Storycoding - Programming Physical Artefacts for Research Through Design

Jussi Mikkonen
University of Southern Denmark
Kolding, Denmark
jumi@sdu.dk

Christina Fyhn
University of Southern Denmark
Kolding, Denmark
cfni@sdu.dk

ABSTRACT
Making programmable physical artefacts and prototypes has inherent value for Research-through-Design (RtD) based HCI. Furthermore, the abstractions and representations within RtD and programming are vastly different, such as between observations, storyboards and the code. Studies have shown that the program of an artefact influences the RtD outcome, but there is also a disconnect between the observations of use and the abstractions involved in the programming. How can we program an artefact so that the code can be created, modified and reflected upon based on directly observable and non-technical abstractions? In this paper we present Storycoding, a computational-thinking based method for programming that focuses on bridging the representational abstractions. Using Storycoded artefacts, we examine programming in light of RtD. We discuss how Storycoding enables programming that is directly observable between the use and the abstractions, being respectful towards RtD. Finally, we conclude with implications towards HCI research and practice.

Author Keywords
research through design; human-computer interaction; artefact behaviour; computational behaviour; programming.

CCS Concepts
+Human-centered computing → Interaction design process and methods;

INTRODUCTION
Making programmable physical artefacts and prototypes has inherent value in Research-through-Design (RtD) based Human-Computer Interaction (HCI) [45, 80, 81]. Constructing, using, discussing, and documenting interactive "behaving artefacts" is fundamental - the interactive behaviour and how it can be varied has been discussed in those seminal works. The scope of the myriad abstractions and representations (e.g. observations, sketches, storyboards, paper-prototypes, wizard-of-oz, programs) involved in the development of those artefacts and prototypes is immense [16, 38, 45, 48, 57, 60, 77, 80].

Studies have shown that the program (the computational behaviour) of an artefact influences the outcome of RtD process and the resulting UX (e.g. through design decisions [26, 50], prototype platform [34], or algorithm [28]). Furthermore, there is also a disconnect between the observations of use and the abstractions involved in the programming in general - indicating major challenges towards building real-world prototypes [18, 44, 47, 49, 65, 66, 67].

While there are studies that focus or discuss the implications of computational behaviour (e.g. [25, 29, 57, 56, 71], we have not been able to find papers which would propose a systematic way of programming computational behaviour for RtD. In fact, "more research is needed for computational behaviour" [56]. Furthermore, we point out needs originating from design-inclusive UX RtD: to "Provide rich prototypes to study aspects of experience and interaction", and a call for "new (lower level) design methods, tools and techniques" [72].

How can we program an artefact so that the code can be created, modified and reflected upon based on directly observable and non-technical abstractions? In this paper we present Storycoding, a computational-thinking based method for programming that focuses on bridging the representational abstractions. These abstractions (storyboard, state machine, flowchart and a program) link together to form an organised and structured model of the artefact’s computational behaviour. Storycoding utilises Goodwin’s contextual configurations [33] to extract the artefact’s behaviour from a storyboard depicting it’s use.

We evaluated Storycoding by prototyping functional artefacts, conducting semi-structured interviews, and examining developed representations. By analysing the collected material using thematic analysis, we continue to examine Storycoding in light of RtD. Our results show that Storycoding answers to the needs [56, 72] by providing a powerful and flexible method for programming functionality that is observable and useful for interdisciplinary discourse. We discuss how Storycoding enables programming that is directly observable between the use and the abstractions, being respectful towards RtD. Finally, we conclude with implications towards HCI research and practice.
RELATED WORKS

For RtD, constructing, discussing, and documenting "behaving artefacts" is natural. This interactive behaviour and how it can be varied has been discussed in the seminal works [45, 80, 81]: this spans RtD very broadly, between designing critical reflection for the showroom, to providing product ideas and development opportunities to engineers. The idea of behavioural iterations of the research artefact, researching through design(ing) is the very essence of RtD. As an example, the iterative approach is described through Significant Screwdriver and Whispering Wall in the paper by Bardzell et al. [9]. Thus, we understand that a programming a behaviour for RtD artefact should be easy to create and have variations to, during and after construction. Furthermore, it should be documentable so that it can be discussed and even reproduced [80].

Our related works has a specific focus on instruction-based programming in RtD (e.g. block-code, C/C++, Java). Fernaeus and Sundström note that "A specific programming language or protocol may for instance make it easier for certain actions to be performed" [28]. In the context of this paper, we focus primarily on instruction-based programming and similar approaches, such as the flow-based systems typical to musical systems (e.g. [46]), are beyond the scope of this paper, and thus left for future work.

Perceiving Computational Behaviour

Computational Behaviour is used to describe how an artefact behaves in a situation, reacting to users and changes in the environment, based on its programming. Thus, there is a clear intent to program a certain way of behaviour.

Computational behaviour has been discussed in RtD context in [19, 24, 25, 56, 57, 71]. They all identified the role of the behaviour as being central, influential, unintendedly influential, or called for further research on the artefact behaviour. The behaviour may be combined with for instance Wizard of Oz, as was done in [8, 40], but in general we identified it as something that provides character to the artefact. Further research towards behaviour is called by [56], specifically stating "more research is needed for computational behaviour".

However, the programmed functionality has been seen to be of lesser value in terms of perception. A study encompassing several RtD projects states that "...artifacts are indeed seen for their mediations rather than their mere instrumental or functional purposes, the studied technologies do also have clear functionalities with at times close-ended purposes." [37]. This difference between the actual behaviour and the participants interpretation has been noted in an experiment with a physical facade [13], where the users began imagining programmed reactions which were actually not programmed. Being able to discern the interpretation from the real behaviour suggests that the intended and interpretable computational behaviour should be traceable to code, as it allows identifying the origin of the experience.

This points towards User Experience (UX) as being central to RtD, focusing on emotions and experiences related to using artefacts. The effect of an algorithm towards the experience has been proposed [3], suggesting an interest with going deeper towards the program. Design-inclusive UX has been analysed by Vermeeren, Roto and Väinänen [72], who describe how design can contribute to the UX in RtD. We raise out two elements they suggest for design-inclusive UX RtD: to "Provide rich prototypes to study aspects of experience and interaction", and a call for "new (lower level) design methods, tools and techniques". In a different study, Güldenpfennig et al. [34] specifically mention "real user experiences" made available using Arduino and mobile phone APIs.

Computational Behaviour Over Time

The connection between the program and the artefact's behaviour was mentioned in several studies to be related to time within RtD. Thus, Temporal Domain consists of ways to program behaviour with timed elements or interactions that change over time. The time-scale behaviour of an artefact can be limited to short-term interactions (e.g. [28]) or longer term change (e.g. [57]). Timed elements, inputs and outputs requiring a continuous timed characteristic have been described or explored in [4, 6, 11, 23]. Time has also been used as a kind of contextual variable [35, 71]. Fernaeus and Sundström specifically make a point on taking care for designing against delays and timed elements that make the interaction slow or passive [28]. For the computational behaviour, we interpret that timed interactions should be fluent to design for, whether they happen simultaneously or over time. For the programming, it implies that this information should be available more clearly from a storyboard or a scenario, and that the temporal algorithms do not interfere with other behaviour in the program.

Abstractions within Computational Behaviour

A physical, programmable artefact can be interacted within the limits of its programming, i.e. the program is typically fixed, and any interpretations rely on perceiving this behaviour the program creates. However, understanding a concept of that artefact relies mostly on the representation, such as Sketches [16, 30], Scenarios [17, 61], Storyboards [21, 42, 70], Wizard-of-Oz [43] and analogue prototypes of different fidelities [38, 39, 59]. This scale of representations poses a fundamental challenge for programming a computational behaviour - how do they translate to an algorithm?

This difference in abstraction between a functional artefact and the concept has been specifically mentioned in several instances [18, 44, 47, 49, 65, 66, 67] as the major challenge towards building real-world prototypes. This challenge manifests through three aspects, and affects both designers and engineers. The first one is the unclear representation of time, which complicates the management of the time-related differences between a prototype and the user [47]. The second falls under the inconsistent and varying perspectives used in the representations of the behaviour, when transforming a concept to a prototype [44, 65, 67, 66]. The third is a constraint within the software used for creating the programs for the prototypes. The constraints have been mentioned to be due to the lack of representative capability between the real world and the abstractions used in programming languages and compilers [18, 49]. We call these challenges towards implementation as the 'concept to prototype gap'. We define it as an apparent lack of
means to build a functional interactable prototype, towards the computational behaviour implied by a concept.

**Programming Computational Behaviour**

As interactive artefacts need programming, they rely on exact computational representations. Computational Thinking (CT) has been coined to address the increasing complexity of computational systems; “Computational thinking is using abstraction and decomposition when attacking a large complex task or designing a large complex system.” [78]. It is seen as a way of bridging the abstract and reality; “It shares with engineering thinking in the general ways in which we might approach designing and evaluating a large, complex system that operates within the constraints of the real world.” [79]. CT has been characterised as an approach leading to an implementation: “We consider computational thinking to be the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms.” [1]. Weintrop et al. [76] have further opened up what CT consists of. They defined four practises (Data, Modeling & Simulation, Computational Problem Solving, and Systems Thinking) that each contain several aspects of that practice, to give structure to CT. Of these, our work draws heavily from the latter two practises.

**Visual Methods in Computational Thinking**

For the purposes of this paper, we focus specifically on Storyboards and Finite State machines. Storyboards have been suggested to be used in CT course [75], and have been used in several cases with respect to CT [15, 22, 62, 68]. They have been perceived as rigid in terms of timing, especially if using two storyboards [20]. Additionally, Storyboards have also been used to illustrate a ‘reverse engineered’ behaviour for learning purposes [32]. Thus, storyboards are relatable for describing interactions from a human or a third-person perspective.

State machines have been mentioned as one abstraction of behaviour in CT [79]. They were originally intended as an answer for the increasing use of displays in the development of interactive systems, by providing means for designing visually [54]. State machines have been extensively explored with interactions, such as with event-based virtual systems [58], where the state-transitions are based on state-charts [27, 36]. State-transitions and stable states have been specifically discussed in interaction context [7, 52]; in both cases the stable states are contexts in which somewhat similar or continuous interactions take place, and the transitions are user-documented actions which change the stable state from one to another. We note them as having potential for visual clarity.

**Technical Means**

Bridging the gap has been attempted earlier with different technical methods, such as automating or guiding the coding or behaviour-translation process (e.g. Starting from storyboards [74, 31], using modelling language tool [64], using state machine as visual translations [54], or using flowchart-cards to guide the programming [51]). Furthermore, the complexity of the code or the hardware can be fully or partially hidden (e.g. .NET gadgeteer [73], Arduino Lilypad [14]), and some programming environments utilise altogether different representations for the code (e.g. using block code [53, 63], coding by controlling the data streams [55], [69]). However, these alone do not address the core issue of managing the interactions implied by the complex behaviour. They are primarily focused on solving technical issues - unrelated to making the real-world behaviour understandable, i.e. they are not well suited for the purposes of RtD. In some cases, the visual clutter might even add to the complexity of the task (e.g. large programs when using block code [41]). Tracing the interaction from the code-level directly back to the original concept is difficult, even more so, when the interaction and the computational behaviour become complex.

**STORYCODING**

In this section we describe the Storycoding method, which has been developed for programming computational behaviour. Our method combines existing methods from design and engineering in a specific way - to create a visually representative path that can be examined by those without in-depth programming experience. Storycoding is starts with the creation of a storyboard, followed by transforming it to a state machine, and then a flowchart as intermediate steps, ending in a program. The Storycoding process is shown in Figure 1. Central to Storycoding is the use of "behaviour"-states. These states define an easy to define context, or activity, for the prototype. Each of these states thus contain a function defining a small aspect of the overall behaviour.

The states have direct parallels to Goodwin’s ‘contextual configurations’ [33]. Where Goodwin describes how the contextual configuration changes during a game of hopscotch, we similarly identify contextual situations from a storyboard, and utilise those for creating a state machine. Goodwin describes "semiotic fields", when referring to e.g. gestures and speech, when meanings are formed. Thus, the artefact use in a storyboard can be translated to separate states (for the state machine) by identifying contextual configurations. As contextual configurations count for temporal and experiential activities, changes in the semiotic fields visualised in the storyboard suggest parallels to the real-world conditions that trigger change in the computational behaviour. Since the storyboard and the state machine are both from the human-perspective, perceiving the connection between them is straightforward. This is also why two intermediate steps are needed. The state machine is a technical drawing, which can be connected to a flowchart. However, between the state machine and the flowchart is a perspective switch: the flowcharts are written from the machine-perspective, making them relatable to the final program. Regardless of the perspective switch, the main-level flowchart implementing the state machine’s overall structure retains visual connection to the state machine.

**The Guard’s Life**

Storycoding uses a storyboard to make the real world relatable. To illustrate how the real world relates to the program, we show how the representations connect between the abstractions. Specifically, we draw a parallel between the human behaviour and the states, by examining a hypothetical day of a Royal Guard. The Guard’s duty is to guard The Queen,
Figure 1. The Storycoding process. Starting from the left, A) identify contextual configurations and their changes to define specific states in a Storyboard. This is followed by B) where a state machine is created using the defined states in the storyboard. The states are then C) opened up as Flowcharts, and finally D) transformed to code. Each transformation (B,C,D) is backwards traceable, indicated with small letters.

but The Guard can also be in the barracks, or have time off. What follows is represented as a storyboard (Figure 2.a), with the corresponding state machine (Figure 2.b). These three situations form The Guard’s states, as in each contextual configuration visible in the storyboard, The Guard’s behaviour is very different. The sergeant commands The Guard to go from the barracks to guard duty, as well as back from the guard duty to the barracks. From the barracks, The Guard can also go to time off as commanded. From time off, he can come back at an agreed hour, or if partying too hard, be commanded back to the barracks by the sergeant. Thus, each transition from one state to another can be defined and there can be more than one transition from one state to another. For example, coming back to barracks can be timed, or commanded by the sergeant. Each of these situations or states can be seen as a different contextual configuration.

Figure 2. a) The Guard’s Life has three different ways to behave (three different contextual configurations), and is commanded from above by the sergeant, or controlled by time. Figure 2. b) Below the storyboard is the corresponding state machine. The arrows depict the causes for transitions, which are commands with the exception of one time-based transition.

Similar to The Guard, artefacts created using Storycoding have behaviour states, and transitions which change the behaviour state from one to another. These states and transitions are extracted from the storyboard that has been created from the user’s or a third person perspective. The transitions are triggered by either time, interaction, or information, and each transition is specific and limited to each state. Therefore, each state can be examined as a separate, limited and simple situation of use. It should be noted, that the artefact stays in a state unless it is being influenced by a specifically identified action or event. By combining several states together, a larger use-case spanning several contextual configurations can be integrated to the computational behaviour.

**Shift to Computational Perspective**

Once the state machine has been created, it is used as a structure for series of "if"-"else if"-"else"-structures. These are best visualised using flowcharts, for which we have adapted Flowcards [51]. Flowcards provide two structured flowchart elements that are limited to a process-card (rectangle) or an if-card (a question with two outputs, represented by a diamond). Backward loops are not allowed within functions - only the main flowchart can loop back to the beginning. These limitations make them useful for simplifying the resulting algorithm. An example of the main flowchart relating to The Guard’s behaviour is shown in Figure 3. With this step, the perspective in Storycoding changes: from this point forward, the behaviour is written from the introspective perspective of the artefact. For practical purposes of selecting in which state to operate, the artefact is essentially programmed to ask "in which way should I (the artefact) behave?" Drawing back to Goodwin, the artefact assesses its 'semiotic fields' to determine if it should change its own behaviour. In practice, it will check if a timer has run out, an input has changed, or some other change has happened.
Figure 4. The Guard’s life main code (BBC micro:bit makecode). It is derived from the flowchart in Figure 3. Note that all programs use a variable called pstate to remember "program state".

What makes the Storycoding method unique is that this introspective questioning is constrained to two places. On the main-loop flowchart, seen in Figure 3, the artefact selects what state (contextual configuration) it is currently in. Then, within each state it will execute code related to that situation. For example, if The Guard is in the barracks, it will listen only to commands that change the state to "Guard Duty" or to "Time Off". There is no need to focus on any other command or time, as they have no relevance to that context. This aspect simplifies programming the computational behaviour to simple if-else commands, and each state can be backtracked to their original visual representation in the storyboard.

An example Storycoding of an egg-timer made with Arduino, with full descriptions and comments including a storyboard, state machine, flowcharts and the code can be found in github https://github.com/kryt/eggtimer.

STOR YCODING PROTOTYPES AND DATA COLLECTION

Storycoding has been used and developed for several years. For the purposes of this paper, we examined an intense project to assess how the storycoding can be learned and how it reflects with RtD. Eleven student-teams created a prototype each, in roughly two weeks’ worth of full-time work in a project focusing on learning interaction design programming. While Storycoding has been used with Arduino and Java, these prototypes were built using BBC Micro:bit, which uses instruction-based visual-block programming similar in structure to text-based programming. All teams created functional prototypes that were then used for one day of user testing. We later found out that one team did not use Storycoding due to team-based reasons, thus they are excluded from this study.

Method

Data was collected through Storycoding materials (Storyboards, State machines, flowcharts, programs), observing the prototypes in action, conducting semi-structured interviews [10] with each team, and collecting project reflection students created during a large group discussion after the project ended. These were then analysed using thematic analysis [12]. Out of the ten prototypes we chose two for more detailed inspection, shown in use in Figure 5.

Both of the selected examples are user-identifying, made possible with the help of another micro:bit through the use of wireless communication. This process is the only aspect of both programs that happens outside the Storycoded structure, as a background event. We clarify this up-front, as the events are not part of the Storycoding structure, influencing variables that get updated automatically, becoming useable similar to inputs. The identification operates in a similar way in both, using a prototype-specific pre-determined radio-channel that excludes all other micro:bits from the communication. The micro:bit worn or carried by the user sends a message using the on-board radio. Upon a successful reception, the receiving micro:bit calculates the signal strength, which is a good-enough equivalent to a distance between the two micro:bits.

Postbox

Postbox is a user-identifying mailbox that lights up when there is physical mail within and allows the nearby user to open the lid by waving a leg over a sensor on the floor. The Postbox combines user identification with a light-detecting resistor, to identify if the user’s leg is placed over the sensor. When the user has picked up the mail, they can either wave the leg again or leave, closing the Postbox.

The storyboard of the Postbox is shown in Figure 6, with the corresponding state machine and flowchart in Figure 7. The final full code is shown in Figure 8. Firstly, the actions and results represented in the storyboard can be seen as transitions and states in the state machine. State-specific actions that change the behaviour to another state can be identified between the two. Secondly, the main state machine loop creates the main flowchart structure on the left-hand side. In the main flowchart, each if-card - process-card pair represents one state of the state machine. Six states result in six pairs. Each pair contains a function, which are represented below and on the right-hand side of the main-loop flowchart.

As an example, the first two frames of the storyboard are contained in state "idle" (marked with 1). Change between the second and the third frame indicate an action, which in the state machine correspond to the state change between "idle"(1) and "unlocking" (2). Frame three corresponds to
The next four frames correspond to the state "open" (4), and so on. This is a good match between the storyboard and the state machine.

Similarly, the specific actions in the state machine correspond quite well to individual flowcarded flowcharts (both seen in Figure 7). For example, state "idle" (1) and it’s exit arrow to the next state becomes one if-card: if the condition in if is true (user within 1m), then new state will be "unlocking" (2). A good example of multiple exit exit-arrows are within "unlocking" (2): one exit goes to state "idle" (1), and another goes to "wait for the foot to go away" (3). The corresponding flowcards can be seen under 2 (unlocking), where the former is represented by the first if-card and process-card combo, and the latter with the second if-card and the process-combo. Both if’s positive result in a state-change.

The state machine is drawn using two colours, with the red colour used for modifying the first version. Notable is the inclusion of human activity of filling the mailbox to the first iteration (state called "filling"). This was bypassed, as the light-indication of the mail did not need a separate state. It runs parallel to the Storycoded program, being visible as function "Post_in_out" in Figure 8. Similarly, the use of timer was left out, as it was not needed in the end. The flowcharts were augmented to include extra states of 3 and 5. Additionally, some if-branches in the flowcharts are not drawn clearly, however, they simply skip the next flowcard if the if-card result was negative.

**Medicine Dispenser**

Medicine Dispenser supplies correct medicine between pre-defined intervals, identifying the user and notifying them at the right time. The identification and the notification are built around a wearable device, containing another micro:bit. Both parts of the prototype utilise battery and a set of LEDs, and the main dispensing unit also houses a servo-based mechanism for distributing pills. Once the Medicine Dispenser is loaded and started, it begins counting time until the moment medicine is needed. Once this moment arrives, the dispenser changes colour and sends an activation signal to the wearable. The wearable lights up, while sending an identifying signal. When the user comes right next to the dispenser, the mechanism activates and dispenses pills to the cup. After this, the timer is reset and the process begins again. The storyboard is shown in Figure 9.

This prototype was initially coded without the use of a state machine. This resulted in a flowchart (shown in Figure 10) and a code that was very difficult to debug, remaining only partially functional. The code waited in a backward loop at every step of the way, before it could move forward to the next activity in the storyboard. After the students made a state machine (Figure 11) and re-created the flowchart accordingly, the programming process changed. They were able to debug any issues much faster and create code where the LED-based visual feedback could run parallel to the computational behaviour.

The numbering in the storyboard (Figure 9) is not intended to match the states (Figure 11). However, the identification of the moments when the timer is running is notable. As the transitions from "unlocking" to "open", via "wait for foot to go away" (3). This was added afterwards: the storyboard depicts a full movement of swiping - moving leg over the sensor, and removing it from the sensor. This also explains why two cards are added separately to the main-loop flowchart.
structure is otherwise very similar to the Postbox (containing simple if-else-structures), we show the flowchart (Figure 12) and the code (Figure 13) focusing only on the functions that create the timings.

Every time the state "timer running"(2) is entered, the timer is reset. For example, in state "idle"(1), the if-card corresponding to the state exit arrow contains also instructions to set timer. Thus, every time a timer is needed, it is set so that the following state can check for passage of time. In this case, state "timer running"(2) has an exit arrow captioned "timer has run out", which corresponds to the only if-card in the flowchart "TIMER". This moment (in the program) can then be traced back to the storyboard via the state machine, where the notification is given.

FINDINGS
We present the central findings that we gained by analysing the interviews, project reflection, and all Storycoding materials. We noticed that most students talk interchangeably about the state machine states, calling them functions. This due to each state being represented as a function, to keep similar level of visual clarity in the code as in the state machine and with the flowcards.

New Way of Programming
Storycoding was perceived as a new way of programming by the students. This was noted by all teams, remarking such as T8P2: "It was another way of programming that we had used to, but it made sense.", with one student having prior experience in design prototyping remarking T3P4: "We had to make a really detailed storyboard, regarding process before the interaction and after the interaction... so that was new to us."

The team who made the medicine dispenser (T4) had a more experienced programmer, who had been learning front-end development at his previous education. This influenced the way of programming interactions: T4P1: "We approached programming with a front-end view, using while-loops and for-loops, but in these kinds of parts of code, the code gets stuck during the process, and when the code gets stuck, we can’t do anything else. So that’s why it was a problem for us."

Thus, it was clear that the previous way of programming was first of all unsuited for reactive interactions. By skipping the state-machine, they were trying to program a direct flow based on the storyboard. This insight led into a deeper discussion between two group members the nature of their original code in Figure 10, and the Storycoded flowcharts shown partially in
Figure 12. T4P1: “Before we had the one flow, and based on that flow, the code was kinda like stuck at point until it reached a condition of yes. So this wasn’t functional for approaching this problem.” T4P2: “It was very hard debugging when we started looking for mistakes.” T4P1: “Yeah.” T4P2: “We didn’t know exactly where to start, because we couldn’t really find out where it happened. We also did some debugging in [Storycode], but since every function is split in to a statement that has to be fulfilled, ... to go to the next, it was very easy to say ‘OK, this step doesn’t work’ - so we can actually debug this specific step, and keep the rest as it is.”

T2 and T4 had an experience that was very similar with another team (2), who also made two different versions of code (their own approach vs storycoding). T2P3: “The only thing is that we have the postate 1-4, otherwise everything else was the same and code was working the same.” T2P2: “Both codes worked, but the [storycoded] one is more organised.” T2P3: “It’s more organised, more clear for people who don’t know that much coding to understand, definitely.” All three teams highlighted being able to use their old way of coding and combining it with the Storycoding as a positive aspect.

One student even proposed a best practice on how to create a new structure for a prototype. T3P4: “Always do the coding immediately after you’ve defined the new states. Always go step by step so you have simple structure at the beginning and build from that.” One team became very ambitious, with a member stating T6P2: “Putting three scenarios to one storyboard was too ambitious at first.” They then created separate storyboards and combined them together to create one state-machine, adding individual storyboard contents together one-by-one.

Interactions Becoming Modifiable
Almost all teams commented on the ability to play around with the interactions by using the visual tools. T7P1: “actually we were really guided by the flowchart while doing the code.” There were several mentions of adding new behaviours, such as T2P3: “Easier to manipulate later, if you make it into more complex code.” and T3P4: “What also worked pretty well, because we did that new approach to coding we first skipped the payment function, but then it was super-easy to add the function, because we had pretty simple structured code.”

This perception of flexibility went beyond the program. T3 had an early idea of fully integrated shopping cart, where the cost of the total purchase would be determined automatically by simply placing an item on the cart. When asked if they would be able to work on this concept in terms of making a prototype for assessing user experience: T3P4: “I think so. When we added the payment-option, we also added a new micro:bit. We learned yesterday how easy it is to add new micro:bits... I think we would have ideas to realise the intelligent shopping cart.”

Similarly, deleting and changing the interactions became a non-issue. T5P1 “We also talked about deleting the shake-function, ‘cause we were not sure if it would work in the real life... that way we could just delete that function and then, we could just move on.” Another team commented on changing the behaviour, T8P3: “We realised when we made the code that we missed some of the states, and then we were through the process, like, always making adjustments, so that was also nice to see how it can change the whole state machine, while making the code too.” This ability to adjust or change the behaviour was perceived as very useful, as other teams also noted that their initial hypotheses about the product were not always correct. Thus, in Storycoding, state- and flow- based intermediate steps clarify where the modifications should be implemented.

The teams highlighted the ease of debugging with Storycoding, T5P2 “If we didn’t use the state-way of doing it, and we didn’t use multiple functions, then if we had error, we wouldn’t, it would be hard to debug it - we would have a big line of code we would have to debug.” This is very similar reflection to T4 and T2. The experienced coder from T4 specifically remarked on the ability to fix small details, T4P1: “Small bugs are always taking the most of the time for the debugging, this lead us back to the track, then implementing the code and the flow after that was pretty fast.”
Storycoding and Collaboration
The ability to reflect on the interactions was seen a benefit, by using which ever abstraction was suitable for the task, T6P4: “It think it’s very good to have other people come and look at storyboards, for precisely that reason that it makes it easier to communicate what is it actually that this thing is gonna do.” In terms of coding, T7P2: “It was not a difficulty, sometimes we write the code because we figured out maybe it is important to give feedback at that point and that point... ... it wasn’t [difficult to pinpoint the location], because we got it from the storyboard.” The different abstractions were also seen as a way to overcome personal challenges with skills, giving a different way to manipulate the program: T5P2: “one time I was actually able to use flowcharts in [this room], to visualise it, that was like an eye opener; then we all went in here, this board, to actually do the flowcharts.” T5P1: “then you have something visual without drawing it, so maybe that’s what you are missing, something to explain from, and you don’t have to be a great drawer to do that.” T5P2: “I just needed an artifact, something to manipulate to express it.” Emphasising the tangible nature of the flowcards.

Human-Relatable Programming
Storycoding was not seen as just writing code, but as combining the human perspective to the program, the machine perspective. This spanned several teams. T3P1: “It’s like human language because you have the storyboard and you just transform it to code, so you’re not thinking during coding. ...transforming an image from something a human can understand as a storyboard or a storycode to a real code, which is easier for non-programmers as well to work with.” This led to an enhanced perception of what is actually relevant between the human and the machine. T2P1: “In the storyboard, drawn, people approaching the prototype to start the whole thing. It’s just, for the machine, that’s not very important.” This student put it most succinctly: T4P2: “There’s two different kinds of processes: the human, the user pressing a button, and then there’s the machine thinking, what does this button actually do.”

DISCUSSION
Programming and computational behaviour are at the centre of using programmed artefacts in RtD. Storycoding was used to develop functional prototypes, all of which were useable for user evaluation. Storycoding enables programming physical artefacts for Research Through Design approach, by making all programming elements and visual representations referable within the scope of the designed interaction.

Constructing Behaviour
Storycoding creates a modular "scaffold" for the program beginning with a storyboard. By going through the state machine and flowchart to the program, a visually rich documentation is created that goes all the way to the code. This also reduces the complexity caused by large programs when using block code [41]. Because of the relatability and the organised structure, this program can be expanded, altered, debugged, or made smaller with pin-point accuracy. While this makes programming interaction more straightforward, the approach may not suitable for all types of programs. Our findings show that Storycoding works with interactions reliant on interaction-history and contextual behaviour. The context can be defined visually, carried to the states, and to the code. As such, it is different from the simple contextless reactive programs, and programs which read a sensor, modify the value, and pass it to an output.

This context-centric capacity is respective of RtD [81, 80], where refining and iteration are central to the approach. Storycoding gives means for building artefacts for exploration. Finite state machine-based approach creates functional visual states, which allow the designer to alter only those states where the change is needed, leaving other parts of the program untouched. This supports the construction of rich experiences through incremental addition, and the parallel nature of the reactive code supports simultaneous interactive elements (especially temporal domain, avoiding delays).

Individual interactions or computational elements can be combined together over time, to create a rich interactive experience. This answers directly the need for rich interactive prototypes and the methods for creating them [72]. There are no theoretical limits to the amount of states or transitions, however, the detailed exploration of the scale, sub-states and effective way to develop parallelism is left for future research.

Being able to imagine and define the behaviour with a concept description is fundamentally important for manifesting the designers intentions and in RtD [26, 50], having been called upon in previous studies [56]. Ability to translate that behaviour from a concept to computational steps is needed to physically manifest the design idea, i.e. to overcome the representational gap [44, 65, 66, 67]. Equally important is the selection of a suitable programming tool [28], to avoid constraints of programming languages and compilers [18, 49]. Storycoding bypasses these by providing several interrelatable abstractions, and supports the programming of computational behaviour by providing manipulable mental model and structure. With regards to the difficulty of working with time [47], Storycoding uses state-based structure to provide clearly identifiable temporal anchor points.

Storycoding is suitable for wide audience, irrespective of level of coding experience. It allows an interdisciplinary team to discuss details of the computational behaviour by referring to any representation, abstraction or perspective. Students were empowered by the method, going as far as saying T3P4: “For me that was a really good approach, and I think I will continue approaching coding like that, because it’s really understand-able.”

Research Through Designing Behaviour
The ability to walk the path from a concept to a prototype and back is beneficial for the reflection and analytical approach of RtD. The students demonstrated the ability to flexibly change and modify their code based on the observations. They also showed capacity to anticipate difficulties in the interactions. These properties turn Storycoded programs into research-objects, where a hypothesis can be placed, evaluated and analysed. This extends to iterations, being able to make
We presented Storycoding as a way to construct artefact's prototyping structures, benefiting research with iterative, or Wizard-of-Oz, development processes. The results directly indicate that being aware of the machine's perception with respect to researcher's understanding of programmed computational behaviour. Benefits of understanding this conceptual difference was seen in a previous study, when the participants' imagination brought new behaviours, when compared to the programmed behaviour [13], p.164).

The results directly indicate that being aware of the machine's perceptual needs with respect to the interaction improves constructing the Artefact’s behaviour. Specifically, Storycoding enables exploring questions within RtD that arise from the artefact behaviour, including modifying and re-designing the behaviour. For example, the artefact can be designed to contain vastly different ways of behaviour that can be triggered by designing the behaviour, or combining a state with e.g. Wizard-of-Oz, benefiting research with iterative, or Wizard-of-Oz-approaches [9, 8, 40].

Being able to trace the visual observations through the representations within Storycoding enables more specific control over RtD-explorations, akin to augmenting the artefact behaviours such as with PeP+ by Deckers et al. [25]. Thus, Storycoding has a direct influence on understanding the impact of the artefact, as students were able to manipulate the behaviour to their needs and goals. The notion of the "Artefact’s ability to perceive the world" also calls for future research on how to design from the machine perspective outside engineering, suggesting complementary knowledge towards aiding technological research, it being one of the outcomes of RtD [80].

Storycoding provides means to discuss these RtD-outcomes, giving ability to provide insights towards the technical opportunities, and even link the computational behaviour and the human behaviour through the contextual configurations and visual representations.

Implications for HCI

We presented Storycoding as a way to construct artefact’s program. For early phases of prototyping our method brings a structure, which can be used as a minimal scaffold for building increasingly complex computational behaviours. Since Storycoding has visual origin, the behaviours could be expanded e.g. through microsketching [30]. On the other hand, by keeping the overall computational behaviour fixed, the designer could explore the same behaviour by trying out different input-sensors and output-actuators. This aspect can feed directly to identifying the technological opportunities. When applied to the development process, Storycoding supports parallel development in at least two different ways. Firstly, the behaviour can be tested and developed to a certain extent with just a serial monitor, using text-based interaction. Secondly, the inputs and outputs can be developed independently, separate from the rest of the behaviour or situations. State machine makes a clear separation with respect to contextual configurations, and the structured use of a flowchart separates the inputs from being tied to any specific state. Thus, as the inputs, outputs, and the computational behaviour can be separated, development is possible without overlapping influence.

This property implies Storycoding to be useful for the development of parallel systems as well, such as IoT or as part of an AI system. This suggests vast potential for future research, where visually rich storyboards could be connected to the AI/IoT backend. On the other hand, this capability of separation also calls for future work with regards to RtD using functional systems that have massively parallel behaviour-alternatives.

Storycoding ties Computational Thinking to HCI by providing a concept-informed computational base. This “algorithmification” of a storyboard implies directions for future research by bringing the experiential methods of design closer to computation. Thus, there is a research direction to form a connection to earlier phases of design exploration and developing a theory of computational thinking in human-computer interaction. Finally, we believe that bridging the experientially vague and detail-rich world of human activities to the computational analysis using activity recognition [5] or deep learning [2], and then utilising the resulting extracted features for designing extremely complex artefacts. We also see research possibilities in the opposite, i.e. by providing a rudimentary scaffold using Storycoding, and letting the AI create variations to the behaviour-scaffold and to the state machine, the flowchart and the program. This would promote development of trans-disciplinary design methods, between AI and HCI communities. Finally, Storycoding could also be beneficial for the humanities-led AI development, where the human is at the now-understandable front.

CONCLUSIONS

In this paper we presented Storycoding, a method for developing programs for RtD artefacts based on a storyboard, a state machine, and a flowchart. Storycoding creates a path that is easy to follow, bridging over the representation gap between the human-relatable world and the program in the artefact. This program can be easily created, modified and reflected upon regardless of the technical expertise, as the abstractions created using Storycoding are linked together. Thus, the program becomes directly observable and utilisable, enriching HCI-RtD as a relatable entity. Finally, we hope our research offers further means for the interdisciplinary collaboration in HCI community and beyond, bringing in the focus to human in the computational behaviour.

ACKNOWLEDGEMENT

Funding support from the Danish It-vest. We thank all our students, and specifically Kausika Simnaraj, Ida Enggaard Petersen, Henrietta Farkas, Ondrej Mastalir, Jakub Tlach, Maria Rochler and Andreas Blumenkranz for their contributions.
REFERENCES


