Norwegian University of
Science and Technology

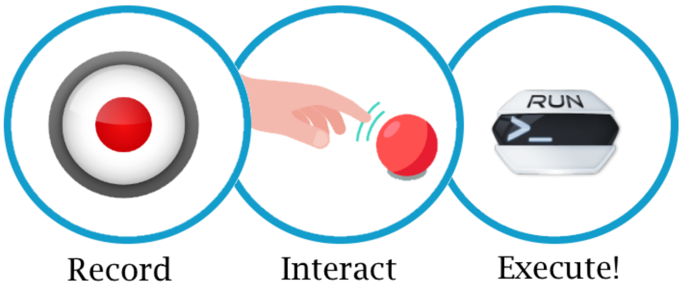
Exploring Programming Paradigms with IoT and Tiles for End-Users

Daniel Alexander Satcher

Master in Information Systems
Submission date: June 2017
Supervisor: Monica Divitini, IDI
Co-supervisor: Simone Mora, IDI

Norwegian University of Science and Technology
Department of Computer Science

Exploring Programming Paradigms with IoT and Tiles for End-Users



Daniel Alexander Satcher

Supervisor: Monica Divitini
Co-Supervisor: Simone Mora

Norwegian University of Science and Technology
Trondheim, Norway

Spring Semester 2017

Abstract

The Tiles project is an Internet of Things (IoT) toolkit designed to help developers create applications by providing a common set of hardware, a common application server hosted in the cloud, and a common gateway to connect devices to the cloud. With Tiles, one can turn everyday objects into smart objects that can be interacted with in meaningful ways. Sunglasses can be tilted to save a marker of your location in Google Maps or a vase could be tapped to the control the lights in a room, for instance. However, with IoT being a new field in general along with Tiles, there is a need to bring Tiles development to many users, some of which have little or no experience in either programming or programming for IoT. To address this problem, this paper examines the contemporary work done in end-user development as it relates to IoT. Specifically, a focus has been concentrated around different programming paradigms for developing applications such as textual, visual, and physical/-tangible programming paradigms. This research into programming paradigms is then applied to Tiles to find a new, innovative approach to end-user development. Finally, this research presents a prototype of *Tiles Recorder!*, a tangible and visual programming application that follows the mantra "Record, Interact, Execute!" in order to simplify development for Tiles using a rule-based engine.

Acknowledgments

I'd like to first thank my supervisors for both their feedback and support during this research. Without them providing guidance and pushing for more innovative solutions, this work would not have been possible. I'd also like to recognize the efforts of researchers working in the field of end-user development whose knowledge and expertise were leveraged in creating Tiles Recorder. Finally, I'd like to thank the users who volunteered to test the system on their own time. Their contribution will make future versions of Tiles Recorder more user-friendly and feature-rich.

Contents

List of Figures

List of Tables

1 Introduction

The following chapter will briefly introduce the concepts regarding this project to the reader. This includes the problem definition, motivation, research method and the research questions that this project serves to answer. Furthermore, an overview of the paper's remaining structure will be provided.

1.1 Problem Definition

The Internet of Things (IoT) can be defined as a network of sensors and actuators that are used to measure responses from the environment and react to these stimuli. These reactions can range from sending messages about environmental data (temperature, wind, audio levels, and user interaction for instance) to sending messages about the state of the sensor itself or the ecology of connected sensors. This interplay between sensing the environment and actuating upon stimuli creates a ubiquitous "computing everywhere" [10] environment where technology disappears into the background. With ubiquity, users no longer have to consciously interact with abstract interfaces to control their surroundings, but can use smart objects instead.

Smart objects are everyday objects embedded with network sensors (both physically embedded or logically embedded by a network) that are able to respond to changes in their environments [18]. A common example of a smart object would be smart lights, such as Philips Hue, which can turn off/on, change color or blink based on movement, temperature changes and various other stimuli. Ecosystems of smart objects act as the foundation for IoT and using them in novel ways can provide solutions for many challenges both small and large in scale. As an example, one could use a smart faucet to detect household water consumption, while a network of the same technology could provide municipal water companies an idea of average water use per home for different parts of the city.

However, for those uninitiated in the world of IoT, creating such applications is a complex, difficult task without the aid of toolkits. The Tiles toolkit is one such suite of tools that aims to holistically assist users with rapidly designing prototypes and developing applications for interactive objects. This is accomplished by providing common hardware shared by all interactive objects in the system through the use of a tile that houses input sensors and output actuators. The tile itself understands a set of interaction primitives, which mainly deal with physical interaction (shaking, rotating, and tapping for instance). These tiles are then attached to common objects, which are used as tangible user interfaces (TUIs) [16]. Next, the design process is simplified through the use of Tiles Cards, a card game created to facilitate creative thinking for IoT applications with an emphasis on interactions between smart objects and data streams [22]. Last, the toolkit provides a library in several programming languages to help guide developers who are unfamiliar with IoT programming. Included in the toolkit is a common cloud for running developed applications as well. More information on Tiles can be found in chapter 2.

While the hardware, software libraries, cloud hosting, and card game have been favorably evaluated by their respective target groups, there is a gap in knowledge regarding how end-users can transition from end-user design to end-user development of Tiles applications. Tiles Cards itself is useful as a tool for end-users to creatively design an application that uses objects, data streams, and interaction primitives, but during workshops with non-developers there has been noted difficulty when moving from the card game to coding. This has currently been attributed to the technical barriers that textual programming raises and conceptual differences between programming for sensors versus the cards. For instance, when programming using the provided libraries, a large focus is on how the sensor/actuator Tile responds to user input, while the actual card game uses smart objects as the interface instead of the tile directly.

This research effort will focus on how end-user development paradigms differ between Tiles and other IoT toolkits and how this paradigm changes what tools, languages, and programming metaphors are needed to make the transition from end-user design to end-user development easier for Tiles. Tiles itself is unique in that the focus is on how users generate interactions between multiple smart objects and how these interactions coalesce into an application that solves a specific problem. Other toolkits, such as Arduino, don't focus on how the user can combine different interactions with objects, thus creating a need for an expanded paradigm for IoT end-user design.

1.2 Research questions

From the above problem definition, a main research question (MRQ) along with supporting sub-questions (SRQs) can be formulated as such:

MRQ - What development paradigm(s) can greatest assist end-user development for Tiles in an innovative way?

SRQ1 - What aspects of Tiles are unique for its domain in IoT and for application development?

SRQ2 - How do current tools and programming paradigms fulfill the requirements for development with Tiles?

SRQ3 - How can the paradigms best be used to create a tool or language prototype for Tiles?

1.3 Methodology

The research methodology for this effort has been primarily focused on:

- Literature review
- Prototyping
- Observation/group interview

According to Rowley [29], literature review "identifies and organizes the concepts in relevant literature." Literature reviews are therefore helpful in understanding the history of knowledge regarding a field of study and exploring the state-of-the-art in the field. Building a knowledgebase of previous work is important in order to identify gaps in knowledge present in the current literature and to avoid reproducing studies that have been proven sound. Chapter 3 delves through the literature for this paper and serves as the knowledgebase to develop a prototype involving different programming paradigms.

Prototyping is an essential part of design research, which is "a pragmatic research paradigm that calls for the creation of innovative artifacts to solve real-world problems."[12] In this case, the artifact created is a software prototype that can be tested by users in order to enhance our knowledgebase regarding the problem and to test the effectiveness of the prototype in the field. Hevner, in an attempt to expand upon his design research theory, created a three-cycle view of this research paradigm represented in figure 1.1 [13] below.

The first of the cycles, known as the relevance cycle, concerns itself with how the artifacts created meet the requirements of the users in the field. Real-world problems are used to create requirements that are used for either the initial design artifact or to add to the functionality of an existing artifact. Typically, human actors are involved in this cycle, although occasionally software or hardware is an actor that is being tested. An artifact used for testing the latency between between two communication protocols over a wide network would be an example of an artifact that is tested using a technical actor.

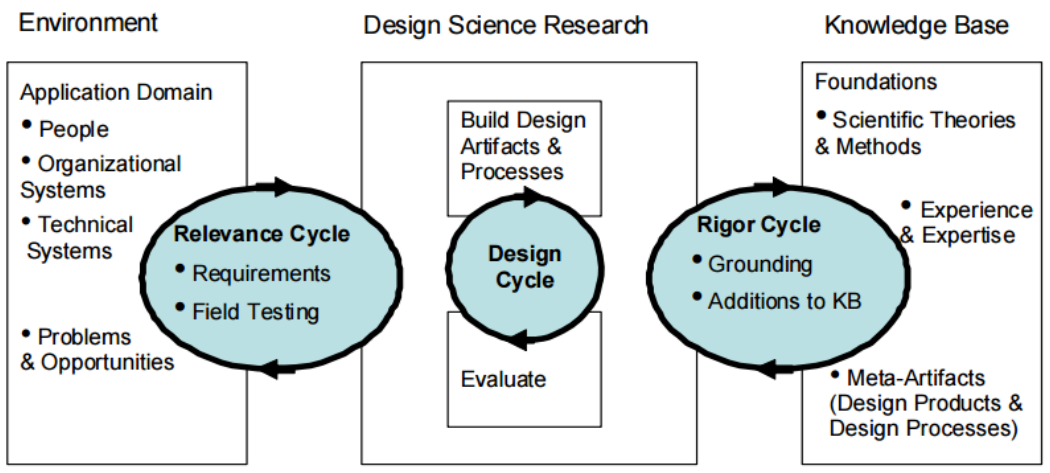


Figure 1.1: Design science cycles

The design cycle is the main activity is design science research. Both the relevance cycle and the rigor cycle contribute to the overall design of the artifact and the evaluation of the artifact. The artifact itself can also affect the rigor cycle by furthering the knowledgebase surrounding the artifact. Showing that one communication protocol is faster than another adds to our knowledgebase about communication protocol speeds, for instance.

The last cycle is the rigor cycle. This cycle concerns itself with how the existing knowledge about a subject can used to ground the design of an artifact and how the artifact itself increases our overall understanding of a subject. Therefore, the meta-artifacts mentioned in the diagram are an essential component of this cycle. These artifacts can influence the overall design of the implemented artifact.

To tie everything together, imagine that a researcher wanted to prove that visual programming was quicker than textual programming for low-complexity projects. The researcher may start by interviewing developers in the industry (relevance cycle) and/or reading the current literature regarding the subject (rigor cycle). From this research, he or she can create a list of requirements for the main artifact and ground the design of the artifact based on the accumulated knowledge provided by the rigor cycle. After designing and building the prototype, the researcher would first evaluate the artifact (a software prototype) internally with user testing before setting out into the field to observe how quickly developers can code with a visual language compared to a textual one. Finally, the researcher would take his findings and modify the artifact as well as document his results into the collective knowledge about visual programming.

This research effort lies primarily in the design science research and knowledge base sections of the diagram. The literature review serves as the rigor grounding the design of the prototype, while the user testing is part of evaluating the design artifact in the design science research section.

Observation and group interviews were used during user testing in order to measure the intuitiveness of the prototype and to gather opinions on the overall concept. Due to the subjective nature of user-experience, these approaches were taken over other more quantitative approaches such as surveys. With observation one can see the struggles users have while using the prototype in real-time and do not have to rely solely on the user's ability to articulate themselves [28]. More information on user testing of the prototype can be found in chapter 7.

1.4 Results

As a result of this research, a literature review of end-user development for IoT has been undertaken and a prototype has been designed, implemented, and user tested. Chapter 3 presents the literature studied for this paper while chapter 4 relates the literature to the Tiles platform. The prototype created is a new foray into tangible programming for Tiles and allows future researchers a unique opportunity to continue researching tangible programming and IoT. The design of the prototype is presented in chapter 5, implementation in chapter 6, and testing results in chapter 7. Based on the feedback from the user testing, it can be deduced that the tangible aspect of Tiles Recorder has generated some excitement. Users felt that programming the rules using actual sensors was quicker and more intuitive than creating a listener in code. The next steps based on these findings is presented in chapter 8.

2 Tiles

This chapter will give a brief introduction to the Tiles system as well as the Tiles card game, which was used when designing the look and feel of the prototype for Tiles Recorder.

2.1 What is Tiles?

Tiles is an IoT toolkit that serves to simplify building IoT applications by providing:

- Tiles squares – Standard hardware
- Tiles Gateway – Communication Protocol
- Tiles Cloud – Application server

2.1.1 Tiles squares

Tiles squares provide a standard set of hardware that is consistent for all Tiles applications. This simplifies the development process by freeing the developer from having to worry about the compatibility or capabilities of different IoT devices in a network. The primary use for tile squares is to be attached to everyday objects that be interacted with using *interaction primitives.* Interaction primitives are simple physical interactions that can be woven into a more complex sequence of interactions. In addition to interactions, tile squares have a standard set of output devices for responding to the user. Table 2.1 summarises the types of available interaction primitives and outputs for each tile square.

Input/Output	Description
Tilt	Fires when the gyro-meter passes a threshold
Tap	When a tile is tapped - can be differentiated by single and double taps
LED	Output. Turns off, on, blinks or fades. Can be green, red, blue, or white in color
Haptic Feedback	Vibration of the tile in short burst or one long vibration

Table 2.1: Interaction/outputs for tile squares [21]

2.1.2 Tiles cloud

Built upon NodeJS, Tiles Cloud is an application server that executes applications written for Tiles [20]. The cloud is also responsible for handling user profiles as well as the status of tile squares in the field. This centrally managed approach is becoming common in IoT as opposed to having application logic on the hardware devices themselves [11].

2.1.3 Tiles gateway

In order to transmit messages between tiles squares and the application server, a gateway must act as relay between the short-range Bluetooth communication on the tiles squares and Tiles Cloud on the internet. The gateway itself is a mobile application that receives messages from the tiles squares and sends them to the cloud server via an MQTT queue. MQTT is a resource-inexpensive messaging protocol that has been prevalent in IoT due to the power-efficiency demands of IoT devices.

2.2 Tiles Card Game

The tiles card game is a design activity meant to engender creativity when designing IoT applications [22]. The game takes essential IoT concepts and concepts unique to Tiles and divides them into sets of cards than can be arranged during the course of the game in unique ways. The categories for the cards are:

- Things
- Data Channels
- Feedback
- Human actions

Things represent the objects that tile squares are attached to while human interactions include the previously mentioned interaction primitives. The feedback cards deal with the output devices on the tiles and the data channels represent abstract data sources such as Twitter, Google Maps, banking web services, etc. The interaction and feedback cards also correlate with the interaction primitives and outputs of tile squares. There are cards for tapping, tilting, lights blinking, and vibration, for instance.

A typical game will involve players selecting a problem domain and group of target users. Smart city travel, managing home environments, recreational activities are common examples of problem domains that could be chosen. Players then choose a set of things cards that they wish to implement in their application. In the smart city domain, an example set of things used could be a bicycle and a car. Next, players choose data sources and human interactions to act as triggers for response events. In our example, players could choose tapping for the bicycle and car to signal the begin/end of a trip the bicycle/car. Data channels and feedback can be used for event responses. Choosing an online web service that estimates the fuel saved by choosing the bike would be an example of an event fired after tapping the bike twice. Players then evaluate their solution and can continue playing rounds to find a varied set of solutions.

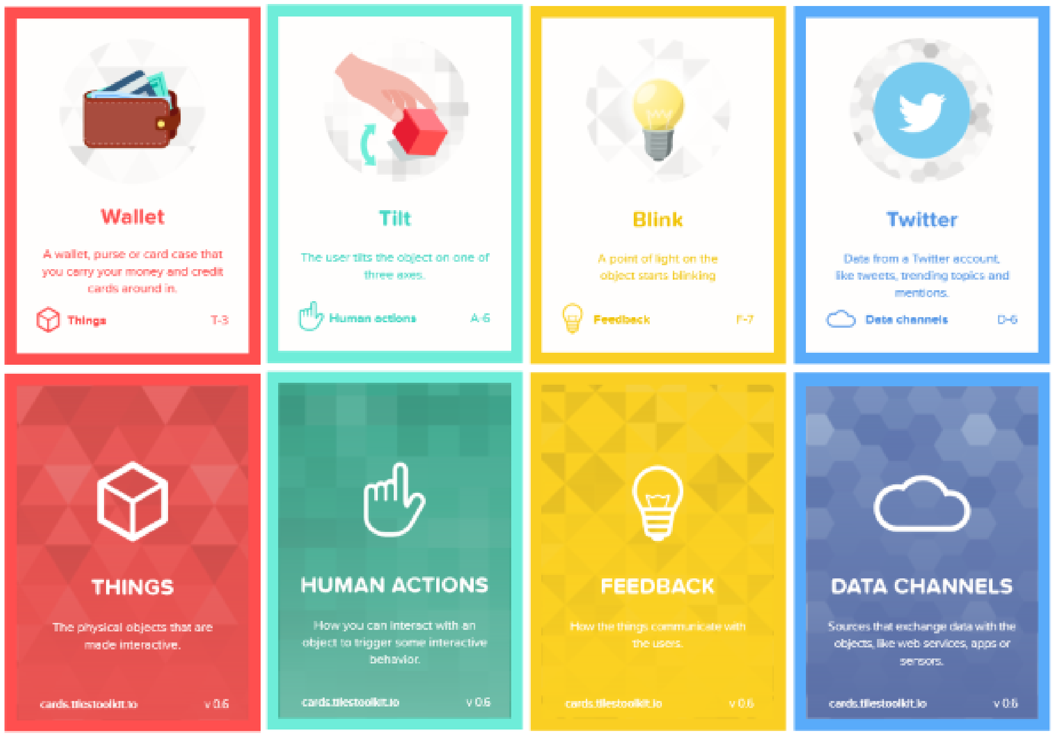

Wallet
A wallet, purse or card case that you carry your money and credit cards around in.
Things
T-3
Tilt
The user tilts the object on one of three axes.
Human actions
A-6
Blink
A point of light on the object starts blinking
Feedback
F-7
Twitter
Data from a Twitter account, like tweets, trending topics and mentions.
Data channels
D-6
THINGS
The physical objects that are made interactive.
cards.tilestoolkit.io
v 0.6
HUMAN ACTIONS
How you can interact with an object to trigger some interactive behavior.
cards.tilestoolkit.io
v 0.6
FEEDBACK
How the things communicate with the users.
cards.tilestoolkit.io
v 0.6
DATA CHANNELS
Sources that exchange data with the objects, like web services, apps or sensors.
cards.tilestoolkit.io
v 0.6

Figure 2.1: Examples of cards from each category

For this paper, the tiles card game is relevant for the design of the system's UI. Synergising the two will allow for users to easily transition from design activities to development activities with Tiles Recorder.

3 Literature

3.1 End-User IoT Development

In section 3.2, a definition for an *end-user* has been provided along with argumentation for why end-user development is rapidly becoming more pressing for new IoT applications. This chapter will present findings from previous work in end-user development with a focus specifically for IoT. The intent of this literature review is to understand where the body of academic research in this field has led to and how this research may further understanding of end-user development for IoT.

3.2 Defining the End-User for End-User Development

End-user development, a term with a seemingly self-evident definition, is somewhat difficult to define and its meaning has been refined over the years. In order to get a better understanding, one must first accept a common definition for what an end-user is and, since some level of skill is involved in development no matter the tool used, what their capabilities are. An earlier way of looking at an end-user would be to see them as a "user of an application program" that "uses a computer as part of daily life or work, but is not interested in computers per se." [6] However, as computing has become more pervasive, looking at the end-user as an isolated entity using a pre-defined program for one specific task has become outdated [31].

End-users are no longer those who just consume data, they are also entities that create, tailor, and share data to other entities in a complex system. In this way, some applications, especially those for IoT, can begin to see the end-user as both a source and endpoint for data. Furthermore, end-users are becoming increasingly aware of their ability to tailor applications to their domain knowledge. As a basic example using a house heating app, end-user John has domain knowledge over what temperatures he prefers and what times throughout the day he will occupy his home. Thus, John can tailor the house heating application to only turn on during the hours John will likely be at home and to which specific temperature he prefers. A designer or isolated developer would not know these domain details and could not possibly design or program an application that inherently accounted for John without John's interest and input into the system.

Concerning skill and interest, these two factors have also changed the definition of an end-user through time. Interest in end-users to become agents who can modify a system has increased as technology has become more social and tailored towards solving problems in both professional and personal domains (more about 3rd wave HCI in the next section). However, end-users can be thought of as having more interest in what devices and applications can do for them rather how they accomplish this in technical terms. While end-users could be programmers, the general consensus is that, for the purposes of defining and end-user, they possess a lack of interest in imperative coding, or mandating how exactly an application should perform its duties in a technical sense. Nardi gives an apt definition of an end-user as someone who "does not want to turn a task into a programming problem..." [1].

By looking at the willingness of users to participate in a system versus their skill level in programming, we can therefore collate these disparate definitions come to a useful definition of our own for an end-user. An end-user, for the purposes of end-user development, can be defined as

> " A person with personal domain knowledge and an interest in modifying existing or creating new software for purposes of tailoring that software for their needs without running into the technical barriers of programming "

3.3 Defining End-User Development

3.3.1 EUD general concepts

Provided below is a list of terms relating to EUD to facilitate the understanding of concepts presented later in the research:

adaptability, extensibility
How easily a system can be modified or how easily new functionality can be created and used by a system. Examples of a highly adaptable system would be Google Chrome's extension widgets.

meta-design
Software design techniques to allow for greater extensibility of a system by an end-user. Examples meta-design would include creating *platforms* that execute highly de-coupled software modules or providing simple interfaces to mix system functionality to create emergent functionality.

low-threshold, high-ceiling [23]
: The ease of use for a design tool and the amount of expressive complexity afforded by that tool. This can be similarly referred to as cost-of-learning and scope mentioned previously.

cognitive dimension framework
: A set of design principles created by Green and Petre [9] to evaluate the usability of a system or programming language. Contains factors such as viscosity, abstraction gradient, visibility and others to assess programming design decisions. More detailed information provided in Chapter 4.

domain-oriented design environments
: Programming environments fixed to a specific domain that typically follow a visual programming paradigm [7].

End-user development can take many forms depending on the level of modification that an end-user wishes to impart on a system. A system with a high level of modifiability by the end-user is referred to as adaptable by Trigg et al [34]. Trigg mentions that systems can be adaptable in four different ways:

- System provides generic, abstract objects that can be utilised differently by different users
- System provides parameters standard behaviors that can be customised by users supplying preferred parameters
- System can be interfaced with easily by other facilities in the environment
- System can be tailored by adding functionality or specialising behavior

Mørch [2] takes the tailoring aspect and furthers it by creating three different classes of tailoring: customisation, integration, and extension. Customisation is the first class of tailoring put forth by Mørch. This tailoring activity typically involves the look and feel of interfaces that fills the gap between the presentation layer and the underlying implementation code. Users may also edit the attributes associated with those interaction elements (such as providing a max range for a slider, for example). Integration regards how end-users add previously existing functionality to the system while extension is adding completely new functionality to a system.

However, Costabile et al [5] provides a simpler classification system that divides end-user activities into two classes. The first class deals primarily with parametrisation. Users supply parameters to affect system behavior that will ideally match their needs. The second class is a much more involved set of activities that allows the user to modify actual software functionality or create new pieces of software. This can be accomplished with tools provided by the software itself or by a programming interface specifically tailored for it.

More recently, Ko et al has divided EUD based on whether or not users are creating new pieces of software or modifying existing software. End-user development is generally referred to as encompassing both activities, with Ko et al defining end-user programming (EUP) as end-users specifically creating their very own programs. End-user software engineering (EUSE) is another term that appears often in the literature as more contemporary reaction to poorly performing end-user applications. EUSE is primarily concerned with how to teach end-users good programming practices while balancing the amount of effort needed by the end-user to create efficient programs. Figure 3.1 has been provided as a reference for the different levels of EUD.

There are many paradigms to enable end-users to develop new software artifacts without needing a deep understanding of the system's implementation code or knowing how program in a general purpose language. Ultimately, as explored by Fischer [8], the EUD sweet spot is to have technology that has a low cost of learning with a high scope of application. Figure 3.2 provides a trade-off diagram for the cost of learning versus the scope of work that the technology can cover. Different programming paradigms can assist in lowering the cost of learning and are further explained in the section 3.4.

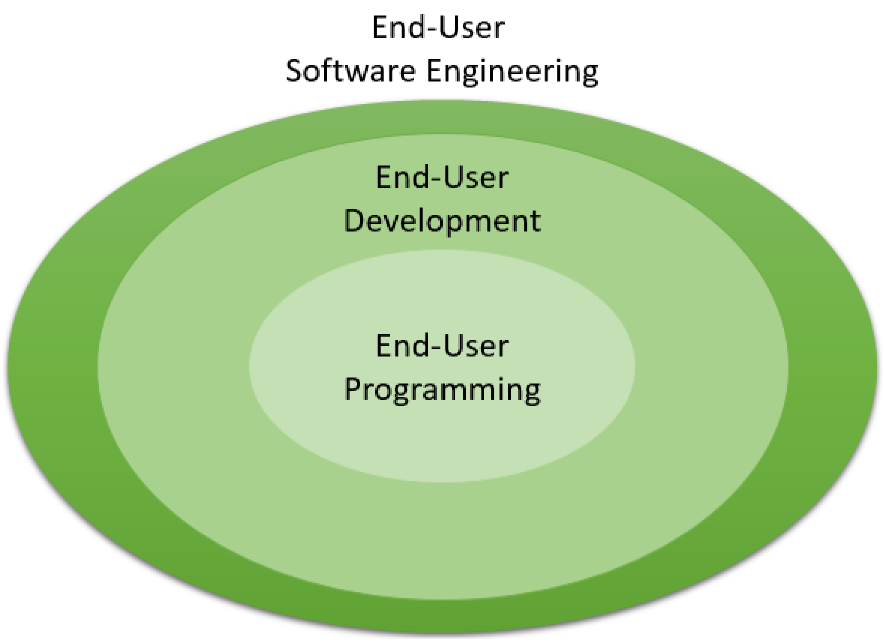


Figure 3.1: Three levels of EU-X

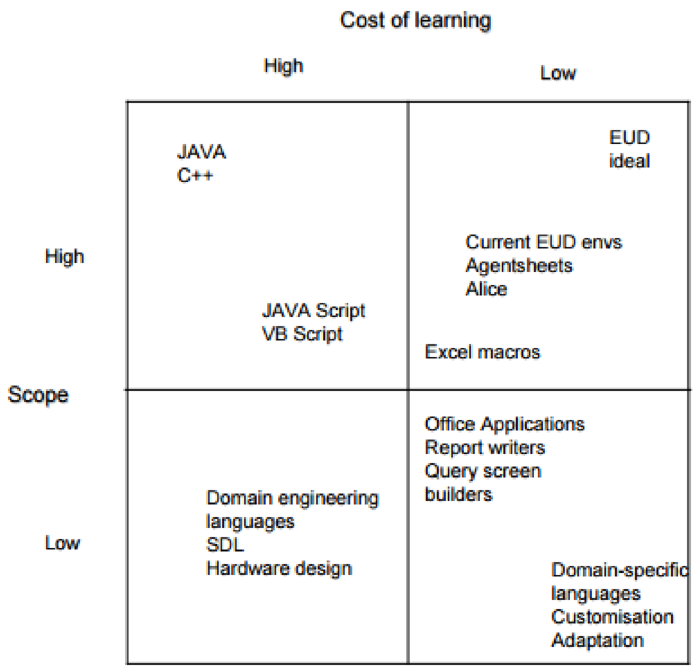


Figure 3.2: COL and scope trade-offs [8]

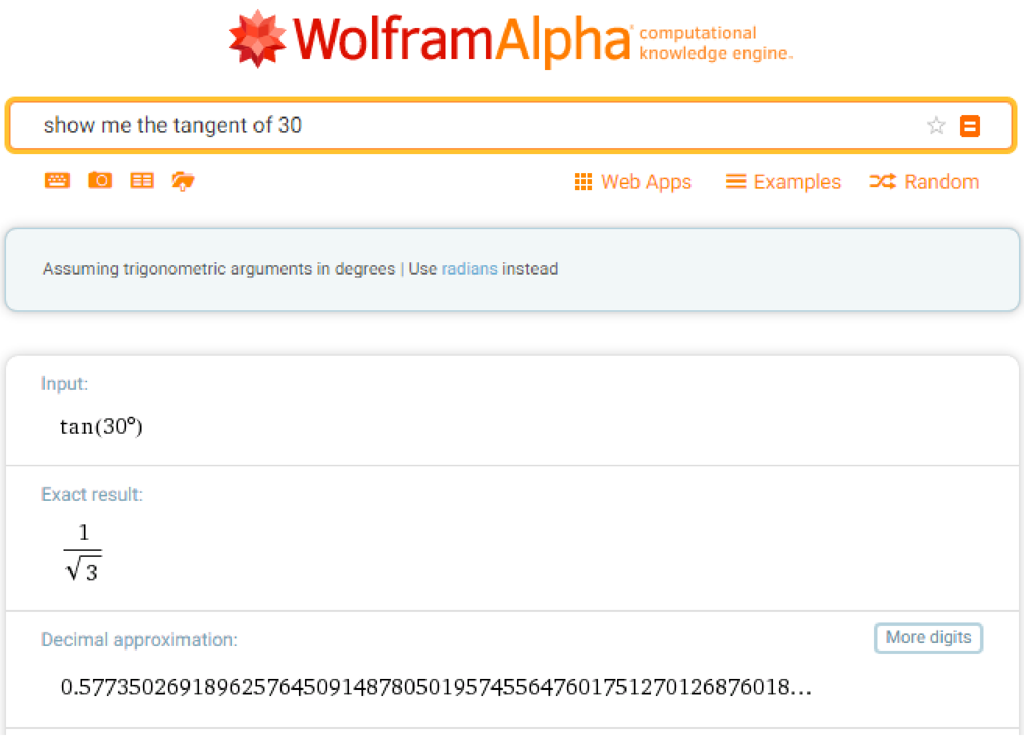
3.4 End-User development paradigms

The ultimate goal of any generic end-user development paradigm is to simplify the development process so that end-users, having little interest in the act of programming for the sake of programming itself, can modify an existing system or create new software artifacts. This is typically accomplished by relying on the end-user's domain knowledge or through creating high-level programming metaphors that end-users can utilise to create an application. There are many approaches on how to best create these metaphors and how end-users can interact with the development language to ease the development process.

3.4.1 Textual programming paradigm

Earlier in EUD theory, textual programming languages could be simplified by creating domain-specific languages or mini-languages. The hope was that a mini-language written for the end-user's context would be far simpler than having the end-user partake in general-purpose programming. While domain-specific languages are useful, they may still incorporate progammatic-esque thinking. These languages have involved into a newer concept called *natural language programming*, which can be used to create *natural language programs.* Typically, an NLP will take a source language such as English and interpret English sentences with the help of a pre-defined ontology. For instance, the sentence "The quick brown Fox jumps over the lazy Dog" could be interpreted as having (2) actors with properties (such as lazy or brown) where one actor performs a procedure (the fox). In this case, a taxonomy for an animal/actor would have already been created previously before compilation. In addition, some known procedures could also be stored (such as jump) and could be referred to in the language with assistance from autocomplete. Some examples of natural programming languages include Shakespeare [1], Inform7 [2], and Wolfram Alpha [3]. Shakespeare is a particularly interesting language in that it also incorporates punctuation as a metaphor for programming concepts as well,

[1]http://shakespearelang.sourceforge.net/
[2]http://inform7.com/
[3]https://www.wolframalpha.com/



show me the tangent of 30

Assuming trigonometric arguments in degrees | Use radians instead

Input:

tan(30°)

Exact result:

$\frac{1}{\sqrt{3}}$

Decimal approximation:

0.57735026918962576450914878050195745564760175127012687601 8...

Figure 3.3: Wolfram alpha interpreting a sentence to find an operation and data as input

such as using question marks at the end of sentences to signify an if-statement.

3.4.2 Visual paradigm

Visual programming has also been of high interest to end-user design due to its ability to abstract programming concepts into a recognisable set of icons. Icons, in this case, refer to an image with its associated semantic reasoning. This is also referred to as dual-coding theory [25], where the mental image is referred to as an analog code while symbolic coding refers to the words and ideas that we associate to the analog code. The use of icons is necessary for lowering the barriers of entry as complex programming concepts that non-programmers find difficult to understand (such as iterations and flow control [27]) can be associated with an image that can be used as a heuristic. An arrow curving around itself in a circle may indicate a loop or boxes with arrows can indicate flow control, for instance. Puzzle pieces, such as those used in Google's Blockly or Scratch, are another common iconic metaphor used in visual programming languages to help users understand which chunks of functionality can interface with others. The way in which icons are used creates many paradigms within visual programming:

Diagrammatic

The earliest of visual programming styles, diagrammatic or flowchart visualisation uses abstract icons (typically some form boxes and arrows) to represent programming states and data flows. Diagrammatic programming languages typically deal with data inputs, processing, and outputs, which makes them ideal for parallel computing [33] because operations typically run independently and are executed as soon as data becomes available. Thus, information about program state between these flows becomes irrelevant. Due to the similarity of this style of visualisation and the pipe-and-filter software development pattern, diagrammatic is also referred to as 'pipe-based' as well [15]. This type of visual representation is of interest to Tiles and other message-intensive systems due to the focus on data messages and reactions to them.

Model-Driven

A hybrid textual-visual paradigm that focuses on data models that are translated into executable code. Executable UML being the most prominent of this group.

Block

Mixes a visual representation using building-blocks with textual execution of code. Typically blocks represent loops, if-statements for comparisons, and can block lines of code into a discrete method.

Example-based

A wide scope of technologies that allow a program to execute "remembered" commands given by a user. For instance, moving a graphical representation of robot by the programmer can be translated into code that the physical robot can execute.

Through the use of visual aids and domain-specific knowledge, we create

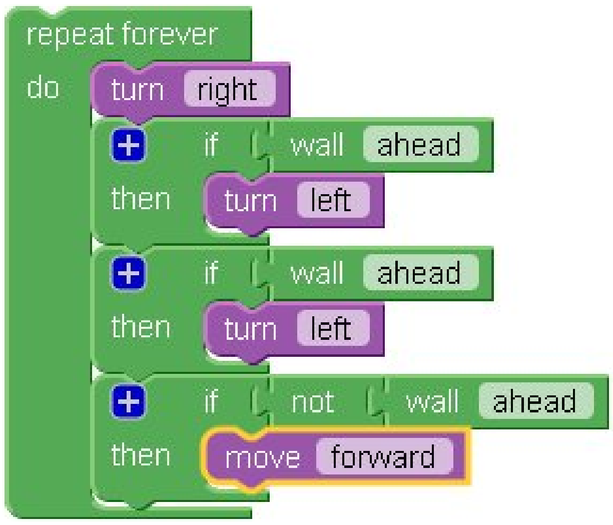


Figure 3.4: The jigsaw metaphor used to combine programming concepts

domain-oriented design environments. The main issue faced by visual DODEs as explored by Fischer is their lack of use for general purpose programming. In order to simplify the programming process, most paradigms rely on metaphors that relate directly to a specific domain. However, more general rule-based paradigms in conjunction with example-based programming have been successfully utilised by works such as HANDS [26]. HANDS is a mixed visual-textual programming language that uses cards as a metaphor for many programming tasks. Users take the role of a player that has access to data, event, and rule cards, and game cards. Rule cards give the programmer simple if-then rules that execute other cards when the condition is met. Data cards store variables while events are occurrences that affect the whole system. Users can then create many programs using the rules that could fit several domains.

3.4.2.1 Visual programming vs program visualisation

Within the field of visual programming, there can be some confusion between visual programming and program visualisation. The two concepts often overlap and the definitions blur, but as a rule visual programming creates code as an artifact. The technologies and paradigms introduced in the previous section are examples of this. Program visualisation, on the other hand, primarily focuses on using graphics to visualise code without creating an executable artifact [24].

Program visualisation can also be further defined by what kind of code is being visualised. If the tool or program displays the program or code structure, then the visualisation is referred to as static structure visualisation. For showing the dynamic state of a program, such as the value of variables during execution, a run-time visualisation is used.

Where the line blurs is when some change to the system is possible directly from the visualisation. A classic example of this is in model-driven design frameworks such as Eclipse's Modeling Framework (EMF) [4]. EMF can visualise text-based classes into a UML-like diagram. While this diagram can be viewed statically, developers may also edit the classes within the diagrams, which changes the underlying classes in the .java files. Figure 3.5 [5] shows how EMF visualises a set of classes in a program.

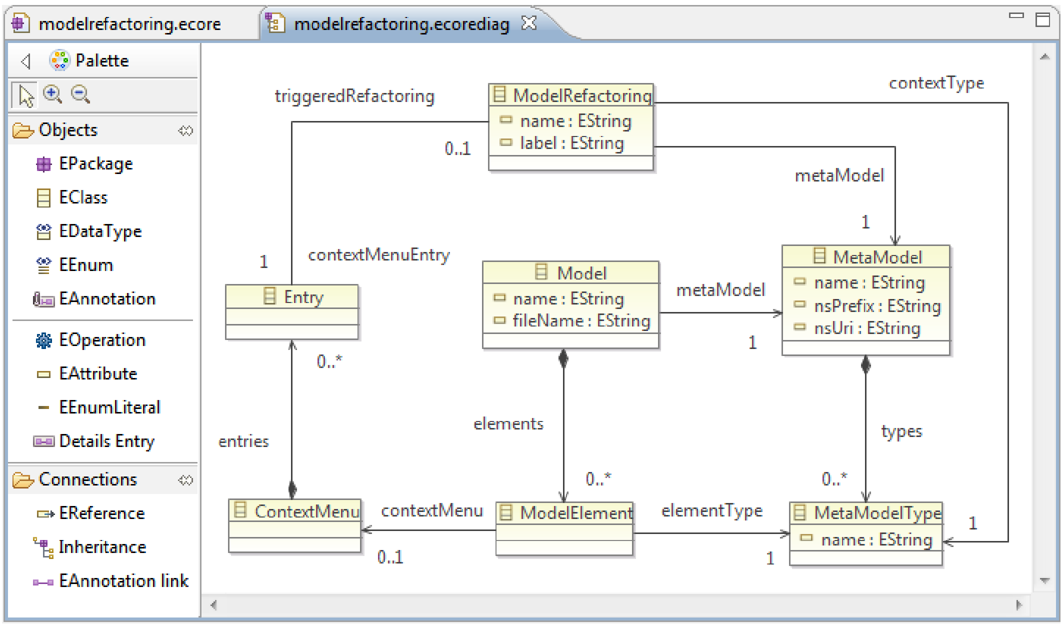


Figure 3.5: EMF visualisation of classes

[4]https://eclipse.org/modeling/emf/
[5]https://www.eclipse.org/emf-refactor/refactoringsjava.php

3.4.3 Physical paradigm

Physical programming is another paradigm that can be used for EUD. Physical programming entails using physical objects or gestures as metaphors for programming. By using physical objects, programmers can develop using tangible user interfaces (TUIs) as opposed to graphical user interfaces (GUIs), which are frequently used to supplement textual programming and always used in visual programming. Hornecker and Shaer have studied the use of TUIs extensively, and according to their research:

> "Tangible Interfaces have an instant appeal to a broad range of users. They draw upon the human urge to be active and creative with one's hands, and can provide a means to interact with computational applications in ways that leverage users' knowledge and skills of interaction with the everyday, non-digital, world."[30]

By leveraging the mechanics of the physical world, TUIs can be more efficient and easier to learn than a virtual analog. Furthermore, TUIs mesh easily with the idea of pervasive technology, which proposes that technology should blend into the background of society. However, this area is still quite novel and most physical programming systems so far have been geared towards teaching non-programmers programming concepts [14].

Google's Project Bloks [6] is a major forerunner for developing small applications using tangible media. Bloks works by connecting different pucks that represent either programming logic or entities to one another on an electronic board. These pucks are them compiled to underlying hardware code after development. The unique aspect of these pucks is that they themselves can contain a physical interface. A "volume" puck, for instance, includes a radial dial that one can use set the volume level.

Other physical programming attempts, such as Michael Horn's Robot Park [14], use computer vision to recognise objects, establish a control flow, and translate those objects into executable code. In Robot Park, a series of blocks

[6]https://projectbloks.withgoogle.com/

are read from left-right and top-down to create simple programs. Horn also created a GUI version of the exhibit and compared how appealing the TUI exhibit was to the GUI exhibit. Horn's findings (shown in figure 3.6) show that users were more interested in the TUI exhibit. These end-users were not technical, so this study provides evidence that tangible user interfaces can be a valuable tool to allow for non-technical users to develop applications.

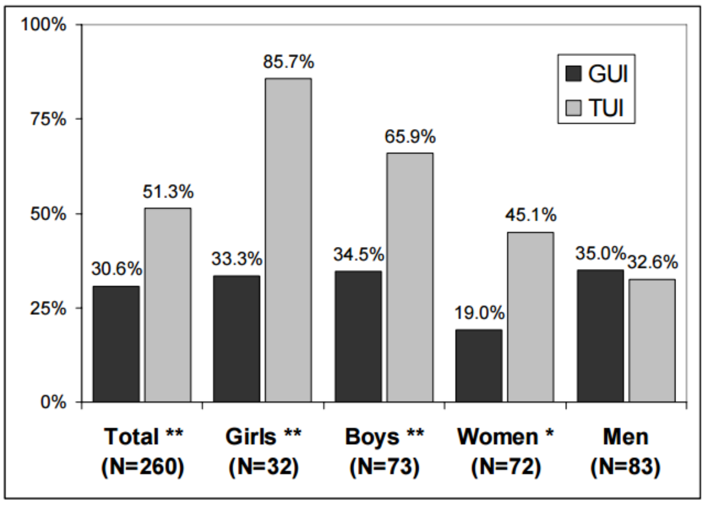


Figure 3.6: Percent interested in GUI exhibit vs TUI exhibit [14]

3.4.4 Programming by example

Another paradigm for creating dynamic applications is programming by example. Programming by example, PbE, can be used to impart instructions to a system by having a user generate a series of events that the system saves and executes. One example of this includes DEXMART[7], which combines both PbE and physical programming paradigms. In DEXMART, an actor is placed into a motion capturing (mocap) suit and then performs actions on objects in the environment. The actor's recorded movements are then serialised into a format that a robotic hand can execute. One can see that this approach greatly abstracts the code needed to manipulate the robotic hand by mapping natural movements from ones own body instead of requiring a developer to program how each motor in every joint behaves.

[7]http://www.dexmart.eu

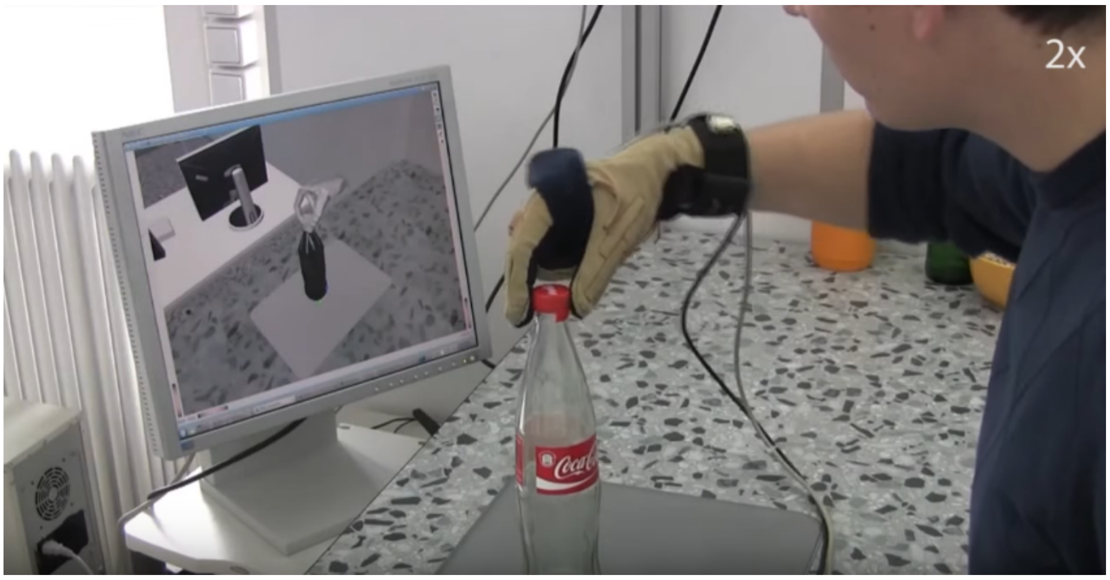


Figure 3.7: Mocap programming by example for how to open a cap on a coke bottle source(https://www.youtube.com/watch?v=jWYkzLSNmuc)

3.5 Why EUD for IoT?

In order to understand why end-user development is important for IoT, it is helpful first to look at some of the prevailing factors of third-wave human-computer-interaction (HCI). Third wave HCI moved away from the strictly work-focused participatory design of second-wave HCI to artifacts that were used in both work and private use. No longer were artifacts designed for a simple context - they had moved into everyday lives and now had a highly social aspect. This shift necessitated that the end-user was no longer just an actor in a context that an artifact was pre-designed for, but a central component of design that had to bring their life experiences to the design of the artifact [4]. As the context of the user changes, so must the artifact and, thus, there is a higher need for the artifact to change during *use-time* [19]. End-users are now human crafters rather just actors, and now provide data and customisations necessary to evolve software during use [19].

IoT can also be seen as product of third-wave HCI in that the pervasive, ubiquitous nature of many IoT applications relies on data from human end-users through many contexts. Barbara Rita Barricelli and Stefano Valtolina

[3] have taken an end-user centric approach to defining ecosystems where they have highlighted important elements that many IoT ecosystems are inevitably comprised of. In their model, the user can affect each element and can knowingly affect the system or unknowingly affect or develop the system. The user can knowingly create behavioral rules for sensors or applications if given the right tools, and they can unknowingly affect the system by their sensor data that affects the recommendation system of the ecosystem, for instance. As a result of being the center of the user-centric IoT ecosystem, it is inevitable that end-users would need tools to tailor their experience of applications and devices in the ecosystem as their context changes. Imagine a household system where an alert threshold for a noise pollution sensor is non-configurable or where the output source of that alert cannot be modified by the user.

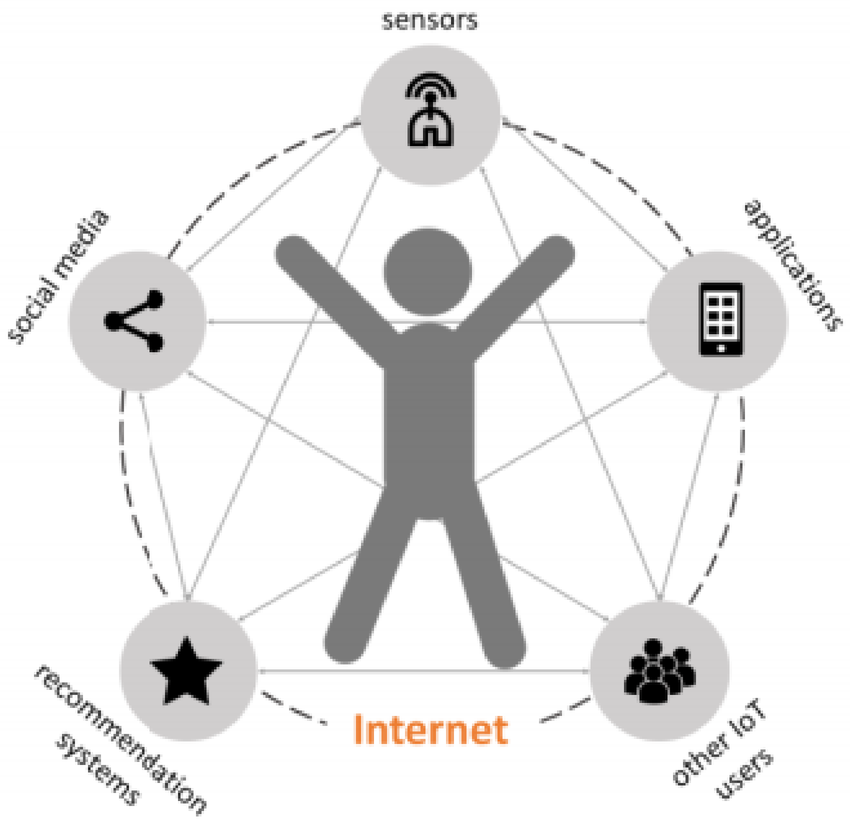


Figure 3.8: The user is at the center of the IoT ecosystem [3]

Finally, it must be realised that end-users are likely the holders of domain-specific knowledge rather than programmers who pre-design and then implement an application. Going back to the noise pollution example, there is no way that a designer would know what levels of noise in a household equates to unwanted pollution by the residents. The designer can, however, use meta-design methodologies to ensure that the end-user can act as both designer or consumer of an artifact in the IoT ecosystem.

3.6 EUD paradigms for IoT

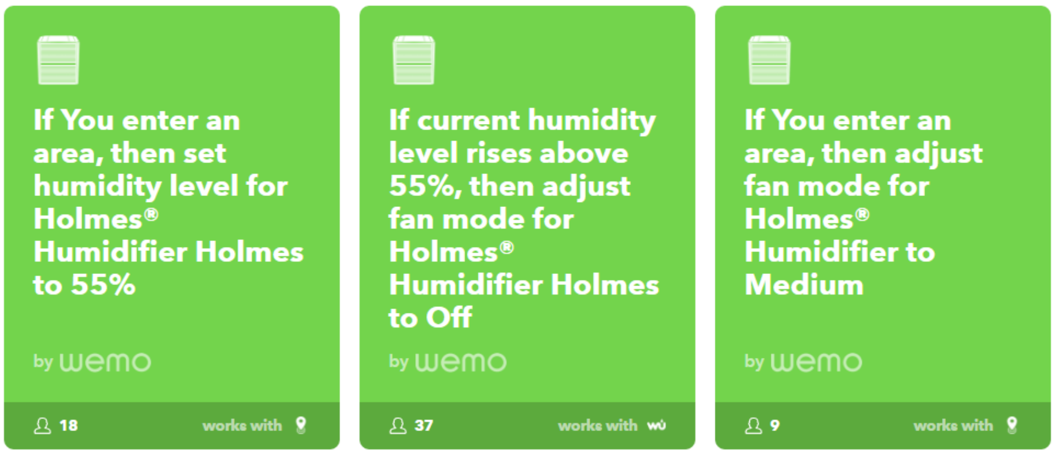
The basic tenets of smart EUD design, namely accessibility and scope of possible modifications, remain critical for IoT EUD. However, a major difference is that those designing applications for EUD in the IoT space need to understand that the applications created or modified by end-users will involve multiple devices with different protocols. Therefore, it is important to develop programming paradigms and tools that take into account hardware, software, and data together [3].

3.6.1 Data-mashups and rule-based systems

One such paradigm is data mashups. Mashups involve taking data from disparate sources with differing protocols, serialising the data into an acceptable output format, and then sending the resulting data forward. If-This-Then-That (IFTTT), for example, can take a tweet using HTTP web services and then, based on a condition, send a signal to hardware on a device using MQTT. Rule-based data mashup services like IFTTT [8] and WebHookit [9] are becoming popular due to them requiring no programming experience and using simple language to express the data rules. These services typically provide a web page where users may connect their devices to different data sources or other devices. Additionally, social features are typically provided by data mashup services. IFTTT is particularly apt at allowing users to create *applets*, or a bundle of two data services with an "if this, then that" rule binding them, that can be shared with other users who have the same devices. Figure 3.9 below

[8]https://ifttt.com
[9]https://github.com/neyric/webhookit

illustrates recipes that any user may activate after providing routing details.


If You enter an area, then set humidity level for Holmes® Humidifier Holmes to 55%

by wemo

18 works with

If current humidity level rises above 55%, then adjust fan mode for Holmes® Humidifier Holmes to Off

by wemo

37 works with

If You enter an area, then adjust fan mode for Holmes® Humidifier to Medium

by wemo

9 works with

Figure 3.9: Example IFTTT services for a smart humidifier

A rule-based approach is so successful for handling hardware and data concerns that it has been highlighted by Barricelli et al [3] as a state-of-the-art when combined with visual programming, where dataflow languages are king. Dataflow languages, sometimes called pipe-based languages, generally involve piping data from one input source to an output source with some data transformation in-between. Visually speaking, this is represented with the "boxes and arrows" approach that allows users to easily track data flow direction at a glance. An example dataflow language written explicitly for managing IoT devices is Node-Red [10]. Node-Red is a web-based visual programming language with several pre-defined nodes that act as data sources or data endpoints. Data can be transferred between nodes based on events with transformations (user-defined functions) available between sources. Social aspects, a common trend in EUD, are also available in Node-Red where users can create custom nodes for others to use in their applications.

3.6.2 Visual and rule-based challenges

However, Barricelli et al points out some flaws with rule-based systems, like IFTTT, and visualisation programs. If-Then rules have limitations regarding expressive power and the ability to have an event kick off based on multiple

[10]https://nodered.org/

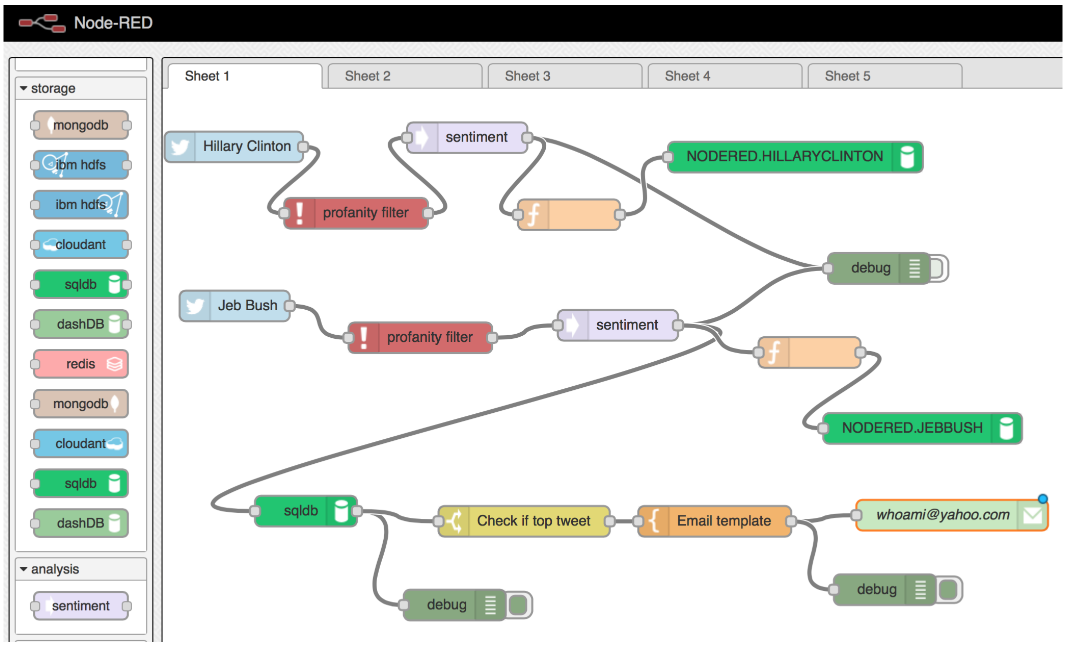


Figure 3.10: Node-Red in use (credit: https://www.ibm.com)

conditions and temporal conditions. For instance, one may require "if this AND this then that" or "IF this AT time THEN that" logic for their applet in IFTTT, which is currently not possible. As for visual programming tools, while Node-Red may be intuitive for someone familiar with programming, Barricelli et al concluded that these visual programming languages were still too complex for EUD. Node-Red in particular assumes that users understand the underlying APIs of devices and data sources, which " puts at risk the success of the EUD approach." [3]

However, it has been noted by the author that the ability to create custom nodes, if created around a known domain, could be useful in creating a mini-language within Node-Red (or any similar dataflow tool). Other visual programming paradigms, such as block-based or model-driven, are geared more for teaching users how to program or minimizing the time spent between designed code and implemented code. Google's Blockly, for example, teaches how to fit variables to control a loop by using puzzle pieces, but end-users in various studies [27][2] have been shown to be disinterested in the act of programming itself, which makes an educational approach less enticing.

3.6.3 IoT & Physical programming

Physical programming paradigms for IoT have not yet been firmly established, although their use for smart objects is applicable. Combining tangible interfaces with physical programming could prove beneficial for creating a sequence or for tuning rules that require an object to be manipulated. A hypothetical application could have to listen for a "shake" event on an object where the threshold for registering a shake is "taught" to the application by the user who shakes the object while the application is in a learning state. Similar to DEXMART, this example would be combination of both programming by example and physical programming, but for the IoT space.

Outside the hypothetical, Smith et al [32] in 2016 performed a study where objects could be used to control lights or a fan based on the proximity of the two objects. In this case, computer vision was used (as in Robot Park) to calculate the proximity of the two objects (a staff and a calabash, a frog and a fan) to determine if the lights should be on and if the fan should be on. However, while the application used connected objects, there was no shown method for the users, who had no technical experience, to configure the distance between the two objects needed before the lights/fan turned on. Still, it is an interesting case in how physical objects can be connected to one another based on proximity and, for Tiles, having connected smart objects is key.

3.6.4 Physical programming challenges

Looking at physical programming, there are some challenges for IoT. Chief among them being how to scale for large applications and how to handle large requirements in both space and time dimensions. Physical objects take up physical space and will naturally limit the size of the program. One solution to this would be to take a functional programming approach where users build functions that are saved digitally and then combined to form a larger program. This method would require some computational thinking on behalf of the end-user and would require that the physical components used to make the application are not core to the application itself (as in Google's Bloks).

In addition to program complexity, the pure scale of possible devices in an IoT ecosystem presents challenges. One would need to 1.) Find reasonable representations of each device and 2.) develop a strategy for iterating over devices or device groups while maintaining the benefits of physical programming for the developer. Again, a hybrid approach between physical programming and program visualisation is recommended in order to keep up with the magnitude objects and object interactions. This is especially pertinent in Tiles, where an application consists of several custom smart objects with different methods of interacting with other objects and services.

Space and time are also major components of most IoT ecosystems. In the example provided in the last section, two objects where configured to activate lights/a fan when they were in close proximity to one another. This works for small scale domains (such as smart homes), but having a physical representation of this for programming would be difficult for wide-area domains (such as smart cities). The same applies to time when the event flow is not simultaneous or events should not follow in rapid succession. One method in which these issues can be addressed is using tangible interfaces. For instance, one could envision a range tangible interface that could be utilised to abstract space and time components. The only configuration for the range slider would be the min-max values and the step values. This would allow for an end-user to intuitively fine-tune an application when the limits of physical programming as a metaphor present themselves.

Taking these challenges into account, it is easy to see why most physical programming research has centered around education domains and haven't been used for large-scale projects. However, for small-scale projects, physical programming could be beneficial for end-users because of its ability to lower barriers of entry. It is also likely that end-users would be disinterested in creating large applications due to complexity being a dissuading factor for end-users to begin with.

4 EUD and Tiles

This chapter will discuss how Tiles behaves as an IoT platform and what challenges present themselves for introducing end-user development to Tiles.

4.1 Unique Aspects of Tiles

Tiles primarily differs from most other IoT platforms in that it is used for creating both:

- user-defined smart objects via attaching tiles hardware
- user-defined interaction for custom smart objects and interactions between other smart objects

Many IoT platforms are concerned with pre-defined devices that can interact with one another in a simple, consistent manner. In Tiles, the object can be ambiguous and, while interaction primitives are already defined for a tile, the interaction can be ambiguous as well in the case of interaction sequencing. This makes defining a framework difficult because one is not only concerned with wiring disparate devices and services but having users create objects as well.

To put this into perspective, Barricelli et al. created a useful diagram [3] to evaluate devices based on time, space, and social aspects. Time represents whether or not a device collects data synchronously (such as continuous environmental readings) or asynchronously (such as user input). The space dimension evaluates whether or not the object is meant to be mobile or situated in one environment. Finally, the social dimension determines how many users are using the device. Combinations of these three aspects are used to create octaves that a device will typically fall into. For Tiles, any device/smart object can really be in any octave. Even with interaction primitives, which one would assume would lend to asynchronous interaction, a tile can also actuate from web services, which may not be asynchronous.

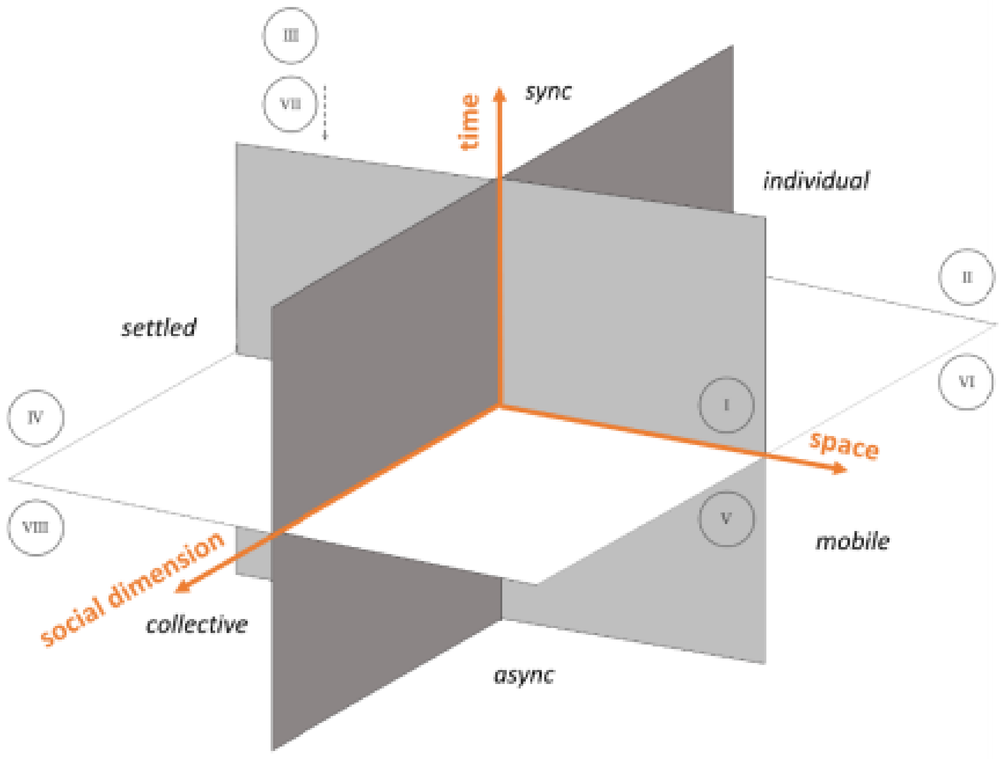


Figure 4.1: Octaves for the time, space, and social dimensions

However, there are some aspects of Tiles that make creating a framework easier. Tiles is unique in that it provides consistent hardware, a common data transfer protocol, and a set of libraries for controlling said hardware and data exchanges. Tiles also has the advantage in that, since it used for rapid prototyping, concerns such as security, authorisation, data privacy, and network efficiency aren't at the forefront. This allows for EUD and Tiles to avoid concerns with end-user software engineering, an aspect that can complicate development by pushing users to employ common software quality techniques.

4.1.1 EUD Criterion for Tiles

In order to draft a set of criterion for Tiles EUD, one must first look at a general set of principles that make EUD possible and then the set of factors unique to Tiles that need to be accounted for. In the previous chapter, an outline of what EUD is and who the end-users are was presented. In this outline, ease of use was a key feature for any EUD activity - whether class 1 application behavior modifications or class 2 application development. Thomas Greene and Martin Petre created a cognitive dimensions framework [9] to determine what level of usability a programming notion has. For studying EUD and Tiles, these dimensions can be used as a general metric to create a scorecard for different pre-existing technologies or a new artifact created for Tiles EUD. The dimensions are given in table 4.1.

Beyond these general dimensions, it is important in IoT to evaluate how a notation handles multiple devices and how these devices interact with one another and various disparate data sources.

Dimension	Description
Abstraction gradient	Bare minimum level of abstraction. What is encapsulated?
Closeness of mapping	Coupling to domain-problem
Consistency	Determines how much a notation follows a recognisable pattern
Diffuseness	Number of symbols needed to represent meaning
Error-Proneness	How likely it is to create errors in logic or syntax
Hard mental operations	What's the cognitive load of using the notation? How much the notation does the user have to look up?
Juxtaposability	Can notions be compared side by side?
Premature commitment	Is there an order of tasks for the user to perform in order to use the system?
Progressive evaluation	How easy it is to receive feedback with an incomplete program
Role-expressiveness	Is the role for each notation obvious?
Secondary notation	Measures if notation has secondary characteristics that carry meaning (shape, color, positioning for example)
Viscosity	How easy does the notation make the program to change?

Table 4.1: Cognitive dimensions framework

4.2 Recommended paradigms for Tiles EUD

When thinking about the possible paradigms to use for Tiles EUD, one must take the needs of the end-user (rapid, easy-to-use development) and the unique aspects of Tiles. Interacting with multiple tangible objects is the key to Tiles, which makes a physical programming approach seem the most intuitive. Additionally, there is scarce data regarding physical programming in the existing literature, which makes this paradigm attractive for the innovative aspects.

As for ease in development for end-users, having the users interact with tile squares attached to familiar objects would be more intuitive than writing abstract code. As an example, a user would have to write several statements in order to determine if a tile event matches a certain tile and what type of interaction was performed. The following code would have to be written in order to listen to a tile square, evaluate the message, and print a response:

```
var client = new TilesClient('TestUser','138.68.144.206',1883).
    connect();
var tileA = {
            name:"tileA",
            id:"12C97U"
            }

client.on('receive', function(tileId, event){
if(tileId === tileA.id && event.properties[0] === "tap"){
  console.log('Single tap received from ' + tileA.name);
  }
});
```

With physical programming, the user could instead just tap on Tile A to save the tile ID and interaction type (single tap). This would prevent the need for writing most of the above code with exception to the response (console.log in this case).

Most applications written for Tiles follow this pattern of listening for an event, figuring out the tile id and interaction type of the source, and then providing a response. This pattern is also highly prevalent in data mashup and rule-based engines such as IFTTT. In this light, it is not difficult to converge the ideas of object interaction with a rule-based system. The interactions and the data surrounding it (tile ID, interaction type, time received, etc) then acts as the rules for the engine. Therefore, creating a rule-based system is in line with most of the current literature and state-of-the-art. Using physical programming with tangible tile squares provides a simple way to create rules without the need for coding.

However, there are issues concerning physical programming and Tiles. As mentioned in the previous chapter, physical programming for IoT runs into issues of scalability and in the complexity of what they can express. The scalability issues involves both the number of devices and the space in which one can program using tangible interfaces. Using a board to program for Tiles could restrict the types of interactions the user wants to have with tile squares as well as constrains their programming environment to one location. In the future when geo-locational capabilities are added to tiles, this may become an issue if users want to record when and where they interacted with the tile. As for the number of devices, there would need to be some mechanism to display device rules and physical space once again becomes an issue as the number of device interactions increases.

Merging several paradigms is one method for providing mechanisms to handle physical programming's weaknesses. Program visualisation and visual programming could come in use to represent the software that users create by interacting with tiles. Program visualisation solves the issue of needing a board to implement logic or to keep up with all the device interactions. Physical programming could be used to program the responses to the rules when conditions are met. A user could use a GUI to program that a Tweet should be sent in the case that tile A is tapped, for instance.

4.2.1 How current technologies fit Tiles

While a tangible, rule-based system with visual components seems viable, it is important to find gaps in the current state-of-arts in order to discover their weaknesses and strengths for Tiles. There is no reason to reinvent the wheel when Tiles can be adapted to existing technologies, of course. Table 4.2 lists the positive and negative factors of the two-state-of-the-art technologies presented in the last chapter. Node-Red was chosen to represent a strong visual programming state-of-the-art that serves both general-purpose and IoT applicaton development needs. IFTTT was chosen as it is a strong representative for data-mashup, rule-based programming that molds perfectly with IoT.

These factors have been grouped in accordance to some of the relevant cognitive framework dimensions and the tile-specific dimensions highlighted in the last section.

Dimension	Node-Red	IFTTT
Abstraction gradient	+flow of data abstracted into boxes and arrows -all symbols look similar -details of each node	+ all underlying technology hidden outside of configuration for devices
Closeness of mapping	+applicable to complex domains	-one purpose per application
Diffuseness	-one symbol used for all nodes, even for functions +arrows useful metaphor for data flow	+- one symbol for each device
Error-Proneness	-no error checking in function nodes -no messaging for incorrect configuration of nodes +input nodes can only match to output nodes	+ checking of configuration + notifications about missing statements
Hard mental operations	+ A to B data flow is easy - Function nodes complicate applications	+ Only configuration data needed to look up
Secondary notation	+ input/output nodes have different markings + node types have different colors + nodes can be position to further show flow of events	+ different color for condition vs effect - little secondary notation outisde of this
number of devices	+many devices possible + each device can have multiple inputs/outputs - distinguishing devices is difficult due to notation	- one device per condition/effect
device interactions	- each interaction requires new node	- each interaction requires whole new rule

Table 4.2: Pros/Cons with leading technologies

From the table, it can be inferred that while Node-Red has more expressive power applicable to many domains, IFTTT is a much simpler tool that abstracts the data flow details from devices and data sources. When it comes to handling multiple devices with several interactions, both technologies struggle. While Node-Red and IFTTT cover many types of devices, this is irrelevant to Tiles as each tile square contains the same hardware. In Node-Red, a Tile square would be represented as an MQTT message, which is not immediately intuitive if you are unfamiliar with IoT technologies. This makes users think in terms of "when MQTT message received, then send this MQTT message in response", which is not how non-technical users would think about an application. For IFTTT, having one device would be fine, but each interaction requires a whole new service from IFTTT. Naturally, making complex interactions this way is cumbersome, but it works fine for simple applications requiring minimal interactions with tile squares.

Thus, it appears that sequencing of interactions would be a main issue for using IFTTT while the more technical and general-purpose nature of Node-Red would be difficult for most end-users. Opportunity then exists in creating a prototype that focuses on sequences of device interactions in an easy to understand manner. This can be accomplished by using both physical and visual paradigms and by taking some of the positive aspects of both Node-Red and IFTTT. The inter-connected node approach shown in Node-Red makes following a sequence very intuitive, for instance. The simple interface of IFTTT where you are clearly shown inputs -> outputs could also be leveraged to help users easily understand the cause and effect of tile interactions in their application.

5 Prototype Design

This chapter will give a brief overview of the audience, functionality, and scenarios of use for software merging the paradigms needed to help develop for Tiles.

5.1 Main Idea

Based on the information gained in the previous chapter, there is a need to create software using multiple paradigms that can allow for end-users to more easily develop applications with Tiles. As discussed in the literature, a rule-based system implementing some visual representation of objects is the state-of-the-art. However, due to the unique, tangible interaction focus of tile squares, it is beneficial to research ways of incorporating tangible programming to align with the purpose of Tiles. Programming-by-example is one such beneficial method that can be used to program a rule engine while the end-user is naturally using the smart objects with attached tile squares. For instance, it is easier to perform the intended action (such as tapping smart speakers in a room) in person than thinking about what sequence of interactions should be performed using mental abstractions.

Therefore, the author has proposed Tile Recorder, a rule-based system that records interactions with tile squares to generate rules and a visual interface to program what responses accompany those rules. In this way, users can map the rules (tile inputs) created in real-time from tangible interactions with the outputs that are created in the user interface. The visual aspect includes this interface along with a visual representation of the executing program's state (i.e. where in the recording sequence the program is evaluating next).

5.1.1 Why build a prototype?

As mentioned in the methodology section of the introduction, a prototype is a design artifact that is essential for testing theories in the field. A well-designed prototype can focus the attention of the audience towards the general concept behind the prototype by providing a concrete artifact that they can use. A proper prototype achieves this by providing the core functionality needed to convey the concept and can be developed rapidly to meet the demands of users after a pilot. Furthermore, the initial prototype should be basic so that users do not debate extraneous functionality or details regarding the prototype so that the researchers may study the central idea. These guidelines have been created by Warfel [35] to help developers in creating new prototypes:

Undestand the audience - Know which stakeholders the prototype is targeted toward and what their concerns are. The audience has been detailed in this chapter.

Plan a little, prototype the rest - Plan roughly 70 per cent of the functionality and expect the rest to change. For Tiles Recorder, only the main functionality was planned before implementation.

Set expectations - Define the key behavior of the prototype to your audience. Scenarios presented in this chapter can set the expectations for what can be achieved with Tiles Recorder.

You can sketch - Use mockup tools to keep the design generic enough so that look and feel discussions around the prototype do not become too detail oriented. Photo editing tools and online mock ups were used to design a UI before implementation.

It's not the Mona Lisa -The prototype does not have to be aesthetically perfect nor technically perfect, but good coding practices still lend to reusability. The implemented prototype uses basic CSS and exists on the client side only (see chapter 6.

If you can't make it, fake it - Use designer tools such as Adobe Fireworks to mimic system execution with little to no coding required. The prototype uses real code, but future functionality could be mimicked with static data (ie a common Twitter account to send tweets).

Prototype only what you need - Implement only the most essential functionality. Recording and executing implemented with limited output options (see chapter 6) as example of only programming what is needed.

Reduce risk—prototype early and often - Similar to Agile methodologies, present the prototype early and often to assess how well it fits requirements. Presented prototype design often to supervisors and undertook user-testing as part of the prototype's pilot.

Additionally, a prototype must be designed correctly in order to reap the full benefit in the field. The design presented in this chapter will follow the 8 principles of good prototype design as presented by Warfel.

5.1.2 Target Audience

Through looking through the literature focusing on end-users, an end-user is defined as someone who is

" A person with personal domain knowledge and an interest in modifying existing or creating new software for purposes of tailoring that software for their needs without running into the technical barriers of programming "

Therefore, it is beneficial to create a system that is not overly technical for end-users that also fits into the strengths of Tiles (such as physical interaction primitives).

5.1.3 Explanation of System

According to Barricelli et al in Designing for End-User Development Design for IoT [3], the state of the art for many end-user development tools is a rule-based system that allows a condition (typically, an event) followed by a reaction. Therefore, the proposed system is ultimately a rule-based system tailored for the unique challenges presented in Tiles (namely, handling sequencing of interactions). The system merges two paradigms: physical and visual programming. Physical programming is used to capture the interactions with Tiles for the condition of the rule, while a visual form can be used to edit the reaction.

On startup, the tiles for a set of users are loaded. To capture the interaction primitives, a user can click a record button to let the application "listen" for events with a set of filters. The user(s) then perform interactions with their tiles to mimic the desired behavior. Captured interactions are listed in the system so that users may delete unintended interactions to reduce noise. A differentiator will be used in the list (a different color for instance) to show when Tiles belonging to different users are interacted with. Recordings can have 1 to many interactions.

Additionally, users may set options and filters for each recording. These options include whether or not Tile IDs are important, whether or not the order of the sequence is important, a time-frame for which the application listens for events and filters for the different interaction primitives. After recording, each sequence is placed into a list with an output. The output is a drop-down with

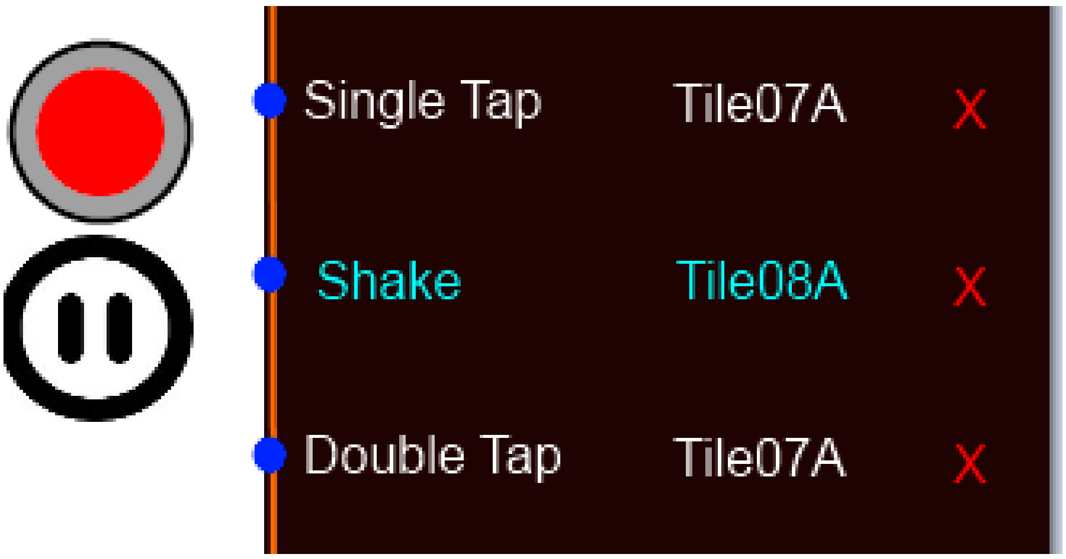


Figure 5.1: Sequence list mockup

several pre-configured responses. For example, Twitter could be selected by the user, the user enters in their Twitter data for login and the text they want to tweet when the recorded sequence is completed. Outputs for Tiles would exist among the selections too, of course. Finally, as with any rule-based system, the rules that were created must been listened to and evaluated. Users can 'activate' their recording to have an executor listen to incoming events and compare them with the interactions saved by the recorder.

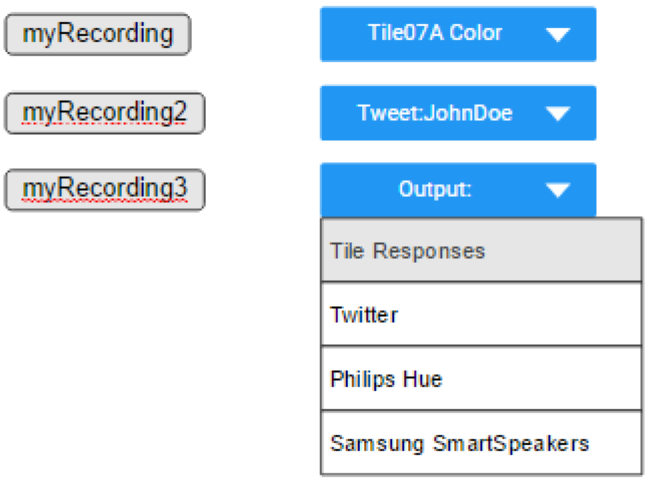


Figure 5.2: Output

5.1.4 Software Components

The following components (or modules) need to be developed in order to have a fully functioning system.

Recorder

The recorder is the essential component that will listen for tile events over MQTT and save them into a data structure that can be parsed by the executor. Functionality of the recorder includes recording and pausing as well as filtering out undesired interactions. These filters have been foreseen as a necessity given the sensitive nature of tile square interactions. Sometimes a double tap is recorded as two single taps or vice-versa, for example. The main output for the recorder is a sequence of device interactions that the executor component can listen to while executing the user's program.

Outputs

With each interaction the user can set outputs that execute depending on whether or not the user matched the recorded output. If a user has recorded that Tile A must be single tapped, they can could tell Tile B to light green when Tile A is single tapped or set Tile B's light to red if a Tile besides A is tapped.

Executor

The executor, when activated, listens for device interactions and compares them with the sequence recorded by the recorder component. The executor evaluates if the interaction meets the Tile Id, interaction type, and timing restraints as set by the recorder. Upon the successful completion of an interaction in the sequence, the executor either:

- Continues to listen for the next interaction if the current interaction meets the requirements
- Executes the success output if present and then continues
- Fails and continuing waiting for the correct response if the interaction does not meet the requirements

- Fails and executes the fail output if present

The executor should also have the capacity to visualise how far in the execution chain the program currently is. For instance, if the list of recorded events is represented as a chain of nodes, the executor should highlight the current interaction (node) that is going to be evaluated.

Events

The events are primarily a data structure that contains data about recorded device interactions (when they occurred, what tile was interacted with and how) as well as output commands. Each event can have one output for success and one for failure.

5.1.5 Scenarios of use

In this section, two different scenarios will be presented that use the Tiles Recorder. These scenarios were created in order to both:

- brainstorm what functionality was required to fulfill each scenario
- show the breadth of what could be accomplished by the system

5.1.5.1 Learning a process

Mr. Henderson runs a fast food burger chain that often hires seasonal help. These temporary employees don't know much about assembling burgers or cooking and its time-sensitive nature, so Mr. Henderson needs to develop a system for training these employees. In the past, Henderson used diagrams and videos, but employees seemed to forget most details when the time came to actually use the kitchen equipment. Therefore, Henderson looks to Tiles as a way to integrate feedback while employees learn using real equipment and not a virtual simulation. Henderson has Tiles attached to the frying/oven top to represent the cook.

A separate set of Tiles are attached to containers with the toppings/ingredients to represent the burger assembler. Finally, a tile is attached to a serving tray to represent the runner who brings the food to patrons. Through the software, Henderson first creates a recording for cooking the fries and meat for the burger. For the fries, he sets a two minute window for when they should be cooking. Henderson tells his employees to shake the fry basket at least three times while they cook. To mimic this in the system, he shakes the fry basked three times, which makes the system record a shaking event for the tile attached to the basket's handle. For the burgers, Henderson wants his employees to tap the tile attached to the cooker when the meat is brown. Therefore, the recording for this piece is three shakes and a tap coming from the cooker. Henderson also notes in the system that the order of the sequence is not important, just the three minute time frame.

Next, the assembler needs to build the burger in the right order. Henderson creates a 1.5 minute window for this. For every ingredient that is placed onto the burger, he instructs his employees to tap the container of the ingredient to show that it has been placed on the burger. First comes cheese, then lettuce, then tomato, and finally ketchup and mustard. For this recording, the order is important. For the final recording, the runner must bring fries, the burger, and a soft drink to a serving tray and bring the tray to a table. The first sequence needs to be done within 1 minute and involves tapping the Tile on the tray for each necessary item placed onto it. The order doesn't matter here. Once the food is at the table, the employee needs to double tap the tray once he/she is back to the work station. This signals the end of the training exercise.

For the outputs, Henderson wants a tweet to show when each recording is completed. In the software, he can see the three recordings and selects the twitter option for each recording. Additionally, he wants the Tile on the tray to vibrate in bursts after the second recording is completed so that the runner knows that the food is cooked. He accomplishes this by selecting Tile output on the selection box and "vibrate-burst" for the output type.

After a few test runs, Henderson is happy with the results from his new employees but wants them to be faster. As they get better in the exercises, he narrows the timing windows for each sequence.

5.1.5.2 Making a game

David's family has expanded with its newest member, Brock. Brock is an 8-month year old that loves to play with his toys and listen to music. David is interested in teaching Brock about patterns using a playful program that he developed using Tiles. Brock has 8 toys with a tile attached to each. David wants to teach Brock to mimic which toys David chooses out of the 8 and what he does with them. Before laying out the toys in front of Brock, David opens a GUI for the recorder. He tells the recorder that both order and the ID of the tile matters and then proceeds to interact with 3 of Brock's toys. He shakes the first, taps the second, and then shakes the third. Once the recording is done, David chooses a generic response from the GUI – playing a song on his smart speakers in this case. Next, David acts out these motions in front of Brock. Brock then imitates these actions with the correct toys and in the correct order to hear his favorite song.

5.2 Niche in EUD for IoT

From the review Node-Red and IFTTT in the previous chapter, the recording application was creating to fill the gap where Node-Red was too complex with device interaction for end-users and IFTTT was too simplistic. Table 5.1 shows pros/cons of the application

Based on these positive and negative factors, the recording application, like the similarly rule-based IFTTT, is a tool that abstracts the logic behind taking device inputs and executing outputs. The main difference being that the recording application can handle multiple device interaction more cleanly and can thus create more complex conditionals for rules. Additionally, the recording application has the tangibility element to expedite the process of setting up device interaction sequences. The intent is for users to develop their sequences while on site and is inspired by IoT's goal of merging digital applications with our physical environment.

Dimension	Recording App
Abstraction gradient	+ device behavior abstracted - configuration on output sources still necessary
Closeness of mapping	- limited to subsets of many domains (primarily situated learning)
Diffuseness	+ well known symbols for recording/pause + can integrate with Tiles cards as known symbols
Error-Proneness	+ error proneness lies in user logic and not syntax or computer logic
Hard mental operations	- having many recordings/outputs can get confusing when combined to make one application
Secondary notation	+ device notation is the object itself in physical space + node types have different colors + nodes can be position to further show flow of events
number of devices	+ many objects possible - scaling limited by physical space
device interactions	+ device interactions listed plainly + device interaction is tangible - one output for recording

Table 5.1: Pros/Cons with recording app

6 Prototype Implementation

This chapter serves primarily as both a guide to using Tiles Recorder and to detail how the application was built in order to expedite future development of the prototype. A look at the user-interface (UI) will begin the chapter and the following sections will explain implementation details of those components. It should be noted that the implemented code makes use of the standard Apache 2.0 license [1].

6.1 Looking at the UI

The prototype for Tiles Recorder was written as a web application utilising the React [2] framework and Node JS. Node JS was chosen as Node is currently used to run the Tiles application server and the current Tiles MQTT client

[1]https://www.apache.org/licenses/LICENSE-2.0
[2]https://facebook.github.io/react/

is written in JavaScript. Therefore, using Node and JavaScript would reduce integration concerns during later development and allow the software to use the official client for Tiles.

React was chosen due to the highly dynamic nature of the system's views, which must update frequently when new inputs are recorded or to visualise the progress of the recorded sequence during execution. Thus, it is logical to start with outlining the UI and then detail each UI component. Fig 6.1 below shows a screen shot of Tiles Recorder during use with the UI components numbered. These numbers correlate with the descriptions of the components on the following pages.

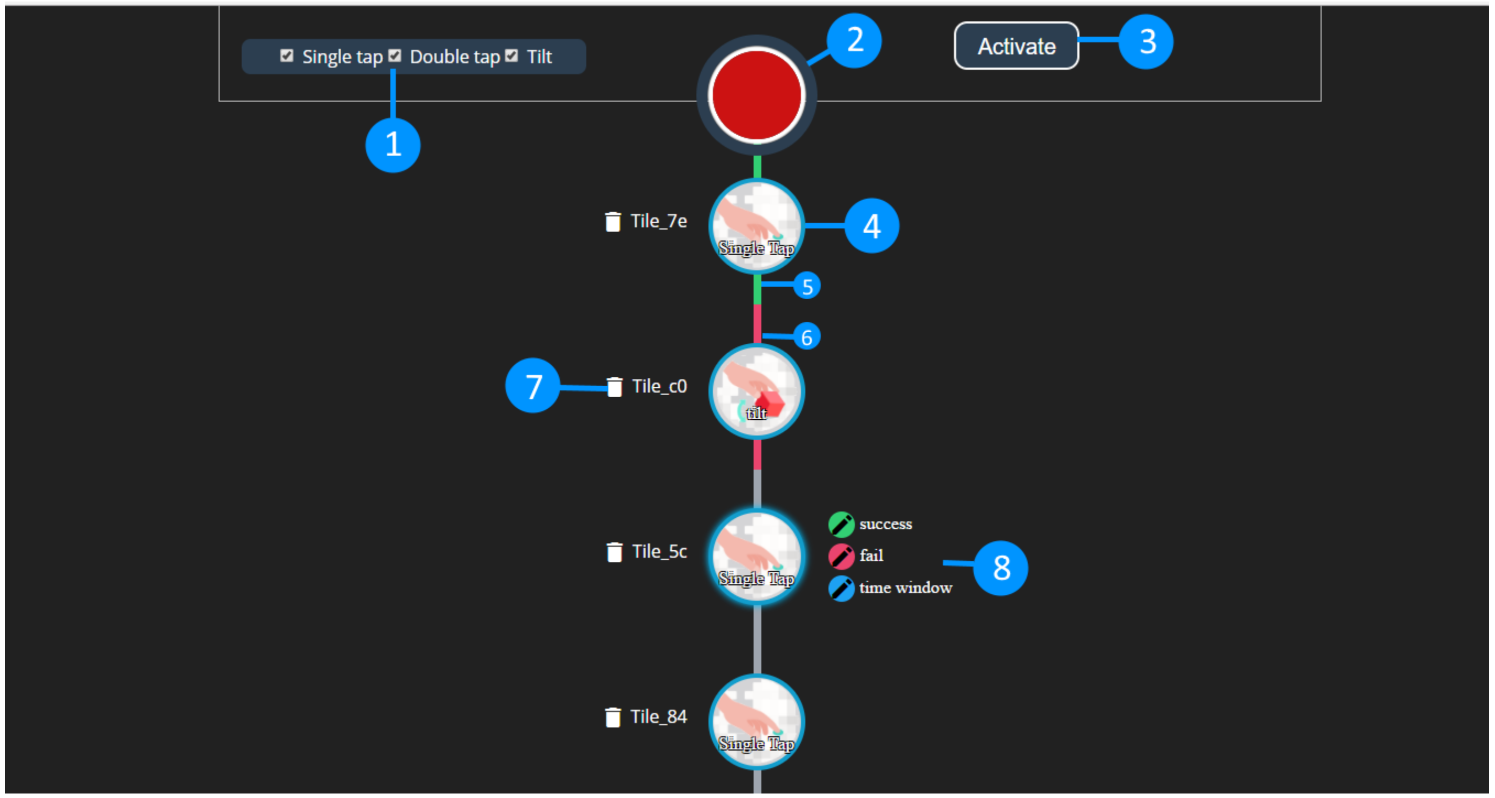


Figure 6.1: Tiles Recorder UI

1. Filter area: Users can click these checkboxes on/off in order to disregard or listen to certain tile interaction events. Sometimes interactions can fire unintentionally from a square, so these checkboxes serve as protection against false-positives.

2. Record button: When clicked, allows for tile interactions to be saved and displayed in the node chain below. Icon changes to a pause button to pause recording. When executing the sequence, the button cannot be clicked and the color changes to a dulled red to signal its inactive status.

3. Execute button: As with every rule-based engine, it must listen to events and evaluate those events based on rules. In Tiles Recorder, the execution button starts the listening side of the application. Once listening, the button can be clicked again to allow the user to record again.

4. Tile Interaction Node: When an interaction is recorded, a new node will be created in the node chain. The type of interaction is displayed with an icon and a text description is overlaid on top. The name of the tile is displayed to the left.

5. Success highlight: During execution, when a user completes a rule (such as single tapping Tile_7e in this case), the color of the chain changes to blue.

6. Fail highlight: During execution, when a user fails a rule (such as not single tapping Tile_7e in this case), the color of the chain changes to red.

7. Delete icon: Removes the current node in the interaction chain

8. Option area: Add/remove success and fail responses for whenever an interaction is passed/failed. Can also add a time limit condition. Only appears while hovering over a node in order to prevent information overload for the user.

Figure 6.2 and figure 6.3 shows the option dialog boxes that appear after clicking on a button in the option area. Note: Success and Fail dialogue boxes only differ in color, not content.

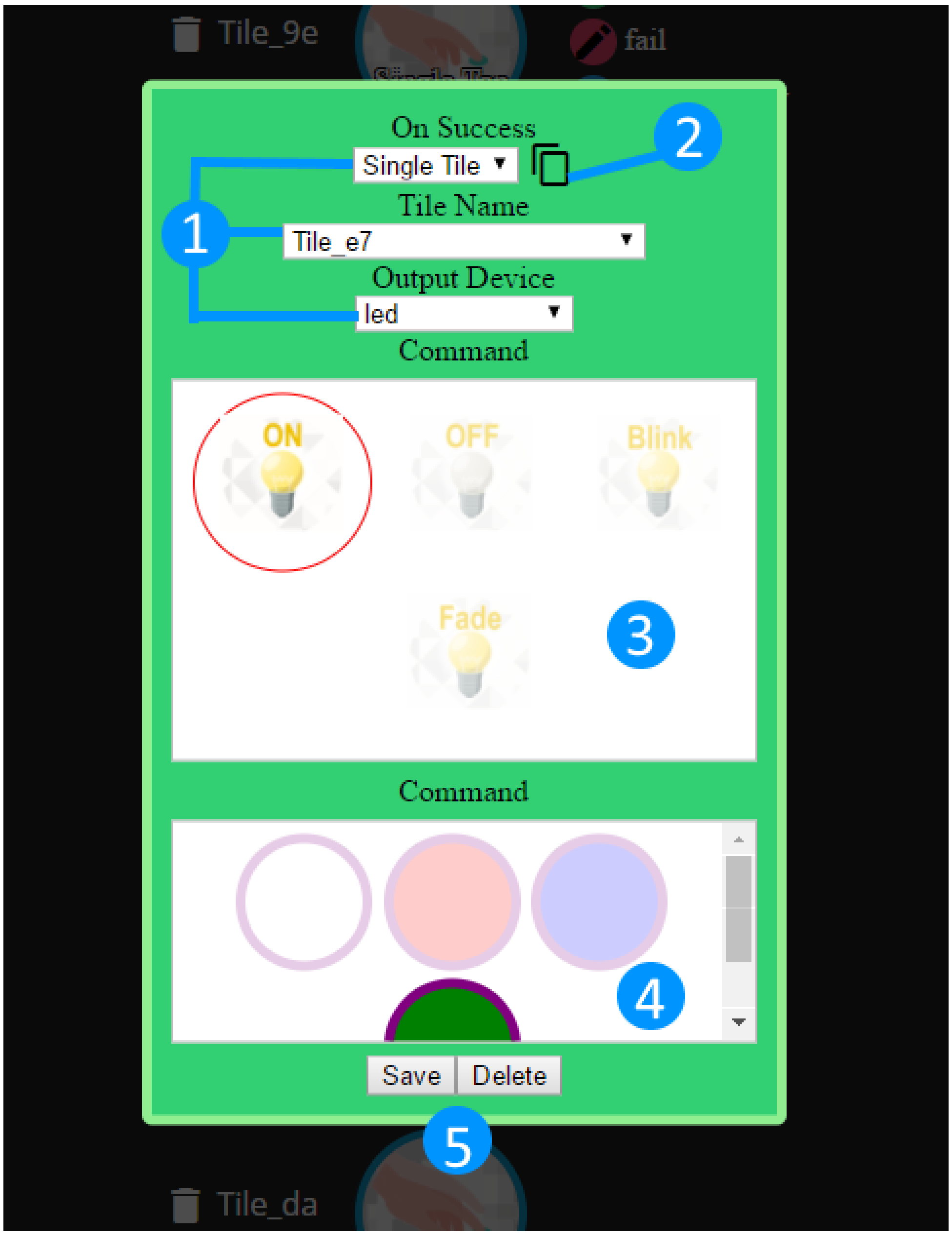


Figure 6.2: Success/fail option window

1. Options area: includes output type (currently only tile outputs are available) and based on the type selected builds the rest of the form. For a tile output, one can specify the tile for output and the output to execute. The list of tile names is gathered via a REST call to the Tiles application server.

2. Copy from previous: This button copies the success or fail condition from

the previous node onto the current node. Useful for scenarios where steps have similar needs for fail/success.

3. Tile command area: Each tile output can have up to two command properties. In this case, the led can be turned on, off, blink, or fade out. The haptic feedback component of a tile is also implemented. The icons for the commands have been taken from the tiles card game to help users with the system after they design with the card game.

4. Command property area: For commands that have a secondary property (led in this case), the secondary property options will be shown here. In the case that the user chooses 'off' as the led command, this area will not appear.

5. Save/Delete buttons: User can save the success/fail response (closes dialogue box) or delete, which removes the success/fail response for the node and closes the dialogue box. Clicking outside the dialogue box area also closes the box, but does not save or delete changes.

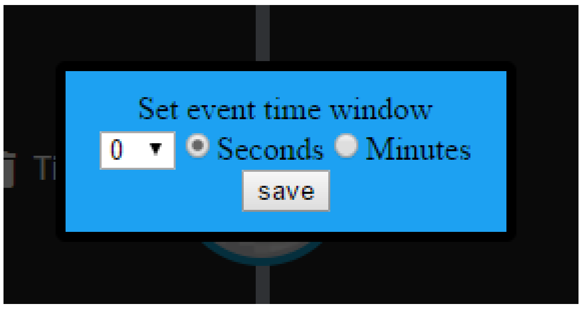
j 

Figure 6.3: Time limit option window

Finally, the last option box is the time limit box shown above. The time limit can be set via the input box and a unit selector.

6.2 Implementation Details

As stated previously, Tiles Recorder has been built primarily as a client-side application using React and socket.io. This decision was made in order to fit in with the principles of effective prototype design as mentioned in chapter 5. That is, the core functionality (recording, executing) has been developed in order to quickly perform a pilot with test users. Extra flavoring of the application, such as persistence in a database for recordings and extra output options, has not been in scope for this iteration of the prototype. Additionally, React allows for the rapid development of a stateful web application and can be changed easily as new improvements for the prototype are found. This section will detail the project structure, client details and the limited server details.

6.2.1 Project structure

The project uses a standard web template for npm projects. Client-side code (including all of our React components) goes into the client folder. Currently, all components are contained within the app.jsx file. The routes folder contains our server-side code for socket.io. This includes getting tile names via an HTTP request, setting up the tiles client to listen to tiles events, and receiving messages from the client. In the root directory, app.js starts the express server and loads in the routes.

For building the project, Browserify [3] is used to compile the js files into the public folder, which also contains the index file for the html page. Babelify [4] is also used to in order to convert the syntactic sugar of React's JSX to plain JavaScript.

6.2.2 Server implementation

The application currently runs on a standard NodeJS express server. The tiles MQTT client is used server-side to collect messages and to push messages to the connected clients. While it is possible to place the tiles client into the client's browser via browserify, it is recommended to keep the client in the server for future development. For instance, in the future where a user or users

[3]http://browserify.org/
[4]https://github.com/babel/babelify

has many different devices with different active recordings, a server-side client can make message routing less hectic. Additionally, the states of programs created by Tiles Recorder could potentially be managed server-side instead of the current client-side implementation, which would create one less 'hop' of data transfer whenever the tiles client receives a message.

The socket.js file contains all of the socket.io setup between the server and client. Below is the code snippet for the routes when a message is received.

```
tilesClient.on('receive', function(tileId, event){

  console.log('Message received from ' + tileId + ': ' + JSON.
      stringify(event));

  socket.emit('tileEvent', {
    tileId: tileId,
    event: event
  });
});

socket.emit('init', userTiles);

 socket.on('tileCmd', function (msg) {

   tilesClient.send(msg.tileId, msg.cmdString,msg.param1,msg.
       param2);
  });
```

As one can see, when the tiles client receives a message (interaction event), it is logged on the server and then the tile event properties are sent to all clients. When a browser client connects, they receive an "init" message containing all of the user's tile names. This is used for building out the list of tile names in the options windows but will also be useful if user profiles are implemented. The last interaction is when the server receives a tile command message from the client. This happens when an interaction is correct/incorrect and the user has set up a success/fail tile response.

6.2.3 Client implementation

On the client side, there are four main components:

- Recorder
- Event
- Options Window
- Executor

These components correspond with their descriptions in the design section. The following subsections will detail their function and any notes for future developers.

6.2.3.1 Recorder

The top level component. Contains the executor and event components as child components. Outside of rendering its child components, the recorder component is also responsible for rendering the filter buttons, record button, and activate button.

```
<div className="exec-buttons">
      {!this.state.executing &&
    <div className="activate-btn" onClick={this.finishRecording
       } value="Finish" >Activate</div>
    }
    <div className={this.props.executing ? "hidden" : ""}>

      {this.renderExecs()}
    </div>
```

Listing 6.1: Activate button display logic

The activate button determines if the execution button should be shown. The important aspect here is that the recording component contains a state that determines if the application is executing or recording. If executing, the execution component is shown (but is always rendered even if hidden). React can render components conditionally, but the reason the executor is always rendered is to synchronise its state. If left conditionally rendered, the state between the executor component and the recorder component would become out of sync. The rest of the recorder component is fairly straight-forward and boilerplate. However, for developers who may be unfamiliar with React, there are some functions declared in the recorder component that are passed down to child components in order to manage their state. This is in accordance to React's best practices [5] where manipulation of top-level components should be passed down to children via properties. Figure 6.2 shows how the event child component is rendered. Notice that the typical CRUD operations are passed from the recorder component (this) to the event component.

```
renderEvents(){
      return this.state.events.map((event, index) => (
      <Event key={index} propKey={index} tileId={event.tileId}
         tileName={event.name} eventType={event.type}
           updateEvent={this.updateEvent}
         deleteEvent={this.deleteEvent}
         status={event.status}
         getPreviousEvent={this.getPreviousEvent} />

      ));
  },
```

Listing 6.2: passing properties down to children

Another note to make is the normalizeEventType function. This function was created in order to normalise how tile event properties are managed by the application. At this time, different tiles submit their properties in different serialisations and must be normalised to simplify development.

6.2.3.2 Event

The event component is responsible for rendering the interaction chain nodes. This is primarily achieved by using a flexbox layout [6] for the tile name and deletion area, node area, and option buttons area. The conditional rendering

[5]https://facebook.github.io/react/docs/thinking-in-react.html
[6]https://css-tricks.com/snippets/css/a-guide-to-flexbox/

here involves setting the interaction type in order to choose the correct interaction icon and reading the state of the event during execution to change the color of the chain to represent success or failure. Another odd addition to some who are new to React is the inclusion of the property named propKey as well as the regular Key property. In newer versions of React, the Key property is reserved for use within the framework itself and, thus, the developer must set a different property if he/she wishes to pass or keep track of the index of the component.

6.2.3.3 Option Window

This component is used to generate the tile response form. The methods and data used here are fairly straightforward outside of how the buttons and modal are displayed. The component has a state that determines if the user needs to view the edit button or the option dialogue window. Changing this state is done via the setEditing function (shown below). Since there are only two states, taking the opposite of the current state without evaluating the current state is possible .

```
setEditing(){
      this.setState({editing:!this.state.editing});
    },
```

6.2.3.4 Executor

The executor component is simple in its rendering logic, but complex in its evaluation logic. As a general note, while this component is currently a React component to help expedite development, the evaluation functionality could be moved to the server in the future. For instance, the server could evaluate messages once the browser client activates its sequence and then send status messages to the client (for success and failure of steps).

To begin with, one should note that this component's internal state is limited to only lastSucessTime. The component itself does not have an internal list of interactions when it evaluates. While this may seem unintuitive, this is in accordance to the best practices of React and general JavaScript programming. Earlier in development, the executor component set its initial state based on properties passed by its parent component. For example, the getInitialState function would have lines such as:

```
Return {eventList: this.props.eventList}
```

However, this is problematic because changes made to the executor component's state would persist up to its parent component as JavaScript objects are passed by value. Therefore, the state in the recorder component is used as the "golden source" and is passed down to all child components. The list of interaction events is never edited directly by the executor and a success/fail signal is sent to the recorder component in order to set the status of the current event being evaluated.

The evaluate function fires whenever an interaction is recorded and has this sequence of checks:

1. Check if the executor is active
2. Check if the tileId matches
3. Check if the tile interaction type matches
4. Check if time condition exists
   (a) Check if time condition is passed

If the above are true, then the evaluation checks if a success response is set on the event (more on this in the next section) and fires that success response. Additionally, a "success"/"fail" is sent to the recorder component in order to set the status of the event that was just evaluated and to increase the current index on the event list if successful (event here means the current tile interaction). On fail, a similar check is used to determine if there is a fail output and sends a status. Currently, the index will not change on a fail so the user can try multiple times if they fail a step.

6.2.3.5 Response Classes

Response classes are models used for executing an action when a user completes a step in the sequence or fails one. Currently, only the tile response class exists, but future classes (such as Twitter) should follow the inversion of control principle [17] by having a generic execute() method. The response class will only concern itself with controlling how execute() operates, but should not determine when the overall application calls this function. For the application, it doesn't care what type of response is attached to an event and only needs to call the assigned execute() method. It should be noted that these classes can also exist on the server should the evaluation function be migrated.

```
tileResponse = function(tileId, cmdString,param1,param2){
  this.tileId = tileId;
  this.cmdString = cmdString;
  this.param1 = param1 ? param1: null;
  this.param2 = param2 ? param2:null;

  this.execute = function(){
    socket.emit("tileCmd",this);
  }
}
```

Listing 6.3: example response constructor

7
User Testing & Future Steps

This chapter will detail the user testing that was performed in order to evaluate the initial implementation of the prototype. The chapter will describe the testing procedure and the results and feedback gathered from testing.

7.1 Why Perform Testing?

As mentioned in section 1.3 in chapter 1, an essential component of design science research is the development of design artifacts/prototypes. As part of this cycle, prototypes must be tested in order assess their usefulness in the field. During this research, user testing was performed in order to gather suggestions on how to best enhance the current prototype. With this feedback, a more robust prototype can be further developed before conducting experiments with more users.

7.2 Test Setup & Proceedings

A group of students was selected to fulfill tasks that mimicked the fast food scenario given in chapter 5. Some of the students had developed with Tiles before, although only in a textual and programmatic manner, while others had no experience in programming for tile squares. Students with a technical background were chosen as they were more likely to articulate their concerns with the prototype itself in terms that could be easily reduced to software requirements.

The tasks given to the students ranged from simple to complex. Students had to:

1. Create a sequence of two interactions (turning on the grill)
2. Add a time limit to the second interaction (simulate when the food is cooked)
3. Create four new interactions to represent adding ingredients with each interaction having a success output (they could choose the output freely)
4. Add a fail event for the previous four interactions. Use the copy from previous buttons to quicken your work.

The students were rotated for each of these tasks while the author observed. Points of interest included how long each student needed to complete a task, how often they asked the author for an explanation of functionality, and what actions they performed when given degrees of freedom. In addition to passive observation, students were encouraged to narrate their feelings on the system during use. Finally, after all tasks had been completed, a brainstorming session was held where students could give their thoughts on future functionality of Tiles Recorder. If the conversation came to a still, the author prepared a set of questions to generate discussion such as:

- Did the tangible aspect feel useful and realistic?

  - Would you prefer this aspect over coding by hand or by filling out forms instead?

- Did you feel that you had enough (or too much) information presented in the UI?
- Would a graphical interface for selecting outputs (such as a virtual tile) be useful instead of web forms?

7.3 Feedback and Observations

Overall, the students found the tangible aspect to be innovative and unique. They expressed that they had never worked with tangible programming before and agreed that setting up a sequence by interacting with the tiles squares was much easier than coding by hand. They also found the GUI pleasing, with the students mentioning that they liked the minimalistic style of the interface. One student, upon noticing the tile output icons from the Tiles Card Game in the options menu, expressed that using icons he was familiar with was a good addition. However, they also noted some potential improvements:

- The activate button needs to be more clear in meaning.
  - One student mentioned renaming it to “Go” or “Run”
- Add a tooltip to the copy icon as the icon is not clear without a previous explanation
- The failure color (“red”) on the interaction chain links was noticeable, but the blue color for success was hard to see from a distance
- Hiding the success/fail buttons until hovering over them is helpful to prevent information overload, but a summary of added success/fail outputs would be beneficial once they are created
- The time limit form could be expanded to add a “before” and “after” limit (after 1 minute but before 10 minutes has elapsed, for example)

After discussing the UI, we began brainstorming new functionality that would allow for the solution to be more expressive. While the author has noted all suggestions for the sake of completeness, these new, non-core functionalities would need to be assessed to determine their usefulness for end-users.

Sub-sequencing

The ability to create sub-sequences would reduce the overhead for creating outputs or timings on each step. For instance, if a user needed to complete all the steps in a recording within a minute, they'd have to set individual time limits for each interaction node that added up to a minute. Having sub-sequences would remove that need and it would allow for a logical grouping of interactions that isn't represented by the current prototype. Sub-sequencing could also be configured so that the order of the sub-steps don't matter while the order of the overall sequence does. This would create a logical AND without needing order.

Branching

If a process deviates in some manner based on user interaction, branching sequences could be beneficial. Going back to the fast-food example, it is possible that an order could be wrong and a new set of steps has to occur. Thus, there needs to be a branching path where (1) the user interacts with a tile to say that the order is complete and (2) the order is incomplete and needs more attention. Students noted that it should also be possible to merge branches to reduce overall visual complexity. Figure 7.1 provides a mock-up of how a branch in the recording might look like.

Looping

Another suggestion is that the executor should be able to fall back to a previous state on fail. Going back to our previous example, one can see the use of a loop in the case of a wrong order. The sequence would need to loop back to the beginning of the assembly process in most cases. Figure 7.2 shows how a loop could be represented on the interaction chain.

Aliases

Users also thought it helpful to alias tile names in the app itself. While

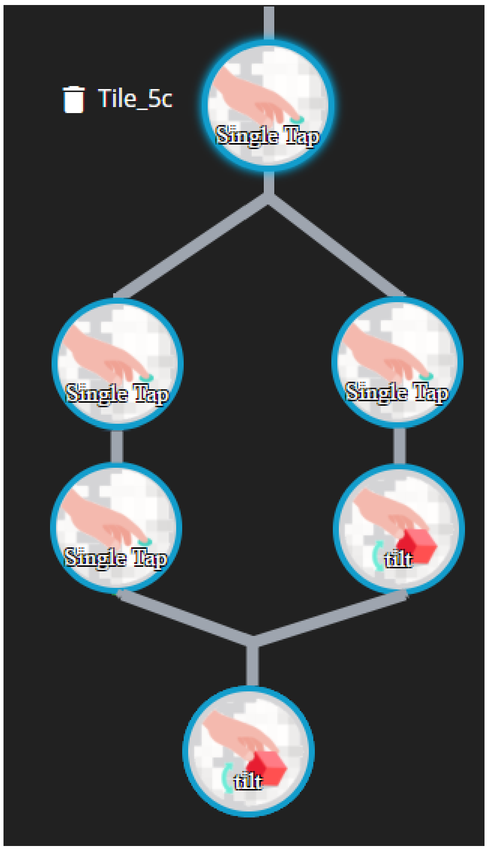


Figure 7.1: Mock-up of branching functionality

it is currently possible to give tiles aliases (such as "Lettuce") in the Tiles Mobile Gateway application, the students still thought it was easier to have the aliases attached to Tiles Recorder rather than the mobile app.

7.4 Reflection

Overall, it is pleasing that the physical programming/tangible aspect was well-received and testers thought that using TUIs was more intuitive than using textual code. There was no critique of the tangible aspect from the students, but the author did note that the tile squares had some issues with false-positives. As mentioned in the design chapter (chapter 5), this was a known problem and was handled by using filters. However, during the execution phase, tiles would sometimes misfire interactions, causing failures or successes unintentionally.

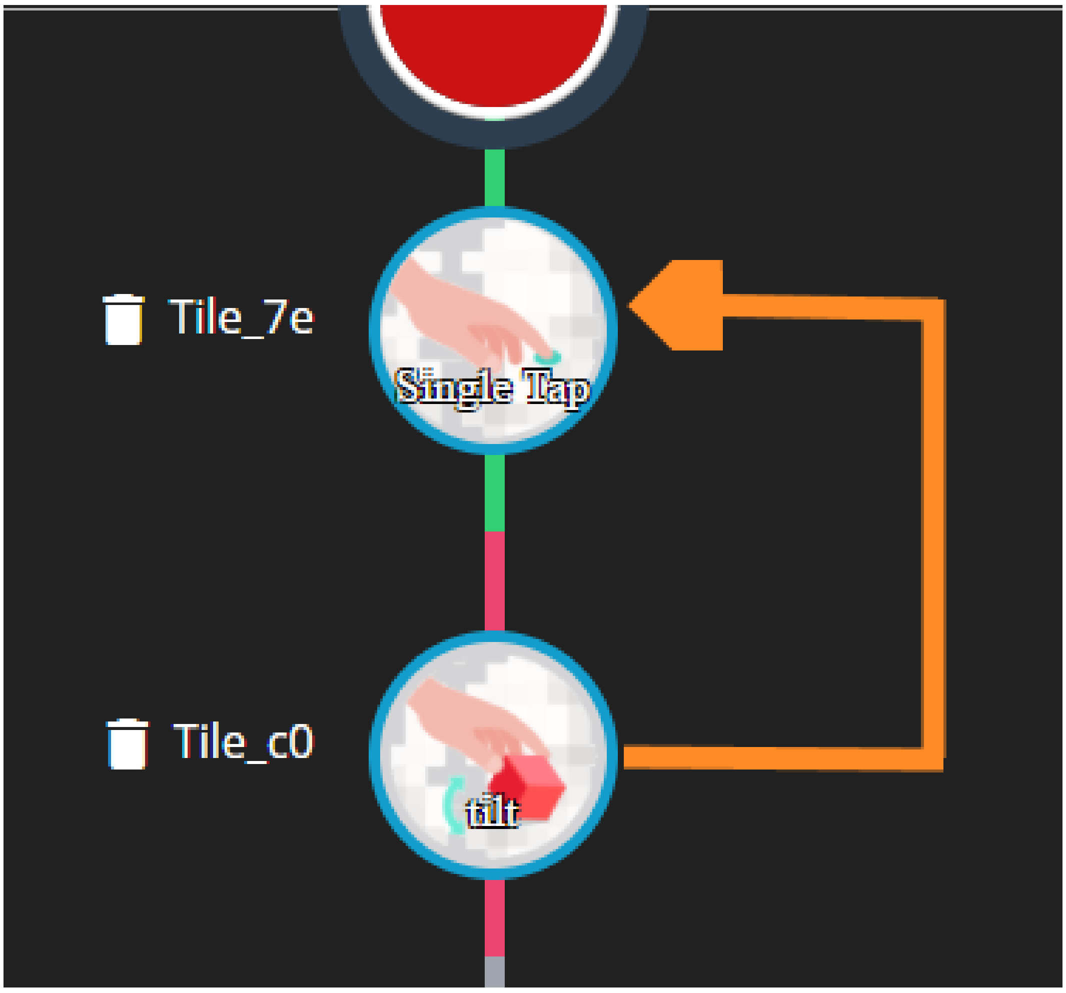


Figure 7.2: Looping

Thresholds for interaction is a problem in general with tangible programming, and hopefully future versions of the tile squares' firmware will allow for more consistent recognition of user interactions.

As for the visual aspects, it appears that, if the suggested functionality from the students is implemented, Tiles Recorder will become more involved in visual programming. As Tiles Recorder stands today, there is more program visualisation than visual programming occurring. That is, showing the state of the program as it executes rather than programming logic or functionality visually. One can see that with branching, grouping/sub-sequencing, and looping, there is a need for more diagrammatic controls in order to handle sequence flow. It is in the author's opinion that, while diagrammatic programming can be beneficial, future research should also explore how these functionalities could be implemented in a tangible way.

8 Conclusion and Future Steps

At the beginning of this research, the author was presented with the problem of finding a new way to easily create applications for Tiles. In order to solve this problem, a methodology was created following the principles of design science research. This methodology included studying the existing literature, assessing the state-of-the-art, developing a prototype based on the literature and own research, and testing that prototype.

In chapter 3, a foundation of knowledge was created by researching previous projects regarding end-user development. This knowledge included defining who an end-user is and what programming paradigms exists to help ease development. These paradigms included textual DSLs, visual programming (including diagrammatic and block-based), physical programming and finally programming by example. This understanding of end-users and paradigms was then translated into end-user development and IoT domains. A look at the state-of-the-art for end-user development in IoT was presented and examples such as Node-Red and IFTTT were highlighted.

Chapter 4 took the knowledge from the literature review and examined how Tiles differed from traditional IoT applications. The needs of Tiles were compared to the state-of-the-arts in order to find a gap between what is available today and what is missing due to the unique aspects of Tiles. The interactivity of tiles squares was particularly noted and was the major factor in determining to go with a mixed-paradigm approach between program visualisation, physical programming, and programming by example. The literature review along with the discussion of their relevance to Tiles sufficiently covers SRQ1 (What aspects of Tiles are unique for its domain in IoT and for application development?)and SRQ2 (How do current tools and programming paradigms fulfill the requirements for development with Tiles?).

With a set of paradigms selected, chapter 5 introduces the reader to Tiles Recorder, a prototype for developing Tiles applications. Included in the design was a discussion regarding what niche Tiles Recorder fits and how the cognitive dimensions framework could be applied to the designed application. Chapter 6 goes over how the prototype was actually implemented as well as giving an overview of the code. Being that a large portion of the software's value is graphical, an overview of the UI is included. Finally, chapter 7 summarises user testing that was performed with the implemented prototype. Users gave feedback on both the UI and how they felt about the overall concept of programming by interacting with physical tiles in their environment. With the prototype and testing, SRQ3 (How can the paradigms best be used to create a tool or language for Tiles?) is answered.

User testing also serves as a launching point for future research. The immediate next steps would involve implementing the desired UI changes as recommended by the testing group. Next, an evaluation of the usefulness of the suggested functionalities could be performed. Once the prototype has been enhanced, further field testing within various domains with more users would be advisable.

Bibliography

[1] B.A. . Nardi and J.R Miller. An ethnographic study of distributed problem solving in spreadsheet development. In *An ethnographic study of distributed problem solving in spreadsheet development*. ACM Press, 1990.

[2] H. Asand and A Mørch. Super users and local developers: the organization of end user development in accounting company. *JOEUC*, 18, 2006.

[3] Barbara Rita Barricelli and Stefano Valtolina. *Designing for End-User Development in the Internet of Things*, pages 9–24. Springer International Publishing, Cham, 2015.

[4] Susanne Bødker. When second wave hci meets third wave challenges. In *Proceedings of the 4th Nordic Conference on Human-computer Interaction: Changing Roles*, NordiCHI '06, pages 1–8, New York, NY, USA, 2006. ACM.

[5] M. F. Costabile, D. Fogli, P. Mussio, and A Piccinno. End-user development: the software shaping workshop approach. *End-User Development*, pages 183–205, 2006.

[6] A. Cypher. Watch what i do: Programming by demonstration. *The MIT Press*, 1993.

[7] G. Fischer. Domain-oriented design environments. *Automated Software Engineering*, 1(2):177–203, 1994.

[8] Gerhard Fischer, Elisa Giaccardi, Yunwen Ye, Alistair G Sutcliffe, and Nikolay Mehandjiev. Meta-design: a manifesto for end-user development. *Communications of the ACM*, 47(9):33–37, 2004.

[9] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131 – 174, 1996.

[10] Adam Greenfield. *Everyware: The dawning age of ubiquitous computing.* New Riders, 2010.

[11] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswam. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(1645-1660):11, jan 2013.

[12] Alan R. Hevner. Design science in information systems research1. *MIS Quarterly*, 28:75–105, mar 2004.

[13] Alan R. Hevner. A three cycle view of design science research. *MIS Quarterly*, 19, jan 2007.

[14] Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. 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*, CHI '09, pages 975–984, New York, NY, USA, 2009. ACM.

[15] Steven Houben, Connie Golsteijn, Sarah Gallacher, Rose Johnson, Saskia Bakker, Nicolai Marquardt, Licia Capra, and Yvonne Rogers. Physikit: Data engagement through physical ambient visualizations in the home. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1608–1619, New York, NY, USA, 2016. ACM.

[16] Hiroshi Ishii. Tangible bits: Beyond pixels. In *Proceedings of the 2Nd International Conference on Tangible and Embedded Interaction*, TEI '08, pages xv–xxv, New York, NY, USA, 2008. ACM.

[17] Johnson and Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, jul 1988.

[18] Mike Kuniavsky. *Smart things: ubiquitous computing user experience design.* Elsevier, 2010.

[19] Monica Maceli and Michael E. Atwood. From human crafters to human factors to human actors and back again: Bridging the design time - use time divide. In *Proceedings of the Third International Conference on End-user Development*, IS-EUD'11, pages 76–91, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] S. Mora, F. Gianni, and M. Divitini. Rapiot toolkit: Rapid prototyping of collaborative internet of things applications. In *2016 International Conference on Collaboration Technologies and Systems (CTS)*, pages 438–445, Oct 2016.

[21] Simone Mora, Monica Divitini, and Francesco Gianni. Tiles: An inventor toolkit for interactive objects. *The International Working Conference on Advanced Visual Interfaces AVI*, 29, jun 2016.

[22] Simone Mora, Francesco Gianni, and Monica Divitini. Tiles: A card-based ideation toolkit for the internet of things. In *Proceedings of the 2017 Conference on Designing Interactive Systems*, DIS '17, pages 587–598, New York, NY, USA, 2017. ACM.

[23] B. A. Myers, S. E. Hudson, and R. Pausch. Past, present and future of user interface software tools. *ACM Transactions on Computer Human Interaction*, 7:3–28, 2000.

[24] Brad A. Myers. Taxonomies of visual programming and program visualization. sep 1989.

[25] Allan Paivio. *Mental Representations: A Dual Coding Approach*, volume 9. Oxford Psychology Series, 1986.

[26] JF Pane. *A Programming System for Children that is Designed for Usability.* PhD thesis, Carnegie Mellon Univeristy, 2002.

[27] JOHN F. PANE, CHOTIRAT "ANN" RATANAMAHATANA, and BRAD A. MYERS. Studying the language and structure in non-programmers' solutions to programming problems. Technical report, Computer Science Department and Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA, 15213, USA, Computer

Science Division, EECS University of California, Berkeley Berkeley, CA 94720-1776 USA, dec 2002.

[28] K. Popper. *The Logic of Scientific Discovery.* Routledge, Taylor & Francis, 1959.

[29] Jennifer Rowley and Frances Slack. Conducting a literature review. *Management Research*, 27(6), 2004.

[30] Orit Shaer and Eva Hornecker. Tangible user interfaces: Past, present, and future directions. *Found. Trends Hum.-Comput. Interact.*, 3(1&#8211;2):1–137, January 2010.

[31] B. Shneiderman. *Leonardo's Laptop: Human Needs and the New Computing Technologies.* MIT Press, 2002.

[32] Andrew Cyrus Smith, Nomusa Dlodlo, and Nobert Jere. Towards an internet of things tangible program environment supported by indigenous african artefacts. In *Proceedings of the First African Conference on Human Computer Interaction*, AfriCHI'16, pages 176–181, New York, NY, USA, 2016. ACM.

[33] Tiago Boldt Sousa. Dataflow programming concept, languages and applications.

[34] RH Trigg, TP Moran, and FG Halasz. Adaptibility and tailorability in notecards. In *INTERACT*. Elsevier Science Publishers, 1987.

[35] Todd Zaki Warfel. *Prototyping: A Practitioner's Guide.* Rosenfield Media, nov 2009.

Appendices

A | Code Repository

The code used to develop the prototype may be found on GitHub at the following link: `https://github.com/dasatcher/tilesRecorder`

How to Install and Run

You'll need first install both NodeJS and NPM. After installing these two system dependencies and cloning the repository, you'll need to run two commands to install the project dependencies.

From the base directory of the project, run:

npm -install

This will install all dependencies for the application. Next, run:

npm run build

This will build the application into the public folder. Finally, run:

npm start

This will run the application on localhost:3000

Note: Using the application requires Tiles squares as well as the Tiles gateway application.

B | Glossary

Design Science A set of cycles making the design science research methodology. Includes a relevance cycle from field testing, a design science cycle for building and testing design artifacts, and a rigor cycle for grounding the design of artifacts.

Domain-specific-language A language that is limited in expressive power to a certain domain. HTML is a DSL of XML because HTML is simply XML regeared for the web domain.

DODEs Domain-oriented design environments. Visual representation of domain knowledge used in visual programming.

EUD, EUP, EUSE End-user development, end-user programming, and end-user software design. EUD is for describing the actions end-users take to modify software or creating new software artifacts. EUP is just the actions taken to create new software artifacts while EUSE concerns itself with the quailyt of software that end-users create.

IoT Internet of things. Networked devices that sense from their environment and each other.

MQTT Lightweight messaging protocol often used for transmitting messages between IoT devices.

Node Interaction Chain In Tiles Recorder, the node interaction chain is the list of nodes (interactions) that is displayed under the recording button.