







# NewYouthHack: Using Design Thinking to Reimagine Settlement Services for New Canadians

Christopher Schankula<sup>1</sup> , Emily Ham<sup>1</sup>, Jessica Schultz<sup>1</sup>, Yumna Irfan<sup>1</sup> ,  
Nhan Thai<sup>1</sup>, Lucas Dutton<sup>1</sup>, Padma Pasupathi<sup>1</sup>, Chinmay Sheth<sup>1</sup>,  
Taranum Khan<sup>1,2</sup>, Salima Tejani<sup>2</sup>, Dima Amad<sup>2</sup>, Robert Fleisig<sup>1</sup>,  
and Christopher Kumar Anand<sup>1</sup>  

<sup>1</sup> McMaster University, Hamilton, Canada  
{schankuc, irfany1, anandc}@mcmaster.ca

<sup>2</sup> Brampton Multicultural Centre, Brampton, Canada  
Salima.Tejani@bmccentre.org,  
<http://outreach.mcmaster.ca/>  
<http://bmccentre.org/>

**Abstract.** In 2018–2019 we applied Design Thinking (DT) to reimagine settlement services for refugee and immigrant youth in Canada. DT continues to gain followers as a practical approach to incorporating human factors into the design process. One insight motivating DT is that design is a series of experiments in which we learn about our users. Iterative prototyping and user feedback are paramount. But we also wanted to expose them to career pathways related to software design and development. In this paper we report on (1) the NewYouthHack process, (2) the resulting app and central role played by social interactions, and (3) the framework we developed to support this work.

We launched with a two-day designathon with 12 identified problems and proposed solutions. Social interaction and community supported by software were threaded through almost all of the solutions. This presented two new challenges: securing iteratively developed network software for vulnerable users, and meaningfully engaging the youth in necessarily complex software. Previously we had developed an outreach curriculum with tool support based on a library in Elm for stand-alone graphical web apps. We taught interaction using state diagrams. In Petri App Land (PAL), we generalized this, with tokens representing users visiting places within the app. Transitions now capture user interactions. To facilitate significant changes from iteration to iteration, much of the code is (re)generated based on a PAL spec.

**Keywords:** Design Thinking · Petri net · Collaborative platform · Mentorship · Social network · Immigration · Refugees

---

Supported by Immigration, Refugees and Citizenship Canada.

© Springer Nature Switzerland AG 2020  
S. S. Rautaray et al. (Eds.): I4CS 2020, CCIS 1139, pp. 41–62, 2020.  
[https://doi.org/10.1007/978-3-030-37484-6\\_3](https://doi.org/10.1007/978-3-030-37484-6_3)

# 1 Introduction

The goal of the paper is twofold: (1) to describe a new model-driven web framework and (2) to describe its use in NewYouthHack, a project to reimagine youth settlement services in Canada. In Sect. 1 we provide context for this project in a larger body of similar research. In Sect. 2 we provide essential background information on several relevant topics, including Design Thinking, our outreach program, our partner in this project, the Elm language, and Petri nets. In Sect. 3, we describe the design of this project and its progression throughout. In the Sects. 4 and 5, we introduce our new platform for multi-user interaction, with an example application in Sect. 6. Finally, we conclude with results in Sect. 7 and ongoing and future work in Sect. 8.

## 1.1 Related Work

The advantages of Human-Centred Design and Design Thinking in designing networked services has been established. Natvig et al. used observations, interviews, innovation games and paper prototyping to develop enhanced support for collaboration in the transport sector [11]. Like our case, they needed to understand the problem from the points of view of multiple stakeholders. Design Thinking has also been successfully taught to children, including by Stanford researchers who enabled the children to redesign systems at their school [2].

Tech4SocialChange is a platform to connect universities, and especially undergraduates, with social problems [16]. The developers identified undergraduates' need for relevant real-world problems matching numerous unmet needs in the community. This differentiates their platform from HeroX and OpenIDEO which also match developers and community needs, but are not specifically geared to students. SOCRATIC is both an online platform and a step-by-step process for identifying problems, forming teams to solve them, and iterating prototypes [19]. It includes analytics to measure typical Design Thinking processes, like number of ideas generated, and it has been designed based on an analysis of several previous initiatives, with the explicit goal to be able to produce scalable solutions to large scale challenges such as migration [5]. The Experts in Teamwork project addresses the same problems, but their focus is on equipping potential social innovators with the skills they need [14]. This Stanford study concluded that Design Thinking “fosters the ability to imagine without boundaries and constraints”, which is instrumental in developing children’s creative confidence and that “design thinking may help students become empowered agents in their own learning who possess both the tools and the confidence to change the world” [2]. Since the youth involved were mostly high school students, we get this secondary benefit that they can apply DT to their own learning. This mirrors the emphasis that we placed on Design Thinking training for all of the participants, not just the McMaster University students involved in the project.

What distinguishes our approach to NewYouthHack, is the attempt to build a technology platform which supports iterative development and the integration of novice programmers. It is too early to say that this platform will lead to

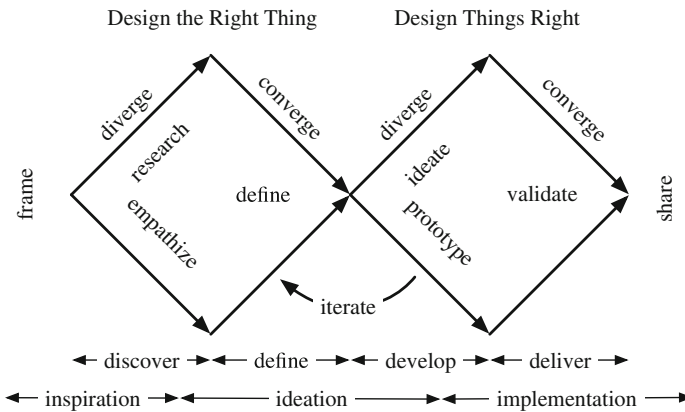
more sustainable solutions, but we are fairly certain that software developers will remain in high demand in the next decades, so these features are important.

One valuable service which other groups have been able to develop, but we have not considered in this project are services to support on-line design communities [1, 7].

## 2 Background

### 2.1 Design Thinking

Design Thinking (DT) is a human-centred methodology that focuses on the end-user and iterates rapidly through conceptual prototyping to produce innovative and creative solutions to complex problems [15]. The DT process is designed to avoid the creation of technically perfect but unwanted or incomprehensible products by focusing on the end-user while creating the product, while considering three dimensions: technical feasibility, economic viability, and desirability to the user [15]. Many people contributed to the development of DT, starting with Cross’s study of designers in several fields and identification with Simon’s *Sciences of the Artificial*<sup>1</sup>, and Norman’s definition of human-centred design.



**Fig. 1.** This modified Double Diamond shows the key processes in Design Thinking, blending elements of the *British Design Council* and Stanford’s *d.school*.

DT assumes that the end-user is complex and that an understanding of their needs requires experiment and inquiry. Working with end-users is not a validation process. It is a discovery process. Hypotheses are not formulated as precisely, and are not about natural phenomena, but about the user’s needs and experience.

There are multiple models and ways of thinking about DT. One of them includes three phases: inspiration, ideation, and implementation. Inspiration

<sup>1</sup> See [6] for a discussion focused on software.

is where the designer empathizes with the end-user, understands their hopes and desires, and uses this to understand the depth of the challenge. Ideation involves making sense of the research, generating ideas, identifying opportunities for design then testing and refining the solutions. Implementation is bringing the solution to life, as well as figuring out how to market it while maximizing its impact on the world [8].

Another view into DT is the double diamond process model developed at the British Design Council in 2005, see Fig. 1. There are divergent thinking stages followed by convergent stages where ideas are narrowed down towards the best one [18]. When designing, some people ignore the left side of the diamond, which leads them to focus on solving the wrong problem. This is why, in DT, *discovering* the problem through empathizing and research, as well as *defining* the right problem, are integral to the process. The *develop* stage involves developing prototype, testing, and iterating. Finally, the *deliver* stage is when the product is finalized, produced, and launched [18].

## 2.2 Software: Tool for Change Program

The McMaster University Outreach Program “*Software: Tool For Change*” has been operating for the past decade. It consists mainly of volunteer undergraduate and graduate students who develop lesson plans and deliver free computer science workshops to schools, public libraries, and community centres in the Hamilton, Ontario, Canada area [12]. We have taught over 15,000 students.

To support these workshops, we have developed tools, including:

1. An open-source Elm graphics library, *GraphicSVG* [17].
2. An online mentorship and Elm compilation system incorporating massive collaborative programming tasks, including the Wordathon<sup>2</sup> and comic book storytelling<sup>3</sup>.
3. A curriculum for introducing graphics programming designed to prepare children for algebra [4].
4. A type- and syntax-error-free projectional iPad Elm editor, *ElmJr* [13].
5. Educational iPad apps, *Image2Bits* which supports binary image encoding, sharing and decoding; *TouchMRI* which mixes instruction about spin physics with an interactive  $k$ -space game; and *MacVenture* a tap-and-type editor for text-and-picture adventure games.

## 2.3 Brampton Multicultural Community Centre

BMC is a nonprofit organization with a mandate to

1. To enhance the capacity of newcomers to participate more effectively in our communities.

<sup>2</sup> <http://outreach.mcmaster.ca/#wordathon2019>.

<sup>3</sup> <http://outreach.mcmaster.ca/#comics2019>.

2. To partner with other service providers and organizations to strengthen the response capacity of the settlement sector.
3. To work collectively with other community actors to facilitate better use of newcomers' knowledge and talents in Canadian workplaces.

## 2.4 Elm Language

We use Elm (<https://elm-lang.org/>), a language designed for the development of frontend web applications [3], to teach beginners, so it was natural to use it for this project. Its syntax, based on the ML family of languages, is intentionally simple. For example, it has no support for user-defined type classes. In addition to strictly enforcing types, the Elm compiler also forces programmers to follow best practices, such as disallowing incomplete case coverage in case expressions. Elm apps use a model-view-update paradigm that keeps pure code separate from code with side effects without the need for monads as in Haskell. Elm code compiles down to JavaScript which provides many practical advantages for deployment and visualization.

While it may seem like this type of language should be reserved to expert users, many of these features useful to experts (strict types, pure functions) are very useful for beginners. In addition to practical implications of compiling to JavaScript, Elm's combination of simple syntax, strict typing, and purity which matches students' pre-existing intuition about math prove to be an asset to our Outreach program. These features allow the development of tools and curricula which would not otherwise be easy or possible in an imperative language with side effects such as Python.

## 2.5 Elm Architecture

All Elm programs follow a common architecture, “The Elm Architecture”, with different variations enabling or restricting certain features as they are needed. These built-in “app” types interact with Elm's JavaScript-based runtime system, enabling pure code to interact with the outside world in a predictable manner, without runtime errors.

Elm's overall architecture consists of three main components: the *model*, the *view*, and the *update*.

**Model.** The *model* of the program is a type that encodes all the possible states the program can be in. In this way, it models the problem domain of the application. The type can be as simple as an alias for a basic type, such as `String` or `Int`, or as complex as needed. Since Elm is used to create web apps, it is recommended that the program be subdivided into top-level states or “Pages” using a union type.

For example, one could encode a very simple program having three states: `MainMenu`, `About` and `Contact`, using an algebraic data type. Additionally, we show the addition of a string to the `MainMenu` screen to display the user's name on screen, for example:

```

type Model = MainMenu String
           | About
           | Contact

```

The `view` portion of the Elm Architecture is a pure function which renders the current state of the program in the browser by returning a representation of html capable of sending messages of type `Msg` (see the following *Update* subsection for a discussion on messages): `view : Model -> Html Msg` Elm’s runtime then uses a virtual DOM diffing strategy to efficiently display the elements described by the `view` function in the browser window.

The `update` function transforms the model state according to the message received: `update : Msg -> Model -> Model`

In this example, the `Msg` type is a “message” type that describes possible actions or events in this app. In our example from above we could have the following `Msg` type:

```

type Msg = GoToMainMenu
         | GoToAbout
         | GoToContact
         | ChangeName String

```

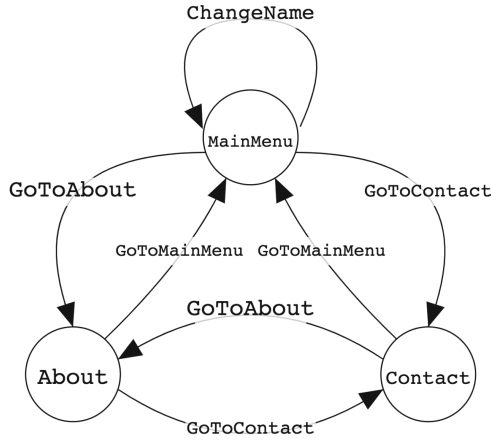
Together, the `Model` and `Msg` types can be visualized as a state diagram, encoding the states as circles and the transitions as arcs from one state to another state. Figure 2 shows how one might encode this example visually. Note that the type system does not enforce that the user adheres to their design. For example, the user’s `update` code must account for the case where the `ChangeName` message is received when the user is in the `About` state, even though that case may be impossible based on the `view` function.

**Commands and Subscriptions.** In more advanced versions of Elm apps, the update function returns a tuple (`Model, Cmd Msg`). Commands (represented as the `Cmd` type in Elm) are a description of an asynchronous action for the Elm runtime to evaluate, passing the result back as a message to the user’s `update` function if the command succeeds. This allows the app to perform impure actions, such as sending a message over the Internet or generating a random number.

Additionally, a *subscription* is a passive listener which waits for events and sends a message when the event occurs. For example, an Elm app may subscribe to a timer to receive a message at a set interval, receive a message when the user enters the app from another tab, or when the size of the app’s window changes.

## 2.6 Petri Nets

A Petri net is a particular kind of directed bi-partite graph. There are two kinds of nodes: *transitions* and *places*. Places are *marked* by *tokens* which are consumed from input places and generated into output places upon the *firing* of a transition (which may occur non-deterministically upon fulfilling certain conditions).



**Fig. 2.** An Elm app can be encoded as a state diagram, with circles being the top-level states in the app and the transitions being the arcs between them, named by the message sent to initiate the transition. When describing an Elm app this way, the programmer has to take care to follow the state diagram in the logic of the program as it is not enforceable at the type level.

**Formal Definition.** A Petri net is a 5-tuple,

$$PN = (P, T, F, W, M_0)$$

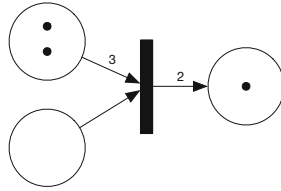
where [10]

$P = \{p_0, p_1, \dots, p_m\}$	is a finite set of places
$T = \{t_0, t_1, \dots, t_n\}$	is a finite set of transitions
$F \subseteq (P \times T) \cup (T \times P)$	is a set of arcs (flow relation)
$W : F \rightarrow \{1, 2, 3, \dots\}$	is a weight function
$M_0 : P \rightarrow \{0, 1, 2, 3, \dots\}$	is the initial marking
$P \cap T = \emptyset$	and
$P \cup T \neq \emptyset.$	

Petri nets are conventionally represented as graphs with circles for places, rectangles for transitions and dots for tokens. Figure 3 shows a small Petri net example.

The behaviour of a Petri net is determined by the *firing* of transitions according to the following rules

1. A transition  $t$  is enabled if each of its input places  $p$  is marked with at least  $W(p, t)$  tokens (where  $W$  is the weighting function of the arc)
2. An enabled transition may or may not fire (nondeterminism)
3. If an enabled transition does fire,  $W(p, t)$  tokens are removed from input place  $p$  and  $W(t, q)$  tokens are added to each output place  $q$ .



**Fig. 3.** A Petri net, showing a single transition, represented by the rectangle, two input places, and one output place. The arcs are weighted, meaning the transition requires as one or more tokens as input and produces one or more tokens as output. Tokens are shown as dots within places.

**High Level Petri Nets.** In order to describe more complex systems in a manageable way, extensions to classical Petri nets such as *Predicate Transition Nets* that utilize predicates in weighting functions and token-colourizing have been developed [9]. “Colouring” of tokens allows the model to distinguish amongst different tokens and the use of predicate expressions in weighting functions allows transitions to fire in several ways based on those colours.

### 3 Methods

BMC advertised the hackathon through their contacts and nine supporting community organizations, and held interviews and focus groups with interested youth and their parents. Thirty new youth, including new immigrants and refugees, aged 15 to 20 and speaking 10 languages, were selected. Graduate students or graduates in design were tasked with preparing and helping to deliver design-thinking training and the hackathon. Twelve undergraduate mentors were recruited and underwent a one-day Design Thinking together with staff from BMC and community partners.

Through multiple coding sessions, students were introduced to coding and learned to write code in Elm. To further develop their coding and collaboration skills, students worked in groups to code an animated comic book depicting their journeys to Canada. At one of these events, the Peel Regional Diversity Roundtable provided diversity and inclusivity training, for mentors and new youth alike.

NewYouthHack, a two-day “hackathon” was held at McMaster, over a weekend, including an overnight stay. Participants were challenged to use DT to improve the experience of settling into Canada. Twelve teams consisting of two to three youth, one BMC mentor, and one undergraduate mentor were guided through an abbreviated DT training and then asked to consider four themes: access to higher education, access to the labour market, access to information and services, and community connections. They were not directed to look for software solutions to the identified problems, but all of the undergraduates had post-secondary programming experience. A panel of judges was assembled in



order to evaluate the effectiveness, business value, and technical feasibility of the designs that each team would develop.

The hackathon activities were modelled after the steps in the DT process. The 12 teams, in an effort to understand the problem from a users point of view, conducted user research by interviewing other participants, with coaching on interviewing provided by mentors. The teams then engaged in the ideation process, by defining their problem space, *how might we?* questions, user persona, and user pain points. At this point, the teams were encouraged to iterate on their ideas, by conducting more user interviews which either corroborated their ideation process results, or informed them that they had to pivot towards a better solution for the user. A practice presentation ended the first day, and a pitch competition the second.

At coding workshops subsequent to the hackathon, the new youth used their coding skills to code illustrations and animations that were incorporated into the app. For instance, students developed new features for an avatar creator, such as glasses and jewelry, and helped prototype the resume format, see Fig. 4. These coding sessions gave the students a means to directly apply their feedback on the app by coding their own contributions.



```

1 | while appears at the top of the browser screen (or on the tab)
2 | title | String
3 | title = "The Doolittle"
4 |
5 | --experience | String -> String -> String -> List String ->
6 | experience | jobTitle location date stuff -
7 | { GridRow ()
8 |   [ GridCol (col.em) --increase number after em to make this or
9 |     [ HTML | style "font-weight: bold" | HTML.text.jobTitle
10 |   ]
11 |   , GridCol (col.em)
12 |     [ HTML.text.location
13 |   ]
14 |   , GridCol (col.textalign Text.alignright)
15 |     [ HTML.text.date
16 |   ]
17 | ]
18 | }
19 | ++
20 | list.map (l.oneStuff -> GridRow [])
21 | { GridCol (col.offsetend) --increase number after offsetend to
22 |   [ HTML.text.oneStuff
23 | ]
24 | }
25 | stuff
26 | }
27 |
28 | blankLine = GridRow [] --this is an empty row (to add blank space)
29 | [ GridCol ()
30 | ]

```

**Fig. 4.** Avatar showing a customization developed with the new youth (*left*) and the on-line programming interface, showing the resume template used in teaching, and later included in the app (*right*).

Focus groups were a way to reach out to newcomer youth and gain their feedback on the current status of the app. Including focus groups in the development process allowed us to take their input into consideration during the development process and adapt the app accordingly. By watching students use the app and holding discussions, we were able to gather information on the usability of features and the user experience. The first focus group consisted of newcomer youth in high school, some of whom had attended NewYouthHack. Students brought up the app on their cellular devices and we displayed the app on a projector. We received feedback on the appearance and usability of the app, and discussed which features they found most helpful. Focus groups afterwards were conducted in a similar manner, with students occasionally using laptops rather than cellular

devices. Focus groups were a great way to involve students in the development of the app and ensure the app met their needs and requirements.

## 4 Platform for User-Driven Innovation

### 4.1 Introducing Interaction with State Diagrams

In *Software: Tool For Change*, we have found teaching interaction using state diagrams to be an effective way for students to understand and design their programs. The simple graphical representation of the state diagram affords much understanding about the overall design of the application at a glance (see Fig. 2). We first teach them about states and transitions and then how to map both states and transitions to algebraic data types. This recipe produces a one-to-one mapping of states and transitions to constructors, which we have found is easy for students to understand once they have seen an example. One downside to this approach is that all possible transitions have to be handled in all states; “impossible” cases (according to the diagram) must be handled by making no change to the current model.

### 4.2 Petri App Land

It was our hope to involve the youth in as many phases of product development as possible. Initially, we were hoping that one or more stand-alone web applications would emerge from the Design Thinking phase, because we had a recipe for teaching children how to build such apps in Elm. But the problems surfaced by NewYouthHack clearly called for mentorship, sharing and commenting on resumes, etc., definitely requiring significant client-server interaction. Thus, we expanded upon our state diagrams to create Petri App Land (PAL), a client-server framework. While students did not contribute to the server-side code, we held workshops for students to contribute to the client frontend.

## 5 Petri App Land Runtime System

PAL contains code on both the client and the server acting as the “runtime system” for all PAL apps. The framework is available as open-source<sup>4</sup>.

**Client-Side.** On the client side, the PAL framework generates code which handles connecting to the server, encoding outgoing messages and decoding incoming messages. It calls the user’s `init`, `view`, and `update` functions from generated case expressions, handling the impossible default cases so they do not complicate the user’s code. A WebSocket library is used to create a two-way communication channel with the server. Messages in a WebSocket connection always arrive in order, or else the connection is closed and must be restarted.

<sup>4</sup> <http://github.com/cschank/petri-app-land>.

**Server-Side.** On the server-side, PAL uses a Software Transactional Memory FIFO queue which is read in a state loop, processing messages and updating state atomically, similar to the Elm Architecture. Haskell threads are used to offload the tasks of encoding and decoding values and handling WebSocket connections with clients. In the future, different places on the server could be split into multiple threads to scale to much bigger services. In addition, we support an asynchronous command architecture accessed in a very similar way to Elm’s, whose events run in different threads, so complicated calculations can be off-loaded in this way, if necessary.

## 6 Example Application

Figure 5 shows an example PAL. For NewYouthHack, the PAL network was encoded using Haskell data structures, which also specified the associated message types, and type definitions. Subsequently, we created a graphical specification tool, PALDraw. This example was made using that tool, but PALDraw and data modelling in PAL is beyond the scope of this paper.

### 6.1 Client-Side Modules

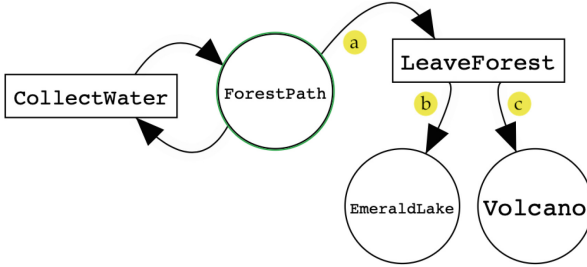
Users must fill in three types of modules on the client side after generating code from their specification, in order to complete their app. These consist of the `Init` module, the `View` modules and the `Update` module.

The `Init` module provides the initial state of the starting place on the client. This is where the user defines the initial data that is associated with the place. Below is an example `ForestPath` place from the example app in Fig. 5 (with imports removed for brevity):

```
init : ForestPath      -- the initial state of the starting place
init = ForestPath 0    -- player’s water amount
                (Backpack [] 0) -- player’s backpack
                0      -- player’s health
```

Each place on the app has an associated view module, with functions corresponding to the Elm Architecture’s (see above) `view` function. Note that we provide separate functions for each place, each with a unique model parameter type. This input type is a generated single-constructor version of the constructor describing the client-side state. This enforces at the type level that the user receives the data corresponding to the place in their view. A specialized `Msg` type is generated, limiting (again at the type level) which messages can be sent from which view functions. The programmer can also make use of Elm’s subscriptions and change the title bar of the browser inside this module, by modifying the default implementations generated for the functions:

```
subs  : ForestPath -> Sub Msg
view  : ForestPath -> Html Msg
title : ForestPath -> String
```



**Fig. 5.** A small PAL app consisting of three places: `ForestPath`, `EmeraldLake` and `Volcano`, and two transitions: `LeaveForest` and `CollectWater`. `ForestPath`'s green outline indicates that it is the place clients are placed in upon connecting to the server. The `LeaveForest` transition has one incoming place, `ForestPath` (labeled as `a`)), and two outgoing places, `EmeraldLake` and `Volcano`. Clients in the `ForestPath` place can trigger the `LeaveForest` transition, causing the server to respond with either `LeaveForest4EmeraldLake` or `LeaveForest4Volcano` (labeled as `b` and `c`, respectively), according to rules related to the amount of water a player is carrying. The `CollectWater` transition keeps the user in the `ForestPath` and can be used to modify the user's current amount of carried water. (Color figure online)

**Update Module.** The Update module contains functions for each of the possible transitions the client can make when receiving a message from the server. For example, in our Fig. 5 example, the labels `b` and `c` represent the messages `LeaveForest4EmeraldLake` and `LeaveForest4Volcano`, respectively. As such, the following function stubs are generated:

```
updateForestPathLeaveForest4EmeraldLakeEmeraldLake : Environment ->
  LeaveForest4EmeraldLake -> ForestPath ->
  (EmeraldLake, Cmd EmeraldLakeT.Msg)
```

```
updateForestPathLeaveForest4VolcanoVolcano : Environment ->
  LeaveForest4Volcano -> ForestPath -> (Volcano, Cmd VolcanoT.Msg)
```

Note that, like in the view function, single-constructor types are used to once again enforce adherence to the app's model and to eliminate the need for (visible) case expressions with default cases for impossible cases.

## 6.2 Server-Side Modules

The Haskell server is structured similarly to the Elm side, but without view modules.

The `Init` module on the server initializes the place state for each place, as all places must be stored on the server at once. In our simple example, the `ForestPath` has a certain amount of water, but the other places, which are not shown, have no associated data:

```
initForestPath :: ForestPath
initForestPath = ForestPath 1000 {-initial water in forest-}
```

The `Update` module on the server contains a function for each transition in the PAL diagram. The function is called when a client initiates a transition by sending a message of the same name. The function is similar to the client update functions, but must consider multiple clients and states for all connected places. For example, the `LeaveForest` transition has an associated function on the server whose type signature is:

```
updateLeaveForest :: Environment -> ClientID ->
  LeaveForest -> ForestPath -> EmeraldLake -> Volcano ->
  [ForestPathPlayer] ->
  ( ForestPath, EmeraldLake, Volcano
  , (ClientID, ForestPathPlayer) -> LeaveForestfromForestPath
  )
```

The programmer must return a new state for all involved places, as well as a function that is mapped over all of the players (users) in the `ForestPath` place. This function returns a specialized type (`LeaveForestfromForestPath` in this case — see below) which determines which place to send the user to (or whether to keep them in the same place). In this way, users cannot be placed in multiple places at once or be forgotten.

The `LeaveForestfromForestPath` union type encodes only the possible transitions that are specified in the user’s PAL model:

```
data LeaveForestfromForestPath =
  LeaveForest_ForestPathtoVolcano
  -- state of player when they go to volcano
  VolcanoPlayer
  -- message sent back to client
  LeaveForest4Volcano
| LeaveForest4EmeraldLake
  ... two more cases ...
```

In the `updateLeaveForest` example, the user who issued the message to initiate the transition (determined by their unique `ClientID`) can go to the `Volcano` place if they have enough water in their player state. Otherwise, they stay inside the `ForestPath` place.

### 6.3 Helper Functions

Places and transitions are generated as constructors with fields for data, with simple data acting as the fields of a record-like structure. Algebraic data types were chosen to have consistency of the data across Elm and Haskell, which share the same basic syntax and semantics. For convenience, we produce `get*`, `update*`, and `alter*` functions for place states and one-constructor data types.

These helper functions are kept up to date as the specification of the app evolves, which replaces the need to pattern-match to extract information inside of the app's `view` and `update` functions. As such, using these helper functions greatly improves the maintainability of the app as new pieces of data are added. Furthermore, if data is removed or renamed, the helper function will also be removed from, or renamed in, the generated helpers module. The resulting definition error signals to the programmer that changes are required in the app's code in order to meet the new design of the app. Helper functions work the same way on both the client and server, leading to consistent code.

Below, we show the types from the generated templates for selected helper functions to access and update the `backpack` data type contained in the `ForestPath` state (from Fig. 5) generated for use in the client.

```
getBackpack : ForestPath -> Backpack
updateBackpack :
  Backpack -> ForestPath -> ForestPath
```

The argument order makes it easy to compose functions using `(|>)` : `a -> (a -> b) -> b`, Elm's left-to-right pipe, as in this fragment:

```
forestPath -- current client state
  |> updateBackpack newBackpack {- new backpack -}
  |> alterHealth (\h -> h - 1) -- reduce health
```

This produces readable code that self-documents the changes made to models or pieces of data. In particular, the `alter*` functions provide the user with flexibility to compose other semantically-relevant, high-level functions, e.g. one could imagine creating some helper functions for backpacks:

```
addSandwich      : Sandwich -> Backpack -> Backpack
eatNSandwiches  : Int -> Sandwich -> Backpack -> Backpack
```

which could then be used in conjunction with the `alterBackpack` function, by partially evaluating all but the final `Backpack` argument.

## 7 Results

### 7.1 The Hackathon

The 12 teams each settled on a problem, and made an initial presentation at the end of the first day. Omitting their well-thought-out detail, the ideas were:

1. Technology Learning for Academic Success
2. Information on courses - career pathways
3. Connections, mentorship at institutions
4. An app to make and submit resumes
5. Employer interaction, interviews, networking and hiring

6. Social interaction, services and events
7. Information on transit, maps and attractions
8. Vlog to help with school system and settlement
9. Multi-lingual resource to find friends and resources
10. Find volunteer opportunities activities and interests
11. Community specific multi-lingual interaction
12. Build connections and get information on services

Although NewYouthHack was more of a designathon than a hackathon, we wanted to capture the spirit of a hackathon by producing some prototypes during the weekend. The way we did this was to listen for self-contained app ideas within the presentations of the twelve ideas made at the end of the day on Saturday. A team of three developers then picked what they judged the greatest value/effort ideas, and implemented two of them: a simple chat client, to capture the idea of on-line mentoring, and an interactive high-school math pathway explorer, see Fig. 6.

Of the 12, the top 3 selected by the judges were, the video log, employer interaction app, and post-secondary education navigation.

After the hackathon, a steering group evaluated the twelve ideas with respect to implementation effort, barriers to adoption, privacy and safety issues, and the existence of competing solutions. It was decided that the video log posed safety issues which could be mitigated by restricting access to individual schools, and this idea was communicated to the Peel School Board. Information on courses is already available from [myBlueprint.ca](http://myBlueprint.ca), but a focus group determined that new youth knew this, so a presentation to [myBlueprint.ca](http://myBlueprint.ca) was made, including the MathPathways prototype. Resume builders exist, but new youth needed trusted feedback, so it made sense to duplicate such functionality within a mentoring framework.

## 7.2 Independent Evaluation

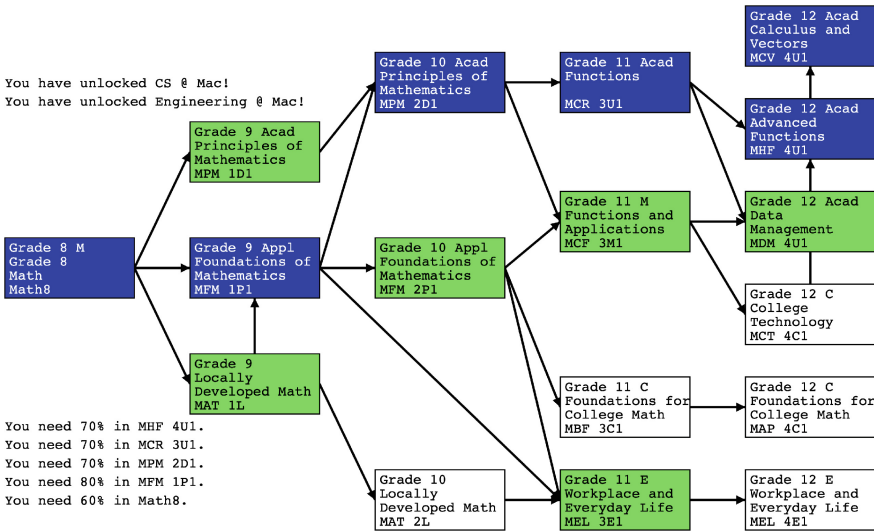
The NewYouthHack project was evaluated using a developmental evaluation approach. It provided timely feedback and data to inform decisions on an ongoing basis, focusing on meeting stated goals, innovating effective solutions and identifying best practices. The newcomer youth and the participating stakeholders were engaged by the independent evaluator through in-person surveys, focus group meetings, key informant interviews, direct observation and informal conversations, to evaluate interaction with stakeholders through the entire process.

We quote the independent evaluator's findings that the NewYouthHack Project achieved its objectives in both developing ideas and empowering youth:

1. Successfully developed an app that aims to assist newcomer youth in multiple areas such as careers, education pathways, volunteer engagement and access to programs and services.
2. The successful launching and implementing of the app now depends on: (1) securing additional funding to further engage in consultation with relevant stakeholders, (2) piloting the app to fine tune, (3) implementing locally in Peel and (4) scale it up by implementing the app in Ontario and Canada.

3. Collaboration and partnership with multiple stakeholders in the community contributed to the successful implementation of the NewYouthHack project that brought sponsorships, stakeholder participation and feedback.
4. The initiative helped engage newcomer youth from diverse background that spoke 10 different languages and included disabled and LGBTQ persons.
5. An environment of trust and safety was created by BMC to address the concerns of parents of the newcomer youth ensuring their full participation.
6. The project provided a platform for newcomer youth to (1) develop a sense of belonging to their new home in Canada, (2) know that they are not alone; (3) open up to share their opinion freely on issues facing newcomer youth at and subsequent to the hackathon; (4) learn, identify, analyse and find solutions to the problems in their day to day life; and (5) expand their social networks.

Click on available courses and try to unlock programs!



**Fig. 6.** Interactive Math Pathways explorer. New Youth, who were not used to having as much flexibility as allowed in the Ontario school system, found it difficult to make course selections and understand the implications in terms of post-secondary education. In this prototype tool, we allowed them to pick courses in the math stream one at a time, with new courses opening up once they have taken the prerequisites. At each stage, the selected courses are highlighted in blue, and the courses available to them are highlighted in green. As new courses are chosen, any entrance conditions (i.e. attaining a specific grade in a previous course) are listed, and undergraduate programs whose entrance requirements are met are also shown. This is one pathway for meeting the math requirements for Computer Science at McMaster University. (Color figure online)



## Square One

Welcome to Square One (the main menu). Click below to take the GO bus to your desired destination!

<p><b>Universities and Colleges</b></p> <p>Connect with mentors from universities and colleges in Ontario</p> <p><a href="#">To Universities and Colleges</a></p>	<p><b>Universities and Colleges</b></p> <p>Connect with mentors from universities and colleges in Ontario</p> <p><a href="#">To Universities and Colleges</a></p>
<p><b>Profile Room</b></p> <p>Make yourself pretty! Edit your Avatar, name, email address and more here.</p> <p><a href="#">To Profile Room</a></p>	

## Square One

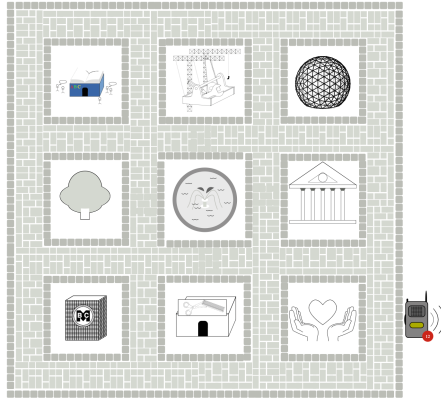
Welcome to Square One (the main menu). Click below to take the GO bus to your desired destination!

<p><b>Universities and Colleges</b></p> <p>Connect with mentors from universities and colleges in Ontario</p> <p><a href="#">To Universities and Colleges</a></p>	<p><b>Universities and Colleges</b></p> <p>Connect with mentors from universities and colleges in Ontario</p> <p><a href="#">To Universities and Colleges</a></p>
<p><b>Profile Room</b></p> <p>Make yourself pretty! Edit your Avatar, name, email address and more here.</p> <p><a href="#">To Profile Room</a></p>	

**Fig. 7.** Visualization of the landing page as it evolved based on focus group feedback. The initial landing page (*top*) was quite simple, and in response to feedback, we added a map (*bottom*). Note that Square One is a shopping area and central landmark in Mississauga, the largest municipality from which participants were drawn. *Continued in Fig. 8.*

## Square One

Welcome to Square One (the main menu). Click below to take the GO bus to your desired destination!



**Fig. 8.** *Continued from Fig. 7.* The map that was seen as confusing, so we tried to make it look like city blocks. *Continued in Fig. 9.*

We also quote from the evaluator’s surveys of the youth:

- 100% of youth said they have improved their communication skills
- 96% of youth said they have learned work in group settings
- 96% of youth said they have learned to design thinking to develop solutions
- 95% of youth said they have learned about design thinking and its benefits

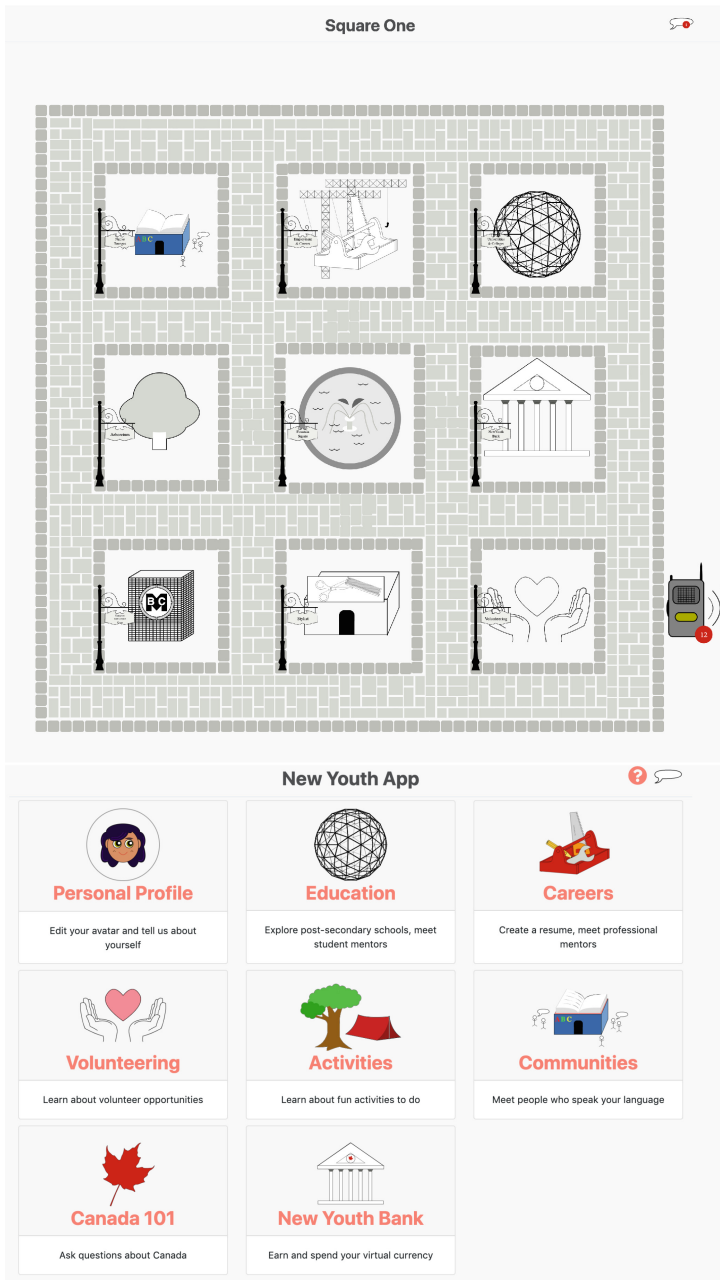
“Thanks to this experience I have noticed how important my role is in Canada. I have the power to drive change and make a positive impact and other youth can do so wherever they are. Using technology as a tool means that we all can change things and make them better for newcomers across Canada, overseas and for the next generations of people coming into our county.”

- 96% of youth said they have increased their self-confidence and self-esteem
- 96% of youth said they have expanded their social network with new friends
- 96% of youth said they have learned to work collaboratively with others

“It means a lot that this time I had the opportunity to design and have a say in creating a solution for barriers I faced as a newcomer youth.”

- 96% of youth said youth and adults have learned from one another
- 92% said they were able to express their opinion on newcomer youth issues
- 92% of youth said their opinions were listened and valued by others
- 88% of youth said the youth and adults worked collaboratively

“STEM has been an interest of mine from an early age but attending Design Thinking, Coding Sessions, and Hackathon made me realize that I might actually look into computer engineering as a career choice.”



**Fig. 9.** *Continued from Fig. 8.* The street map introduced a different type of confusion, causing us to add signposts (*top*), which elicited more concrete feedback and led us to go back to a button interface (*bottom*), but now with colourful icons.

In one of the final focus groups, 75% reported they would recommend the app to other newcomer youth.

### 7.3 The App

The final application contained over 55,000 lines of code across the client and server, over half of which was code generated by the PAL framework. Much of the hand-written code was on the client side, defining the user interface which consists of bandwidth-sparing programmatically-created graphics, as follows:

	Spec (Haskell)	Client (Elm)	Server (Haskell)	Total
Written	3621	15498	7418	26537
Generated	N/A	14224	14556	28780
Total	3621	29722	21974	55317

Code generation was critical to development due to the rapid evolution of the design required to respond to user feedback. The changes are most visible through the evolution of the landing page, as shown starting in Fig. 7.

The final app consisted of the main menu shown in Fig. 9 (*bottom*), leading to each of the features of the application. The icons are roughly ordered to represent the order in which users are likely to use the app. The *Personal Profile* allows the student or mentor to create an avatar and write about their goals or career. The *Education*, *Careers* and *Volunteering* sections provide mentorship and information about those respective topics, as well as resume creation and sharing, job posting, on-line interviewing, etc. *Activities* describes some common Canadian activities and foods, such as hockey or poutine. *Communities* provides language-based forums for youth to meet others who speak the same language. *Canada 101* is a moderated FAQ page for youth to ask questions about their new country and community. *New Youth Bank* provides a virtual currency that can be earned by completing tasks in the app, and then customizations can be purchased (many gamification ideas from focus groups await implementation).

## 8 Conclusion and Future Work

When we started this project, we applied for one year of funding because it was a new three-way collaboration and introduced new ideas to all of the participants. At the end of the year, all of our expectations were exceeded, especially in terms of the impact on the participating new youth and the robustness of an app built without requirements. We are seeking funding to continue work on this project.

Building on this experience, we have collectively undertaken follow-on projects: the *#BTCHACK - Re-Imagining Youth Civic Engagement* hackathon,

focusing on Reimagining Youth Civic Engagement, was hosted by BMC in partnership with McMaster’s Faculty of Engineering and Big Brothers Big Sisters of Peel, funded by the Ontario Trillium Foundation (OTF); and *Software: Tool For Change* hosted an *Internship-Style Summer Camp* over four weeks with 10–12 children ranging from ages 9–14, in which the children were given crash courses in Design Thinking and Cognitive Science and asked to imagine and develop a multi-player math game, with the resulting app<sup>5</sup> exceeding our expectations. In preparation for this camp, we created a prototype visual design tool called PALDraw for modelling the PAL, associated algebraic data types and their documentation, and including embedded state diagrams in each place. PALDraw generates the PAL spec as well as Elm modules which also work stand-alone. As a next step, we will extend PALDraw into an Integrated Development Environment (IDE), merging it with our web-based editor [4].

## References

1. Ahmed, F., Fuge, M.: Capturing winning ideas in online design communities. In: Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW 2017, pp. 1675–1687. ACM, New York (2017). <https://doi.org/10.1145/2998181.2998249>
2. Carroll, M., Goldman, S., Britos, L., Koh, J., Royalty, A., Hornstein, M.: Destination, imagination and the fires within: design thinking in a middle school classroom. *Int. J. Art Des. Educ.* **29**(1), 37–53 (2010)
3. Czaplicki, E.: Elm: Concurrent FRP for functional GUIs. Senior thesis, Harvard University (2012)
4. d’Alves, C., et al.: Using elm to introduce algebraic thinking to K-8 students. In: Thompson, S. (ed.) Proceedings Sixth Workshop on Trends in Functional Programming in Education, Canterbury, Kent, UK, 22 June 2017. Electronic Proceedings in Theoretical Computer Science, vol. 270, pp. 18–36. Open Publishing Association (2018). <https://doi.org/10.4204/EPTCS.270.2>
5. Dinant, I., Floch, J., Vilarinho, T., Oliveira, M.: Designing a digital social innovation platform: from case studies to concepts. In: Kompatsiaris, I., et al. (eds.) INSCI 2017. LNCS, vol. 10673, pp. 101–118. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-70284-1\\_9](https://doi.org/10.1007/978-3-319-70284-1_9)
6. Gregor, S.: Building theory in the sciences of the artificial. In: Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology, DESRIST 2009, pp. 4:1–4:10. ACM, New York (2009). <https://doi.org/10.1145/1555619.1555625>
7. Hajiamiri, M., Korkut, F.: Perceived values of web-based collective design platforms from the perspective of industrial designers in reference to Quirky and openIDEO. *ITU AZ* **12**(1), 147–159 (2015)
8. IDEO: The field guide to human-centered design (2015). <http://www.designkit.org/resources/1>
9. Jensen, K.: High-level Petri nets. In: Pagnoni, A., Rozenberg, G. (eds.) Applications and Theory of Petri Nets. *INFORMATIK*, vol. 66, pp. 166–180. Springer, Heidelberg (1983). [https://doi.org/10.1007/978-3-642-69028-0\\_12](https://doi.org/10.1007/978-3-642-69028-0_12)

<sup>5</sup> Escape From Math Island <https://macoutreach.rocks/escapemathisland/>.

10. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
11. Natvig, M.K., Wienhofen, L.W.M.: Collaboration support for transport in the retail supply chain. In: Fahrnberger, G., Eichler, G., Erfurth, C. (eds.) I4CS 2016. CCIS, vol. 648, pp. 192–208. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-49466-1\\_13](https://doi.org/10.1007/978-3-319-49466-1_13)
12. O’Farrell, B., Anand, C.: Code the future!: teach kids to program in elm. In: Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, pp. 357–357. IBM Corp. (2017)
13. Optimal Computational Algorithms Inc.: ElmJr (1.0). iOS App Stores (2018). <https://apps.apple.com/ca/app/elmjr/id1335011478>
14. Pappas, I.O., Mora, S., Jaccheri, L., Mikalef, P.: Empowering social innovators through collaborative and experiential learning. In: 2018 IEEE Global Engineering Education Conference (EDUCON), pp. 1080–1088. IEEE (2018)
15. Plattner, H., Meinel, C., Leifer, L.: Design Thinking: Understand-Improve-Apply. Springer, Heidelberg (2010). <https://doi.org/10.1007/978-3-642-13757-0>
16. Reis, A., et al.: Tech4SocialChange: technology for all. In: Fahrnberger, G., Eichler, G., Erfurth, C. (eds.) I4CS 2016. CCIS, vol. 648, pp. 153–169. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-49466-1\\_11](https://doi.org/10.1007/978-3-319-49466-1_11)
17. Schankula, C.W., Anand, C.K.: GraphicSVG [elm package] (2016–2019). <http://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest>
18. Tschimmel, K.: Design thinking as an effective toolkit for innovation. In: ISPIM Conference Proceedings, p. 1. The International Society for Professional Innovation Management (ISPIM) (2012)
19. Vilarinho, T., et al.: Experimenting a digital collaborative platform for supporting social innovation in multiple settings. In: Hodoň, M., Eichler, G., Erfurth, C., Fahrnberger, G. (eds.) I4CS 2018. CCIS, vol. 863, pp. 142–157. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-93408-2\\_11](https://doi.org/10.1007/978-3-319-93408-2_11)