blockly/guides/create-custom-blocks/>



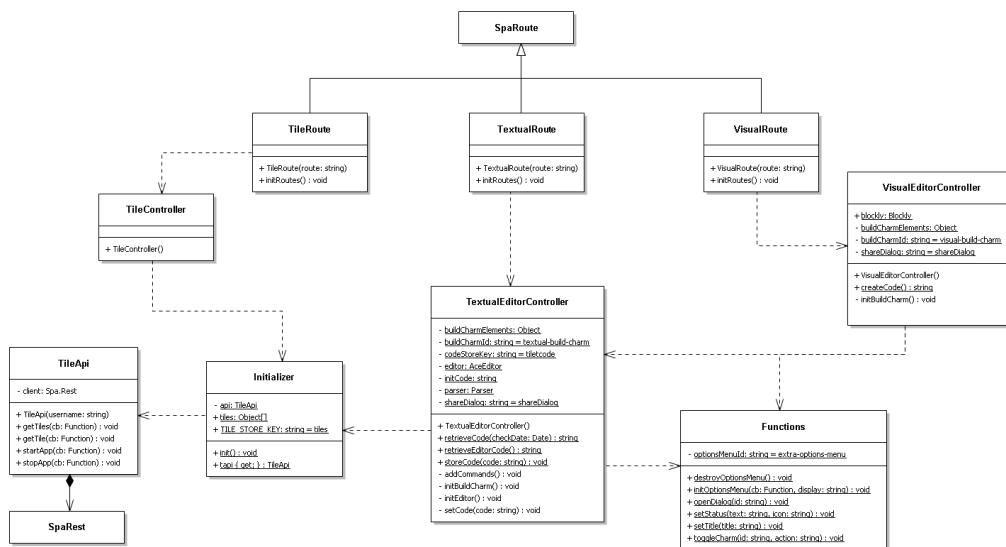


Figure 7.8: Class-diagram Development Environment

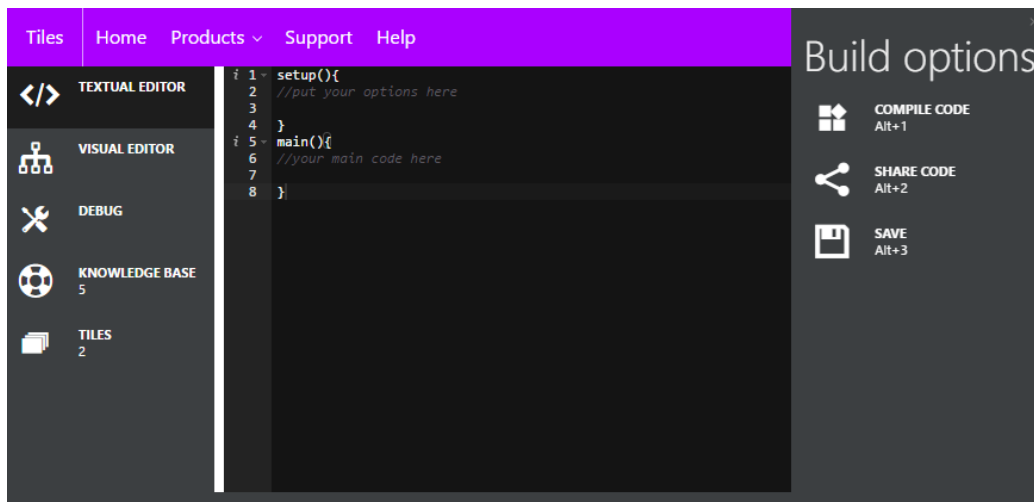


Figure 7.9: Development Environment - textual editor layout

in a JavaScript array, and loaded by the Blockly instance. The figure below also shows an example of a custom created block. Specifying which blocks to show in the visual editor is done using XML, and an example of loading a

overview

block is shown in listing 7.11. The development environment support creating visual code, but because of time-restrictions, visual representations of the implemented DSL is not created.

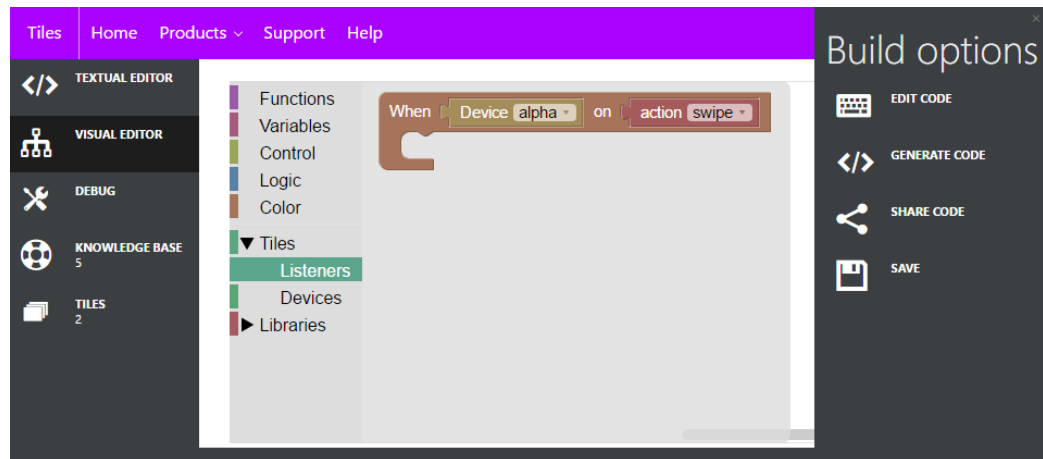


Figure 7.10: Development Environment - visual editor layout

```

1 <category name="Listeners" colour="160">
2   <block type="tile_on_device">
3     <value name="id">
4       <block type="tile_device"></block>
5     </value>
6     <value name="action">
7       <block type="tile_action"></block>
8     </value>
9   </block>
10 </category>

```

Listing 7.11: Blockly - Custom block loading

An overview of the registered TILES is retrieved from the TILES infrastructure, and dynamically inserted within the TILES layout, as shown in figure 7.11.

## 7.3 Emulator

The emulator is a web-application, built with TypeScript. The emulator is stand-alone, but integrated within the development environment, to simply

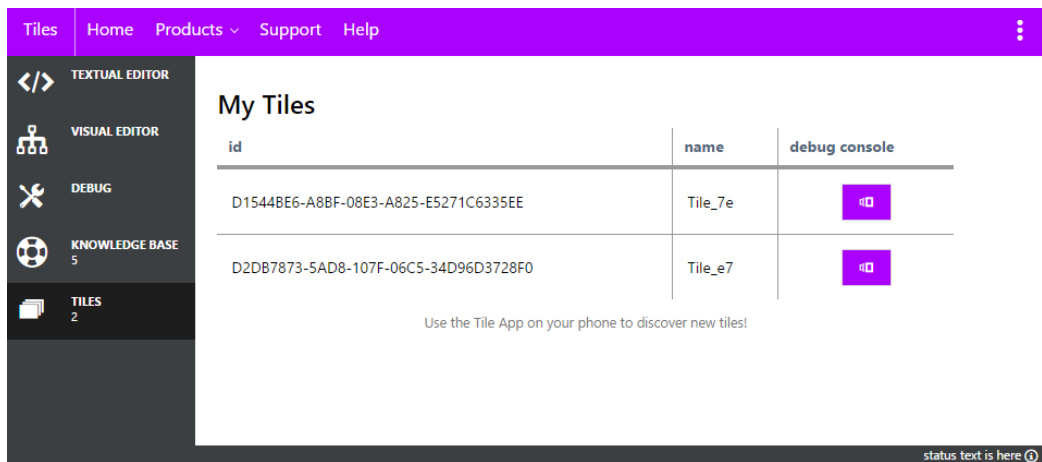


Figure 7.11: Development Environment - TILES layout

emulate an actual device. Multiple emulators are able to run simultaneously, and are not dependent of each other. The emulator works as a client connected to the TILES infrastructure, supporting user interaction. Both receiving commands and sending events is supported by the emulator, and as an actual TILES device, it will control its output interfaces. The emulator is a simpler web-application only consisting of a single static HTML file. The application logic changes the status of the HTML elements, to simulate controlling the output interfaces, and sending events based on interactions on its buttons. A UML class-diagram of the emulator is shown in figure 7.12.

The `Client` class maintains a connection to the TILES infrastructure, for receiving commands and sending events. The `Device` class maintains the HTML elements which represents the different output interfaces and event interactions. It's the main class for the emulator, acting as a facade for the other classes. The class handles all user-interactions and events creations based on a triggered user-interaction, along with passing a command to the appropriate class representation for the output interface. Each output interface is represented with their own class for parsing a received command, and changing the status of the HTML element based on the incoming command. Each output interface class inherits the same interface, `iOutput`. The interface defines functionality which each output interface is required to support. Extending the emulator with a new output interface requires only an inheritance of the `iOutput` interface, and an implementation of appropriate methods for controlling its HTML element.

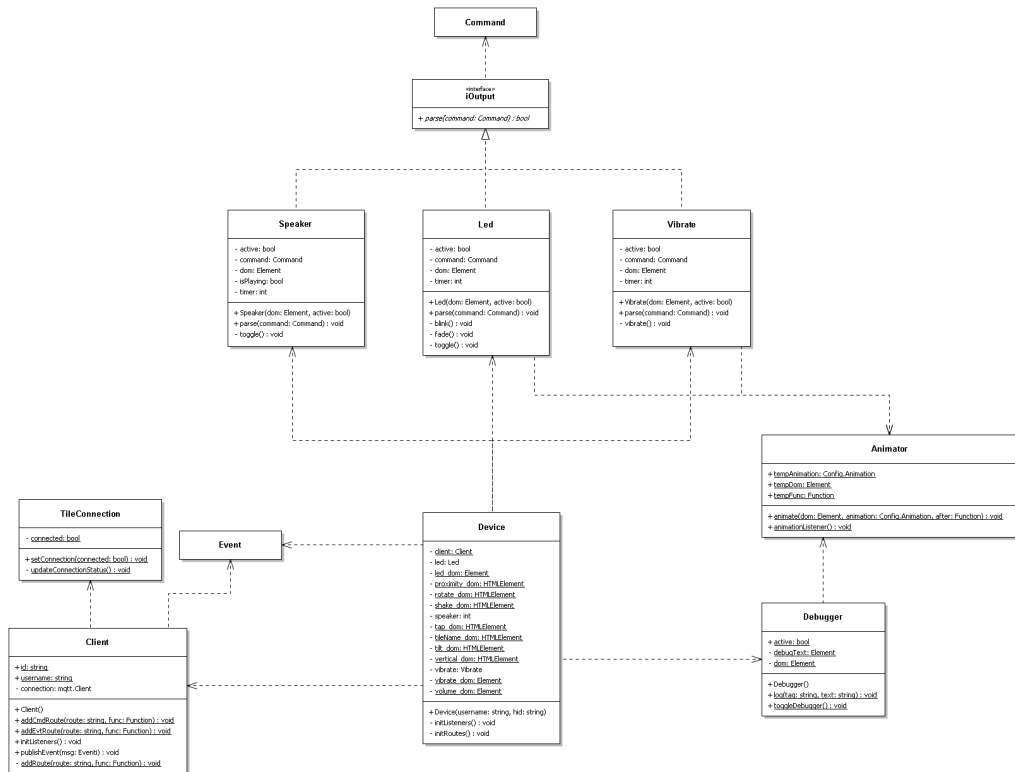


Figure 7.12: Class-diagram Emulator

The layout of the emulator is shown in figure 7.13, and includes highlighters for easier description. Highlight 1 outputs the name set for the emulator. In the picture below, this is *alpha*. Highlighter number 2 is a status indicating whether the emulator is connected to the TILES infrastructure or not. It values can either be *connected* or *disconnected*. Highlighters 3-5 shows the output interfaces for the emulator: LED, speaker, and vibrate indicator. A debug console is also available by clicking the button indicated by highlighter 6. The debug console is shown in figure 7.14.

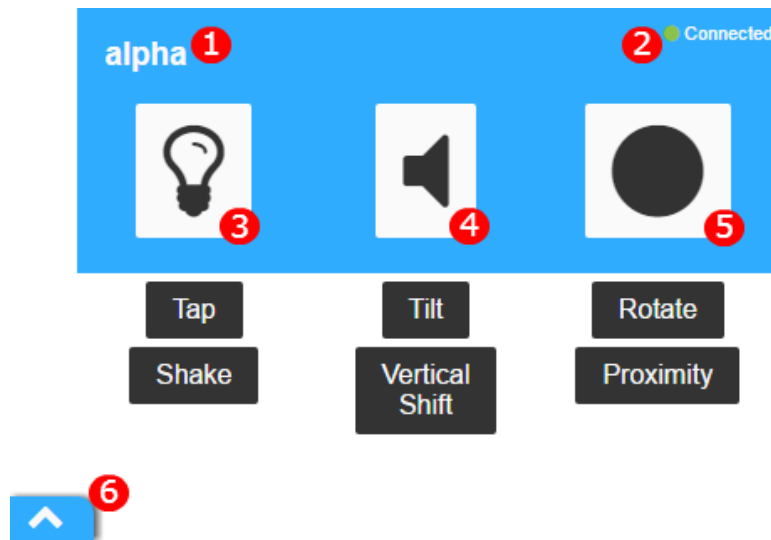


Figure 7.13: Emulator layout

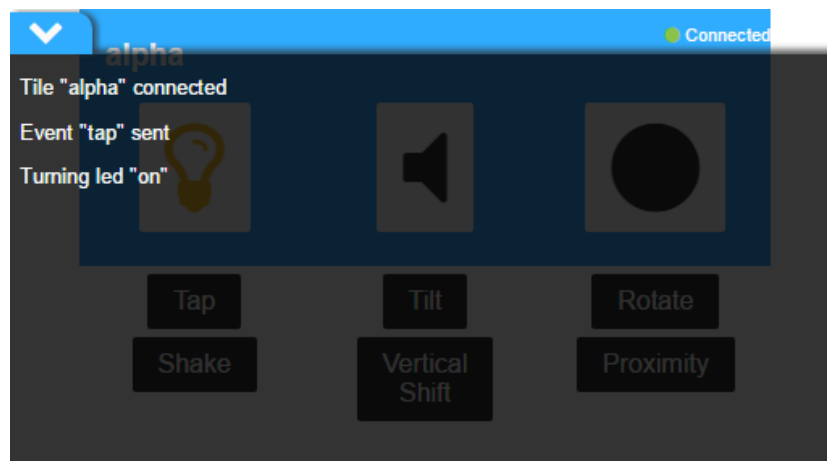


Figure 7.14: Emulator debug layout

## 7.4 CLI

A simple command-line-interface is provided for compiling and running the written DSL locally. The CLI uses a library called *minimist*<sup>18</sup> for parsing command line arguments. The CLI is invoked by using Node.js locally, and passing arguments in the following format:

<sup>18</sup><https://github.com/substack/minimist>

```
node <file.js> <input> <output> <options>
```

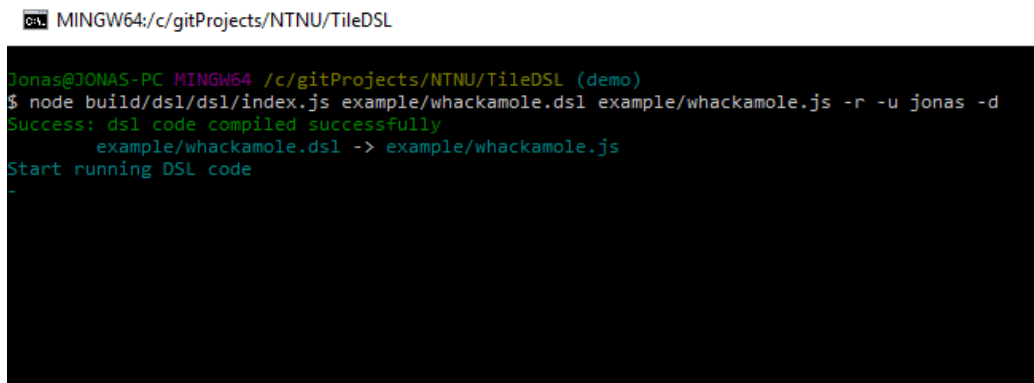
Each parameter is split by space. `<file.js>` refers to the Node.js file to execute. All parameters provided after the first parameter is parsed by `minimist`. `<input>` is the path to the DSL file to be compiled, `<output>` is an optional parameter, stating a path and filename, on where to store the compiled code locally. The different `<options>` available are shown in the table below.

Parameter	Optional	Description
<code>-u</code>	<code>--username</code>	User to run application as
<code>-r</code>	<code>--run</code>	Run application immediately
<code>-c</code>	<code>--code</code>	DSL code to compile. Overrides <code>&lt;input&gt;</code>
<code>-d</code>	<code>--debug</code>	Print status of the compilation process

Example usage of the CLI is shown in figure 7.15, running the following parameters:

```
build/dsl/dsl/index.js example/whackamole.dsl
example/whackamole.js -r -u jonas -d
```

The first parameter refers to the Node.js file to execute, which is the compiled version of the TypeScript source. The second parameter refer to the `.dsl` file containing the DSL code to compile. The output of the compilation is provided by the third parameter. As this is an optional parameter, it can be left out, and the default it `output.js` in the current directory run from the command-line. The rest of the parameters specifies the application should be run immediately after compilation, connect to the TILES infrastructure using the username `jonas`, and that the status of the compilation process should be outputted back to the console-line.

A terminal window with a black background and white text. The window title is 'MINGW64:/c/gitProjects/NTNU/TileDSL'. The prompt is 'Jonas@JONAS-PC MINGW64 /c/gitProjects/NTNU/TileDSL (demo)'. The user enters the command '\$ node build/dsl/dsl/index.js example/whackamole.dsl example/whackamole.js -r -u jonas -d'. The output is 'Success: dsl code compiled successfully' followed by 'example/whackamole.dsl -> example/whackamole.js' and 'Start running DSL code'.

```
MINGW64:/c/gitProjects/NTNU/TileDSL
Jonas@JONAS-PC MINGW64 /c/gitProjects/NTNU/TileDSL (demo)
$ node build/dsl/dsl/index.js example/whackamole.dsl example/whackamole.js -r -u jonas -d
Success: dsl code compiled successfully
example/whackamole.dsl -> example/whackamole.js
Start running DSL code
```

Figure 7.15: CLI example usage





# Chapter 8

## Evaluation

### 8.1 Pre-implementation evaluation

Before the implementation of the DSL and the development environment, an online focus group were conducted, using an online tool called *FocusGroupIt*<sup>1</sup>. The focus group contained 7 topics, where 6 of them required an answer from the participants. The participants come from different backgrounds, including computer science students, makers, and professional software developers. The group consisted of 8 people, where 6 of them replied to all required topics. The topics which required an answer from the participants are found in appendix C.

The focus group were conducted to get feedback on the different scenarios for developing TILES applications, to see if someone have had any previous experience with the same approaches. The participants were asked to explain their previous experience on each topic, and if they had any good or bad experiences following the explained approach. Their answers were to help with planning the functionality for a development environment for TILES. The bad experience could help with either not follow a certain approach, or making sure the same bad experience is not replicated within the TILES development environment. The good experiences would help by trying to replicate the same behavior in the process of creating applications for TILES.

Only a few of the participants had any experience with developing in a web-browser. Those with previous experience pointed out the benefit of not re-

---

<sup>1</sup><https://www.focusgroupit.com/>

quiring a setup process to start developing, and the benefit of being provided always up to date system, as an update would be made accessible to all clients at the same time. As quoted by a participant coming from a maker background: *"My experience is that the browser is today's best approach for all development (not high-end gaming)"*. Another participant had experience through different courses in school, but mainly editing a few lines using the textual-editor in GitHub<sup>2</sup>, an online source code management system. The same participants had concerns about reliability, mainly concerning losing any work if the web-browser might refresh its state. This was backed up by another participant, that though an auto-save feature would be a nice feature when coding in a browser. Another concern was not having standard IDE features available, like keyboard-shortcuts and code highlighting available. A participant also said coding in a browser is not a problem as long they are not working with a large code-base, including multiple files with a great number of lines of code. Having the ability to code anywhere were discussed as a benefit of using a browser-environment for coding.

Most of the participants had a great experience with the use of libraries and APIs. One participant noted the struggle of understanding the different definitions between a DSL, library, and an API, as the participant felt they were the same. A great library which is easy to use would require a well-written documentation, which was pointed out by all participants with a previous experience using either libraries or APIs. Some also mentioned the importance of providing samples on how to use a library, and that it had a high usability, easy to understand for the users. The only negative experience the participants had was with poorly documented libraries or APIs.

All participants were familiar with using a textual approach for development. Their background on JavaScript were however different, where some used it more as a necessity for creating web-pages, others were using it in their daily work. The road of JavaScript, from being fairly used, to today being required and used "everywhere", were discussed regarding JavaScript usage in larger applications. One participant also felt JavaScript were following poor standards because it was not intended for heavy applications. Other participants had no problem, because of its large community and large support. Regarding a textual approach for development, one participant noted: *"A textual approach allows the application to be comprehensive and complexed as possible with no restrictions except the language itself."* Another participant were thinking about other approaches than textual, and came in favor for a textual approach: *"A textual approach gives better possibilities for the users,*

---

<sup>2</sup><https://github.com/>

*creating applications by their own imagination and not restrained by a toolbox given to them through a graphical toolbox.*". The users are here referred to as makers and computer science students wanting to create applications for TILES.

No participants had any great experience with visual programming, other than simple testing. These were participants with a textual programming background, and many of them saw a visual approach as a restricted way of programming. *"I feel that you might cap your complexity by using the visual approach"* was one of the concerns a participant had. Even though the participants were familiar with a textual programming approach, they still saw some advantages: *"... this would be really great for handling things like smart-home events..."*, and some argued that a visual programming approach would provide a wider range of people to be able to create applications for TILES because of its simplicity. Not all did agree with this statement however, and saw the visual approach as a more time consuming process to learn and master properly. *"The end user who wants to build application-logic by GUI, most likely also wants a GUI to interact with as well"* was another important statement made by one of the participants, as this participant started to think about the whole environment around application development for TILES.

The use of emulators was familiar to the participants from both Android<sup>3</sup> programming and web-development. All participants saw the use of an emulator as a good approach for testing. Especially if the TILES devices themselves are missing some hardware features that may be emulated in an emulator, or if someone has no immediate access to a TILES device. One participant also noted that the use of emulator is a faster approach for testing application than on an actual hardware device. For TILES use-case, a participant did not see the benefit of an emulator however. Because JavaScript doesn't require any compilation, this participant thought publishing the application directly, and test it on the fly, could be an equally good alternative as an approach provided by an emulator.

### 8.1.1 Discussion

The focus group had some very interesting discussions and brought up many aspects of importance for a development environment for TILES. A web-browser environment for TILES seemed to be a great way to go, but with

---

<sup>3</sup><https://www.android.com/>

some concerns. Regarding these concerns and suggestions, a save feature was implemented in the TILES Development Environment. The code is stored in the local storage of the browser, and retrieved once the text-editor is brought back up. With some minor tweaks, this could be changed to an auto-save feature, saving the work after the writer stops typing. The ACE-editor provides multiple sets of features, including many found in standard desktop IDEs. Configuring keyboard shortcuts is a feature which the ACE editor provides out of the box. This feature was implemented within the TDE as well, by moving the cursor to the first line in the editor when `ALT+G` is pressed. The ACE-editor also supports creating custom code highlighters. As the DSL provides its own syntax, a custom syntax highlighter would be required for the DSL to be recognized by the ACE-editor. This were not however prioritized, because of time-restrictions. As the DSL is defined, creating applications for TILES is not meant to include a large number of files. All of the application logic will be defined within a `main` method. The number of line numbers depends on the complexity of a TILES application.

Regarding the concerns of a visual approach, some may find a graphical approach of creating applications more easy to grasp. Especially those with less programming experience. As the participants were very familiar with a textual approach, they are used with the freedom which a textual-approach provides. Others may find a textual approach providing too much freedom, and don't know where to start. Rather complex applications are also possible to create using a visual approach, but may require another way of thinking than a textual approach. Simple command and event handling were pointed out by one of the participant to be a good fit for a visual approach, which correspond with the TILES infrastructure and data-source communication.

The emulator approach had both some positive and negative notes from the participants. The missing hardware emulation, and being able to more easily test an application with multiple emulators were found to be some strong points for developing an emulator. As the TILES device is still under development, an emulator would help regarding testing the functionality before it's actually supported by the device itself. When a new feature is added to the hardware device, it can be already thoroughly tested in the TILES infrastructure because of the emulator. Regarding JavaScript not requiring any compilation, the DSL however does. This is a process supported within the development environment, to make sure no syntax errors are crashing the application when it's running in the TILES infrastructure. Even though an application is syntactically correct, it may not act as the developer intended. Testing the application logic requires the application to be communicating with the TILES infrastructure. Regardless if an emulator is provided, the

application still requires to be uploaded to the TILES infrastructure. The emulator is therefore seen more as a tool while developing applications for TILES, where makers and computer science students are able to emulate their own devices, or add custom emulated devices. The makers and computer science students are then helped with the whole development process, from writing code, to compilation, and lastly testing, all from their web-browser. The emulator removes any hassle it may be to connect their TILES devices to the TILES infrastructure, and as pointed out by one of the participants, provide a quicker way of testing.

The last topic, where the participants were to propose something of their own ideas for a TILES development environment, none of the participants came with any suggestions. The questions on this topic may have been too open, compared to the previous topics, where direct questions on a given functionality were provided.

## 8.2 Implementation evaluation

An evaluation of the implemented development environment for TILES were conducted at an event hosted by TELL, a cooperation between NTNU<sup>4</sup> and SINTEF<sup>5</sup>. The event was presenting different ways of learning, using IoT, and was held at DIGS in Trondheim. Around 30 persons attended the event, in which the TILES were presented and demoing the TILES Development Environment. The demo consisted of the full TILES infrastructure. A running demo of the development environment were shown, along with the implemented DSL, some example applications written using the DSL, and the compilation, and running of the TILES application. Three TILES Devices were brought to the demo, which were made accessible for user-interaction. Multiple people from different backgrounds visited the TILES demo, and were curious about learning what TILES is all about.

The demo was showing multiple aspects of the DSL language, including fetching data from third-party-services, listening and waiting for a single and a sequence of events triggered by the TILES Devices. The demo started with a presentation of the TDE for creating TILES applications. To help explain the DSL, development environment, and its control over the TILES devices, the text-editor contained an example code of the game *WhackAMole*, as described in section 6.3.3. The compilation and uploading process were shown,

---

<sup>4</sup><https://www.ntnu.no/>

<sup>5</sup><http://www.sintef.no/>

and a random TILES Device turned on its LED. The participants were very intrigued on what just happened, and after a simple explanation of Whack-A-Mole, they started quickly to play around with the TILES devices. As most of the participants attending the demo came from a more technical background and already familiar with programming, they were very impressed that just three lines of simple code were able to control the output interfaces of the TILES Devices, and respond based on their interaction with it. One participant that was looking into user-interacting objects at work, knew how difficult it's to create applications that respond to user-interactions. The participant were very interested that the demo were showing the whole process from writing code, to uploading and running the application in the infrastructure, and stated: *"I haven't seen the process to be this simple before"*.

Other participants that attended the demo also wanted a thoroughly walk-through of the whole development process for creating applications for TILES, and had questions about the implementation of the DSL. One participant in particular, a former computer science student, meant the DSL would not be required for a future implementation of a visual programming approach, as the DSL were just compiling the code to JavaScript. This participant was already familiar with Blockly, and meant plain JavaScript code could be defined for each block. When the process of the DSL was explained, and showing the JavaScript equivalent code, which the parser compiles to, the participant quickly understood the benefit of using a DSL as a bottom layer for each programming approach, as it would be more simple to defined each custom block, and it's equivalent code output. The same participant observed after interacting with the TILES Devices, and restarting the running application, that the infrastructure were not storing any state of the output interfaces for the TILES Device. Two of the devices had its LED turned on at the same time. The participant explained how storing the state of an LED could be implemented, but would require some change on the hardware device, which in turn would drastically complicate the hardware code.

### 8.2.1 Discussion

The demo served its purpose by getting feedback from participants with background in IoT, and programming experience. Participants with no programming experience were however more careful regarding testing the development system, and rather inspected the provided example code. The participants were able to interact with the whole development environment, from hardware to software. The feedback from the implementation evalua-



Figure 8.1: Demo of TILES

tion, were only positive regarding the simplicity of the DSL language, and the potential for TILES because of it.

### 8.3 TDE requirement evaluation

This section will evaluate the implemented features based on the requirement specification for the TILES Development Environment, see section 5.3. The requirement evaluation is conducted, to make sure the implementation, as described in chapter 7, meet the requirements needed for a development environment for TILES.

A web-application, supporting both a textual-environment and a visual-environment, using ACE and Blockly respectively, were implemented early in the development process. This complies with the functional requirements: *FR1*, *FR2*, and *FR5*.

Through the implementation of the DSL modules, section 7.1.1, the abstract functionality facilitate an easier implementation of a custom DSL for TILES. The logic behind the DSL is put behind two abstract facades. The first facade is encapsulating the DSL modules, which is the actual software code for handling communication with the TILES infrastructure. The second facade

is the language itself, abstracting the DSL module facade further. The second facade consist of a parser, which is responsible of generating calls that is understood by the DSL module interface. Proper communication between these facade requires proper interoperability, done through a tactic called *orchestrate*. The orchestrate tactic is when an interaction between software components are scripted, which is the case for the DSL parser. A set of language types is translated to an equivalent call to the DSL module interface, to execute the interface's appropriate methods based on the parsed language type. The interoperability of the encapsulated DSL modules is done by each TILES application on execution. Modifiability is also one of the quality attributes which the DSL modules provide. The modules have no coupling between each other, making it more easy to do any future changes on the modules, as a change will one affect that particular module. Focus on increasing the cohesion, where common responsibility has been moved to the same module, have been a concern from the start of the design process. The decision of creating modules through division by responsibility, have helped with the development distributability, in which a module could be implemented one at a time, in any order. Abstracting common services through inheritance, polymorphism, and defining classes as abstract, have helped with implementing a service in a more general form, supporting easier extension of the DSL modules. These are all tactics complying with *FR3*.

From the definition of the language syntax, and the use of PEG.js, as described in section 7.1.2, the DSL is built with JavaScript, as defined by *FR4*.

Section 7.2 describes the process of generating custom visual block elements. Translating the combined blocks to JavaScript is also explained, with the use of the `VisualEditorController`, complying with *FR6*. *FR7*, regarding visual representations of the DSL, only a simple sample were provided because of time-constraints. A fully functional visual programming approach is therefore not implemented. Because of the requirement's priority, a visual approach was not prioritized over implementing a functional textual programming approach. Requirements dependent on *FR7* is however met, meaning the system supports the implementation of visual blocks.

Both *FR8* and *FR9* are a feature described by the participants in the first focus group, described in section 8.1. Storing the DSL code locally is described in section 7.2.

The implementation evaluation at DIGS, described in section 8.2, included a running demo of uploading and executing TILES applications in the TILES infrastructure. The demo both meet the requirement of *FR10* and *FR11*.



The deployability of a TILES application includes how an application is uploaded to its running environment, and how it can be integrated into the existing TILES infrastructure. The DSL code created in the development environment were uploaded using the HTTP protocol through the *POST* method. The uploading and executing of a TILES application do not affect the existing execution of the TILES infrastructure. The infrastructure was extended, by adding a child-process to the infrastructure, in which the application were run in a sandbox environment. Any TILES applications executed are not able to influence any resources it should not have access to, by running the application within a child-process. The application is however dependent on its parent process, meaning if the infrastructure were to stop its execution, all child-processes would stop executing as well. The parser for the DSL are both needed in the development environment, and to be integrated within the TILES infrastructure. The portability of the parser is great, as it can easily be run in different environment: both client and server side, with a simple change in its built configuration. The availability of a running TILES application is supported through exception handling. Each DSL module, and the DSL module interface, provides multiple exception handlers, to make sure the TILES application will continue its execution regarding if an error should occur. The exception handling does however only prevent minor errors to affect the application. If a more severe error occurs, including logic error, which is created by the developer of the TILES application, the application will stop execution.

The DSL parser also provides a command-line interface, as described in section 7.1.2. The command-interface can both compile DSL code, and run the compiled application directly from the command-line. The application running will then be like just another client connected to the TILES infrastructure. From the CLI-interface, the application is running in its own environment, where the state of the application is fully accessible from the command line. This provides a way of testing an application further, but requires extra setup, then using the development environment. The CLI-interface complies with *FR12*.

The functional suitability of the development environment is meeting almost all of its requirements, except *FR7*, where only the fundamental software methods are provided. The implementation of the TDE is however seen as successful, even though the visual DSL representations are missing. As most of the work is completed regarding supporting custom visual blocks, little effort is required for creating a fully functional visual programming approach. As already stated, because of time-constraints, the creation of custom blocks was not prioritized. The TILES Development requirement

can easily be further extended by adding new modules, because of its focus on modifiability.

## 8.4 DSL requirement evaluation

This section will evaluate the implemented features based on the requirement specification for the DSL, as described in section 5.4. Each requirement is described as a user-story, with some acceptance criteria. Each user-story will be compared to its acceptance criteria, followed by an example on how the DSL is supporting a user-story.

The DSL provides syntax rules for successfully recognizing and parsing commands that control the output interfaces for a specified TILES device. The supported output interfaces implemented in the language is: **LED**, **speaker**, and **vibration**. A language specification of the commands supported is found in appendix C.2.5. The acceptance criteria for user story 1 are all met. The user story 2, where a presentation of the available commands that are available to use, is provided by the DSL parser itself. If a syntax error is found, the parser provides an error message consisting of the legal syntax to use as an example. If an unknown output interface is referred to in code, the parser will notify the error, and list the available output interfaces.

Appendix C.2.3 describes how different TILES devices may be referenced within the application. All registered TILES devices may be stored within a variable, or only a single TILES, as long as its name is provided. Both **TILES** and **TILE.<tile name>** is defined as keywords within the language. The TILES devices available are retrieved from the database within the TILES infrastructure, to make sure that only TILES devices that belong to the user writing a TILES application, is referenced in code. TILES referencing complies with user story 5, 6, and partly 10, where the name of a TILES is used as a reference, rather its longer MAC-address. A **Me** keyword is provided of the language as well, for accessing user stored data, like name and id. The specification is given in appendix C.2.4.

Listening for events triggered on a TILES device, is provided through the **if** statement, described in appendix C.2.7.

```
if (TILE.alpha.tapped) { ... }
```

This statement will set up a listener, and wait for the TILES device called `alpha`, to be interacted with a tap. The code defined within the brackets will be executed once the event is triggered. This examples also shows that the application requires parsing of the received event data, to properly identify whether a `tap` event occurred. Events can also be further identified, by providing an extra parameter on the event, as described in appendix C.2.6. Adding `.double` at the end of the parameter within the `if` statement, the code within the brackets will only be executed as long as a double tap occurred. Both user story 3 and 4 are shown with the provided example to be successfully implemented.

The language compatibility with third-party sources is described in appendix C.2.8, which both list examples compliant with user story 7, 8, and 9. The support of pre-define third-party sources is implemented within the language itself, to help creating syntactically correct calls to the DSL modules. The examples provided within the language specification (see appendix C.2.9) shows example of retrieving data, posting data, and integrating the retrieved data within the application logic of a TILES application.

The `sync` keyword described in the language are listening for events triggered in a given pattern. Its language specification is given in appendix C.2.7, with a provided example of its use. The provided order makes sure the code within its block are not executed before all events are triggered in the exact order.

All user stories are successfully supported by the DSL. The language specification, given in listing 6.1, defines the syntax rules to accomplish these features, while the implemented DSL modules carry out the logic behind the code. Each user story is tested using the CLI of the DSL, making sure it's both compiled and run as expected.

## 8.5 Language evaluation

An evaluation of the language was conducted with two professional programmers, with a background as computer science students, as participants through a group interview. They were already familiar with TILES, as they also were a part of the focus group session described in section 8.1. Because of the participants' background, they are familiar with multiple programming languages, and use programming as part of their daily work. Before the evaluation started, they were presented with the language specification

included in appendix C.2. The participants were given some time to go through the language specification, before providing their feedback on how they perceived the language. The evaluation was conducted to be provided with a more technical feedback on the language itself, and the perceived usability of the language.

One of the first comments from the participants were the similarity of the DSL with the Arduino programming language<sup>6</sup>. The Arduino programming language provides a specified starting point, including a `main` method, running in an infinite loop. The DSL is inspired of the Arduino programming language, and provides similar `setup` and `main` methods, where the main method can optionally run in an infinite loop.

Both participants found the language easy to understand, especially as it was built on similar syntax rules they were familiar with from other programming languages. One of the participants commented on the if statement however:

```
if ... then ...
```

This participant found the syntax to be similar with pascal and basic, two programming languages using the same syntax as shown above. This is all about preferences programmers have, and the participant noted he would preferably support an `if` statement without the use of brackets, which are supported in most major programming languages, like *C*, *C++*, *Java*, *PHP*, and *JavaScript*:

```
if (...) ...
```

As the DSL is built on top of JavaScript, and follow many of its patterns, the participant argued that following the same conventions would be more beneficial for makers and computer science students. Since JavaScript can be mixed with the DSL, removing the `then` keyword, would make the DSL to be more close with JavaScript in terms of syntax, and also not require three different methods of using the `if` keyword.

Both participants were familiar with using Node.js, and the hardware device Raspberry pi<sup>7</sup>. They have been introduced to the use of libraries, for controlling different hardware devices like GPS and camera, attached to a Raspberry

---

<sup>6</sup><https://www.arduino.cc/en/Main/Software>

<sup>7</sup><https://www.raspberrypi.org/>

pi device, running on NodeJS. The simplicity of communicating with these hardware devices were compared to the DSL. If larger and complex application is required, having the ability to use plain JavaScript along with the DSL, were considered to be a great benefit for the language. The DSL is not only making it more easy for makers and computer science students to create applications for TILES, but also support a TILES application to be rather complex because of the power of mixing JavaScript and the DSL.

The language was well received by both participants, and perceived as something they could quickly start to use, and create simple applications with. Even one of the participants asked about the possibility of using this DSL with his own IoT devices at home. As the DSL is created with TILES in mind, porting to other devices would require a change for the code-generator, after parsing the DSL. This is however a whole different area, not concerned for the TILES toolkit. Creating a "cook-book", a set of example usage, of the DSL language was proposed by the participants, to easier show how the language could be used. A cook-book could also benefit as a documentation and starting-point of the language, in which makers and computer science students could base their applications on. One participant also asked how the DSL were simplifying work, as opposed to just using a library. The next section 8.6, will answer this question, in regards of comparing DSL code and its equivalent compiled DSL code.

## 8.6 DSL Compilation evaluation

To show how the DSL helps in simplifying the process of developing applications for TILES, a set of DSL examples are provided, along with its JavaScript equivalent. The compiler generates JavaScript code that calls methods from the DSL module interface. The `dsl` keyword, is a reference to the current scope in which the application is running. This evaluation is meant to show how much the development process is simplified for makers and computer science students. The compiled version of the TILES application, is assigned to a new function, and passed to the `TileDSL` class, using the `set userFunction` method, as shown in figure 7.2. The compiled version of the DSL, is not executable itself, but needs to be run within the scope of the `TileDSL` class. Listing ??, show how the compiled DSL may be added to the scope of the `TileDSL` class, by replacing the compiled code with `parse.code`. The DSL modules itself, contains over 2000 lines of code, which sets up the application environment, initializes communication with the TILES infras-

tructure, and handles database management. The simple example of the game *WhackAMole* can be defined in the DSL as follows:

```

1 main() {
2   var tile = random(TILES);
3   tile.led;
4   if tile.tapped then tile.led.off
5 }

```

The compiled Javascript, generates the following output:

```

1 var main=undefined;
2 var options=undefined;
3 main = function(dsl){
4   var tile = dsl.random(dsl.tiles);
5   dsl.led.turnOn(tile);
6   tile.once("tap", function(event){{
7     dsl.led.turnOff(tile);
8     return main(dsl);
9   }});
10 };
11 return output(options,main)

```

The compiled version of *WhackAMole*, uses listeners, and the structure of the DSL is still found within the compiled code. Line 2 in the DSL example:

```
var tile = random(TILES);
```

is equal to the compiled JavaScript:

```
var tile = dsl.random(dsl.tiles);
```

Not much differs between these lines, other the introduction of the `dsl` variable. Once the application grows in size and complexity, the true benefit of the DSL is shown:

```
1 setup(){
2   Facebook.options.apikey="myapikey";
3   Twitter.options.consumer_key="ckey";
4   Twitter.options.consumer_secret="csecret";
5   Twitter.options.token="token";
6   Twitter.options.token_secret="stoken";
7 }
8 main(){
9   if(TILE.alpha.tapped){
10    var test=1;
11  }
12  if TILE.alpha.tapped.double then TILE.beta.led.
    green
13  repeat(3){
14    Facebook.post("my repeating facebook post
    message");
15  }
16  var i=0;
17  while(i<10){
18    ++i;
19  }
20  sync(TILE.alpha.tapped.single TILE.beta.shaked
    TILE.ceta.tapped){
21    Facebook.post("test");
22    Twitter.tweet("hello");
23    TILE.beta.led.white;
24  }
25  random({"test":"hello","teit":"hmm","jadda":"tenk"
    });
26  random(["test","hello"]);
27  var tile=random(TILES);
28 }
```

The compiled version of the provided DSL example, is as follows:

```
1 var main=undefined;
2 var options=undefined;
3 options=function(dsl){
4   dsl.initFacebook('myapikey');
5   dsl.initTwitter('ckey','csecret','token','stoken')
6   ;
7 };
8 main = function(dsl){
9   dsl.tiles.alpha.once("tap", function(event){{
10    var test = 1;
11  }});
12 dsl.tiles.alpha.once("tap", function(event){if(
13   event=="double"){
14   dsl.led.turnOff(dsl.tiles.beta);
15  }});
16 for(var i=0;i<3;++i){
17   dsl.facebook.post("my repeating facebook post
18   message");
19 }
20 var i = 0;
21 while (i < 10) {
22   ++i;
23 }
24 dsl.initPipe("pipe0"
25   ,{tile:dsl.tiles.alpha,event:{name:"tap",event:"
26   single"}}
27   ,{tile:dsl.tiles.beta,event:{name:"shake"}}
28   ,{tile:dsl.tiles.ceta,event:{name:"tap"}});
29 dsl.once("pipe0",function(){
30   dsl.facebook.post("test");
31   dsl.twitter.tweet({status:"hello"},function(data
32   ){});
33   dsl.led.turnOff(dsl.tiles.beta);
34   return main(dsl);
35 });
36 dsl.random({
37   "test": "hello",
38   "teit": "hmm",
39   "jadda": "tenk"
```



```
35  })
36  dsl.random(["test", "hello"])
37  var tile = dsl.random(dsl.tiles);
38  };
39  return output(options,main)
```

From this example, the structure becomes more complex to compare to each other. A lot of calls are made to the `dsl` variable, and most of the keywords in the DSL is not used within the compiled code. The `sync` keyword, in line 20 of the DSL example, is compiled to the output shown in line 21 - 24 in the listing above. Instead of all being defined in a simple line, three lines are required for the compiled code. The structure of the pipe-and-filter patterns, described in section 7.1.1, becomes clearly visible. How the `sync` keyword inner-working are is not visible at all, neither should it be. As in programming, the language abstraction helps with hiding away details that is of no concern to those who use it, as long as it's working.

The provided examples are also in reference to how the DSL simplifies work, as opposed to using a library. As described in section 8.5, this was a question from one of the professional programmers. When using the DSL, the code can be parsed before compilation, and therefore check if any syntax errors are found. Using a library, this would not be possible before actually running the code, which would have caused a type exception to be thrown, and stopped the application from running. Any syntax errors found from the DSL parser, will also contain a descriptive error message, to help fix the error, and includes a list of keywords that the parser were expecting. This will clearly benefit in the development process of TILES applications, as errors are found before publish any applications to the TILES infrastructure. The DSL and the equivalent JavaScript code show how much code is simplified by using the DSL. The DSL also removes the term of listeners and asynchronous functions, and replaces them with statements, like `if`, usually found in programming languages. A statement is in the form as shown below, where `logic` is the input parameter to the statement, while `code` refers to the code to execute when the `logic` has occurred, or equals to `true`.

```
STATEMENT(logic){code}
```

## 8.7 Application creation

The following section will describe how makers and computer science students are able to use the development environment, and start creating applications for TILES. This description should show how easy the development environment has made it for makers and computer science students getting started with TILES.

The development environment is run in a browser. The layout which makers and computer science students will be using to create TILES applications are shown in figure 8.2. The figure contains multiple highlight blocks, and the explanation of each is shown in table 8.1.

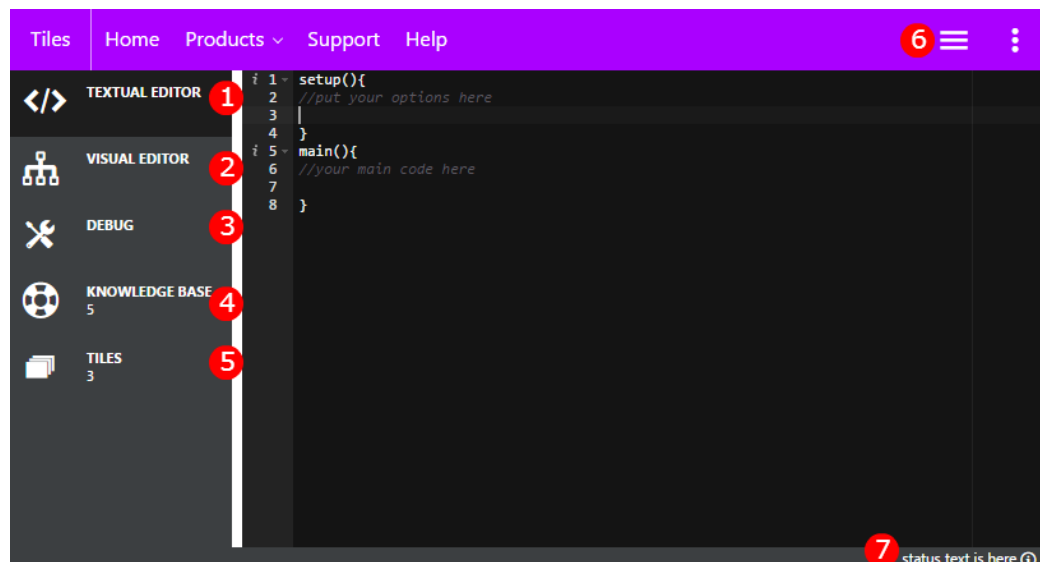


Figure 8.2: TDE layout

#	Type	Description
1	Left-sidebar	Start the ACE text-editor for creating textual applications
2	Left-sidebar	Start the Blockly visual-editor for creating visual applications
3	Left-sidebar	Not implemented. Debug interface logging data sent to and from TILES infrastructure
4	Left-sidebar	Documentation to help with creating TILES applications, includes the language specification (see appendix C.2)

5	Left-sidebar	Overview of registered TILES devices
6	Header-menu	If shown in current view, provide optional features in a right-sidebar
7	Footer status bar	Prints important status messages, whether an error occurred, or if a task was triggered successfully

Table 8.1: DSL types

The provided text-editor is pre-defined with the `setup` and `main` methods as shown in the TDE layout figure. Makers and computer science students able to write DSL code, compile it, and see it running in the TILES infrastructure from the textual-editor layout. The compilation is run within the development environment, making sure no syntax errors are found before uploading the TILES application to the TILES infrastructure. The compilation can be triggered using a keyboard shortcut, `ALT+1`, or using the header-menu, which will bring up a right-sidebar menu shown in figure 8.3.

The build options menu provide functionality for compiling the current code typed in the textual-editor, or store the code locally, which is able to be restored the next time the development environment is run using the same browser. The share code functionality is not implemented. The compilation process, and the execution of TILES applications are shown in figure 8.4. Figure 8.5 show the error message created if a syntax error found within the DSL code.

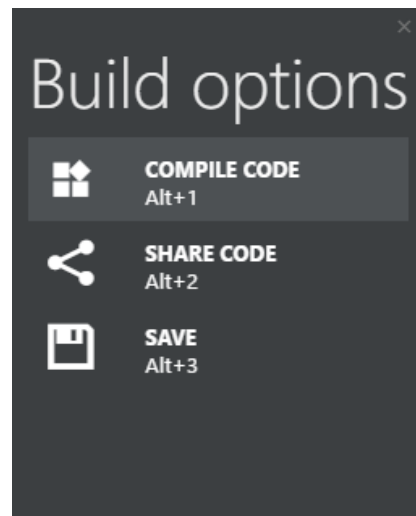


Figure 8.3: Build options menu

Using emulated TILES devices, instead of physical is also possible when developing applications for TILES in the development environment. Figure 8.6 shows as example of using four emulators with the game WhackAMole. Each emulator act as a TILES device, connected to the TILES infrastructure, as-

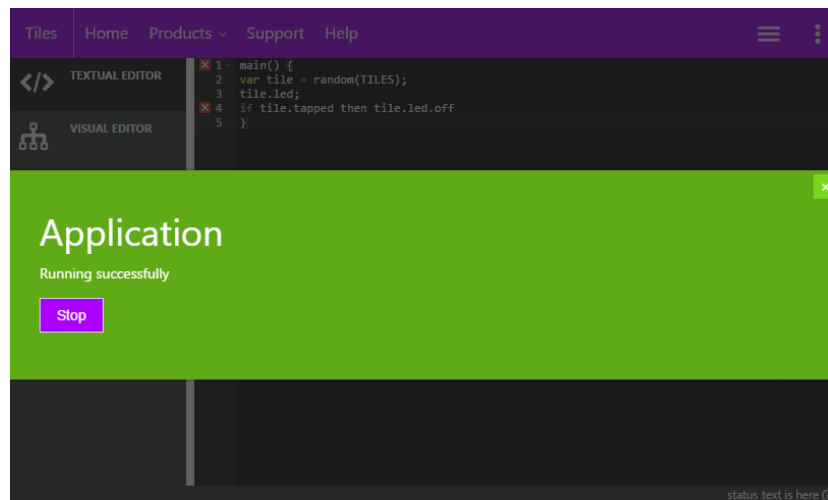


Figure 8.4: Running TILES application

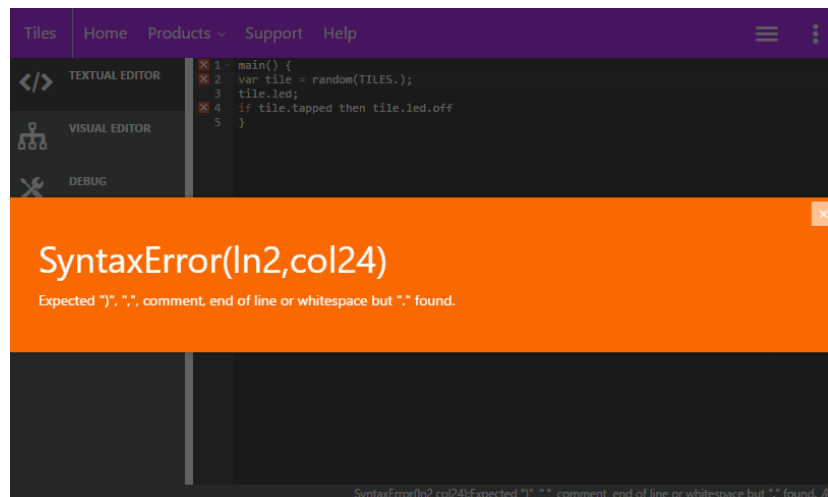


Figure 8.5: Syntax error TILES application

signed with its own identifier name. The figure below shows the emulator "test1", top-left, using its speaker output interface, and the "test3", bottom-left, emulator to having its led turned on.

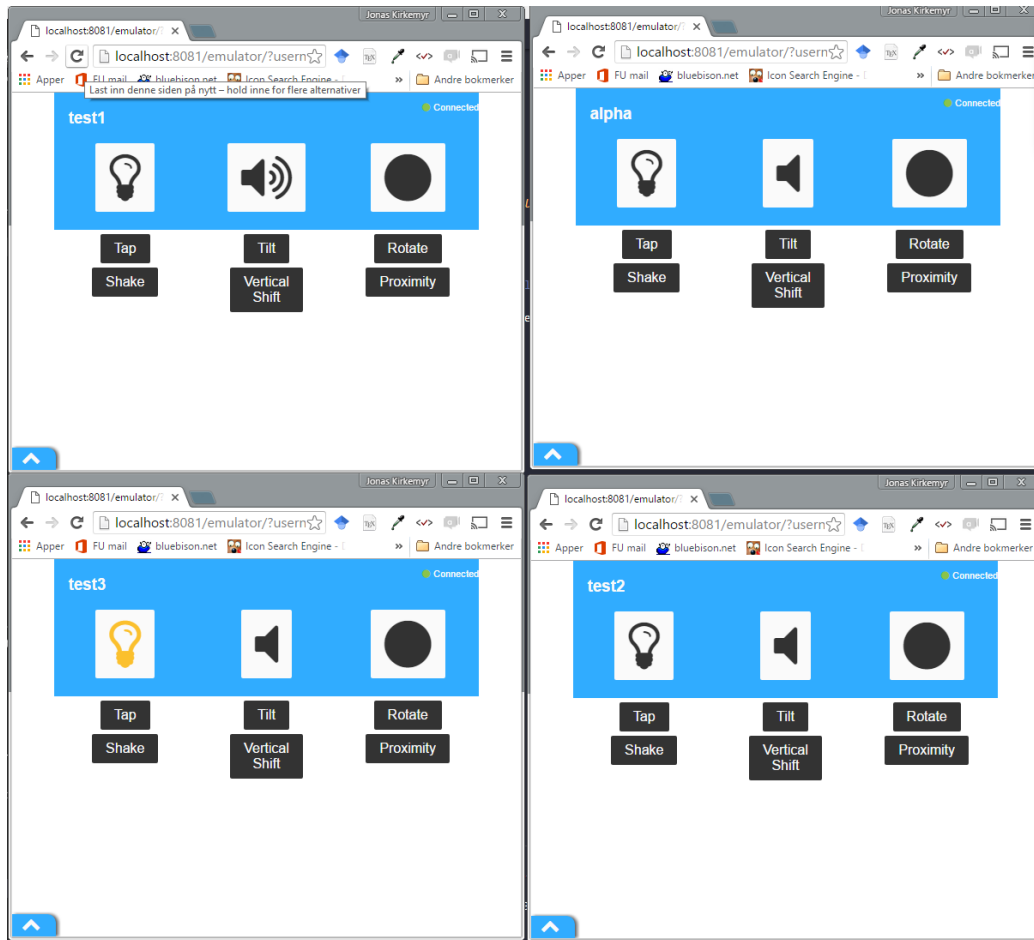


Figure 8.6: WhackAMole using emulators



# Chapter 9

## Conclusion

### 9.1 Results

Multiple evaluations were conducted on the components which makes up the whole development environment. The first evaluation was conducted to help with the implementation of a development environment for TILES, by using the targets users to help create requirements. The second evaluation were conducted to see how the target users were able to use the system, from writing code to make it run in the TILES infrastructure. The third evaluation were done on the DSL itself, including professional programmers, to help find any constraints there may be in the language, and whether it follows standard patterns usually found in other programming languages, making sure the DSL is a simple language to use and learn.

Simplifying the process of creating applications for TILES has been the goal for this work, which have been made possible through a simple development environment, and the use of a Domain-Specific Language, designed for TILES. The results regarding the research section denoted in section 1.3, will be discussed further.

### **9.1.1 MRQ: How to implement a development environment, specialized for makers, and computer science students to easily create applications for TILES?**

A development environment for TILES were successfully implemented, for creating TILES applications. The created applications are able to be run within the TILES infrastructure, communicate with TILES devices through MQTT, and other third-party data-sources over HTTP. Application logic can easily be created around these communication channels, by either running a set of instructions when certain data are received, or sending pre-defined data through a communication channel. The development environment is written in TypeScript, which is compiled to JavaScript, and able to be run in a web-browser. It supports textually and visually approaches for application creation, by using the implemented DSL as a bottom layer for each paradigm.

Simplifying the application creation were done by creating different software modules, which abstract tasks that are difficult to accomplish from scratch. The modules abstracted the communication with the TILES infrastructure, by parsing incoming events from the TILES infrastructure, to easily respond and listen for a single, or a set of events received, and easily control the output interfaces for TILES devices using pre-defined methods, instead of handling raw-data that are supported and understood by the TILES infrastructure. The modules also abstract the communication with third-party services, by implementing a general HTTP-client, and HTTP-clients for more commonly used data-sources, like Facebook, Weather, and Twitter.

### **9.1.2 PRQ1: Which programming paradigm is best fit for makers, and computer science students to create applications for TILES?**

Multiple programming paradigms were evaluated in this work: textually, visually, and physically. For each paradigm, there are multiple style of programming, e.g. visually programming includes both a block and flowchart approach. Each paradigm has its strength-points, depending on the target users, its use, and in what environment. For TILES, textually, visually, and physically approaches could be used, and would be a good fit for the project. Supporting multiple paradigms, more people could start creating applica-



tions for TILES, with an approach they would feel more comfortable with, and fitting their background knowledge. Multiple solutions today for makers and computer science students are usually using a textual approach, but the visual approach is growing in support. The decision of creating a DSL for TILES, were made to make it more easily to implement and integrate other approaches. By having a common language for TILES, the DSL could become a potential building block for other paradigms. This work implemented a fully functional textual approach for creating applications for TILES.

As makers and computer science students most usually have experience in textual programming, this approach were prioritized. The implementation of the DSL, also made it possible to easily support a textual approach of programming, as the DSL parses some text of code, and outputs its AST representation.

### **9.1.3 PRQ2: How should a cross-platform development environment for TILES be designed and implemented for its target users?**

The environment created for developing applications for TILES is running in a browser. This makes it possible to use the environment independent on what platform is used by the makers and computer science students, as a web-browser is only required. By implementing the environment as a web-application, updates are reached to everyone immediately, as the same end-point solution is targeted. A web-application also requires no prior installations on the client-side, so that makers and computer science students are able to start developing applications for TILES without any setup process.

The TILES Development Environment were deployed and running at <http://tilestoolkit.io/dsl/>.

## 9.2 Future work

### 9.2.1 Development environment

#### Functionality

In the development environment it's possible to create textual applications, retrieve data from the TILES infrastructure, and start emulators of existing or virtual TILES devices. In future work, the development environment should be further improved, by adding more functionality. The environment for using a visual block approach is implemented, but requires future work according to implement block representations of the DSL. This should make it possible to not only create textual applications for TILES, but also visual. The ACE text-editor used in the development environment comes with syntax highlighting, and provides functionality for highlighting lines with undefined syntax use. The ACE editor should be extended to support the DSL types, and preferably provide an auto-complete feature to further help in the development process for creating TILES applications. As the DSL is not recognized by the ACE editor, legal syntax in the DSL is highlighted as errors. A syntax highlighter, and an auto-completion feature would benefit makers and computer science students, to write code more quickly, as the editor would suggest keywords as defined by the DSL, and further make sure syntactically correct code is created. An application manager, providing an overview and management of the running applications, and storing all created applications for later alterations, would benefit makers and computer science students by giving them full control for starting, stopping, and replacing running applications. They should at all times be provided with an overview of all their created applications, in which they can continue their previous development. Providing a set of example applications as recipes, in which could be used as a basis for new applications, would make it more easy to get started with TILES application development.

#### Evaluation

Further user-evaluations of the development environment would be beneficial by testing the environment's usability. Providing user-scenarios, a set of instructions a group of people would need to carry out, and observe how they use the development environment, would help finding parts of the system in which is more hard to understand or use. Because of time-constraints, a

closed user-evaluation using user-scenarios were not conducted.

### 9.2.2 DSL

#### Documentation

A simple language specification of the DSL is created (appendix C.2), but as mentioned by one of the participants in the language evaluation in section 8.5, a cook-book could become beneficial for makers and computer science students. A cook-book should provide more examples of the DSL, and benefit as an introduction of the language. For each example, it's important to state what the expected output is, along with minor explanations of the more complex code. Providing a set of recipes in the development environment would also be beneficial for the DSL and could be used as documentation for its use.

#### Extending

The DSL provides simple functionality for communicating with TILES devices. As the TILES device provides more functionality, and possible output interfaces, the DSL should be extended to support these as well. The language could also be extended in supporting more language keyword to accomplish different types of interaction from multiple TILES devices. Support for user-interactions triggered at the same time, and a delay functionality, where the time between user-interactions are triggered are functionality that in future work could be implemented. Also, one of the participants from the language evaluation in section 8.5, were missing support where the order of a sequence of events were not important.

Extension of the DSL capabilities, requires both the parser, code-generator, and DSL-modules to be updated with the new functionality. The DSL-modules needs to implement the feature itself, the parser need to recognize the new keyword and assign it to a type in the AST, and the code-generator needs to be able to generate code from a new AST type.

### 9.2.3 Programming paradigms

The development environment supports creating applications textually, and the foundation of supporting a visual approach. Future work of supporting

the physical programming paradigm would benefit TILES more, and introduce a new target group which are able to interact with physical objects instead of digital tools. The physical approach could be built on top of the DSL, making it more easy to introduce a new programming paradigm to the TILES toolkit, as the fundamental logic is already defined in the DSL.

# References

- [1] craft ai — the maturity of visual programming. <http://www.craft.ai/blog/the-maturity-of-visual-programming/>. (Visited on 02/03/2016).
- [2] Barbara Rita Barricelli and Stefano Valtolina. Designing for end-user development in the internet of things. In *End-User Development*, pages 9–24. Springer, 2015.
- [3] Len Bass. *Software architecture in practice*. Pearson Education India, 2007.
- [4] Andrea Bellucci, Giulio Jacucci, Veera Kotkavuori, Barış Serim, Intiaj Ahmed, and Salu Ylirisku. Extreme co-design: Prototyping with and by the user for appropriation of web-connected tags. In *End-User Development*, pages 109–124. Springer, 2015.
- [5] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. Mini-languages: a way to learn programming principles. *Education and Information Technologies*, 2(1):65–83, 1997.
- [6] Pei-Yu Peggy Chi and Yang Li. Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3923–3932. ACM, 2015.
- [7] Pei-Yu Peggy Chi, Yang Li, and Björn Hartmann. Enhancing cross-device interaction scripting with interactive illustrations.
- [8] David Cuartielles. Opensource hardware and education. In *End-User Development*. Springer, 2015.
- [9] Alexandre Demeure, Sybille Caffiau, Elena Elias, and Camille Roux. Building and using home automation systems: a field study. In *End-User Development*, pages 125–140. Springer, 2015.

- [10] Paloma Díaz, Ignacio Aedo, and Merel van der Vaart. Engineering the creative co-design of augmented digital experiences with cultural heritage. In *End-User Development*, pages 42–57. Springer, 2015.
- [11] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.
- [12] Lars Grammel and Margaret-Anne Storey. An end user perspective on mashup makers. *University of Victoria Technical Report DCS-324-IR*, 2008.
- [13] Saul Greenberg. Toolkits and interface creativity. *Multimedia Tools and Applications*, 32(2):139–159, 2007.
- [14] RG Hague. *End-user programming in multiple languages*. PhD thesis, Citeseer, 2005.
- [15] Juan Haladjian. Tangohapps: an integrated development environment for smart garments. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pages 471–476. ACM, 2015.
- [16] Steve Hodges, Nicolas Villar, James Scott, and Albrecht Schmidt. A new era for ubicomp development. *Pervasive Computing, IEEE*, 11(1):5–9, 2012.
- [17] Seth Holloway and Christine Julien. The case for end-user programming of ubiquitous computing environments. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 167–172. ACM, 2010.
- [18] Van Holm and Eric Joseph. What are makerspaces, hackerspaces, and fab labs? *Hackerspaces, and Fab Labs*, 2015.
- [19] Michael S Horn and Robert JK Jacob. Designing tangible programming languages for classroom use. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 159–162. ACM, 2007.
- [20] Michael S Horn, Erin Treacy Solovey, R Jordan Crouser, and Robert JK Jacob. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 975–984. ACM, 2009.

- [21] Hiroshi Ishii. Tangible bits: beyond pixels. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages xv–xxv. ACM, 2008.
- [22] Thomas Kubitzka and Albrecht Schmidt. Towards a toolkit for the rapid creation of smart environments. In *End-User Development*, pages 230–235. Springer, 2015.
- [23] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development: An emerging paradigm*. Springer, 2006.
- [24] Niko Mäkitalo. Building and programming ubiquitous social devices. In *Proceedings of the 12th ACM international symposium on Mobility management and wireless access*, pages 99–108. ACM, 2014.
- [25] Timothy Scott McNerney. *Tangible programming bricks: An approach to making programming accessible to everyone*. PhD thesis, Massachusetts Institute of Technology, 1999.
- [26] Brad A Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *ACM SIGCHI Bulletin*, volume 17, pages 59–66. ACM, 1986.
- [27] Briony J Oates. *Researching information systems and computing*. Sage, 2005.
- [28] John Pane and Brad Myers. Usability issues in the design of novice programming systems. 1996.
- [29] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.
- [30] Albrecht Schmidt. Programming ubiquitous computing environments. In *End-User Development*, pages 3–6. Springer, 2015.
- [31] Varun Sivapalan and Jonas Kirkemyr. Event-driven infrastructure for the internet of things supporting rapid development. 2015.
- [32] M.Divitini S.Mora, F.Gianni. Tiles: an inventor toolbox for interactive object. *The International Working Conference on Advanced Visual Interfaces AVI*, 2016.
- [33] Daniel Tetteroo and Panos Markopoulos. A review of research methods in end user development. In *End-User Development*, pages 58–75. Springer, 2015.

- [34] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.
- [35] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.
- [36] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.



# Appendix A

## Programming Environments

### A.1 Visual Blocks

#### A.1.1 Scratch

Scratch is a VPL aimed at children for creating games, interactive stories, games, and animations. Scratch have created a community around their product, where users can share and build on others work, to better learn from eachother, or just show off what they have been able to create. The system is developed by "Lifelong Kindergarten" group at MIT Media Lab.

Scratch is run inside a "mini-world" where users can engage on actuators, by either controlling their movement, listening for input from users, output text, create logic for controlling their mini-world, and much more. By using drag and drop, users can easily create a simple application, and modify it as they want. The project aims at appealing to people to use their creativity to program simple application, without needing any programming experience. "Scratch is found to help in learning mathematical and computational concepts, being creative, reason systematically and work collaboratively" - ref(Scratch:Programming for all).

Users can easily start using Scratch right away, and any change occuring in their building blocks, are updated and shown in the mini-world, so users can better understand exactly what teh added building block will trigger and do. With the introduction of Scratch, multiple projects have been inspired by it, and used the same concepts which are found in Scratch.

Scratch is open source.

### A.1.2 ScratchX

ScratchX is an extension of Scratch, that are built for use with embedded hardware like Arduino, and connecting with the Internet for retrieving data from e.g Twitter. ScratchX doesn't have the same focus on the community as Scratch has, mainly because ScratchX is still in the experimental phase.

<http://scratchx.org/#home>

### A.1.3 Snap!

Snap! is an extended project of Scratch, reimplemented with the use of Javascript. Users have the ability of building their own blocks to be used within their application, and can better do more advanced processing, than what is found in Scratch. Snap! also has built in support for classe, found in object-oriented languages. These features requires the users to have more of a technical background, and the focus group is therefore high-school or college students, and not kids. The project is open source.

Snap! is built by the University of California at Berkeley.

<http://snap.berkeley.edu/>

### A.1.4 Blockly

Blockly is a system for building VP editors. The library is written in Javascript, and developed by Google. Blockly is a system that isn't meant for direct use by i.e. children, but a tool for creating systems like Scratch. Blockly requires configurations and development for providing custom-built blocks. Multiple solutions for VP are built with Blockly because it's easily extendable. Code can be created from blocks and exported to both Javascript, Python, and PHP. The project is open source.

<https://developers.google.com/blockly/>

### A.1.5 App Inventor

App Inventor is a block drag and drop tool used for introduction to programming and creating applications for Android. The system focus on users with

no programming experience, to easily create applications with minor effort. App Inventor is created by MIT and Google.

### A.1.6 Ardublock

Ardublock is a language using blocks for creating applications for Arduino, builtin with custom blocks to match functions that are available for Arduino. The system is built with Java, and is made open-source by its authors.

<https://github.com/taweili/ardublock>

### A.1.7 Gameblox

Gameblox is a visual editor that is used for building games with the use of dragging and dropping blocks, like Scratch. The main focus is on creating games, and have therefore blocks customized for this purpose. The system is built on Blockly, and creates games in Javascript, that can be run in any browser. <https://gameblox.org/>

### A.1.8 Scriptr;

Scriptr; is a system that interacts with embedded hardware and other online services for fetching and posting data. Scriptr; supports applicatinos written in plain Javascript, and also have an environment for creating applications with the use of blocks, built on Blockly. With the introduction of Blockly, they provide their end-users to easily create applications for acting upon input created from embedded hardware, without the need of programming experience, and to simplify the process of creating applications for IoT.

<https://www.scriptr.io/home>

### A.1.9 Zipato Rule Creator

Zipato is a smart-home solution, providing different devices for controlling a home. Zipato provides its own smart-phone application, in which can be used to control each device inside a home, from Zipato. Zipato also provides an online visual-editor for setting up rules, in which each device can be controlled from.

[http://www.vesternet.com/resources/application-notes/apnt-9#.VrB1x\\_krJ04](http://www.vesternet.com/resources/application-notes/apnt-9#.VrB1x_krJ04)

# Appendix B

## EBNF visual

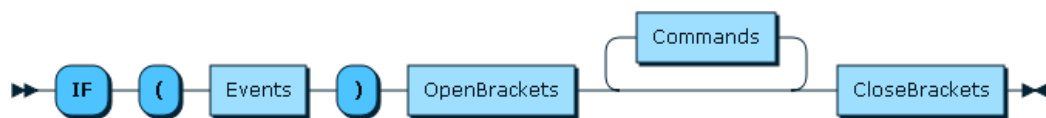


Figure B.1: TILES DSL - EBNF If statement

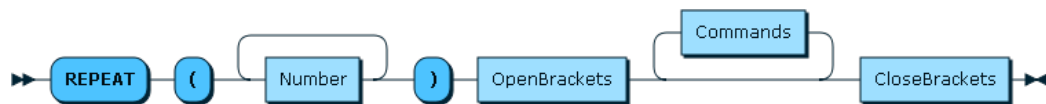


Figure B.2: TILES DSL - EBNF Repeat statement

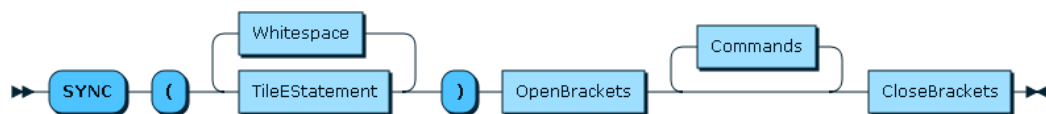


Figure B.3: TILES DSL - EBNF Sync statement

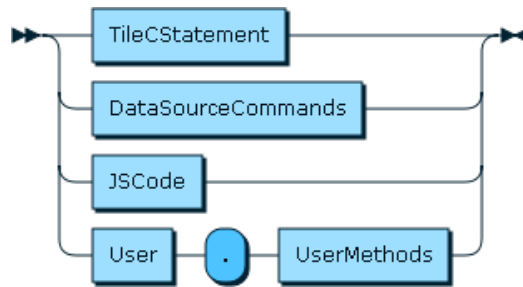


Figure B.4: TILES DSL - EBNF Commands

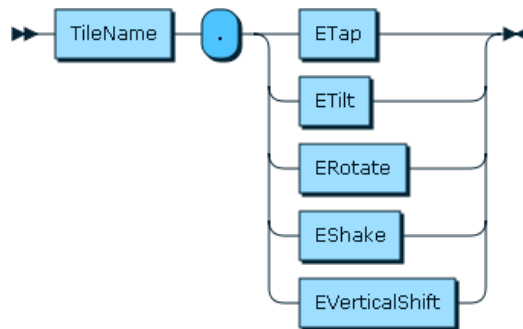


Figure B.5: TILES DSL - EBNF TileEStatement

# Appendix C

## Focus Group

### C.1 Online topics

Topic 1 contained a brief explanation of TILES. A full description of TILES can be found in chapter 3.

**topic 2** Creating applications for TILES is proposed to be a browser-based process, where the users are able to choose either a textual or visual approach.

What is your previous experience of writing code in a browser environment? Can you think of any downsides of creating applications in a browser?

**topic 3** DSL - Domain Specific Language.

The created applications need to be run in the cloud-layer for TILES. By creating a DSL (library), a textual and visual language can be built on top of the DSL, so that both approaches can call functions for controlling the interfaces of TILES (e.g. led, vibrate, speaker) and access third-party services (e.g. twitter, facebook, weather). If the user have multiple TILES devices registered, these can be controlled as well using the DSL.

**Examples:**

- Tile.Alpha.lightOn
- Tile.Beta.lightOn
- var data = twitter.getLatestTweets()

How and when have you been using DSLs/libraries? What would you say is important characteristics of a DSL/library? Have you ever had

any negative experiences with a DSL/library? If so, what didn't work that well?

**topic 4** A textual approach for creating applications for TILES requires calling functions of the created DSL. The cloud-layer is developed using NodeJS (JavaScript), and requires any user-created applications to be written in JavaScript as well.

Along with the available DSL functions, the users are also able to write plain JavaScript code for creating the application logic.

Do you have any experience in using JavaScript, and if so, in what situations? Do you see any cutback with using JavaScript as a programming language to create TILES applications, or in general? What do you think about providing a textual approach for creating TILES applications? (benefits, ease of use etc.)

**topic 5** Creating applications visually is proposed to be done using Blockly (<https://developers.google.com/blockly/>). This is a block/puzzle approach, where different blocks are put together to create applications. Custom blocks can be created to match the DSL, and provides an overview of all the available functionalities. Other block-based approaches for creating applications are 'Scratch', 'Snap!', 'AppInventor'(android)

Do you have any previous experience using a block approach for creating applications, or other visual approaches? What is your thoughts about using blocks to create applications? Do you see benefits or disadvantages?

**topic 6** An emulator is proposed to help with the development of applications, where users are able to interact with a virtual TILES device, to test their applications before publishing it to the cloud-layer.

Have you previously had any experience using emulators for other devices in application development? If so, how did you use the emulator, and was it a positive or negative experience? (please explain)

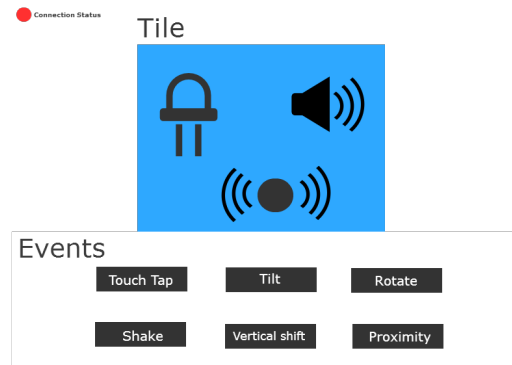
Do you think an emulator for TILES would be of any benefit in a development environment for TILES? Do you have any thoughts about how it could be used?

The attached picture shows example layout of an emulator, and the 'buttons' at the bottom, shows the different user-interactions supported by the TILES device

**topic 7** From the briefly explained project proposals, do you see anything that needs improvements?

Suppose you were in charge of the project, and were able to make one





change that would make the programming environment better. What would you do?

## C.2 Language specification

This document is intended to be an introduction of the TILES DSL. The documentation contains explanation and simple examples describing the features of the language. The TILES DSL is a simple language which supports both plain JavaScript code, and the language features described in this document. The language is compiled to JavaScript, before it's executed and run in the TILES infrastructure.

### C.2.1 Table of contents

1. Starting point
2. TILES Identifier
3. Me
4. Commands
5. Events
6. Statements
  1. If
  2. Repeat
  3. Sync

4. Random
5. Reset
7. Data Sources
8. Example Applications

### C.2.2 Starting point

The layout required by a TILES application is as follows:

```
1 setup(){
2     //my setup
3 }
4 main(){
5     //my application logic
6 }
```

The `setup` method is optional, and can be removed if it's not required by the application:

```
1 main(){
2     //my application logic
3 }
```

Within the setup block, only options for data sources are permitted. Please refer to each data-source for available options. The setup block initializes the applications before running the application logic residing within the main block.

### C.2.3 TILES Identifier

A TILES identifier is a reference to a TILES device registered to your user. The reference to a TILES device is done through a keyword: `TILE`, followed by the name of the TILES device. The identifier can only be used with `nd`. Referring to a TILES identifier is accomplished by the following, where `alpha` is the name of the TILES device which should be referenced:

```
1 TILE.alpha
```

Retrieving all registered TILES is accomplished through the TILES keyword. This will return a nested object, consisting of key and value pairs. The registered name will be the key, and the TILES device, represented as an object, as the value. Each TILES device consist of the following key-value pairs as well: **id**, **active**, **name**, and **state**. The **id** stores the TILES device MAC-address, **active** refers to whether the TILES devices should be made accessible in the TILES infrastructure, **name** the chosen reference name of the TILES device, and **state** contains the latest **event** or **command** retrieved from/to the device.

All TILES can also be stored within a variable for later referencing:

```
1 var myTiles = TILES;
```

The variable `myTiles` will now be a reference to all your TILES devices, containing the following structure:

```
1 {
2   alpha:{
3     id:"123",
4     active:true,
5     name:"alpha",
6     state: {name:'led', properties:['on']}
7   },
8   beta:{
9     ...
10  },
11  cita:{
12    ...
13  }
14 }
```

Accessing the state of alpha

```
1 //will now contain the object: {name:'led',
   properties:['on']}
```

```
2 var alphaState = TILES.alpha.state;
```

## C.2.4 Me variable

Accessing your stored user details are available through the `Me` keyword. Both the name and id are accessible from the DSL language.

```
1 Me.name; //access name
2 Me.id; // access id
3 var myId=Me.id; //store your id in a variable
```

## C.2.5 Commands

Controlling the output interfaces on a TILES device is accomplished by a command. The following output interfaces are currently supported to control: `led`, `speaker`, and `vibrate`.

Each output interface type have their own methods, corresponding to the output interface ability.

An led for a given TILES device can be turned on, turned off, or set to repeatedly be turned on and off. The default color for the led is `blue`. Supported colors are: `red`, `green`, `blue`, and `white`.

```
1 TILE.alpha.led; //default to turning on the led with
  a blue color
2 TILE.alpha.led.on; // default to turn on with a blue
  color
3 TILE.alpha.led.on.green; //turn on led with a green
  color
4 TILE.alpha.led.off; //turn off the led
5 TILE.alpha.blink; //repeatedly turn on and off with
  default color
6 TILE.alpha.blink.green; //repeatedly turn on and off
  with a green color
```

A speaker support start playing a sound, set it to pause, or stop it completely.

```
1 TILE.alpha.play; //play sound
2 TILE.alpha.pause; //pause play
3 TILE.alpha.stop; //stop speaker
```

Vibrate can either be turned on or off for a device

```
1 TILE.alpha.vibrate; //default to on
2 TILE.alpha.vibrate.on; //start vibrating
3 TILE.alpha.vibrate.off; //stop vibrating
```

A variable referencing a TILES device can also be used with the output interface commands:

```
1 var alpha=TILE.alpha;
2 alpha.led.on.green; //turn led on with a green color
   for device alpha
```

## C.2.6 Events

Listening for events triggered on devices are accomplished by an event. Events can only be used within statements, as it requires some logic to be executed when an event is triggered. You have multiple ways of interacting with your TILES device:

- tapped
  - single
  - double
  - hold
- tilted
  - left
  - right

- updown
- rotated
  - clock
  - counter
- shaked
  - horizontally
  - vertically
- shifted
  - lift
  - freefall

Each event includes a more detailed event, for example, tapping the TILES device, can either trigger a **single**, **double** or **hold** event.

```

1 TILE.alpha.tapped //listen for both single, double
  and hold event
2 TILE.alpha.tapped.single //listen for the single
  event
3 TILE.alpha.tapped.double //listen for the double
  event
4 TILE.alpha.tapped.hold //listen for the hold event

```

Please note these events are not supported by themselves, as a command, only in statements!

### C.2.7 Statements

Multiple statements are supported by the TILES DSL, which will be described here

#### If

An **if** statement consist of an event and a corresponding body, encapsulated within brackets { } which will be executed when the provided event

is triggered. Please refer to the events section, to get a list of supported events.

Execute the code within the brackets when TILES device alpha is single tapped:

```
1 if(TILE.alpha.tapped.single){
2   ...
3 }
```

Only a single event is supported for input to the DSL if statement. Operators found in other programming languages like `&&` and `||` are not supported

A more simple if statement is provided as well, which only support an event and a command as input, divided by the `THEN` keyword. Listening for the event will trigger the specified command.

Turn the led on for device beta when alpha is tapped (either single, double, or hold)

```
1 if TILE.alpha.tapped then TILE.beta.led.on.red
```

## Repeat

The repeat statement is a simple loop representation for the well-known `for`-loop.

Repeat body statement 10 times:

```
1 var i=0;
2 repeat(10){
3   ++i;
4 }
5 //i = 10
```

## Sync

The sync statement is a listener for multiple triggering events. The exact order given is the exact order which will be set up for listening.

Beta single tapped, alpha double tapped, beta double tapped:

```

1 sync(TILE.beta.tapped.single TILE.alpha.tapped.
   double TILE.beta.tapped.double){
2   ...
3 }
```

Note that each event is divided by a single space character. Triggering the input events in a reverse order will not execute the statement!

## Random

The random statement returns a random value from a set of parameter passed to it. It supports both objects, arrays and TILES.

```

1 random({"one":"hello", "two":"world"}); //return
   hello or world
2 random(["hello","world"]); //return hello or world
3 random(TILES); //returns a random TILE device
```

The returned value may also be stored within a variable:

```

1 var tile = random(TILES);
```

## Reset

The reset statement will start the application from the beginning of the main entry point.

This sample application will turn the led on for the device alpha, wait for it to be tapped, then turn its led off, and start over again.



```
1 main(){
2   TILE.alpha.led.on;
3   if(TILE.alpha.tapped){
4     TILE.alpha.led.off;
5     reset; // return to start
6   }
7 }
```

### C.2.8 Data sources

Data sources are used for communicating with third-party services. Each data source is here explained, with a explanation of which options are required for them to be setup correctly. Commands to third-party services are sending data, while events are referred to as retrieving data, corresponding with TILES events and command handling.

Because the DSL is compiled to JavaScript, each data source event is requiring a following block of code, which will run when the retrieved data is ready for use.

#### Twitter

The keyword for referring to the Twitter data-source is `Twitter`. The capital T is important for the command to be parsed correctly.

#### Commands:

- `tweet` - For posting a new tweet

```
1 Twitter.tweet("this is my first tweet #helloworld");
```

#### Events:

- `followers`- retrieve a list of your followers
- `following` - retrieve a list of who you're following

The retrieved data is stored within a variable 'twitter'.

```
1 Twitter.followers{
2   twitter.ids; //array containing the id of each
3   follower
4 }
5 Twitter.following{
6   twitter.ids; // array containing the id of each
7   follower
8 }
```

**Options** Required setup for facebook: Accessing the Twitter API is done using `oauth`, and requires 4 different keys.

Required setup for twitter:

```
1 setup(){
2   Twitter.options.consumer_key="my consumer key";
3   Twitter.options.consumer_secret="my consumer
4   secret";
5   Twitter.options.token="my token";
6   Twitter.options.token_secret="my secret token";
7 }
```

Please refer to the twitter application management for creating your keys.

## Facebook

### Commands:

- `post` - post a status to facebook

```
1 Facebook.post("This is my facebook post message");
```

### Event:

- `feed` - retrieves your current feed

- `places` - retrieves your checked-in places

The retrieved data is stored within a variable 'facebook'.

```
1 Facebook.feed{
2   for(var i=0;i<facebook.data.length;++i){//loop
3     through all facebook feeds retrieved
4     facebook.data[i].story; //story for feed
5     facebook.data[i].message; //message for feed
6     facebook.data[i].created_time; //feed creation
7     date
8   }
9 }
10 Facebook.places{
11   for(var i=0;i<facebook.data.length;++i){//loop
12     through all facebook places retrieved
13     facebook.data[i].place; //the place containing
14     it location (lat,lon)
15     facebook.data[i].created_time; //tagged place
16     creation date
17   }
18 }
```

**Options** Required setup for facebook:

```
1 setup(){
2   Facebook.options.apikey="my apikey";
3 }
```

Please refer to the Facebook API for creating a valid api-key.

## Weather

The weather data source only supports fetching weather data, and includes therefore only events.

**Events:**

- **current** - fetch current weather
  - Retrieves a large set of data accessible through its available variable: `temperature mintemp maxtemp humidity cloudiness rain snow weatherInfo sunrise sunset time cityId cityName`
- **forecast** - fetch forecast
  - Retrieves a large set of data accessible through its available variable: `numbOfResult cityId cityName getWeatherInfo(<number>) avgTemp avgMaxTemp avgMinTemp`
- **history** - fetch history weather data
  - Retrieves the same data set as **current**

```

1 Weather.current{
2   weather.temperature;
3   ... //Refer to the event description above for a
4     definition of the data availables
5 }
6 Weather.forecast{
7   weather.avgTemp;
8   ... //Refer to the event description above for a
9     definition of the data availables
10 }
11 Weather.history{
12   weather.temperature;
13   ... //Refer to the event description above for a
14     definition of the data availables
15 }

```

**Options** Required setup for OpenWeatherMap:

```

1 setup(){
2   Weather.options.apikey="my apikey";
3 }

```

Please refer to the OpenWeatherMap API for creating a valid api-key.

## CustomHTTP

The CustomHTTP is a more general HTTP client, with no predefined commands or events. Commands can be triggered by the client through its `command` keyword:

```
1 CustomHTTP.command("post");//post data to set url
```

**Options** Required setup for a custom HTTP data-source client.

The CustomHTTP keyword assigns options to use with a HTTP-request by setting custom data to its option key. The below example shows the structure for assigning data to the custom http. `<myoption>` and `<myvalue>` can here be replaced with anything of your choosing

```
1 setup(){
2   CustomHTTP.options.<myoption> = <myvalue>
3 }
```

Attaching an `apikey` field to the request-data, and specifying its url endpoint:

```
1 setup(){
2   CustomHTTP.options.apikey = "my api key";
3   CustomHTTP.options.url = "my website";
4 }
```

### C.2.9 Example Applications

The following provides multiple example of TILES applications using the described functionality in the DSL.

**WhackAMole:** Turn a random TILES led on, wait for it to be tapped, then turn the led off, and repeat the application. The `repeat` keyword is automatically assigned to the last block of the TILES application, and do not require it to be set manually.

```

1 main() {
2   var tile = random(TILES);
3   tile.led;
4   if tile.tapped then tile.led.off
5 }

```

Turn alpha led on, if current temperature is over 14 degrees. The weather is fetched if a sequence of events are triggered by the alpha and beta TILES respectively:

```

1 setup(){
2   Weather.options.apikey="APIKEY";
3 }
4 main(){
5   sync(TILE.alpha.tapped.double TILE.tapped.double){
6     Weather.current{
7       if(weather.temperature>14)
8         TILE.alpha.led.on.blue;
9     }
10  }
11 }

```

Combine multiple data sources: Here the repeat keyword is required, for the application to be able to listen for all events after its first trigger.

```

1 setup(){
2   Facebook.options.apikey="myapikey";
3   Twitter.options.consumer_key="ckey";
4   Twitter.options.consumer_secret="csecret";
5   Twitter.options.token="token";
6   Twitter.options.token_secret="stoken";
7 }
8 main(){
9   if TILE.alpha.tapped.double then TILE.beta.led.
   green
10  if(TILE.alpha.tapped.single){
11    Twitter.tweet("Tap interaction with the alpha
   device");

```

```
12     repeat;  
13 }  
14 if(TILE.beta.tapped){  
15     Facebook.post("Tap interaction with the beta  
16     device");  
16     repeat;  
17 }  
18 }
```





# Appendix D

## TILES Toolkit

### D.1 DSL Grammar Rules

The listing below shows the grammar rules of the implemented DSL, using PEG.js.

```
1 // TILE DSL
2 OpenBrackets = __ ('THEN' i / '{') __
3 CloseBrackets = __ ('END' i / '}') __
4
5 // Keywords
6 SetupToken = 'setup' i !IdentifierPart
7 MainToken = 'main' i !IdentifierPart
8 FacebookKeyword = 'Facebook' !IdentifierPart
9 TwitterKeyword = 'Twitter' !IdentifierPart
10 WeatherKeyword = 'Weather' !IdentifierPart
11 CustomHTTPKeyword = 'CustomHTTP' !IdentifierPart
12 MeKeyword = 'Me' !IdentifierPart
13 TilesKeyword = 'TILES' !IdentifierPart
14 TileKeyword = 'TILE' !IdentifierPart
15 RandomKeyword = 'RANDOM' i !IdentifierPart
16 ResetKeyword = 'RESET' i !IdentifierPart
17 RepeatKeyword = 'REPEAT' i !IdentifierPart
18 SyncKeyword = 'SYNC' i !IdentifierPart
19
20 TileDSLKeyword = SetupToken
21 / MainToken
```

```

22 / FacebookKeyword
23 / TwitterKeyword
24 / WeatherKeyword
25 / CustomHTTPKeyword
26 / MeKeyword
27 / TilesKeyword
28 / TileKeyword
29 / RepeatKeyword
30 / SyncKeyword
31
32 DsBlock = OpenBrackets __ body:SourceElements? __
    CloseBrackets
33 {return {
34 type: 'BlockStatement',
35 body: optionalList(body)
36 };}
37
38 // Variables
39 FacebookVariable = FacebookKeyword '.' opt:( 'feed'/'
    places') __ body:DsBlock
40 { return{type: 'facebookvar', option:opt, body:body};}
41 TwitterVariable = TwitterKeyword '.' opt:( 'followers
    '/' 'following') __ body:DsBlock
42 { return{type: 'twittervar', option:opt, body:body};}
43 WeatherVariable = WeatherKeyword '.' opt:( 'current'/'
    forecast'/'history') __ body:DsBlock
44 { return{type: 'weathervar', option:opt, body:body};}
45 MeVariable = MeKeyword type:( '.' opt:( 'name'/'id') {
    return opt;})?
46 { return{type: 'mevar', option:type};}
47 TileId = TileKeyword '.' id:IdentifierName
48 { return{type: 'tile', name:id.name};}
49 TilesVariable = TilesKeyword
50 { return {type: 'tiles'};}
51
52 TileDSLVariables = FacebookVariable
53 / TwitterVariable
54 / WeatherVariable
55 / MeVariable
56 / TileId
57 / TilesVariable

```

```

58 / RandomStatement
59 / ResetStatement
60
61 // Options
62 DsOption = 'APIKEY' i __ '=' __ val:StringLiteral
63 { return val; }
64 TwitterOption = 'CONSUMER_KEY' i __ '=' __ val:
        StringLiteral
65 { return {type: 'ConsumerKey', init:val}; }
66 / 'CONSUMER_SECRET' i __ '=' __ val:StringLiteral
67 { return {type: 'ConsumerSecret', init: val}; }
68 / 'TOKEN' i __ '=' __ val:StringLiteral
69 { return {type: 'Token', init:val}; }
70 / 'TOKEN_SECRET' i __ '=' __ val:StringLiteral
71 { return {type: 'TokenSecret', init: val}; }
72
73 OptionList = head: Option EOS tail: ( __ Option EOS)*
        {return buildList(head, tail, 1); }
74 Option = FacebookKeyword '.options.' i val:DsOption
75 { return{type: 'FacebookAPI', init:val}; }
76 / TwitterKeyword '.options.' twitter:TwitterOption
77 { return twitter; }
78 / WeatherKeyword '.options.' i val:DsOption
79 { return{type:'WeatherAPI', init:val}; }
80 / CustomHTTPKeyword '.options.' i val:
        AssignmentExpression
81 { return {type:'CustomHTTP', init:val}; }
82
83 OptionBlock = OpenBrackets __ body:OptionList? __
        CloseBrackets {
84 return optionalList(body);
85 }
86
87 // Events
88 TileTapped = tile:(TileId/Identifier) '.' 'tapped'
        type:( '.' type:( 'single'/'double'/'hold') {return
        type;})?
89 { return {type:'TileEvent', event:'tap', tile:tile,
        param:type}; }
90 TileTilted = tile:(TileId/Identifier) '.' 'tilted'
        type:( '.' type:( 'left'/'right'/'updown') {return

```

```

    type;})?
91 { return {type:'TileEvent', event:'tilt', tile:tile,
    param:type}; }
92 TileRotated = tile:(TileId/Identifier) '.' 'rotated'
    type:( '.' type:( 'clock'/'counter') {return type
    ;})?
93 { return {type:'TileEvent', event:'rotate', tile:
    tile, param:type}; }
94 TileShaked = tile:(TileId/Identifier) '.' 'shaked'
    type:( '.' type:( 'horizontally'/'vertically') {
    return type;})?
95 { return {type:'TileEvent', event:'shake', tile:tile
    , param:type}; }
96 TileShifted = tile:(TileId/Identifier) '.' 'shifted'
    type:( '.' type:( 'lift'/'freefall') {return type
    ;})?
97 { return {type:'TileEvent', event:'vertical_shift',
    tile:tile, param:type}; }
98
99 TileEvent = TileTapped
100 / TileTilted
101 / TileRotated
102 / TileShaked
103 / TileShifted
104 TileEvents = head: TileEvent tail:(__ TileEvent)*
105 { return buildList(head,tail,1); }
106
107 // Commands
108 TileColor = color:( 'red'/'green'/'blue'/'white'/'i)
109 {return {type:'TileColor', color: color};}
110 TileLedCommand = tile:(TileId/Identifier) '.'
    command:( 'led'/'blink')
111 param:(
112 (color:( '.' color:TileColor {return color;}))
113 /(status:( '.' status:( 'on'/'off') {return status;}))
114 )?
115 EOS
116 { return {type:'TileCommand', tile:tile, command:{
    type:'led',command:command}, param:param};}
117 TileSpeakerCommand = tile:(TileId/Identifier) '.'
    command:( 'play'/'stop'/'pause') EOS

```

```

118 { return { type: 'TileCommand', tile: tile, command: {
      type: 'speaker', command: command } }; }
119 TileVibrateCommand = tile: (TileId/Identifier) '.'
      command: 'vibrate' param: ('.' param: ('on'/'off')) {
      return param; }? EOS
120 { return { type: 'TileCommand', tile: tile, command: { type:
      : 'vibrate', command: command }, param: param }; }
121 TileCommand = TileLedCommand
122 / TileSpeakerCommand
123 / TileVibrateCommand
124
125 DataSourceCommand = key: FacebookKeyword '.' 'post' '
      (' __ val: StringLiteral __ ') EOS
126 { return { type: 'DataSourceCommand', ds:
      extractOptional(key, 0), method: 'post', value: val
      }; }
127 / key: TwitterKeyword '.' 'tweet' '(' __ val:
      StringLiteral __ ') EOS
128 { return { type: 'DataSourceCommand', ds:
      extractOptional(key, 0), method: 'tweet', value: val
      }; }
129 / key: CustomHTTPKeyword '.' 'command' '(' __ val:
      StringLiteral __ ') EOS
130 { return { type: 'DataSourceCommand', ds:
      extractOptional(key, 0), method: 'command', value:
      val }; }
131
132 DslCommands = TileCommand / DataSourceCommand
133
134 // Statements
135 DSLIfStatement = IfToken '(' __ event: TileEvent __ '
      )' OpenBrackets body: SourceElements?
      CloseBrackets
136 { return { type: 'DslIf', event: event, body:
      optionalList(body) }; }
137 DSLIfStatementSimple = IfToken __ event: TileEvent __
      'then' i __ command: TileCommand
138 { return { type: 'DslIfSimple', event: event, command:
      command }; }
139 RepeatStatement = RepeatKeyword '(' __ number:
      NumericLiteral __ ') OpenBrackets body:

```

```

    SourceElements? CloseBrackets
140 {return {type:'Repeat', number:number, body:
    optionalList(body)}};}
141 SyncStatement = SyncKeyword '(' __ events:TileEvents
    __ ')' OpenBrackets body: SourceElements?
    CloseBrackets
142 {return {type:'Sync', events:events, body:
    optionalList(body)}};}
143 RandomStatement = RandomKeyword '(' __ random:(
    TilesVariable/ObjectLiteral/ArrayLiteral) __ ')'
    EOS
144 { return {type:'Random', random:random}; }
145 ResetStatement = ResetKeyword EOS?
146 { return {type:'Reset'}};}
147 DslStatement = DSLIfStatement
148 / DSLIfStatementSimple
149 / RepeatStatement
150 / SyncStatement
151
152 // DSL Program
153 MainBlock = OpenBrackets __ body: SourceElements? __
    CloseBrackets {
154 return {
155 type: 'BlockStatement',
156 body: optionalList(body)
157 };
158 }
159
160 Setup = SetupToken __ '(' ')' __
161 consequent: OptionBlock __{
162 return{
163 type: 'OptionStatement',
164 body:consequent
165 };
166 }
167 Main = MainToken __ '(' ')' __
168 consequent: MainBlock{
169 return{
170 type:'MainStatement',
171 consequent: consequent
172 };}

```

```
173  
174 TileProgram = Main / Setup Main  
175 Program = TileProgram
```

Listing D.1: DSL GrammarSmall

## D.2 Primitives

The table below list the different input and output primitives that should be supported by the TILES device, and the TILES infrastructure.










Input Primitives			
Primitive	Degrees of freedom	Example of mapping	
Touch/Tap	Single, double, touch-and-hold	Send a command, log a quantity	
Tilt	Left, right, upside-down	Select a function, binary switch	
Rotate	Clockwise, counterclockwise	Modify a quantity, select an option	
Shake	Horizontally, vertically	Throw a random option, discard a command	
Vertical shift	Lift, freefall	Detect presence, detect falling	
Proximity	to the LEFT,RIGHT, TOP or BOTTOM side of another TILE-based object, to one hand	Exchange resources between two objects	
CUSTOM	See extensibility/hacking section		
Output Primitives			
Primitive	Degrees of freedom	Example of mapping to digital functions	
LED Light feedback	Color shift, blink, fade	Continuous notification about the status of process	
Haptic feedback	Vibration pattern	Discrete notification about the status of process	
Sound feedback	Sound <u>patter</u>	Discrete notification about the status of process	
CUSTOM	See extensibility/hacking section		

Figure D.1: Interaction primitives for TILES. Source: Simone Mora



## **D.3 DSL Class-Diagram**

The following section show the UML-class diagram of the combined DSL modules.

