Refactoring *Local* to *Cloud* Data Types for Mobile Apps

Michael Hilton
Oregon State University, USA
hiltonm@eecs.oregonstate.edu

Arpit Christi
Oregon State University, USA
christia@eecs.oregonstate.edu

Danny Dig
Oregon State University, USA
digd@eecs.oregonstate.edu

Michał Moskal
Microsoft Research, USA
Michal.Moskal@microsoft.com

Sebastian Burckhardt
Microsoft Research, USA
sburckha@microsoft.com

Nikolai Tillmann
Microsoft Research, USA
nikolait@microsoft.com

ABSTRACT

Mobile cloud computing can greatly enrich the capabilities of today's pervasive mobile devices. Storing data on the cloud can enable features such as automatic backup, seamless transition between multiple devices, and multiuser support for existing apps. However, the process of converting local into cloud data types requires high expertise, is difficult, and time-consuming. Refactoring techniques can greatly simplify this process.

In this paper we present a formative study where we analyzed and successfully converted four real-world touch**develop** apps into cloud-enabled apps. Based on these lessons, we designed and implemented, CLOUDIFYER, a tool that automatically refactors *local* data types into *cloud* data types on the touch**develop** platform. Our empirical evaluation on a corpus of 123 mobile apps resulting in 2722 transformations shows (i) that the refactoring is widely *applicable*, (ii) CLOUDIFYER saves human effort, and (iii) CLOUDIFYER is *accurate*.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms Refactoring

Keywords Refactoring, Cloud Computing, Empirical Study

1. INTRODUCTION

The marriage of cloud and mobile computing has transformed our access to data. If users lose their phone, upon activating a new phone, all the data (e.g., contacts, pictures) is there due to the automatic backup of data. A user can start editing a document on their mobile device and resume editing it on their tablet at home, thus seamlessly transitioning between multiple devices operating on the same data. Additionally, the cloud enables rich, multi-user apps. Examples abound from domains such as social networking (e.g., Facebook, Twitter), multiplayer games, and collaborative data collection (e.g., Citizen Science [8]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MOBILESoft'14, June 2-3, 2014, Hyderabad, India.

Copyright © 14 ACM /14/06...$15.00.

To take advantage of the benefits of the cloud, app developers face a high entry barrier. They need expertise on many topics: communication protocols (e.g., web services, REST, SOAP, etc.), data storage (e.g., Amazon S3, Microsoft SkyDrive, etc.), databases, cloud infrastructure (e.g., Amazon EC2, Windows Azure, etc.), programming or scripting languages. Similarly, converting a single into a multiuser app has a high entry barrier: they need to determine the candidate data structures and methods that operate on data structures as well as move them to the cloud. Currently, this process is manual, time consuming, and error prone [16].

In this paper we are lowering the entry barrier to allow even hobbyists and beginner app developers to use the cloud. Thus, we are targeting touch**develop** [32], a programming environment and language developed by Microsoft Research to write apps *on* mobile devices *for* mobile devices. We are employing automated refactoring techniques to convert local data types into cloud data types.

touch**develop** introduced specialized cloud data types [6] that provide an abstraction layer over web service implementation, communication protocols, and storage. In order to make the app responsive, even when the connection to the server is unavailable, cloud data types provide both local copies of the data as well as eventually consistent sharable cloud storage. This paradigm allows programmers to use cloud types in a similar manner to local data structures, but to also enjoy the benefits of the cloud.

In this paper we present the results of our formative study to convert local to cloud-enabled apps. We used four publicly available touch**develop** apps (three productivity tools and one game) and manually converted them into multi-user apps that share data via the cloud. In doing so, we discovered four conversion steps: (i) identify local data structures that need to be moved to the cloud, (ii) for each identified local data structure add a new cloud data structure, (iii) replace local data type API calls with cloud API calls, (iv) initialize the cloud data types. Not all these steps can be automated; some (e.g., identifying data types that need to be moved) require domain knowledge which is best provided by the app developer.

Using the lessons that we learned from the formative study, we designed and implemented a refactoring tool, CLOUDIFYER, to automate the conversion of local `Number Collection` into `Cloud Data Table`, and to transform the API calls. We selected this refactoring because its manual application is challenging for several reasons. In addition to differences in the names of the APIs, there are also differences in the cardinality of the mapping. Sometimes the mapping is 1-to-1 (e.g., `count` is the same for both data types), while other times the mapping is 1-to-many (e.g., `insert at` from `Collection` is transformed into a sequence of 3 operators from `Table`). Sometimes there is no mapping at all, in which case

CLOUDIFYER injects functions to achieve the same computation. For example, `collection->max` needs to be converted into a function that iterates atomically over the elements of the `Table`.

This paper makes the following contributions:

- **Idea:** To the best of our knowledge, we are the first to enable hobbyists and beginner programmers to tap into the power of mobile cloud computing through the use of refactoring techniques.
- **Formative Study:** We have conducted a formative study on four real-world apps to learn what transformations are needed to convert single to multi-user mobile apps.
- **Tool:** We have designed and implemented the analysis and transformation algorithms to refactor *local* data types into *cloud* data types on the touch**develop** platform.
- **Evaluation:** We have evaluated our tool, CLOUDIFYER, on a corpus of 123 mobile apps, resulting in 2722 transformations. The results show (i) that the refactoring is widely applicable: 94% of the candidate local `Collections` were successfully refactored into `Cloud Data Table`. Second, CLOUDIFYER saves human effort: on average it took 3 seconds for each performed refactoring. Third, CLOUDIFYER is accurate: 100% of the applied transformations are correct, and the tool correctly identified 95% of all necessary transformations. The tool and all the data are publicly available at:
`http://cope.eecs.oregonstate.edu/cloudifier`

2. BACKGROUND ON TOUCHDEVELOP AND CLOUD DATA TYPES

touch**develop** is a programming language and integrated development environment (IDE) that allows the development of mobile applications (called *scripts*) on any device, for any device. It provides a development experience that is both ubiquitous and social: ubiquitous because the IDE is optimized for operation on touch-screen devices like phones or tablets, and social because scripts and libraries can be easily shared, forked, commented on, reviewed, and published in the cloud. Both touch**develop** and the touch**develop** scripts are compiled to HTML5/JavaScript, thus they can execute on all major smartphone, tablet, and personal computer brands.

The main purpose of touch**develop** is: (i) to provide an instrument for enticing and educating the next generation of programmers, and (ii) to develop novel programming language features that can simplify and accelerate the development of cloud-connected mobile applications (of which touch**develop** itself is a prime example). An example of the latter are cloud data types [6], which were recently added to touch**develop**.

Cloud types make it easy to share data between instances of the app running on multiple devices, by simply declaring such data to be of a particular *cloud type*. Once declared, the runtime then automatically synchronizes the data across multiple devices that are part of the same *cloud session*. A cloud session may include devices by a single user or by multiple users. For example, a single user may want to synchronize settings, calendars, contact lists, or any other script data between various devices she owns. Examples of multiple-user scenarios include a simple grocery list (to help family members keep track of items to be purchased on the next trip to the store), multi-player games that share game state, or applications that include typical social features like comments, reviews, achievements, high scores, and so on.

There are three categories of cloud data.

Cloud variables have a name and a type, which can be a simple type (number, string, etc.) or a reference to a cloud table row. All cloud variables support `get` and `set` operations. Number variables also support an `add` operation.

Cloud tables are declared to have a name, and any number of named, typed columns. The types available for the columns are the same as for cloud variables. Tables support operations for adding a new row (always appended at the end of the table), deleting a row (deletion is permanent and idempotent), and enumerating over rows (in the order they were appended).

Cloud indexes are made up of entries that contain keys and values. All keys and values have names and types. The index contains exactly one entry for each key or combination of keys, and this entry can be retrieved using the `at` operation. Note that it is *not* possible to add or remove entries from indexes (because there is always exactly one entry per key). Entries for which all values are the default value are considered 'empty'. Indexes support enumeration of all non-empty entries. The number of non-empty entries is always finite, even if the number of entries is infinite (for example, if using a key of type string).

Cloud types are integrated into the programming language and are thus as easy to use as data types used by traditional programming languages, (simple types, objects, and collections) but their interface is more similar to types typically used for persistent storage (such as tables and indexes). The reason for this design choice is that cloud types are designed specifically to support synchronization of replicas in a distributed system with a reliable server and clients that may crash silently. In such a system, garbage collection is impossible as the server never knows if there is some client alive with a reference to a particular piece of data). The memory must thus be managed explicitly, which is facilitated by using globally named variables, tables and indexes. Also, since client devices can read and update local data at all times (whether connected or not), the server must resolve conflicts. To this end, all operations on the cloud types are designed to support automatic conflict resolution, i.e. eventual consistency can be obtained by determining a total order of conflicting updates on the server [6, 30]. Such conflict resolution is easier when the semantics is implicit in the type instead of being encoded in arbitrary heap operations—for example, consider resolving conflicts in a table where two devices add a row, versus a linked list where two devices change a pointer's value somewhere.

As a consequence, programs that use classical heap-based object and collection data types require refactoring to take advantage of the automatic persistence and synchronization provided by cloud types. Roughly, this amounts to making classes correspond to tables, objects to rows, and fields to columns.

2.1 Consistency Model

As multiple clients can concurrently update the same data, even during offline operation, conflicts may need to be resolved after-the-fact, in a manner that guarantees eventual consistency. To give the reader a general idea, we now briefly describe the mechanism used in touch**develop**, which is a simple, specialized form of eventually consistent transactions [5] and guarantees causal eventual consistency.

For each cloud session, touch**develop** stores a log of all updates performed in the cloud. The basic idea is to use what is often called primary replication: Client devices (secondary replicas) stream all locally issued updates to the touch**develop** server (primary replica). The server interleaves all updates in the order received, and stores them in log which reflects the final ordering of the updates. The server also streams the final update log back to the clients, where the updates in the log get applied to the locally stored copy of the data. However, this basic primary replication mechanism is enhanced in several ways:

1. **Local Buffers.** Local updates are made visible immediately, not only after they are echoed (confirmed) by the server. Conceptually, the client maintains a buffer of unconfirmed updates, similar to store buffers in the TSO memory model, and superimposes the effect of these updates over the current confirmed state. The use of such buffers improves the perceived performance and allows seamless offline operation, but breaks strong consistency (which is a necessary consequence of allowing offline operation, as the CAP theorem [4, 14] shows).
2. **Automatic Transactions.** Updates are transmitted not individually, but as a group (called an update transaction). The transaction boundaries are determined automatically based on the event queue: any time there is no event executing (i.e. when the execution is quiesced), the current transaction is ended and a new one is started. The use of automatic transactions helps to avoid typical atomicity errors when updating multiple data items at the same time.
3. **Cloud Types.** The updates in the log are not just simple write operations, but mirror the rich semantics of the cloud types. Cloud types allow programmers to work around limitations of the weak consistency model, because they allow semantically meaningful conflict resolution simply by ordering the updates into a consistent order. For example, the cloud number type provides an `add` method in addition to the usual `get` and `set`. Applications of `add` from multiple clients can be meaningfully merged by the server, whereas `set` has the last-writer-wins semantics.

Because a weak consistency model is used, refactoring for cloud types can introduce consistency errors that were not present in the original program. The responsibility for identifying such problems and correcting them remains with the user.

Example Let's consider a client A performing two updates, `x := x + 2` followed by `x := x + 4`, in an event handler E. Client B is executing `x := x + 3`, where `x` is initially `0`. These are the only possible executions:

- B's update is streamed to server and then to A before E starts executing, in which case `x` will be eventually `9`
- A performs its updates, setting `x` to `6`, and B sets `x` to `3`. Depending in which order the updates reach the server the eventual value of `x` will be either `3` or `6`. Regardless, while E is executing, A will keep seeing `x` as `6` from the point of the second update on (due to local buffers above), and only in the next event it may discover that it is now in fact `3`.

Note that an update from the server cannot arrive between the two updates in E (due to automatic transactions above). Also note that if the updates were `x->add(2); x->add(4)` for A, and `x->add(3)` for B, then the eventual value of `x` would be always `9`, but A could still see `x` as `6` or `9` after its updates.

2.2 TouchDevelop plug-ins

The touch**develop** IDE can be extended with user-provided plug-ins, which are just regular touch**develop** scripts conforming to a specific interface. In general, touch**develop** scripts can query and update abstract syntax trees (ASTs) of scripts installed on the device (after asking the user for permission). A plug-in is invoked by the user when editing a script. The identifier of that script is passed to a specific action in the plug-in, that can then read and write the AST. The AST is represented as a JSON object, which can be manipulated using standard touch**develop** libraries (for purely client-side plug-ins), or shipped over to a cloud service of plug-in author's choice for processing.

Currently, touch**develop** supports plug-in invocation on an entire script, where the plug-in can ask the user to point to a particular AST node to operate on if necessary. In near future, we plan to have plug-ins invoked contextually for a given definition or expression.

When the user taps the plug-in invocation button, touch**develop** shows a list of currently installed scripts marked with `#scriptPlugin` in their description. The user has also an option of searching touch**develop** cloud for more such scripts. We thus reuse the script distribution and rating mechanisms for plug-ins.

After plug-in execution, the user may be shown the diff with the changes performed by the plugin and asked to confirm them.

3. FORMATIVE STUDY

We selected four apps from the touch**develop** script bazaar for our formative study. We chose representative apps from various domains that cover different features and use cases of touch**develop**. We also looked for apps where we could envision multi-user functionality. We then converted these apps from single-user to multi-user apps. In this section we describe the script's behavior, the process to convert them, as well as the changes we needed to make.

The first app we selected was MILEAGE TRACKER, which is publicly available (the script id can be found at [1]). Mileage Tracker is an app that records and calculates fuel usage in Miles per Gallon (MPG) and displays how it changes over time. We began by adding some functionality to the original app. We then converted Milage Tracker into a multi-user app, such that multiple family members using the same family car can collect the fuel usage even across multiple devices.

Business Manager+ [1] is an app to track business contacts. We converted it into a multi-user app so that business colleagues can share their contact list with each other.

CliffHangers [1] is a clone of the popular game "hangman." The player is presented with a series of blanks, and they must guess the letters that fill in the blanks to make a word. The player has a limited number of guesses, and if they cannot guess the word before they reach the limit, they lose the game. We converted it into a multi-player game so that two players can collaboratively work together on separate devices to guess the word.

MyAssignment [1] is an app for students to track their class assignments or projects. A single user can input information about projects, progress, due dates, etc. We converted it into a multi-user app so that multiple students can collaborate on class assignments.

After converting these apps, we ran them with multiple users on three devices to verify that in fact the apps work correctly. They are all publicly available [1].

Based on the lessons we learned from converting these four apps, we designed a process to convert a single to a multi-user app. Our process consists of four steps:

1. Identify data that needs to be moved to the cloud
2. Create new cloud data types to hold the shared data
3. Replace the local usage with cloud usage for the shared data
4. Initialize cloud data

Next we will illustrate these steps using the MILEAGE TRACKER app. In step 1 we identified the data that needed to be shared: (i) the `MilageRecord` is a collection of numbers that holds the Miles Per Gallon for the past usage consumption, (ii) `MaximumRecordEntries` is a number that determines how many records should be stored, (iii) `UseUSUnits` is a boolean that stores the preference between metric or imperial units.

To illustrate step 2, let us consider one of the shared data types, the `MilageRecord` collection. We created a new `Cloud Data Table` named `MileageRecordTable` to hold the data. Notice that the

original data is stored in a one-dimensional data structure, whereas the `Cloud Data Table` is a two-dimensional data structure.

In step 3 we replaced all the uses of the local `MilageRecord` with uses of `MileageRecordTable`.

In step 4 we initialized `MileageRecordTable`. Since the data is now persistent on the cloud, we need to change the initialization code from eager to lazy in order to avoid erasing all the data every time the app is launched. In addition, we need to make a choice between using the cloud (i) as a backup for the data for one user across multiple devices("just me session") or (ii) to enable collaboration between multiple users across multiple devices ("everyone session"). These changes required adding 3 LOC and updating 2 LOC.

Notice that steps 1 and 4 require domain knowledge in order to choose and refine the proper end-user experience.

Based on the type of the input as well as the local data that we needed to migrate to cloud, we have identified three kinds of refactorings. Each refactoring can be performed using the general steps that we identified above. First, we refactored primitive data types into the corresponding cloud-enabled primitive type (e.g., from `Number` to `Cloud Number`). Second, we refactored local `Data Table` into `Cloud Data Table`. Third, we refactored local `Collection` into `Cloud Data Table`.

When carrying out the refactorings, we noticed that steps 1 and 4 are hard to automate as they require understanding the original program and the desired end-user experience. It took the second author an average of 6 minutes to perform steps 1 and 4 for each app in the formative study. Steps 2 and 3 have different degrees of complexity. The first two kinds of refactorings are trivial because there is a perfect match between the local data and the cloud data type, so the change is as simple as prepending the keyword `Cloud` to the variable declaration.

The third kind of refactoring is non-trivial: it requires changing the program from using a flat, one-dimensional `Collection` to a two-dimensional `Cloud Data Table`, as shown in Figure 1. The APIs are different enough that sometimes we needed to map one function call from `Collection` into a sequence of calls from `Table`, whereas other times we had to augment the API by writing new functions.

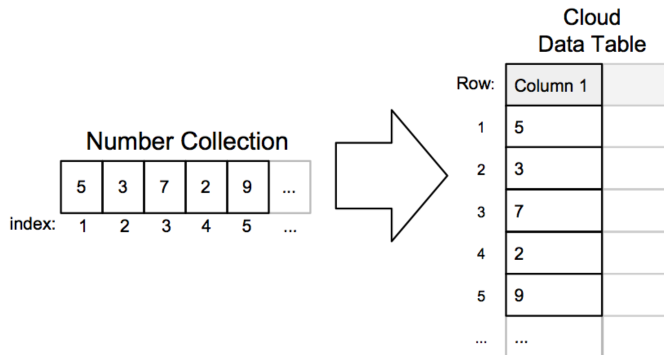

Figure 1. Conversion from Number Collection to Cloud Data Table

For these reasons, we automated this refactoring.

Table 1 lists the total number of refactorings that we applied as part of our formative study.

4. AUTOMATED REFACTORING

This section presents our automated refactoring which converts from a local `Number Collection` to a `Cloud Data Table`. This refactoring is composed of several transformations. The transformations are the individual changes that CLOUDIFYER must apply to the program in order to accomplish this refactoring. We implemented the refactoring in a tool, CLOUDIFYER, which itself is a touch**develop** app.

Table 1 Refactorings used in the formative study

App Name	Primitive to Cloud	Local Table to Cloud Table	Collection to Cloud
MileageTracker	2	0	1
Buisness Manager+	0	0	5
CliffHangers	8	0	2
My Assignments	0	1	0
Total	10	1	8

Section 4.1 presents the workflow of using the tool. Section 4.2 presents the preconditions that must be true before applying the refactoring. Section 4.3 presents the three kinds of transformations needed to perform the refactoring.

4.1 Workflow

CLOUDIFYER is implemented as a touch**develop** app. It can be run using the touch**develop** platform like any other touch**develop** app. Once it is run, the app first prompts the user to select a script on which to perform the refactoring. Once the user has selected a script, CLOUDIFYER will display all the possible refactoring targets for the chosen script, and allow the user to choose which `Number Collection` they wish to convert to a `Cloud Data Table`. CLOUDIFYER will then perform all the needed transformations on the selected `Number Collection` in an automated fashion.

4.2 Preconditions

In the current implementation of touch**develop**, there are some differences in how `Number Collections` and `Cloud Data Tables` are treated. For example, an instance of a `Number Collection` can be passed as an argument or returned from a touch**develop** function. However, `Cloud Data Tables` cannot be passed as function arguments, or returned from functions. Also, `Number Collections` can be assigned to a variable, where as `Cloud Data Tables` cannot. Based on these differences, we developed two preconditions that must be met before CLOUDIFYER can perform a refactoring. These preconditions are inherent to how the touch**develop** language works, not limitations of our tool.

(P1) A `Number Collection` instance must not be passed as an argument or returned from a function.

(P2) A `Number Collection` instance must not be assigned to a variable.

4.3 Transformations

There are four categories of transformations to refactor `Number Collection` to `Cloud Data Tables`: (i) creating `Cloud Data Table` data structure, (ii) direct mapping of APIs, (iii) indirect mapping of APIs, and (iv) injected function to augment missing APIs. Table 2 displays the mapping between the APIs of the two data types.

We will illustrate the transformations using simplified code snippets from the MileageTracker app that we introduced in the Formative Study (see Section 3), an app that enables users to track their fuel usage. Figure 2 shows the code before and after the refactoring. The left column shows the original code. Line 1 shows the declaration of the `MileageRecord Number Collection`, which is the target of the refactoring. We show four functions to display all mileage record, clear existing mileage record, add a new entry to the mileage collection, and compute and display the average mileage.

Table 2 `Number Collection` API's and corresponding `Cloud Data Table` Operations.

Number Collection Operations	Cloud Table Operations	Transformation Type
`clear`	`clear`	Direct
`count`	`count`	Direct
`post to wall`	`post to wall`	Direct
`add`	`add row`	Indirect
`at`	`row at`	Indirect
`set at`	`row at→valueName`	Indirect
`remove at`	`row at→deleteRow`	Indirect
`insert at`	`row at→value`	Indirect
`add many`	NONE	Injected Function
`avg`	NONE	Injected Function
`contains`	NONE	Injected Function
`index of`	NONE	Injected Function
`max`	NONE	Injected Function
`min`	NONE	Injected Function
`random`	NONE	Injected Function
`remove`	NONE	Injected Function
`reverse`	NONE	Injected Function
`sort`	NONE	Injected Function
`sum`	NONE	Injected Function

Create `Cloud Data Table` data structure: Lines 1-3 of the right-hand side of Figure 2 show the newly created data structure, `MileageRecordTable`. We append the `Table` to the end of the name for clarity. This `Table` contains one column, `MileageRecordColumn`. We will use this column to store the values that were originally stored in the `MileageRecord` collection.

Direct Transformations: The simplest transformations are those when the mapping from the `Number Collection` to `Cloud Data Table` APIs is one-to-one and both data types use the same name (e.g., `post to wall`). In this case, the transformation consists of simply replacing the receiver object. Figure 2 (a) line 8 shows the API call before the transformation, while Fig. 2 (b) line 8 shows the code after the transformation.

Indirect Transformations: Indirect Transformations are more complex than Direct Transformations, and cannot be performed with a simple find-and-replace technique. We use Indirect Transformations when the same functionality can be accomplished in both data types, but the APIs are different: the cardinality and/or the name is different. Sometimes, the transformation maps one API call from the `Number Collection` into a sequence of API calls from the `Cloud Data Table`. In addition, sometimes it turns a function call into an assignment, further increasing complexity.

One example of an Indirect Transformation is shown in Figure 2. The function `AddMileageToRecord` adds a number to the `MileageRecord Number Collection`. Line 26 in column (a) shows `Mileage` being added via the API call `add`. In order to correctly perform the transformation, `Mileage` now needs to be added to the `Cloud Data Table MileageRecordTable`. This becomes a two step process. Step 1 is to add a new row to the `Cloud Data Table`, and step two is to assign the value of `Mileage` to that row. This is shown in column (b) lines 26-27.

Injected Function Transformations: We used Injected Function Transformations when some API functionality in `Number Collection` cannot be replicated using APIs from `Cloud Data Table`. In order to transform such missing APIs, CLOUDIFYER inserts Injected Functions that provide the same functionality. Since touch**develop** does not allow `Cloud Data Tables` to be passed to functions, but instead must be used as a global variable, CLOUDIFYER cannot create generic functions to replicate the API functionality. Instead, it must inject a function for each `Cloud Data Table` that has been created as a result of a refactoring. We developed a template for each Injected Function Transformation. These templates have "holes" and CLOUDIFYER then fills up these "holes" with the relevant table name and column name. Thus CLOUDIFYER generates these Injected Functions fully automatically.

One example of an Injected Function Transformation is shown in Fig 2 (a) on lines 32-33. The function `DisplayAvgMileage` calculates the average value of all contents of `MileageRecord`, and then displays it. Once CLOUDIFYER refactors the program to use a `Cloud Data Table` to store the data, there is no longer an API call that will calculate the average value of all contents of a `Cloud Data Table`. Instead, CLOUDIFYER must introduce the injected function `MileageRecordTable avg` (see Fig. 2 (b) lines 36-45) that will compute the average of all the values in the `MileageRecordTable`. Now the API call in `DisplayAvgMileage` is transformed to call this `avg` injected function instead, as seen in Fig. 2 (b), line 32.

Another example of an Injected Function Transformation is shown in Fig 3. Column (a) shows the template for the function that calculates the maximum value of a `Cloud Data Table`. Column (b) shows the same function once it has been injected in the Mileage Tracker app.

Concurrency: Since multiple users can access the cloud data at the same time and perform updates concurrently, one needs to worry about the atomicity of compound operations in our Injected Functions. However, touch**develop** guarantees atomicity at the function level, thus our Injected Functions will execute atomically.

4.3.1 Tool Limitations

The current version of CLOUDIFYER does not support the injected function transformation needed to transform the API call `add many`. We believe that this function could be implemented but it remains as future work.

So far, we implemented CLOUDIFYER only for `Number Collections`. In order to add other types of collections, we would need to write new injected functions in order to provide additional functionality. For example, in order to support the `String Collection`, we

```
1   data MileageRecord : Number Collection
2
3
4
5   private action ShowMileageGraph ()
6   //Shows a graph of the mileage record on the execution wall.
7   wall->clear
8   MileageRecord->post to wall
9   wall->prompt("Click␣\"Ok\"␣to␣continue...")
10
11  ...
12
13  private action ClearMileageRecord ()
14  //This action clears the mileage record.
15  wall->clear
16  if wall->ask boolean("Warning", "Are␣you␣sure␣you␣want...
17  ␣␣to␣clear...")->equals(true) then
18    MileageRecord := collections->create number collection
19  else do nothing
20
21  ...
22
23  private action AddMileageToRecord (
24    Mileage : Number)
25  do
26    MileageRecord->add(Mileage)
27
28
29  ...
30
31  private action DisplayAvgMileage ()
32  var avgMileage := MileageRecord->avg
33  avgMileage->post to wall
34
35
36
37
38
39
40
41
42
43
44
45
46
47  ...
```

(a) before

```
1   cloud table MileageRecordTable
2   columns
3     MileageRecordColumn : Number
4
5   private action ShowMileageGraph ()
6   //Shows a graph of the mileage record on the execution wall.
7   wall->clear
8   MileageRecordTable table->post to wall
9   wall->prompt("Click␣\"Ok\"␣to␣continue...")
10
11  ...
12
13  private action ClearMileageRecord ()
14  //This action clears the mileage record.
15  wall->clear
16  if wall->ask boolean("Warning", "Are␣you␣sureâĂę")
17      ->equals(true) then
18    MileageRecordTable table->clear
19  else
20
21  ...
22
23  private action AddMileageToRecord (
24  Mileage : Number)
25  do
26    MileageRecordTable table->add row
27      ->MileageRecordColumn := Mileage
28
29  ...
30
31  private action DisplayAvgMileage ()
32  var avgMileage := MilageRecordTable avg
33  avgMileage->post to wall
34
35
36  private action MilageRecordTable avg ()
37  returns (
38    avg : Number)
39  do
40    var sum := 0
41    for each ct in  MileageRecordTable table
42      where true
43    do
44      sum := sum + ct->MileageRecordColumn
45  avg := sum / MileageRecordTable table->count
46
47  ...
```

(b) after

Figure 2. Relevant code from Mileage Tracker app. The left-hand side shows the original app, whereas the right-hand side shows the refactored code. The modified code is underlined, and the `avg` is a newly injected function.

```
1   private sync action $TABLENAME max ()
2   returns (
3     max : Number)
4   do
5     max := $TABLENAME -> row at(0) -> $COLUMNAME
6     for each ct in $TABLENAME
7       where true
8     do
9       if ct -> $COLUMNAME > max then
10        max := ct -> $COLUMNAME
11      else do nothing
```

(a) Template Function

```
1   private action MileageRecordTable max ()
2   returns (
3     max : Number)
4   do
5     max := MileageRecordTable table -> row at(0) -> MileageRecordColumn
6     for each ct in MileageRecordTable table
7       where true
8     do
9       if ct -> MileageRecordColumn > max then
10        max := ct -> MileageRecordColumn
11      else do nothing
```

(b) Injected Function

Figure 3. Template showing the injected function `max`. The left hand-side shows the template with "holes", whereas the right-hand side shows the function instantiated to use the `MileageRecordTable`.

would need to write an equivalent injected function for the `join` API method which concatenates the entire collection into one string.

4.3.2 Implementation

CLOUDIFYER refactors touchdevelop scripts *in place*. First, CLOUDIFYER retrieves the source of the target app as an Abstract Syntax Tree (AST) stored in JSON format from the touchdevelop script bazaar. It then transforms the AST as needed. Once all the transformations are performed, CLOUDIFYER completes the refactoring by saving the new AST for the target app.

5. EVALUATION

To determine if CLOUDIFYER is useful we ask the following research questions.
Q1: **APPLICABILITY**: How applicable is the refactoring?
Q2: **EFFORT**: How much effort is saved by CLOUDIFYER when refactoring?
Q3: **ACCURACY**: How accurate is CLOUDIFYER when performing a refactoring?

5.1 Methodology

To create a corpus, we wrote queries against a database of all publicly available scripts in the touchdevelop script bazaar. We wanted to collect popular and mature scripts. Thus, our query returned top scripts (based on the number of times that each script was executed) that contained a `Number Collection`. We retained the top 123 scripts.

Table 3 provides details about our corpus. We report the size in non-blank source lines of code (SLOC) and complexity in number of functions. We also report the number of times the apps were run by the touchdevelop community, as well as the number of installations. Notice that a user can run an app without installing it in the user profile; this feature allows a touchdevelop user to try out an app before installing it.

Table 3 Corpus Characteristics

	Average	Max	Std Dev	Total
SLOC	403	2,526	483	47,185
Functions	21	121	26	2,502
App Runs	46	750	127	5,350
App Installations	5	125	18	635

Table 4 shows the total usage of the `Number Collection` APIs in our corpus. For each API method we report (i) the total number of call sites in our corpus, and (ii) the percentage of call sites among the number of `Number Collection` API calls.

We ran the refactoring in batch mode. Steps 1 and 4 of our process require domain knowledge. For Step 1, we refactored all the `Number Collections` in our corpus instead of trying to identify which data makes sense to be moved to the cloud. For Step 4, we used the same policy that the app currently uses for data initialization, as changing that would involve making design decisions for each app. As far as the data session, we used the default session provided by touchdevelop, the "just me session."

In order to determine the applicability of CLOUDIFYER, we report the number of scripts that our tool was able to refactor successfully, as well as the number of each kind of transformations performed inside a refactoring.

In order to determine the effort saved by using this refactoring tool we record the number of seconds each refactoring took.

Table 4 Usage of Number Collection API reported in number of call sites

API Name	# Call Sites	% of total `Number Collection` API Call Sites
At	1010	39.25
Set At	632	24.56
Add	532	20.68
Count	109	4.24
Contains	107	4.16
Clear	84	3.26
Remove At	53	2.06
Sum	12	0.47
Avg	10	0.39
Sort	8	0.31
Min	4	0.16
Post To Wall	4	0.16
Reverse	4	0.16
Max	3	0.12
Random	1	0.04

In order to validate the accuracy of CLOUDIFYER, we randomly chose 20 scripts from the corpus of 123 refactored scripts. We ran the scripts and exercised the features for three minutes before and after each refactoring to ensure that the refactorings did not change the scripts' runtime behavior. Indeed, we did not observe any runtime differences before and after the refactoring.

In addition to running these 20 scripts we also carefully inspected the refactored code. We counted transformations that CLOUDIFYER (i) applied correctly, (ii) applied incorrectly, and (iii) missed. The first two authors manually refactored these same 20 scripts, and created a set of transformations for these refactorings. We define this set of transformations that an expert would apply as our *gold standard*. For each transformation that CLOUDIFYER applied, we manually compared the result of the automated transformation with the gold standard to determine if that transformation had been applied correctly. We define a *TruePositive* as a transformation that CLOUDIFYER identified and applied correctly. We define *FalsePositive* as a transformation that CLOUDIFYER applies to the code, but it is not part of the gold standard. In other words, CLOUDIFYER transformed code that should not have been transformed. We define a *FalseNegative* as a transformation that CLOUDIFYER did not apply, but it is in the gold standard. In other words, CLOUDIFYER missed applying a needed transformation.

Using these metrics we calculate precision and recall for CLOUDIFYER, using the standard definitions:

$$precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

$$recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

5.2 Results

Next we present the results for each of the three research questions.

Applicability: Table 5 shows the results for running CLOUDIFYER on 123 scripts. Out of the 123 scripts that CLOUDIFYER attempted to refactor, 116 (94%) met the preconditions for the refactoring.

Only 7 scripts were unable to be refactored. Out of these, 4 scripts did not meet precondition **(P1)** (the `Number Collection` was passed as an argument to a function), and 1 did not meet precondition **(P2)** (`Number Collection` was assigned to a local variable). The remaining 2 scripts exposed a limitation of the

current implementation of CLOUDIFYER (they were using a currently unsupported function, `add many`). Thus, we think that our refactoring is widely applicable to touch**develop** scripts that contain the `Number Collection` data type.

Table 5 Total Scripts Refactored

Total Scripts	123
Number Successfully Refactored	116
Failed Precondition **(P1)**	4
Failed Precondition **(P2)**	1
Tool Limitations	2

Effort: Table 6 shows the number of transformations performed per each refactoring. The first three rows show the type of these transformations (i.e., direct, indirect, and injected function). The fourth row shows the total for all types of transformations, while the last row shows the running time for each refactoring.

The first column shows the average per script, while the last column shows the total for all the scripts. Each script had, on average, 24 transformations applied in 3 seconds. Given that the touch**develop** editing experience is optimized for small screens on touch devices, our refactoring tool saves significant effort.

Moreover, most of these transformations are non-trivial. Figure 4 shows the percentage of each kind of transformation applied. By far the most common transformation kind is Indirect Transformation, which involve mapping one API method call to a sequence of one or more API method calls that do not share a common name with the original method. Thus, we believe that automation saves significant effort.

Table 6 Refactorings Applied Per Script

	Avg	Max	Std Dev	Total
Refactorings	1.84	8	1.38	211
Direct Trans	2.22	30	4.45	249
Indirect Trans	19.88	466	56.62	2227
Injected Function	2.20	30	5.01	246
Total Trans	24.30	466	56.56	2722
Time [sec]	2.64	27	5.18	296

Accuracy:

By manually checking refactorings applied in 20 scripts, we found CLOUDIFYER's *precision* to be 100% and its *recall* 94%. This means that all the transformations that CLOUDIFYER applied are correct. However, CLOUDIFYER missed 8 potential transformations out 145 transformations that it should have applied. We identified the root cause in our current implementation, and we expect a future version will fix this.

5.3 Threats to validity

Construct Validity: Do our metrics indeed measure the advantages of using CLOUDIFYER? Can we measure development effort by simply counting the time and number of transformations per refactoring? Ideally, we would have performed an experiment with real touch**develop** developers and observe them while refactoring. However, given the cutting edge nature of the Cloud Data Types, we could not find such developers. Thus, we chose to use indirect metrics, as it is commonly done in the literature [15, 33]. Second, are we sure that the performed refactorings are accurate? Ideally, we should have run tests before and after each refactoring. However, very few scripts in touch**develop** have any tests at all. In order to mitigate this we manually inspected a random set of 20 apps and

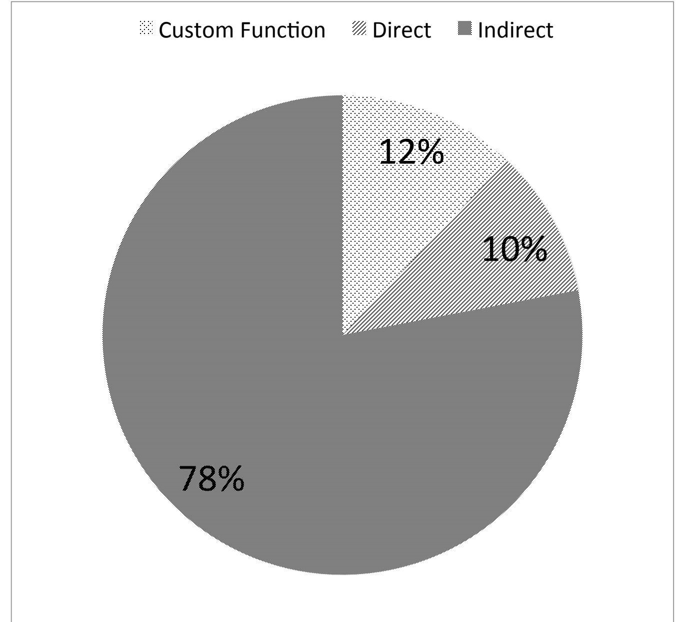

Figure 4. Transformations by Percentage

exercised their functionality to determine that the runtime behavior did not change.

Internal Validity: How did we mitigate bias during manual inspection? We randomly sampled from the set of our corpus, until we had sampled from a variety of app domains. Also, we carefully constructed the set of transformations in the gold standard *before* we applied CLOUDIFYER. The authors are all experts on the touch**develop** platform.

Also, the corpus that we used in the evaluation predates the introduction of Cloud Data Types, and thus the developers were unaware of our later study and could not bias their code to help CLOUDIFYER.

External Validity: Do our results generalize? For our corpus, we used 123 of the most commonly run scripts, but we put no restrictions on the type of the scripts. Our corpus includes scripts from various domains: entertainment, sports, games, education, productivity, etc. Moreover, the apps in our corpus are developed by 83 authors from a diverse end-user community. We do not foresee any reasons why our results would not generalize to other touch**develop** apps. Currently there is no widely accepted standard for cloud data types. In order for our results to generalize, cloud storage providers will first need to agree on Cloud Data Storage standards.

Reliability: Is our evaluation reliable? The apps that we used to evaluate our tool are all available on the touch**develop** bazaar, and CLOUDIFYER is published on the touch**develop** website. All these URLs are available on our webpage:
`http://cope.eecs.oregonstate.edu/cloudifier`

6. DISCUSSION

There are two reasons why touch**develop** is well suited for refactoring to the cloud. The first is that cloud types in touch**develop** have well-defined but constrained semantics which includes the eventual consistency model as described in section 2.1. The second is that touch**develop** does not allow `Number Collection` data objects to be aliased.

In order to generalize our results beyond touch**develop**, one must deal with the problem of aliasing. This is a problem when the object that is being moved is referenced by another alias in the code. Consider a local object that one wishes to convert into a cloud data type. If there are aliases to the local reference, any subsequent method calls via that alias will cause the application to crash, unless one finds all aliases and changes their data type as well. The reason this is not a concern for our tool is that touch**develop** only allows cloud data types to be global objects, therefore ensuring there are no aliases to the object that was moved to the cloud.

In order to deal with these issues in environments that allow aliases (e.g., in Java), researchers [31] have been forced to implement distributed shared memory via a combination of code transformation/generation.

7. RELATED WORK

We group the related work in the following categories: (i) cloud-related refactorings, (ii) multi-user apps, and (iii) tools for the end-user development community.

Refactoring and the Cloud: There is a lot of research on the generic topic of refactoring. While the original research was using refactoring to improve the design of existing programs, the more recent work used refactoring techniques to retrofit concurrency [11, 34], functional features [15], etc. As far as we know, we present the first automated tool to introduce mobile cloud computing via an automated refactoring tool.

The closest research to our current work is by Kwon and Tilevich [19] who proposed *Cloud Refactoring*, a tool to refactor methods of an enterprise application system into services that can be ported to the cloud. Their tool also determines if a method is a good candidate for refactoring using static and dynamic analysis. However, even though their tool automatically refactors methods into services, the programmer must still move them to the cloud manually. Our tool, CLOUDIFYER, performs the refactoring fully automatically, including moving data to the cloud.

Strauch et al. [28] describe a methodology for refactoring applications to move local data to cloud data. They propose several cloud data patterns, and describe migration scenarios for these patterns. Although this work provides a methodology at a high level, the application of the refactoring is completely manual and left up to the programmer. Ling et al. [22] describe a systematic refactoring approach using category theory to formally define an approach to convert Object Oriented systems into Service-Oriented Architecture systems. Our work is different from these in that we provide a tool to perform the refactoring in an automated manner.

Researchers have used many approaches to harness the power of the cloud. One of the more common uses of the mobile cloud is to offload computing in order to make apps more energy efficient and thereby preserve battery life, such as Spectra [12], Slingshot [29], and MAUI [10]. Kwon and Tilevich [18] proposed an approach to offload computation to the cloud to save battery life that is tolerant to network outages or unavailability.

CloudCone [7] also attempts to harness the power of mobile cloud computing, but their approach is significantly different then ours in that they create clones on the cloud, and then move the execution of the application as well as it's state between the cloud and the device.

Researchers [3, 26] have also used refactoring techniques to migrate from a version of the library to another version. Balaban et al. [3] propose refactorings that automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements, and an analysis based on type constraints determines where the replacement can be done. Nguyen et al. [26] designed a technique to identify API changes automatically. Their analysis compares two library versions and extracts knowledge about how the updated client code has migrated to the newer version of the API. Since this line of work is investigating API migration between two versions of the same class, the changes are less dramatic than in the case of migrating between a one-dimensional data type (`Number Collection`) and a two-dimensional data type (`Cloud Data Table`). Such a dramatic API change requires custom program transformations, and cannot be specified with a generic rule system.

Multi-user apps: Many have studied multi-user apps in the following domains: collaboration [23, 36], gaming [20], Geospatial Applications [13] (e.g., Google earth), information sharing [25], and music [27]. This research focuses on how multi-user technologies impact human interactions. In contrast, we are focused on the technical challenges of refactoring from single to multi-user apps.

End-user Development community: Lewis et al. [21] describe the need of End User Programming tools to make End User Programming more acceptable.

touch**develop** is a programming platform developed by Microsoft Research that allows hobbyist, novice programmers and end users to program *on* their phone for their phone. There are other programming environments that are targeted to a similar audience, including: App Inventor [35], Scratch [24], Alice [9], and Greenfoot [17]. Our recent work [2] provides refactoring tools for users of Excel. However, our current paper is the first paper to engage the mobile end-user in a refactoring workflow.

8. CONCLUSION

Mobile cloud computing can be harnessed to enable rich access to data. In this paper we presented a formative study to convert four apps to use the touch**develop** cloud data types. Based on these lessons, we designed, implemented, and evaluated a tool, CLOUDIFYER, to refactor local data types into cloud data types. Our empirical evaluation on 123 real apps, shows that the tool is applicable, relevant, and saves human effort. Using our refactoring tool in combination with a well designed, powerful end-user programming platform (touch**develop**) can enable even novice programmers to take advantage of the power of the cloud.

9. ACKNOWLEDGEMENTS

We would like to thank Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, Alexandru Gyori, Semih Okur, Yu Lin, Cosmin Radoi, Eli Tilevich, and the anonymous reviewers for feedback on earlier drafts of this paper. This research is partly funded through NSF CCF-1213091 and CCF-1219027 grants, a SEIF award from Microsoft, and a gift grant from Intel.

10. REFERENCES

[1] Refactoring to cloud data. March '14, `http://cope.eecs.oregonstate.edu/cloudifier/`.

[2] S. Badame and D. Dig. Refactoring meets spreadsheet formulas. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 399–409, 2012.

[3] I. Balaban, F. Tip, and R. M. Fuhrer. Refactoring support for class library migration. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 265–279, 2005.

[4] E. A. Brewer. Towards robust distributed systems (abstract). In *19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 7–, 2000.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and M. Sagiv. Eventually Consistent Transactions. In *European Symposium on Programming*

(ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS, volume 7211, pages 64–83, 2012.

[6] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 283–307. 2012.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *6th conference on Computer systems (EuroSys)*, pages 301–314. ACM, 2011.

[8] J. P. Cohn. Citizen science: Can volunteers do real research? *BioScience*, 58(3):192–197, 2008.

[9] S. Cooper. The design of alice. *ACM Transactions on Computing Education (TOCE)*, 10(4):15, 2010.

[10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *8th international conference on Mobile systems, applications, and services (MobiSys)*, pages 49–62. ACM, 2010.

[11] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering (ICSE)*, pages 397–407. IEEE Computer Society, 2009.

[12] J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 217–226. IEEE, 2002.

[13] C. Forlines, A. Esenther, C. Shen, D. Wigdor, and K. Ryall. Multi-user, multi-display interaction with a single-user, single-display geospatial application. In *19th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 273–276. ACM, 2006.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[15] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *9th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 543–553, 2013.

[16] A. Khan, M. Othman, S. Madani, and S. Khan. A survey of mobile cloud computing application models.

[17] M. Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14, 2010.

[18] Y.-W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 586–595. IEEE, 2012.

[19] Y.-W. Kwon and E. Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, pages 1–28, 2013.

[20] K. Leichtenstern and E. André. Studying multi-user settings for pervasive games. In *11th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI)*, pages 25:1–25:10. ACM, 2009.

[21] G. Lewis, D. Smith, L. Bass, and B. Myers. Report of the workshop on software engineering foundations for end-user programming. *ACM SIGSOFT Software Engineering Notes*, 34(5):51–54, 2009.

[22] H. Ling, X. Zhou, and Y. Zheng. Refactoring from object-oriented systems to service-oriented systems: A categorical approach. In *International Conference on Service Sciences (ICSS)*, pages 214–218. IEEE, 2010.

[23] R. Lopez-Gulliver, H. Tochigi, T. Sato, M. Suzuki, and N. Hagita. Senseweb: Collaborative image classification in a multi-user interaction environment. In *12th Annual ACM International Conference on Multimedia (MM)*, pages 456–459. ACM, 2004.

[24] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[25] M. A. Nacenta, M. R. Jakobsen, R. Dautriche, U. Hinrichs, M. Dörk, J. Haber, and S. Carpendale. The lunchtable: A multi-user, multi-display system for information sharing in casual group interactions. In *2012 International Symposium on Pervasive Displays (PerDis)*, pages 18:1–18:6. ACM, 2012.

[26] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 302–321. ACM, 2010.

[27] H. Sørensen and J. Kjeldskov. The interaction space of a multi-device, multi-user music experience. In *7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design (NordiCHI)*, pages 504–513. ACM, 2012.

[28] S. Strauch, V. Andrikopoulos, and T. Bachmann. Migrating application data to the cloud using cloud data. In *e 3rd International Conference on Cloud Computing and Service Science, (CLOSER)*, pages 36–46. SciTePress, 2013.

[29] Y.-Y. Su and J. Flinn. Slingshot: deploying stateful services in wireless hotspots. In *3rd international conference on Mobile systems, applications, and services (MobiSys)*, pages 79–92. ACM, 2005.

[30] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29:172–182, December 1995. ISSN 0163-5980.

[31] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19 (1):1:1–1:40, Aug. 2009. ISSN 1049-331X.

[32] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD)*, pages 49–60. ACM, 2011. ISBN 978-1-4503-0941-7.

[33] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 173–182, 2009.

[34] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 173–182. ACM, 2009.

[35] D. Wolber. App inventor and real-world motivation. In *42Nd ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 601–606. ACM, 2011.

[36] N. Yuill and Y. Rogers. Mechanisms for collaboration: A design and evaluation framework for multi-user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 19(1):1:1–1:25, May 2012. ISSN 1073-0516.