Declaratively Programming the Mobile Web with Mobl

Zef Hemel
Delft University of Technology & Cloud9 IDE Inc.
z.hemel@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

A new generation of mobile touch devices, such as the iPhone, iPad and Android devices, are equipped with powerful, modern browsers. However, regular websites are not optimized for the specific features and constraints of these devices, such as limited screen estate, unreliable Internet access, touch-based interaction patterns, and features such as GPS. While recent advances in web technology enable web developers to build web applications that take advantage of the unique properties of mobile devices, developing such applications exposes a number of problems, specifically: developers are required to use many loosely coupled languages with limited tool support and application code is often verbose and imperative. We introduce *mobl*, a new language designed to declaratively construct mobile web applications. Mobl integrates languages for user interface design, styling, data modeling, querying and application logic into a single, unified language that is flexible, expressive, enables early detection of errors, and has good IDE support.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.11 [*Software Engineering*]: Software Architectures; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Design, Languages, Verification

1. Introduction

With the rapid growth in sales of modern smart phones and tablets, such as iPhone, iPad, Android and BlackBerries, the web becomes available on an increasing number of powerful mobile devices equipped with modern web browsers. However, today's websites are optimized for personal computer browsers and environments, whereas mobile devices are used in different contexts, and have different features and constraints than personal computers, for instance:

- Internet access is not always available, reliable or fast;
- Screen estate is limited;
- Expected user interaction patterns are different, such as touch controls and gestures such as tapping, swiping and pinching;
- Applications are expected to respond to changes in context, such as holding the device in portrait or landscape mode, or changes in location.

Consequently, hundreds of thousands of custom *native* mobile applications are being developed. Examples include communication applications (e-mail, messaging), content viewers (books, articles, papers, RSS feeds, video, photos, audio) and location-based services (wikihood, foursquare, loopt). While these applications run locally on the device itself, a large class of these applications are *data-driven applications* that communicate with one or more web services to exchange data.

While iOS, Android, BlackBerry, WebOS, Windows Phone 7 and other platforms are similar in terms of interaction, features and restrictions, their development environments are quite different. iPhone and iPad applications are developed using the Objective-C language; Android and BlackBerry applications are built using Java, but using very different APIs; WebOS applications use HTML, CSS and JavaScript; Windows Phone 7 development is done using .NET. Developing software that is *portable* to multiple platforms is difficult. In addition, *deployment* is non-trivial; most platforms come with an application marketplace, some of which require manual testing of submitted applications by the marketplace provider before being published — a process that can take many weeks — and applications can be rejected for seemingly arbitrary reasons.

At the end of the 1990s, mobile phones started to gain access to the Internet through WAP (Wireless Application Protocol). The development model for WAP applications was very similar to the development of regular web applications. Rather than sending HTML, a server would send WML (Wireless Markup Language) to the mobile phone. With the release of the original iPhone in 2007, a new generation of smart phones and tablets started to be released with more powerful browsers that support all modern web technologies. At the same time, advancements in HTML (HTML 5) and CSS (CSS 3) started to enable the creation of web applications that offer a comparable experience to native applications, especially for *data-driven applications*, by supporting application and data caching, detection of touch gestures and access to geographical position information (GPS). The portability and deployment advantages of web applications make the use of web technologies for building mobile applications very attractive.

Similar to *native* applications, mobile web applications can now be developed that run completely disconnected from the server, requiring a different development model than regular web applications. When a mobile web application is first launched through the web browser, its application code is cached on the device. The application can use local SQL databases to cache data obtained from a server for offline use. When no Internet connection is available, the mobile browser retrieves the application from its cache and continues to operate. All application logic, written in JavaScript, resides on the device rather than on the server as is the case in regular web applications. Communication with the server, similar to native applications, happens by performing web service calls using AJAX (Asynchronous JavaScript and XML). At the time of writing, HTML5 is well supported by the iPhone, iPad, Android, WebOS and BlackBerry (6+) platforms.

While HTML5 makes it *possible* to develop offline-capable mobile web applications that are portable and easy to deploy, development of such applications exposes a number of problems.

First of all, web development does not enforce a particular application architecture; application concerns (such as data modeling, user interface and application logic) can be mixed arbitrarily – an approach that does not scale well. Therefore, a *structured architecture* is required to develop mobile web applications. A common architectural style in organizing user-facing software is the Model-View-Controller (MVC) pattern [6]. The MVC pattern separates the Model (data, e.g. in a database) strictly from the View (the user interface) by making the Controller responsible for communication between the two. While the separation of View and Model is good, the MVC pattern results in *boiler plate code* that needs to be written to glue the application together.

Second, mobile web applications are built by mixing a number of *loosely coupled languages* including HTML, CSS, JavaScript, SQL and a cache manifest. While the use of domain-specific languages in web development support a declarative programming model, they are not very well *integrated*. In previous work we have studied the current state of server-side web frameworks [8] which, similar to mobile web development, take advantage of multiple loosely-coupled languages. The consequence of this design is the same both in mobile and regular web development: lack of static analyses detecting inconsistencies results in *late detection* of failures. In addition, developers have come to expect excellent *IDE support* for their languages, including in-line error highlighting, reference resolving, outlines, code completion and refactoring support. The dynamic nature and loose coupling of the web languages complicates the construction of IDE support.

Third, web languages such as HTML and CSS do not support basic *abstraction* mechanisms, complicating the reuse of user interface elements. As a result, HTML and CSS artifacts contain a lot of code duplication.

Fourth, JavaScript in the browser is a single-threaded environment, forcing developers to use asynchronous APIs for performing expensive computations, including database queries and obtaining GPS coordinates. These asynchronous APIs require the developer to write code in the unnatural *continuation-passing style*, one example of *accidental complexity* in mobile web development.

In this paper, we introduce *mobl*[1], a high-level, declarative language for programming mobile web applications, which addresses these problems. Mobl is our second case study in the design and implementation of *syntactically integraded DSLs*, DSLs that integrate sub-languages for multiple application aspects, enabling static verification of the entire application. Previously, we developed WebDSL, a DSL to develop data-driven web applications. While covering a different domain, many ideas from WebDSL are reused in the design of mobl. Mobl integrates languages for user interface design, styling, data modeling, query and application logic into a single, unified language. The language is *high-level* since it avoids accidental complexity such as continuation passing style and supports the definition of reusable user interface elements. The language is *declarative* since it ensures automatic updates of the user interface through reactive programming and automatic persistence of data in the client-side database.

Mobl implements the *Model-View* (MV) pattern, a variant of Model-View-Controller where the role of Controller has been *automated*, data model-related logic has been moved to the Model and user interface logic has been moved to the View. The MV pattern reduces the amount of *boilerplate code* that needs to be written compared to MVC.

The integration of the various concerns of mobile web programming into a single language, enables consistency checking across concern boundaries, ensuring early detection of many common errors by the mobl IDE (integrated into Eclipse), which provides in-line error reporting, code completion and reference resolving. The mobl compiler compiles mobl code into a pure client-side web application, implemented using a combination of HTML, CSS, JavaScript and application caching manifests. Mobl applications can be deployed to any web server and are server-technology agnostic.

[1] `http://www.mobl-lang.org`

Outline The rest of this paper is organized as follows: Section 2 analyzes the mobile domain and its problems. Section 3 describes the general architecture and design principles of mobl. Subsequent sections discuss the various aspects of mobile applications and how mobl supports them: data modeling (Section 4), user interfaces (Section 5), navigation (Section 6), higher-order controls (Section 7) and styling (Section 8). Section 9 discusses related work and Section 10 concludes.

2. Mobile Web Applications

The design of a new language for mobile web application development requires a thorough understanding of the mobile domain. This section discusses the architecture of traditional web applications and compares it to the architecture of *mobile* web applications. Subsequently, we identify a number of problems in the *development* of mobile web applications.

2.1 Technical Architecture

The traditional style of web applications, sometimes referred to as RESTful web applications [20], are request-oriented. Objects on the server have the life span of a single request, and are recreated as needed on every request. Since making HTTP requests is relatively expensive, they are used sparingly, when navigating to a new page, submitting a form or performing an AJAX (Asynchronous Javascript and XML) call. The web application server responds to requests from the client (browser). When a request comes in, it is handled by a server written using, for instance, Java, .NET, PHP or Ruby. The server communicates with a database to retrieve or manipulate data, and eventually sends back HTML to the browser which renders it on the user's screen. A server handles multiple users and typically stores data for all its users in a shared database. HTTP requests can also be sent by JavaScript code on the web page, using AJAX calls. Based on the result of such a request, the JavaScript may manipulate the HTML DOM (Document Object Model) to make changes to the user interface without requiring an entire page reload. In addition to performing AJAX calls, JavaScript is used for client-side validation of user input in forms.

There are multiple approaches to developing *mobile* web applications. For older, non-smart phones, processing power is the main bottleneck. Therefore, several thin-client approaches exist [12, 11] where all processing happens on the server and phones are served with pre-rendered pages. However, today's modern smart phones have more powerful processors, thus client-side processing is no longer a bottleneck. Therefore, for these devices applications can be developed in a range of styles. On one end of the scale are web applications that are built similarly to regular web applications, except reducing the amount of data presented on a single page, to fit the screen size of the mobile device. It is relatively easy to adjust a regular web application to produce pages that are more friendly to the smaller screen size of a mobile device.

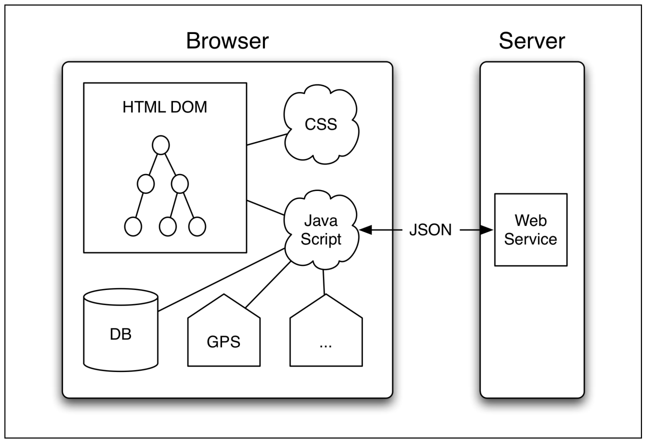


Figure 1. Mobile web application technical architecture

A drawback of this approach is that such applications are not available without an Internet connection. In addition, Internet speeds on mobile devices are on average a lot slower than on PCs, resulting in a bad user experience.

At the other end of the spectrum are *offline-capable* mobile web applications that, once accessed by the mobile browser, are cached locally. They may fetch data from the server and cache it in a local database on the device as well. The development model of this type of application is very similar to desktop applications and native mobile applications and merely use web technologies as an implementation means. All the application logic, written in JavaScript, executes at the client, in the device's browser. This enables much more responsive user interfaces, because a "click" no longer requires a HTTP request be sent to the server. Events can therefore be processed much more granularly than in RESTful-style web applications, and can respond immediately to gestures and key presses. Like desktop applications, mobile web applications are single-user applications that do not require user authentication and access control. This type of application can be used without an Internet connection, after the application and its data is loaded and cached locally. Internet latency on mobile networks is also less problematic because fewer requests have to be sent to the server.

Figure 1 shows the technical architecture of offline-capable mobile web application. The user interface is defined using HTML (HyperText Markup Language) and styled using CSS (Cascading StyleSheets). The runtime representation of the user interface is the Document Object Model (DOM), which can be manipulated at runtime using JavaScript. JavaScript acts as a glue language, manipulating the DOM, calling web services and executing database queries. The application's data is stored in a SQLite database running locally on the device. The database is accessed through an asynchronous JavaScript API that supports the execution of SQL queries. All application resources (such as HTML, CSS, JavaScript and images) are cached locally on the device using the HTML5 Application Cache. When an

Internet connection is available, the application can request data from, and push data to the server.

2.2 Architectural Patterns

There is no particular application architecture enforced in web development. HTML, JavaScript and CSS can be mixed arbitrarily. While lowering the barrier to entry, this unstructured web application development does not scale well for larger applications. Therefore, a number of architectural patterns have been developed for user facing applications. The most commonly used is the Model-View-Controller (MVC) [6] pattern, but alternatives include Model-View-Presenter [19] and Model-View-ViewModel [21].

The Model-View-Controller pattern creates a strict separation between three layers of the application:

1. The **Model** represents the data to be manipulated by the application, e.g. persistent data objects.
2. The **View** defines a user interface, presenting (elements of) the Model.
3. The **Controller** responds to user events and adapts the View and Model accordingly.

While developing WebDSL [22], we studied MVC web frameworks that are commonly used in web development. We observed that the Controller is required to perform a mostly infrastructural role. It is responsible for reading user input, applying requested changed to the Model, and manipulating the View. It impedes the rapid development of applications: minor changes, such as a new property in the Model that has to be editable from the View, requires not only the adaptation of the Model and View, but the Controller as well. Consequently, the use of the Model-View-Controller, as well as similar patterns, result in a lot of *boiler plate code* that needs to be written.

2.3 No Integration

In previous work we surveyed the state of practice in web development [8]. We observed that web frameworks typically rely on a number of *loosely-coupled languages*, e.g. Java, XML configuration files, SQL, HTML, CSS and Javascript. Due to their loose coupling, these framework typically lack tools that can statically verify applications to detect inconsistencies between components of the applications defined using different languages, such as HTML pages that link to non-existing Java controllers, or HTML elements that reference non-existing CSS styles. As a result, errors materialize as runtime faults with obscure error messages that are hard to trace back to their origin.

Mobile web development suffers from the same problem. It too relies on the use of multiple languages, such as HTML for creating user interfaces, CSS for styling, JavaScript for application logic, SQL for database querying and caching manifests for application caching. In addition, since all web languages are dynamically typed, accurate implementation of typical IDE features such as code completion and reference resolving has become challenging. Consequently, tool support for mobile web development is sub-optimal.

2.4 No Abstraction

HTML was architected to define the structure of an entire web page. It does not support the definition of reusable HTML templates, or means to invoke a template. Similarly, CSS's support for abstraction is also limited. Using CSS classes, styles can reused by attaching them to multiple HTML tags, but no parameterization of these styles is supported to vary colors slightly, for instance. SQL does not support abstraction either. A SQL query can only be expressed as a whole, not in reusable parts. Although it is possible to iteratively construct a query by concatenating strings, this is very error prone.

2.5 Accidental Complexity

JavaScript in the browser runs on a single thread that is shared with the page renderer. Therefore, JavaScript calls that take a long time to complete can freeze the browser. As Javascript does not allow developers to create threads, many JavaScript APIs are defined as *asynchronous* APIs. Asynchronous computations are computed on a separate thread (managed by the browser), and call back to the Javascript thread when the computation completes. While synchronous calls return the result of their computation as a return value, asynchronous methods are passed a *callback function* (or *continuation*), which is called with the result when the computation has finished. This style of programming is called *continuation passing style*.

Asynchronous APIs have favorable performance characteristics, because they do not block the user-interface thread. Nevertheless continuation-passing style leads to verbose, difficult to read and maintain code. Effectively, developers have to adapt their programming style as a result of a low-level performance-related issue.

3. Mobl Architecture

We have developed mobl. Mobl is a new statically typed, domain-specific language designed specifically for the rapid development of data-driven mobile web applications.

Mobl *linguistically integrates* all aspects of mobile application development into a single, statically verifiable language. It enables *separation of concerns* by supporting the separation of user interface and data model. It applies *domain abstraction* to abstract from accidental complexity and irrelevant details of the platform/domain. It supports *user-defined abstractions* by enabling users to define reusable screens, controls and styles.

This section discusses the high-level aspects of the language and application architecture. Subsequent sections give detailed descriptions of the sub-languages that mobl comprises.

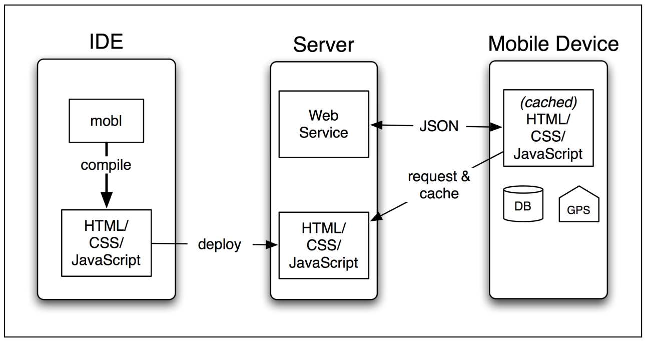


Figure 2. Mobl application compilation and deployment

3.1 Integration and Tooling

HTML, JavaScript and CSS contain numerous cross-references. For instance, CSS selectors rely on the structure of the HTML page and JavaScript is used to manipulate the HTML DOM at run-time, e.g. by attaching or removing CSS classes. Verifying that these cross references are correct, e.g. that a CSS selector matches the right HTML tag and JavaScript attempts to manipulate an existing DOM node, is typically done by *running* the program, resulting in *late failure*. In addition, loose coupling of web languages makes implementing accurate IDE support difficult. Therefore, web development IDEs are not at the level of languages such as Java and C#.

By contrast, mobl integrates the aspects of mobile applications into a single, *integrated language*, rather than using several loosely-coupled languages. Mobl consists of a number of integrated sub-languages for the definition of data models, queries, user interfaces, styles and application logic. Language elements are shared across the sub-languages. For instance, the expression language used in application logic is reused in user interfaces, resulting in a consistent language. This *linguistic integration*, previously also applied in the implementation of WebDSL [8], enables accurate end-to-end static verification of applications, e.g. verifying that controls are invoked correctly, invoked screens exist, the properties of data objects presented in the user interface exist and are of the correct type, and queries filter based on existing properties.

The mobl *compiler* compiles a mobl module to a combination of HTML, JavaScript and CSS. As Figure 2 shows, the resulting compiled files can be deployed to a web server along with any web services that the application may use. A mobile device requests the HTML file, automatically fetching the CSS and JavaScript resources. All application resources are cached in the browser's HTML5 application cache, allowing the application to be launched even when no Internet connection is available. The application runs on the device and has access to a local database, as well as other APIs including Geo Location. The application may call a web service on the server pulling or pushing data, presenting that data and optionally caching it locally in the database.

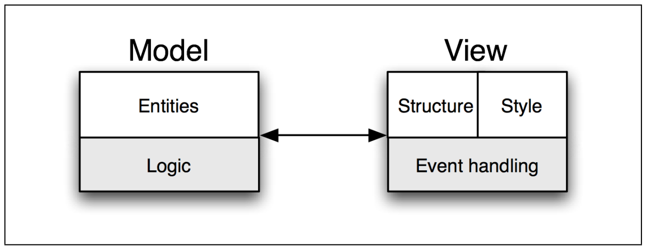


Figure 3. Model-View pattern

The mobl IDE is implemented as an Eclipse plug-in using Spoofax [9]. As Figure 4 illustrates, it offers an outline view, in-line highlighting of verification errors, reference resolving and code completion. The mobl compiler is integrated into the IDE, and triggered on every save of a mobl module. There is also a stand-alone compiler available.

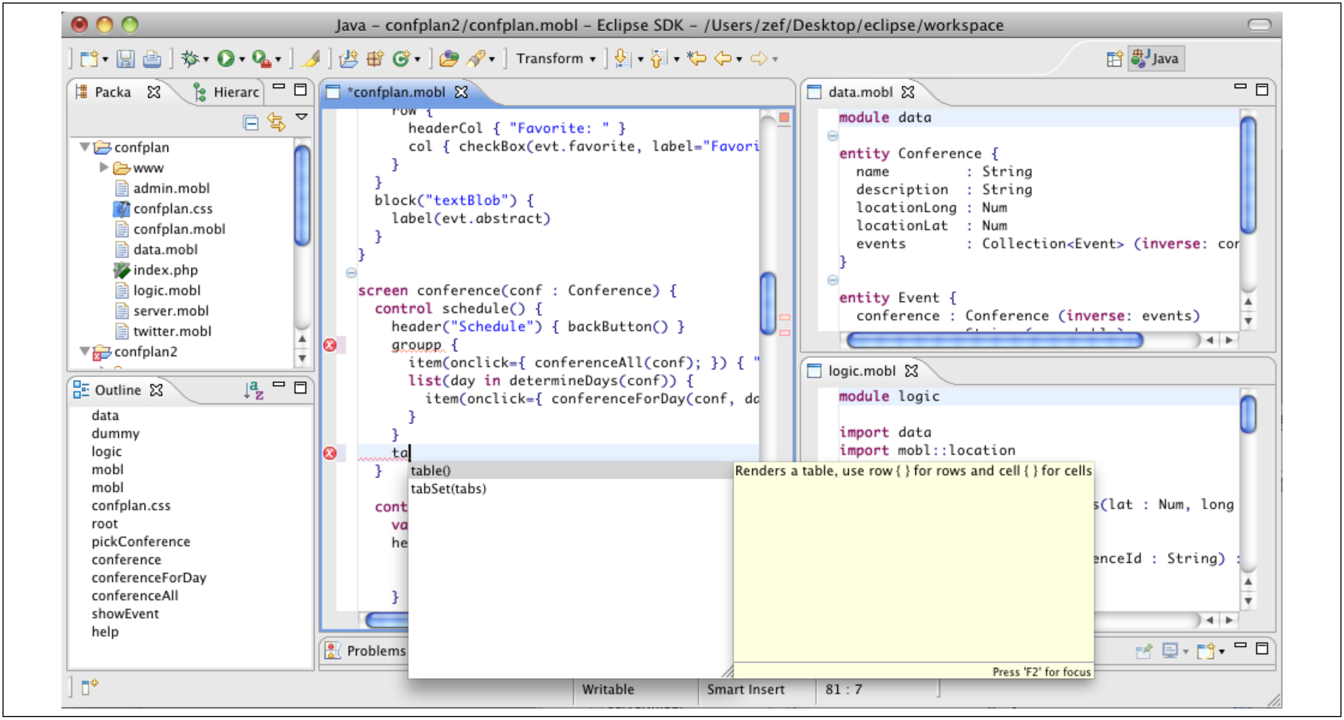
3.2 Model-View Pattern

While the Model-View-Controller pattern is a good organizational tool, it also requires a considerable amount of *boiler plate* code to set up and to achieve simple tasks. This boiler plate is largely caused by the Controller. In a typical application, the Controller has the following responsibilities:

- Read data from the Model and send it to the View;
- Manipulate the Model based on forms defined in the View (user input);
- Persist changes in the Model to database;
- Activate and deactivate (parts of) Views;
- Communicate with external data sources, e.g. web services.

While the core of the application is encoded in the Model and the View, a lots of plumbing code is required in the Controller, while most of the Controller's tasks are very common and infrastructural in nature. Therefore, mobl implements the Model-View (Figure 3) architecture. The MV architecture is an adaptation of the MVC pattern, *automating* the tasks of the Controller rather than letting the developer encode them manually.

In the MVC architecture the Controller is responsible for instantiating Views and populating them with data. In contrast, in the MV architecture Views are the initiators. *Views* can be parameterized with one or more Model objects to present, or they can send a request to the Model themselves to retrieve data. Views are also responsible for handling user input events, such as button clicks and responding to them, e.g. navigating to another View or calling a method on the Model. The *Model* is automatically persisted to a database, no explicit save operations are required. In addition, the Model communicates with web services to synchronize and cache data. *Data binding* establishes a direct connection be-



```
    row {
      headerCol { "Favorite: " }
      col { checkBox(evt.favorite, label="Favori
    }
  }
  block("textBlob") {
    label(evt.abstract)
  }
}

screen conference(conf : Conference) {
  control schedule() {
    header("Schedule") { backButton() }
    groupp {
      item(onclick={ conferenceAll(conf); }) { "
      list(day in determineDays(conf)) {
        item(onclick={ conferenceForDay(conf, da
      }
    }
    ta
  }

  cont
    va
    he

    }
```

Renders a table, use row { } for rows and cell { } for cells

```
module data

entity Conference {
  name         : String
  description  : String
  locationLong : Num
  locationLat  : Num
  events       : Collection<Event> (inverse: cor
}

entity Event {
  conference : Conference (inverse: events)
```

```
module logic

import data
import mobl::location

s(lat : Num, long

enceId : String) :
```

Figure 4. The mobl Eclipse IDE

tween the Model and View, eliminating the need to manually copying data from the Model to the View and vice versa.

4. Data Model

The implementation of an application's data model, as well as the manipulation of data at run-time, is cumbersome in mobile web development because of the lack of *domain abstraction*. This section details the underlying issues and shows how mobl raises the level of abstraction by using declarative data models, its imperative language and integrated query language. It concludes with an example detailing the implementation of a data model for a simple task manager application.

HTML5 data persistence Part of HTML5 is the Web SQL API[2], enabling the creation of local (SQLite) databases in a mobile device's browser. The amount of space available to a database varies from the device to device, but is typically around 5 megabytes. Therefore, the local database is perfectly suited to store small amounts of data and cache data from remote resources.

Since the HTML5 database APIs are new, libraries and frameworks built around them are still limited. Therefore, communicating with an HTML5 database is still done at the level of *low-level SQL statements*, which is not only inconvenient for developers, but also more prone to security problems such as SQL injection attacks. In addition, encoding queries in strings is error prone because developers do not get the support from the IDE that they do get for the rest of the language, including syntax checking, semantic checking and code completion.

SQL queries do not compose well. It is difficult to pass a partial query to a different part of the application where it can be extended, e.g by adding an additional filter condition. Therefore, reuse of queries is limited to what can easily be achieved using string concatenation.

Search Most mobile web applications require simple full-text search functionality, allowing users to quickly search through local data. HTML5 does not offer direct support for this. Therefore, custom solutions need to be built.

Logic Database and web service related Javascript APIs are exposed as asynchronous APIs. This requires the developer to write code in a *continuation-passing style*. For instance, consider the following code written using (hypothetical) *synchronous* JavaScript APIs:

```
var tasksJSON = httpRequest("/export");
tx.executeQuery("INSERT INTO Task ...");
alert("Done!");
```

Javascript's asynchronous APIs, rather than returning the result as the result of a function, are passed a *callback function* and return immediately. The actual execution occurs on a separate thread, managed by the browser. When the computation finishes, the callback function is invoked with the result. Therefore, the above code using asynchronous APIs has to be rewritten as follows:

```
function completed() {
```

[2] http://www.w3.org/TR/webdatabase/

```
Def ::= "entity" ID "{" EBD* "}"

EBD ::= ID ":" Type ("(" {Anno ","}* ")")?
     | "static"? "function" ID
        "(" {FArg ","}* ")" ":" Type
        "{" Stat* "}"

Type ::= ID
       | "Collection" "<" Type ">"
       | "[" Type "]"
       | "(" {Type ","}* ")"

Anno ::= "inverse:" ID
       | "searchable"

FArg ::= ID ":" Type
       | ID ":" Type "=" Exp
```

Figure 5. Data model syntax

```
  alert("Done!");
}
function receiveTasks(tasksJSON) {
  tx.executeQuery("INSERT INTO Task ...",
                  completed);
}
var tasksJSON = httpRequest("/export",
                  receiveTasks);
```

As can be observed, the code in continuation-passing style is written in an inverted order. While this asynchronous code leads to more responsive applications in the browser, it impedes developer productivity. It is a typical example of *accidental complexity*.

4.1 Data

Mobl contains *domain abstractions* for declaratively defining persistent data structures (`entity` definitions), abstracting from the underlying SQL database that implements them. Persistence of data is handled by the mobl runtime transparently.

The syntax of data models is detailed in Figure 5. Data model declarations consist of zero or more `entity` definitions. Every entity has a name, zero or more *properties* and associated *functions* expressing application logic related to the entity. Each property has a name, type and optionally one or more annotations. Its type can be of a scalar type (e.g. `String`, `Num`, `DateTime` or `Bool`) as well as `Collection`s of other entities.

A `Collection` represents a (virtual) collection of entity instances that can be filtered, sorted, paged and manipulated. `Collections` are used to represent one-to-many and many-to-many relationships in models, but also to query persistent data. In addition, the `Collection` abstraction is used for full-text search. The `(searchable)` annotations on textual properties indicate that the property should be included when performing full-text searches on instances of this entity. These searches are performed through a `EntityName.search(phrase)` call, which returns a `Collection` object representing the search results. As with any `Collection`, the results can subsequently be filtered and paged.

An `(inverse: property)` annotation on a property defines `property` as the inverse property of this one. Properties declared as each other's inverse keep each other in sync and are used to declare one-to-one, one-to-many and many-to-many relationships.

Outside data models, mobl also supports variables of other collection types, including arrays and tuples. Arrays are declared using the `[Type]` notation and tuple types using `(T1, T2, T3)` syntax.

Implementation We have developed a JavaScript object-relational mapper [1] (ORM) library called persistence.js[3] to handle data persistence mobl. The library implements transparent data persistence, querying and search. Data models defined in mobl are translated to calls to persistence.js by the mobl compiler. A full-text search index (implementing a simple stemming algorithm [13]) is automatically maintained by the ORM library.

4.2 Logic

Mobl's imperative object-oriented sub-language enables programming in the natural, *synchronous* style, abstracting from the accidental complexity of the asynchronous programming style enforced by HTML5 JavaScript APIs.

Imperative code is written using a JavaScript-like [4] syntax. The language supports variable declarations, assignments, if-statements, for-each and while loops, function and method calls, and various arithmetic expressions. Its full syntax is defined in Figure 6. Mobl comes with an extensive set of libraries[4] containing reusable user interface elements, as well as APIs to call web services, perform web searches and get contextual information such as GPS location and device orientation.

At compile-time, the mobl compiler analyzes mobl imperative code to determine whether it relies on asynchronous methods and functions. If so, it automatically performs the continuation-passing style transform [18], turning code written in a synchronous style to the asynchronous style with callback functions as illustrated in the beginning of this section.

4.3 Query

Mobl's query language is *linguistically integrated* into the expression language part of the imperative language defined in the previous sub-section. The query abstraction is built on the `Collection` abstraction. Collections can be instantiated by the user, but for each entity there is also an `Entity.all()` collection defined, and for each one-to-many and many-to-many property there is a collection ob-

[3] `http://persistencejs.org`

[4] `http://docs.mobl-lang.org`

```
Stat ::= "var" ID "=" Exp ";"
       | LVal "=" Exp ";"
       | Exp ";"
       | "if" "(" Exp ")" Stat
         ("else" Stat)?
       | "foreach" "(" LVal "in" Exp ")"
           "{" Stat* "}"
       | "while" "(" Exp ")" "{" Stat* "}"
           "{" Stat* "}"
       | "return" Exp? ";"
       | "screen" "return" Exp? ";"

LVal ::= ID
       | Exp "." ID
       | "(" LVal "," {LVal ","}* ")"

NamedExp ::= Exp
           | ID "=" Exp

Exp ::= STRING | NUMBER | ID | "true"
      | "false" | "null" | "this" | "!" Exp
      | "(" Exp ")" | "[" {Exp ","}* "]"
      | "(" Exp "," {Exp ","}* ")"
      | ID "(" {NamedExp ","}* ")"
      | Exp "." ID "(" {NamedExp ","}* ")"
      | Exp "." ID | Exp Op Exp
      | Exp "?" Exp ":" Exp | "{" Stat* "}"

Op ::= "||" | "&&" | "==" | "!=" | "<"
     | "<=" | ">"  | ">=" | "*"  | "/"
     | "%"  | "+"  | "-"  | "++" | "--"
```

Figure 6. Imperative language syntax

```
Exp    ::= Exp Filter+

Filter ::= "where" SetExp
         | "order" "by" OrderExp
         | "limit" Exp
         | "offset" Exp

OrderExp ::= ID | ID "asc" | ID "desc"

SetExp ::= ID "==" Exp | ID "!=" Exp
         | ID "<"  Exp | ID "<=" Exp
         | ID ">"  Exp | ID ">=" Exp
         | ID "in" Exp | ID "not" "in" Exp
         | SetExp "&&" SetExp
```

Figure 7. Query syntax

ject as well. The `Collection` type has methods for filtering, sorting, paginating, aggregating and manipulating the collection. For instance:

```
Task.all().filter("done", "=", true)
          .order("due", false)
          .limit(10)
```

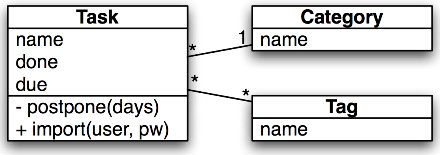


Figure 8. Todo list data model

This expression represents the top ten results of tasks that are not done, sorted by due date in descending order. The disadvantage of encoding queries as method calls is the lack of static checking of property names as well as its verbose syntax. Therefore, mobl defines a thin syntactic layer, similar to LINQ [14] on these methods as defined in Figure 7. This (optional) syntactic layer has the added advantage of enabling code completion support in the IDE. The same expression using the query syntax look as follows:

```
Task.all() where done == true
           order due desc
           limit 10
```

Full-text search queries are formulated using an entity's `search(phrase)` method, returning a `Collection` representing search results, ordered by relevance. Like any other `Collection`, results can be filtered and paginated.

These (virtual) query collections can be *reused* and *extended* by storing them in variables and passing them to functions. The result of a query is only calculated when required (e.g. when iterating over the result). Therefore, it is possible to define a method on an entity that returns a filtered collection (using `where` clauses), which is subsequently called and paginated in the user interface by adding `limit` and `offset` clauses to the method's resulting `Collection` object.

4.4 A Task Manager Data Model

To demonstrate the data modeling language, as well as model-related logic, we describe the data model of a simple task manager (Figure 8). The mobl implementation of this data model is listed in Figure 9. The data model defines three entities: `Task`, `Category` and `Tag`. A task has a name, a done property to keep track of whether the task has been completed or not, a due date, a category it belongs to and a collection of tasks. Implicitly there is a one-to-many relationship between `Task` and `Category`: a task belongs (points to) a category, and a category has many tasks. The `inverse` annotations define the inverse relationship so that a task is automatically added to a category's collection of tasks when the `category` property is set and vice versa.

The function `postpone`, defined on `Task`, postpones the task a number of days, i.e. it moves the `due` date back. The static function (a function that is called on the entity itself, not an instance) `import` takes two arguments: a username and a password, and invokes a web service (located

```
entity Task {
  name     : String (searchable)
  done     : Bool
  due      : DateTime
  category : Category (inverse: tasks)
  tags     : Collection<Tag>
               (inverse: tasks)

  function postpone(days : Num) {
    this.due = DateTime.create(
      this.due.getFullYear(),
      this.due.getMonth(),
      this.due.getDate() + days);
  }
  static function import(user : String,
                         pw : String) {
    var tasksJSON =
      httpRequest("/export?user="
                  + user + "&pw=" + pw);
    foreach(t in tasksJSON) {
      add(Task.fromSelectJSON(t);
    }
  }
}
entity Category {
  name  : String
  tasks : Collection<Task>
            (inverse: category)
}
entity Tag {
  name  : String
  tasks : Collection<Task> (inverse: tags)
}
```

Figure 9. Mobl implementation of data model

on the URI `/export`) to import all tasks defined on the server for the given user and cache them locally in the device's database. Web service results, by default, are returned as JSON[5] objects, a lightweight notation to represent structured data. The service returns an array of JSON objects, each representing a task. The `Task.fromSelectJSON` method is used to convert a JSON object into a `Task` object and cache it locally.

5. Reactive User Interfaces

In the current state of practice, web-based user interfaces are implemented at a low-level of abstraction. A lot of UI-related code is the result of *accidental complexity*. This section identifies the underlying problems and shows how mobl solves them by introducing *domain abstractions* such as data binding and reactive programming and by supporting *user-defined abstractions* such as controls.

Coupling View and Model The interaction between (persistent) application data and the user interface requires a lot of Controller *boilerplate code*. Data values have to be copied into the user interface when it is first loaded and stored back into data objects when certain events occur (e.g. when a "Save" button is pushed). Similarly, changes to data often give rise to changes in the user interface. For instance, when the user of a task manager application creates a new task in the database, the screen that displays all tasks has to be updated. Current frameworks require developers to encode this behavior manually.

Adapting the user interface is done by traversing the DOM and manipulating it in-place. These manipulations are imperative, e.g. "replace this node with this new node" and "remove this node".

Abstraction A HTML page defines the content and structure of a page. CSS styles are used to apply styling to a HTML page (e.g. defining fonts, colors, borders and positioning), based on the knowledge it has about the page structure (using CSS selectors). A feature that both HTML and CSS do not support are *user-defined abstractions*. Reuse of page and style fragments, e.g. to reuse a calendar widget or a grid view control, is not supported by these languages. It also lacks support to *define* such reusable components. Consequently, JavaScript frameworks, such as jQTouch[6] and jQuery Mobile[7] attempt to fix this reuse issue by inventing an encoding. For instance, a framework like jQuery mobile may reinterpret a HTML tag `<div class="calendar"/>` as a calendar control, dynamically adapting the DOM to implement it. Nevertheless, such mechanisms only allow *use* of controls built into the framework, while *definition* of new controls has to be done using imperative JavaScript. Other frameworks, such as GWT[8] and Sencha Touch[9] abstract from HTML altogether with a Java (GWT) or JavaScript (Sencha) API to imperatively construct user interfaces.

Neither of these approaches is perfect. Annotating HTML is declarative, but uses an arcane encoding and attaching new meaning to HTML elements; using JavaScript to build the UI is imperative and low-level.

5.1 Declarative User Interfaces

Mobl supports user-defined abstractions for user interfaces through two core syntactic constructs: `screens` and `controls`. Screens take up the entire size of the physical screen (hence the name) and are composed of controls, state variables, conditionals and loops. Both screens and controls have a name, a set of formal arguments and a body. Screens, in addition, have an optional return type. The full syntax of user interfaces in mobl is detailed in Figure 10.

The body of screens and controls consist of local variable declarations, HTML tags, control calls, conditionals (`when`,

[5] `http://json.org`

[6] `http://jqtouch.com`

[7] `http://www.jquerymobile.com`

[8] `http://code.google.com/webtoolkit/`

[9] `http://www.sencha.com/products/touch/`

```
Def ::= Anno* "control" ID "(" {FArg ","}*
          ")" "{" SE* "}"
      | Anno* "screen" ID "(" {FArg ","}*
          ")" ":" Type "{" SE* "}"

SE ::= "<" HTMLID HtmlArg* ">"
         SE*
       "</" HTMLID ">"
     | Exp "(" {NamedExp ","}* ")"
         "{" SE* "}"
     | "var" ID "=" Exp
     | "list" "(" ID "in" Exp ")"
         "{" SE* "}"
     | "when" "(" Exp ")" "{" SE* "}"

HtmlArg ::= ID "=" Exp
          | "body" "=" Exp

NamedExp ::= Exp
           | ID "=" Exp

Anno ::= "@when" Exp
```

Figure 10. User interface syntax

for conditionally rendering parts of the user interface) and loops (`list`, for rendering UI fragments for every item in a collection).

Local variables are used to store state relevant to the user interface. At the lowest level, mobl embeds HTML tags to construct a DOM. As can be seen from the syntax, HTML attributes in mobl cannot only contain strings, but arbitrary mobl expressions (numbers, variables, calculations, function calls).

Controls are *domain abstractions*, abstracting from low-level HTML. Controls are called by name with zero or more (optionally named) arguments and, optionally, a control body (in-between `{` and `}`). Screen and control arguments are passed by reference, enabling controls to write values back to the variables and properties passed to them, which is an essential element to enable *user-defined abstractions*, as will be demonstrated in the next sub-section.

5.2 Data Binding and Reactive Programming

Mobl user interfaces declare a View of the Model. *Data bindings* establish a direct connection between View and Model. The View is automatically updated when the Model is changed, and the Model is updated when Model properties are changed in the View.

The following fragment of user interface code demonstrates how this data binding works using HTML tags:

```
var name = "John"
<input type="text" value=name/>
<span body="Hello, "+name/>
```

A local variable `name` is used to keep track of the user's name. The `<input>` HTML tag implements an input field and binds its value to the variable `name`. Consequently, the input field displays "John" as initial value and *as the user types* in the text field (on every key stroke), the changed text box value is propagated back to the `name` variable.

The `<span>` tag implements a label in the user interface, whose *body* (the text that appears inside the label) is bound to the *expression* `"Hello, " + name`. Consequently, when the user types in the input field, the `name` is adapted, which, in turn, propagates to the `<span>` whose body is updated to reflect the new value of `name`.

Beside local variables and inline HTML, user interfaces use conditionals and loops which expose similar *reactive behavior*. The `when` construct conditionally shows a part of the user interface as long as a certain condition holds, i.e. when the condition's value changes the `when` construct adapts the user interface accordingly.

As an example, in the following example the validation error remains hidden while the length of `name` exceeds 3 characters in length and appears as soon as the name is shorter than 3 characters:

```
var name = "John"
<input type="text" value=name/>
when(name.length < 3) {
  <span body="Name should be at least three
                characters"/>
}
```

The `list` construct iterates over a collection and renders its body for every item in the collection. Similar to `when`, `list` automatically adapts to changes in application state; it reacts to changes in the collection it iterates over, i.e. if items are added or removed from the collection, it adapts the user interface accordingly.

In summary, rather than imperatively manipulating the DOM to make changes to the user interface, mobl's user interfaces are *reactive* [7] – their structure and content depend on application state and adapts to changes automatically. As a result *boiler plate* code to implement this behavior manually is eliminated.

5.3 Implementation

Mobl's *data binding* establishes a direct connection between the value or an attribute of an HTML node in the DOM (representing an HTML tag) and a variable, property or expression in the Model. For variables and properties, a two-way binding is established: when the DOM is modified (for instance, when a user edits a text input field), this new value is propagated back into the variable or property. When the value of a variable or property changes this change is propagated back to the DOM. When a DOM node is bound to a more complex mobl expression, a one-way connection is established: whenever the value of the expression changes, it is propagated to the DOM.

Changes are propagated by using the Observer Pattern [6]. Any piece of data in a mobl application is observable (including local variables, control arguments and entity properties) and UI constructs subscribe to change events of the observable values that they rely on. For instance, a label that shows a user's full name by concatenating its `firstName` with its `lastName` property, will subscribe to both of these properties and rerender itself whenever any of these two properties trigger a 'change' event.

Similarly, a `list` loop that iterates over a search collection, as is the case in Figure 13, subscribes to changes in the search collection. The search collection, in turn, keeps track of all `Task` objects and their `name` properties (which has been marked as `searchable`) and on every change, reevaluate if they match the search phrase or not.

5.4 Reusable Controls

Rather than requiring the duplication of the same HTML code in multiple places, *controls* can be used to implement *user-defined abstractions* for user interfaces. Control arguments are passed by reference, enabling constructing controls that abstract from low-level HTML while maintaining data binding semantics. Figure 11 demonstrates the implementation of the `textField` and `label` controls. Using these definitions the previous code, using HTML tags, can be reduced to the following, more clean and concise code, maintaining the same behavior. Section 7 gives more complex examples of control implementations.

```
var name = "John"
textField(name)
label("Hello, " + name)
```

Control arguments, as well as function and screen arguments, are passed in order, or can be explicitly named. For instance, `label("Hello")` is equivalent to `label(s="Hello")`. This proves particularly useful for optional arguments.

By annotating controls with a `@when <exp>` annotation it is possible to implement multiple versions of a control, deciding at run-time which implemention to use, based on a run-time condition. An application of this will be demonstrated in Section 7.2.

6. Navigation

Section 5 only considered the definition of single-screen interfaces. However, a typical application requires multiple screens and navigation between them.

```
control textField(s : String) {
  <input type="text" value=s/>
}
control label(s : String) {
  <span body=s/>
}
```

Figure 11. Text field and label control implementation

```
screen prompt(question : String) : String {
  var answer = ""
  header(question) {
    button("Done", onclick={
      screen return answer;
    })
  }
  textField(answer)
}

screen root() {
  button("Ask", onclick={
    alert("Hello " + prompt("First name")
          + " " + prompt("Last name"));
  })
}
```

Figure 12. A screen with return type

The 'regular' web is navigated by clicking hyperlinks, sending the user from one web page to another. Browsing patterns can be random, and websites are not always organized in a strictly hierarchical manner. We observe that in mobile applications, navigation patterns are more stringent. Data-driven mobile applications typically organize information as trees. Some applications present the top-level of the tree as *tabs*, enabling the user to quickly switch between them. Deeper levels of information are presented in *list views*. When the user selects a list item, the current screen slides to the left, and a new one slides in from the right. Navigation between screens usually happens by navigating deeper into the hierarchy or moving back to a higher level (using the *back* button).

On iPhones and iPads, navigation is implemented using a stack of screens where only the top of the stack is visible. When an item is selected, a new screen, representing the item is pushed onto the stack and when the user pushes the back button, the screen at the top is popped off the stack and the previous screen appears. This screen stack has to be managed *manually* by the developer, by pushing and popping screens.

6.1 Multiple screens

This stack-based navigation very closely matches the call stack of function invocations in programming languages, a concept familiar to any programmer. Therefore, in mobl, screens are called as if they were functions and can optionally return a value using `screen return`.

As an example, Figure 12 defines a `prompt` screen, which takes a question as argument and returns the answer as result. The `textField` is bound to a local `answer` variable, which is returned by the screen when the "Done" button is clicked. The `root` screen contains a button, which, when clicked, invokes the `prompt` screen twice: first asking for the first name, then for the last name, then showing

```
screen root() {
  var phrase = ""
  header("Tasks") {
    button("Add", onclick={ addTask(); })
  }
  searchBox(phrase)
  group {
    list(t in Task.search(phrase) limit 20){
      item {
        checkBox(t.done, label=t.name)
      }
    }
  }
}
screen addTask() {
  var t = Task()
  header("Add") {
    button("Done", onclick={
      add(t);
      screen return;
    })
  }
  textField(t.name)
  datePicker(t.due)
}
```

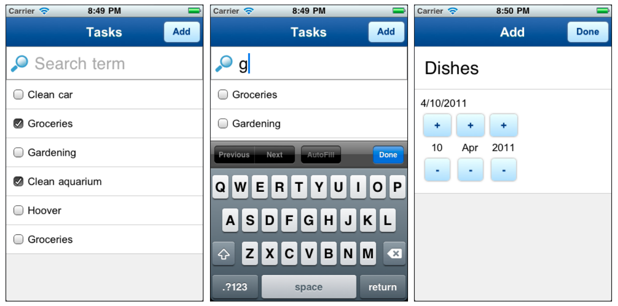


Figure 13. Tasks root screen with search

an alert pop-up box producing a greeting concatenating the results from the two `prompt` screen calls.

6.2 A Task Manager User Interface

Section 4.4 demonstrated how to define a data model for a task management application. Figure 13 shows how to build a simple user interface for this data model. When a mobl application launches, the `root` screen is loaded. Figure 13 defines the main screen of the task manager application. The screen shows a header, a search box and a group of at most 20 tasks that match the search phrase. Each task has a check box that can be used to mark the task as done.

The user interface is realized using a local user interface variable `phrase` to keep track of the search phrase. The search box is *bound* to this variable. The body of the `group` control contains a `list` construct that iterates over the search collection representing all tasks that match `phrase` with a maximum of 20 results. The body of the `list` construct instantiates an item control for every task, containing a `checkBox` which is bound to the `done` property of the task, as well as using the `name` property of the task for the checkbox label.

As the user types a search phrase in the search box, the changed search phrase is written back to the `phrase` variable. The `list` construct iterates over a collection that relies on the `phrase` variable. Therefore, it is recalculated as well. As a result, the list of tasks updates as the user is typing in the search phrase. Whenever new tasks are added to the database that match the search phrase, the task list will automatically be updated as well.

When the "Add" button is pushed, the `addTask` screen activates. The `addTask` screen uses a local variable `t` to keep a new task object whose `name` property is bound to a `textField` control and whose `due` property is bound to a `datePicker` control. The `button` control takes two arguments: a label to put on the button, as well as a named argument `onclick` of type `Callback`. Callbacks are snippets of imperative code, written using the same language as described in Section 4.2, to be executed when a certain event occurs (in this case an on click event). These snippets can be defined in-line in between `{` and `}`. When the user is done editting the name, he pushes the "Done" button, which adds the `t` object to the database and returns the user to the previous screen using a *screen return*.

7. Higher-Order Controls

Mobl comes with a extensive library of reusable controls. These controls have been implemented in mobl itself, concisely defined using its abstraction, data binding and reactive programming features. Section 5 demonstrated how simple controls such as `textField` and `label` can be implemented top of HTML with data binding. This section will describe how higher-level controls are implemented. Specifically, the `tabSet` and `masterDetail` controls are described. The `tabSet` control is a higher-order control, taking other controls as arguments. The `masterDetail` takes control arguments as well, but in addition has two separate implementations: the 'right' implementation is chosen at run-time based on the screen width.

To support higher-order controls, mobl has a set of types to represent controls as values: `Control`, represents a control without arguments. Similarly, `Control1<Num>` represents a control with one argument, of type `Num`. Control arguments are passed as arguments are instantiated as any other control.

7.1 Tab Set

Figure 14 demonstrates how the `tabSet` control is used. It defines two controls: one for each tab. The `root` screen invokes the `tabSet` control with a list of tuples where each tuple represents a single tab. The first element of the tuple is the tab title (of type `String`), the second a reference to the control to use for the body of the tab (of type `Control`,

```
control tab1() {
  header("Tab 1")
  label("This is tab 1")
}
control tab2() {
  header("Tab 2")
  label("This is tab 2")
}
screen root() {
  tabSet([("One", tab1), ("Two", tab2)],
         defaultTab="One")
}
```

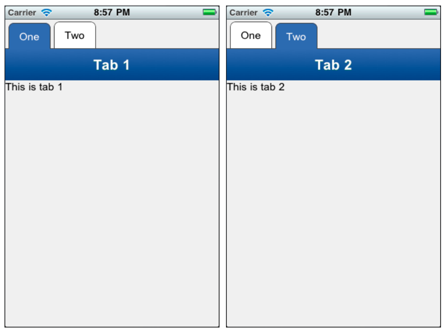


Figure 14. Using `tabSet`

```
control tabSet(tabs : [(String,Control)],
               activeTab : String) {
  list((tabName, tabControl) in tabs) {
    block(onclick={ activeTab = tabName; },
          style=activeTab==tabName ?
                activeTabButton
                : inactiveTabButton) {
      label(tabName)
    }
  }
  list((tabName, tabControl) in tabs) {
    block(activeTab==tabName ?
            visibleTab : invisibleTab) {
      tabControl()
    }
  }
}
```

Figure 15. `tabSet` implementation

a control without arguments). The `defaultTab` argument specifies the title of the tab to activate first. The screenshots in Figure 14 show the result: a tab bar along the top and when a tab is selected, the tab view changes to the selected tab's body.

Implementation Figure 15 details the entire implementation of the `tabSet` control. It takes two arguments: an array of tuples and the currently active tab. The tab set implementation relies on a few styles (styling in mobl will be discussed in Section 8) that are used with `block`

```
control taskItem(t : Task) {
  checkBox(t.done, label=t.name)
}
control taskDetail(t : Task) {
  textField(t.name)
  datePicker(t.due)
}
screen root() {
  header("Tasks")
  masterDetail(Task.all() order by due desc,
               taskItem, taskDetail)
}
```

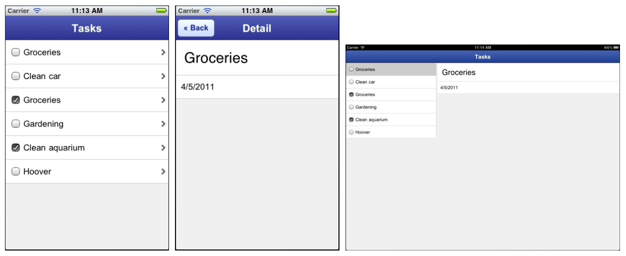


Figure 16. Using the `masterDetail` control

controls. The `block` control is a simple stylable container control. An `activeTabButton` block appears as a selected tab, with rounded borders at the top. An `inactiveTabButton` block is similar, but looks like an inactive tab. The `visibleTab` and `invisibleTab` block respectively are visible and invisible. Thus, when a tab is not selected, its control is still rendered, it is just hidden using styling.

The `activeTab` argument keeps track of the currently selected tab. When a tab is selected, the `activeTab` variable is changed. Consequently, due to the reactive semantics, the styles on the tabs are toggled (tab content gets visible style, tab button gets selected style) and the new tab appears.

The `list((tabName, tabControl) in tabs) { ... }` notation uses tuple syntax on the left-hand side. It binds the first value of each tuple in `tabs` to `tabName` and the second to `tabControl`.

7.2 Master-detail

A common pattern in mobile user interfaces is the master-detail user interface pattern. There are two common implementations of this pattern, based on the available screen estate: On mobile devices with narrow screens, such as phones, initially a list of items appears and after selecting one, its details appear on a separate screen containing a back button to navigate back to the list. On devices with wider displays, such as tablet devices, the list of items appears along the left side of the screen and details of the selected items appear along the right.

Figure 16 shows how the `masterDetail` is used and how it looks on a narrow-screen device (first two screenshots) and on a wide screen (third screenshot). Two controls are defined: `taskItem` is used in the list view and

```
control masterDetail(items : Collection<?>,
    masterItem : Control1<?>,
    detail : Control1<?>) {
  group {
    list(it in items) {
      item(onclick={
        detailScreen(it,detail);
      }) {
        masterItem(it)
      }
    }
  }
}
screen detailScreen(it : ?,
                    detail : Control1<?>) {
  header("Detail") {
    backButton()
  }
  detail(it)
}
```

Figure 17. `masterDetail` implementation

```
@when window.innerWidth > 500
control masterDetail(items : Collection<?>,
      masterItem : Control1<?>,
      detail : Control1<?>) {
  var current = items.one()
  block(sideBarStyle) {
    group {
      list(it in items) {
        item(style=current == it ?
               selectedItemStyle
               : notSelectedItemStyle,
             onclick={ current = it; }) {
          masterItem(it)
        }
      }
    }
  }
  block(mainContentStyle){
    detail(current)
  }
}
```

Figure 18. A wide-screen `masterDetail`

`taskDetail` in the detail view. The `root` screen calls the master detail control with a collection representing all tasks ordered by due date in descending order, the `taskItem` and `taskDetail` controls.

Implementation Figure 17 shows the default implementation of the `masterDetail` control (used for devices with a narrow screen). It takes three arguments: a collection of any type (`?` is syntactic sugar for the `Dynamic` type, representing dynamically typed values), a `masterItem` control that is used for the list view and a `detail` control that is used to show the details of the item. Both the `masterItem` and `detail` are controls that take an item from the `items` collection as argument.

The control iterates over each item and renders an `item` control for it, using the `masterItem` control to render the content of the item. When the item is clicked, the `detailScreen` is called with both the item and the detail control as arguments. The implementation of the `detailScreen` renders a header control with a backButton, that, when click returns the user to the previous screen. It calls the `detail` control, passed as an argument with the `it` argument to render the detail view.

Figure 18 shows an alternative implementation of the `masterDetail` control that is only used when the browser window's inner width is larger than 500 pixels (expressed using the `@when` annotation), i.e. on wider screens. The arguments match exactly with the previous implementation, but the control body differs. A local variable `current` is used to keep track of the currently selected item in the collection. It is initialized to the first item in the collection (the `.one()` method limits the collection to a single item, returning the first one). The `sideBarStyle` is used to show a block to the left of the screen containing the list of `items`. The style (color) used for the item depends on whether it is selected or not. When the item is clicked, it is assigned as the `current` item. The block styled with the `mainContentStyle` appears right of the list and uses the `detail` control to render the currently selected item's details. The item rendered by the `detail` control automatically updates as new values are assigned to `current`.

8. Styling

Cascading Stylesheets (CSS) are used to define the look and feel of a mobile web application. Styles are attached to HTML either automatically (using CSS selectors) or explicitly by attaching `class` attributes to HTML tags. Nevertheless, stylesheets are source of *code duplication* due to its lack of support for parameterization.

For instance, the following style can be attached to an HTML element to implement rounded corners. Due to the current state of browser support for the `border-radius` (a CSS3 feature), it uses browser-specific properties for Webkit and Gecko-based browsers (two common rendering engines) to make it work on all browsers:

```
.rounded-corners {
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
```

However, whenever rounded corners are required with a radius other than 5 pixels, these three lines have to be duplicated and adapted.

```
Def ::= "style" ID "{" StyleProp* "}"
      | "style" "mixin" ID
        "(" {StyleFarg ","}* ")"
        "{" StyleProp* "}"
      | "style" "$" ID "=" StyleVal

StyleProp ::= ID "=" StyleVal* ";"
            | ID "(" {StyleVal ","}* ")" ";"

StyleVal ::= CSSSTYLEVALUE
           | "$" ID
           | "$" ID "." "r"
           | "$" ID "." "g"
           | "$" ID "." "b"
           | StyleVal "+" StyleVal
           | StyleVal "*" StyleVal
           | StyleVal "-" StyleVal
```

Figure 19. Styling language syntax

8.1 Styling in Mobl

In order not to reinvent the wheel, mobl's styling language reuses all of CSS3's styling properties [23]. In addition, it adds styling constants, calculations based on these constants and style mixins. These additions were inspired by Sass[10], an extension of CSS that adds similar features.

Figure 19 defines the syntax for styles in mobl. At the HTML level, style values are attached to the `class` attribute of tags. Typically, controls have a `style` argument (of type `Style`) that is used to pass styles around. For instance, the `block` control:

```
style largeStyle {
  font-size: 100pt;
}
screen root() {
  block(largeStyle) { label("Large text") }
}
```

8.2 Theming

Applications can easily be themed with custom colors by overriding style constants used by the standard mobl library of controls. For this purpose, mobl supports global style constants that can be referenced in styles. When using RGB (Red-Green-Blue) colors, the individual color components can be accessed to build new colors:

```
style $baseColor = rgb(72, 100, 180)
style $textButtonColor = rgb($baseColor.r-50,
                             $baseColor.g-50,
                             $baseColor.b-50)

style buttonStyle {
  color: $buttonTextColor;
  ...
}
```

[10] `http://sass-lang.com`

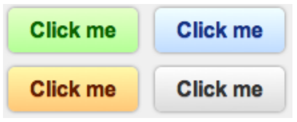


Figure 20. Theme derivation

```
style mixin borderRadiusMixin($radius) {
  -moz-border-radius:    $radius;
  -webkit-border-radius: $radius;
  border-radius:         $radius;
}
style buttonStyle {
  color: $buttonTextColor;
  borderRadiusMixin(5px);
  ...
}
```

Figure 21. Style mixin example

The controls that come with mobl all derive their colors from the `$baseColor` constant. Therefore, simply overriding this constant and changing it to a different color, creates a new color theme based on the given base color. Figure 20 shows how buttons change with different `$baseColor` settings.

Parameterized styles are implemented using *style mixins*. Style mixins can be used and parameterized in other styles. Figure 21 implements a parameterized version of a border radius style taking the border *radius* as argument. The `buttonStyle` uses the mixin to realize a border radius of 5 pixels. Mobl comes with a library of reusable style mixins, including border radius and gradient mixins.

9. Discussion

To evaluate the coverage of mobl we have built a number of applications using mobl, ranging from simple toy applications such as a todo list manager and a tip calculator to more complex applications such as a twitter client, a conference planner application and even simple graphical games and a collaborative drawing application. Mobl receives a lot of interest from industry. Several companies are working on mobile applications built using mobl. Together with our user community we grew a library of reusable controls, ranging from basic, such as labels and buttons, to more complex, such as the tab set, a master-detail, accordion, date picker and context menu controls. The definitions of these controls are all declarative and concise. A member of the community has also developed a framework (using mobl) to enable unit testing of the data model and logic.

This section discusses the limitations of our approach and compares it to related work.

Language Limitations While mobl's type checker checks many program properties, it does not yet check everything. For instance, `item` controls controls have to be nested within `groups` to be rendered properly. Mobl does not yet support declaring such nesting requirements.

Data synchronization with web services currently has to be implemented manually. In the future we intend to support transparent data synchronization web services as part of the mobl language, thereby eliminating the custom synchronization code that need to be written on an application-by-application basis.

Mobile web applications generated by mobl are portable to any mobile platform that supports HTML 5. However, the user interface does not adapt to the look-and-feel of the platform, while mobl *supports* this variability using `@when` annotations, we have not yet developed many platform-specific control implementations.

Performance The performance of mobile *web* applications will always be worse than *native* applications, just as web applications in general are slower than native desktop applications. Nevertheless, by caching both the application and its data locally and the recent performance improvements of (mobile) browsers, performance of mobile web applications is very reasonable. While performance has not been the primary focus of the mobl compiler thus far, it is possible to produce an optimized build which eliminates all unused definitions from the generated JavaScript and CSS files. In addition, unnecessary whitespace is removed and variables are renamed with shorter names to considerably reduce the application's download size.

Good Web Citizenship While mobl uses the web as a medium to deliver applications, and uses web technologies to run applications, a mobl application is not built like a regular web application: a mobl application does not consist of pages with unique URLs; breaks the browser's back button; and is not indexable by search engines. We intend to solve some of these issues. A working back button is relatively easy to implement. Full history support is much more complex, requiring some type of encoding of the application state in the URL of the application. Indexing mobile applications can be useful for some data-driven applications. A tool such as CrawlJax [15] could be used to generate a static, indexable version of the application.

Web Application Limitations While web applications have the advantage of being portable, they have limitations too. HTML5 offers many JavaScript APIs that give access to various device services, but their implementation in mobile devices is not always complete. Access to audio and video services is limited — it is possible to play an audio or video file, but only by launching the dedicated audio or video player. Access to other device-specific features such as bluetooth, the built-in compass, camera and local file storage are not supported yet.

A way around these restrictions is a native/web hybrid approach. PhoneGap[11] allows a developer to build applications using web technologies, and expose additional native APIs including a file storage API and a camera API through JavaScript, an approach that works nicely with mobl. Applications built with PhoneGap can be deployed as native applications through e.g. the Apple AppStore or Android Marketplace.

[11] `http://www.phonegap.com`

Web applications have limitations in user experience as well. It is very difficult to reproduce certain native application behaviors in web applications. Inertia scrolling is one such behavior, where, after a finger flick on the screen, the screen keeps scrolling for a while longer after the finger no longer touches the screen. There are a number of projects that attempt to emulate this behavior in the browser, but it has proven very difficult to do perfectly. Fixed positioning is another behavior that is difficult to achieve in mobile browsers. A control that has a fixed position, does not move when the rest of the screen scrolls. A typical example is a screen header. A header is positioned at the top of the screen and while the rest of the content scrolls, the header remains fixed at the top.

9.1 WebDSL

In previous work we developed WebDSL [22], a domain-specific language for the development of RESTful web applications. From a WebDSL program, the WebDSL compiler generates a Java web application, deployable in any Java servlet container.

Mobl borrows many concepts from WebDSL. For instance, like WebDSL, mobl is statically verifiable [8] and has similar constructs for the definition of data models.

State and Event Handling Syntactically, the definition of WebDSL and mobl user interfaces are similar, but their semantics differ when it comes to the time of data binding. The unit of interaction within a WebDSL application is a HTTP request, either executed using an AJAX call, a form submit or page request. Pages are reconstructed on every request, instead of incrementally updated as is the case in mobl. Incremental user interface updating is cheap when maintaining state locally, while implementing incremental updates efficiently in a client-server application requires *application state* to be maintained on the server as well as client, which would require the storage of application state for potentially thousands or millions of users.

Handling of events in mobl is more fine-grained than in WebDSL: when editing a data object in WebDSL, changes are persisted only when the edit form is submitted to the server, rather than instantaneously as is the case with mobl applications. Since all interaction and persistence happens locally, such continuous persistence is much cheaper to implement. Sending every keystroke to the server would be very expensive.

Extension Mobl has a different philosophy than WebDSL when it comes to language extension. WebDSL developed many abstractions as built-ins, including built-in types, controls and functions. As a result, any modifications or improvements to these constructs requires extension or adaptation of the compiler. Mobl takes the approach of library extension. Rather than hard-coding types and controls into the compiler, they are defined in libraries either encoded in mobl itself, or through the native Javascript interface. The advantage of this approach is that users can easily add new functionality to mobl, without the need to know how its compiler works. This approach is currently in process of being adopted in WebDSL as well.

9.2 Related work

DSLs for mobile development Behrens [2] describes a domain-specific language for creating *native* mobile applications, using a single language from which both iPhone and Android applications can be generated. Similar to mobl, the language comes with an IDE plug-in for Eclipse that supports error high-lighting, code completion and reference resolving. Berhens' language has a number of high-level controls built into the language, including sections, detail views and cells. It can fetch its data from data providers. However, the DSL currently only supports data *viewing* and is not as flexible as mobl; defining custom controls is not supported, for instance.

Kejriwal and Bedekar developed MobiDSL [10], an XML-based language for developing mobile web applications. Unlike mobl, the application is executed on the server and plain HTML is sent to the mobile device. MobiDSL comes with a number of built-in controls, such as query views, page headers and search requests that can be used to build pages. It is not possible to define custom controls, nor is there specific IDE support available.

Google Web Toolkit is a tool that enables client-side web applications using Java. The use of Java has the advantage of having excellent IDE support. A GWT plug-in[12] enables access to HTML 5 APIs such as geolocation and local databases. Like mobl, GWT applications are compiled to a combination of HTML, Javascript and CSS. However, user interfaces using GWT have to be defined using verbose Java code. In addition, GWT does not provide data binding or reactive programming support, therefore requiring a lot of boiler plate code to bind the Model to the View.

Reactive User Interfaces Courtney and Elliot developed Fruit [3], a Haskell framework that applies functional reactive programming [17, 5, 24] to user interfaces. It is based on signals (streams of events) and signal transformers (functions that transform streams of events). On top of these concepts, Fruit builds a purely functional user interface library. Mobl's user interfaces are also reactive, but not based on pure functions. Concepts such as signals and signal transformers are not exposed to the developer in mobl. Instead, events triggered by changes in data or control events, result in updates to the user interface.

[12] `http://code.google.com/p/gwt-mobile-webkit/`

Meyerovich et al. describe FlapJax [16], a language for building AJAX applications. Flapjax is also built on the concept of event streams: streams of events that model, for instance, mouse movements, clicks and web service responses. These streams can be filtered and merged to build responsive user interfaces. Mobl takes a more traditional approach to event handling. Events in mobl trigger event handler logic, which can modify application state potentially resulting in user interface changes.

10. Conclusion

In this paper we introduced mobl, a new language for developing mobile web applications. Mobl *linguistically integrates* languages for data model definition, user interface, styling and application logic. It introduces *domain abstractions* to abstract from accidental complexity and irrelevant details of the platform and domain. Mobl's support for *user-defined abstractions*, data binding and reactive program enable the *reusable* implementation of both simple controls (labels and button) and higher-level controls (tab sets and master-detail). Mobl automates the tasks typically manually encoded in Controller logic, thereby reducing the amount of boiler plate code that needs to be written. Mobl has received a lot of interest from industry. A number of companies have already committed to implement their mobile applications using mobl.

11. Acknowledgments

This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*. We would like to thank Google for providing Android phones for testing and development.

References

[1] D. K. Barry and T. Stanienda. Solving the Java object storage problem. *computer*, 31(11):33–40, 1998.

[2] H. Behrens. MDSD for the iPhone. In *SPLASH '10: Proceedings of Object oriented programming systems languages and applications companion*, 2010.

[3] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *PLI*, 2001.

[4] ECMA. ECMA-262 ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`, December 2009.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[7] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, 1985.

[8] Z. Hemel, D. M. Groenewegen, L. C. L. Kats, and E. Visser. Static consistency checking of web applications with WebDSL. *JSC*, 46(2):150–182, 2011.

[9] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010.

[10] A. A. Kejriwal and M. Bedekar. MobiDSL - a domain specific langauge for mobile web applications: developing applications for mobile platform without web programming. In *Proceedings of the 9th OOPSLA Workshop on Domain Specific Modelling (DSM'09)*, October 2009.

[11] J. Kim, R. A. Baratto, and J. Nieh. pthinc: a thin-client architecture for mobile wireless web. In *WWW*, pages 143–152, 2006.

[12] A. M. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya. Improving web browsing performance on wireless pdas using thin-client computing. In *WWW*, pages 143–154, 2004.

[13] J. B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[14] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and xml in the .net framework. In *sigmod*, page 706, 2006.

[15] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE*, pages 122–134, 2008.

[16] L. A. Meyerovich, A. Guha, J. P. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA*, pages 1–20, 2009.

[17] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002.

[18] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *TCS*, 1(2):125–159, 1975.

[19] M. Potel. MVP: Model-View-Presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[20] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[21] J. Smith. WPF Apps With The Model-View-ViewModel Design Pattern. `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`, February 2009.

[22] E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007.

[23] W3C. CSS 3 working draft. `http://www.w3.org/TR/css3-roadmap`, 2011.

[24] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI*, pages 242–252, 2000.