Under preparation for OOPSLA'14

Declaration-Driven Frameworks: A Language-Agnostic Approach

Paul van der Walt Charles Consel Emilie Balland
Inria & University of Bordeaux I, France
first.last@inria.fr

Abstract

Programming frameworks are an accepted fixture in the object-oriented world, motivated by the need for code reuse, application developer guidance and restriction. A new trend is emerging where frameworks supporting open platform offer domain-specific declarations to address concerns such as privacy. These declarations are used to drive the behaviour of the resulting application. Although most popular platforms (*e.g.,* Android) are based on declaration-driven frameworks, they provide ad hoc and narrow solutions to concerns raised by their openness. In particular, existing programming frameworks are approached from an implementation viewpoint, in that they are specific to a particular programming language and domain.

In this paper, we show that declaration-driven frameworks can provide programming support, constraints, and guarantees in a wide spectrum of programming languages. To do so, we identify concepts that underlie declaration-driven frameworks and apply them uniformly to both an object-oriented language, Java, as well as a dynamically typed functional language, Racket. From this case study, we propose principles for developing declaration-driven frameworks that apply across programming languages, covering a spectrum of domains.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—domain-specific architectures, languages, patterns

General Terms Design, Languages, Verification

Keywords Generative programming, programming frameworks, functional programming, object-oriented programming, architectural conformance, domain-specific languages

1. Introduction

Software reuse is agreed to be a goal in itself, for keeping applications maintainable, facilitating the development process, and avoiding repetition [19]. To this end, software libraries have long met a need in software engineering. Concerns such as ease for the application developer, not wanting to be bothered with managing an application's life cycle, and preventing developers from deviating from the chosen architectural style, have prompted the development of programming frameworks. Frameworks turn the tables on libraries: instead of a developer writing a whole application from scratch and calling routines provided by a library, a framework takes over and manages the control flow, calling the pieces a developer has provided.

Their advantages typically include (1) reducing development effort of applications (through *guidance* of the developer) in whatever domain they are intended for, (2) enforcing a particular architectural style (*restriction*) [22], and (3) fulfilling the needs previously met by libraries, that is, providing easy access to common software artefacts.

More precisely, programming frameworks are defined in Fayad *et al.* [7] as a software engineering technique employing *inversion of control*[1], for creating applications through extension. Contrary to the use of libraries, they can be seen as a technique to arrive at a final application by filling holes with snippets of code (*e.g.,* the framework provides placeholders which may be filled in with the desired behaviour).

Application frameworks are a well-established and widespread tool in the object oriented ecosystem [14], where they are seen to be an effective way of addressing the concerns involved in application development. Frameworks are found everywhere from GUI applications, to web, to mobile computing. We see a trend emerging, where frameworks are ever more frequently making use of some domain-specific declarations as input [3, 20, 21], which, apart from describing, are used to influence the shape and behaviour of the final application. These declarations also allow the framework to better answer emerging challenges such as privacy concerns,

[1] Inversion of control is known informally as the Hollywood Principle: "don't call us, we'll call you". In software engineering, this refers to the situation where the life cycle of the application is not the responsibility of the application developer – they just provide various components which are then used as needed by the run-time system.

providing support to the developer, and giving a user insight into how their sensitive information is being used. In this work we will exclusively focus on these *declaration-driven frameworks*.

Recently we are seeing an explosion of new application domains making use of these modern frameworks, such as smartphones, where the open platform model is being used. When we refer to open platforms, we mean systems with (1) *public programming interfaces*; (2) providing *shared resources* to the applications; and (3) a *run-time environment* for applications. Examples include Android [20] and iOS [4], but also the Facebook platform [8], among many others. Because it is an attractive business model to offer a platform for which 3rd party developers can easily write applications for end users to install, the open platform model is being widely adopted.

These novel application domains pose new challenges. For example, they expose shared resources which are sensitive for the end user (*e.g.,* camera, microphone), making them potentially accessible by 3rd party developers. Secondly, platform providers want to encourage adoption, so are interested in facilitating the development process as much as possible. It is in the domain of open platforms that declaration-driven frameworks are most prevalent and successful, because of the reduced development effort, as well as their greater potential for control from the point of view of the platform, and the insight they can provide to the end-user.

Among declaration-driven frameworks, we identify a spectrum of different approaches to dealing with these declarations and their resulting restrictions: from fully dynamic (as in Android) to static (as in DiaSuite [3]). This spectrum raises questions for us, such as what are the advantages of each approach, and what does such a design choice mean for the framework implementation? Can the choice of static/-dynamic semantics for declarations be made independently from the choice of host language, or do we need certain features in a programming language, such as a strong static type system to be able to enforce the requirements?

Our research questions are therefore, "how can the concepts of programmer guidance and property guarantees arising from declarations be mapped into programming features? What influence does the implementation language have on this mapping?" We will explore a concrete case study of a declaration-driven framework in support of an open platform, in two widely differing programming paradigms. We are interested in identifying the requirements arising from the declarations and in the mapping from requirements into programming language features. This mapping will also be validated by looking at existing frameworks. To make the case study as meaningful as possible, we choose a platform with a highly expressive declaration language, to maximally exercise the capabilities of the underlying implementation language.

Contributions. Our work takes place in the context of open platforms supported by declaration-driven programming frameworks. Our contributions are (1) identifying the language-agnostic minimal requirements to be able to implement the checks and guarantees open platforms call for, (2) showing that open platform requirements we identify map into a wide spectrum of programming languages, *e.g.,* statically or dynamically typed, object oriented or functional, (3) implementing such a framework in two very different programming languages and finally, (4) synthesising general principles from our case study to guide future implementations in a potentially wide spectrum of programming languages.

2. Declaration-driven programming frameworks

In this section, we will explore requirements for a declaration-driven programming framework intended to support the effectiveness of open platforms. We will validate these requirements by comparing them to various existing frameworks, which exist in the open platforms domain. We also highlight their declaration languages' varying expressiveness. Afterwards, we will present a case study that will demonstrate the benefits of these modern frameworks. This case study is based on an existing declaration-driven framework, DiaSuite [3] (introduced in Sec. 2.2).

First, we identify the stakeholders and their requirements in the context of open platforms (Sec. 2.1). Next, we introduce core DiaSpec, which will be the vehicle of our experiments (Sec. 2.2). Finally, we instantiate these requirements for the case study (Sec. 2.3), which will be implemented as a programming framework in Sec. 3.

2.1 Requirements of open platforms

When considering declaration-driven frameworks supporting open platforms, we observe that the different stakeholders have various concerns:

- the *end user* would like clear insight into resource usage (*which* resources are requested, *how* they are used),
- the *platform provider* wants to facilitate the development process as much as possible, to encourage adoption of the platform,
- the *3rd party developer* is interested in high-level programming support, and abstraction from platform details (leading to greater portability thanks to *e.g.,* hardware agnostic implementations).

These concerns can be expressed more precisely as the following requirements.

Req1 The user would like clarity on *which* shared resources are to be used by the application. *Resource declarations* should therefore specify the sources and sinks of potentially sensitive data and side-effects, and whether a given application needs them. The set of possible resources is pre-defined by the platform provider, as in the case of An-

droid (with permissions such as taking a picture with the camera, or sending an email). The developer should list the resources desired by the application, thus allowing the user to make an informed decision whether to install the application.

Req2 The *data flow* should be constrained to maintain the safety of sensitive information. Potential leaks might be predicted, by clarifying whether a control flow path exists between components, which have access to various sensitive resources. If this is done in a coarse-grained manner such as in Android, the user might need more information to know if an application is risky. For example, the application may have access to both the internet and photos, meaning photos could be leaked. Using fine-grained declarations provides a user with more insight into how sensitive data is used. For example, one might define an application in which one component has access to a sensitive source, and another component to a public sink, but where there is no control flow path between A and B. This property is called *data reachability*, as defined in [2].

Req3 Tailored *programming support* for the developer can and should be derived from the declarations, since they give a form of specification for the desired behaviour of the application. For example, if the declarations prohibit a certain resource from being used, its API should not be available to the developer. This avoids confusion and clutter during implementation. Also, run-time exceptions can be avoided, if illegal calls are not available to the developer.

Req4 *Verification* of the conformance of the implementation to the declarations should be performed. If a developer can compile the implementation, it should be guaranteed to conform to the declarations. This way a user knows the declarations can be trusted as being meaningful for the application.

By comparing this list of requirements to the declaration languages provided by some of the most prevalent and successful frameworks, we can validate that indeed these are the emergent requirements of real-world stakeholders. We will show that for each requirement, declaration languages vary widely in expressiveness and precision.

On one end of the spectrum, the simplest declaration language for an open platform might cover only resource usage requirements (**Req1**). For example, the Facebook [8] declaration language allows the developer to specify the resources an application requires globally (*e.g.,* access to remote websites, or the "friend list", or the user's birth date). When installing an application, the user can assess this information to decide whether or not to install it.

The Android declaration language [20] goes further by enforcing an architectural style dedicated to mobile applications – that of different views and communication channels between them called *intents*. The interactions between these components are controlled by the framework (*i.e.,* broadcast receivers are only triggered by system or application events). The underlying declaration language (seen in the Manifest files) provides constructs dedicated to this architectural style, forcing the developer to declare the application components and the resources the app is authorised to use. Having resource declarations in combination with structural declarations potentially allows more fine-grained control over sensitive information. If not only the resources but also the components are declared, the possible paths of information flow can be approximated. Unfortunately, though, in Android the declarations related to data flow are not fully leveraged, and the resource declarations are global to the application. This means that based only on the declarations, they leave a misbehaving application indistinguishable from a reasonable one. For example, if all that is known about an application is that it is allowed to send emails and access personal data, it is unknown what it will send where. On the other hand, if declarations are fine-grained (*e.g.,* only read emails or only send emails, and restricted to certain views) a user might see that an application can only send emails on their command, as opposed to exfiltrating sensitive data in the background. Unfortunately, this is not paranoia, but a real risk, since sensitive data is regularly exfiltrated in real-world Android applications [24].

On the other end of the spectrum, we have approaches like DiaSuite [3]. Like Android, the declaration language is dedicated to an architectural style, in this case *Sense/Compute/Control*. Contrary to Android, the resource usage is part of the architectural style and specified at the component level, not globally (**Req1**). The declaration language also provides constructs dedicated to the interactions between the components [2]. This combination allows the developer not only to declare which components may interact with each other but also how (**Req2**) they interact (*i.e.,* in which order and depending on which conditions). This kind of data and control flow restriction is essential to our approach for ensuring confinement of sensitive data.

Another difference is that Android does not offer application-specific programming support. DiaSuite provides a tailored Java framework generated from the declarations, so that APIs for disallowed resources are unavailable to the developer. This lowers development effort since only essential API calls are available (**Req3**).

iOS[2], Android and DiaSuite all offer restriction of resource usage. Android and DiaSuite verify resource usage according to the declarations (**Req4**), but all three differ in how they enforce the declared permissions. In Android, if a developer tries to access a forbidden resource, a run-time exception is raised, *i.e.,* dynamic verification. This needs to be treated, or else the application will crash. This might only

[2] iOS is not declaration-driven, but since it does have a resource access control mechanism, we consider its alternative approach in the spectrum of possibilities.

be discovered via testing, or worse, by an end-user. DiaSuite, however, checks all resource usage statically, so that once a developer successfully compiles their application, they (and the end-user) can be sure that all permissions required have been granted accordingly. iOS simply prompts the user if and when a sensitive resource will be accessed for the first time (making resource declarations dynamic and implicit).

2.2 Core DiaSpec

For our case study, we will use a simplified declaration language, modelled on DiaSpec [2]. DiaSpec is used to specify applications in the DiaSuite system. The main interests of such a language for this case study are that (1) it applies to open platforms, for which declaration-driven frameworks have shown to be an attractive development approach; (2) it relies on an expressive declaration language, raising the bar for the implementation and exploring further the potential of declaration-driven framework.

DiaSuite. DiaSuite is a development methodology dedicated to the *Sense/Compute/Control* architectural style, promoted by Taylor *et al.* [22]. This pattern ideally fits applications that interact with an external environment. SCC applications are typical of domains such as building automation, robotics, avionics and automotive applications, but we will see that SCC fits the open platform model, too.

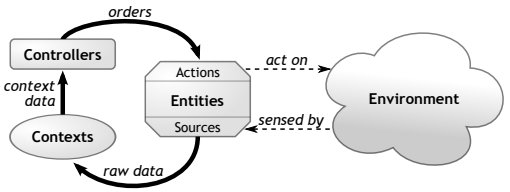

Figure 1. The *Sense/Compute/Control* paradigm.

As depicted in Fig. 1, this architectural pattern consists of three types of components: (1) *entities* correspond to open platform resources, whether hardware or software, and interact with the external environment through their sensing and actuating capabilities; (2) *context components* refine (filter, aggregate and interpret) raw data from the entities; (3) *controller components* use this refined information to control the environment by triggering actions on entities. When targeting a specific application domain (*e.g.,* home automation or mobile phones), the platform owner defines a unique taxonomy of entities (*i.e.,* resources) for all applications in this domain. This taxonomy of entities is a fixed list of resources available to the open platform (*e.g.,* camera, phone book, *etc.* on Android). The description of an application is composed of declarations of context and controller components that interact with the resources.

DiaSpec core language. In this case study, we will use a minimalistic version of the DiaSpec language. Non-essential constructs have been removed (*e.g.,* multiple interaction contracts, non-functional declarations such as timing/QoS constraints). The grammar is shown in Fig. 2.

```
1  CoreDiaSpec      -> Declaration*
2  Declaration      -> Resource | Context | Controller
3  Type             -> Bool | Int
4
5  Resource         -> source srcName gives Type
6                    | action actName takes Type
7
8  Context          -> context ctxName as Type
9                      { ContextInteract }
10 ContextInteract  -> when ( required GetData?
11                           | provided (srcName | ctxName)
12                                      GetData?
13                                      PublishSpec)
14 GetData          -> get (srcName | ctxName)
15 PublishSpec      -> (always | maybe) publish
16
17 Controller       -> controller ctrName { ContrInteract }
18 ContrInteract    -> when provided ctxName
19                     do actName
```

Figure 2. Core DiaSpec grammar. Keywords are in bold, terminals in italic and rules in normal font.

A DiaSpec specification is composed of a list of declarations. The non-terminals `Resource`, `Context` and `Controller` correspond to the three kinds of components. The resources are defined and implemented by the platform owner, therefore we will not consider them here. The contexts and controllers are defined using interaction contracts [2], which describe how they interact with other components and resources. In the SCC architectural style, context and controller components are reactive. A context component can be triggered by either a data pull request (`when required`) or data published from other components (*i.e.,* `when provided (srcName | ctxName)`). When activated, a context component may need to pull data (denoted by the optional `GetData`) from other contexts or sources (corresponding contexts must have a `when required` contract). Finally, a context may publish a new value (*i.e.,* `PublishSpec`. Note that `when required` contexts never publish). Controllers are only triggered by a data publication from a context. When activated, controller components can send orders to resources, using their actuating interfaces (*i.e.,* `do actName`).

Example scenario. As a running example, we will use a small scenario in the domain of building automation. It corresponds to a simple thermostat application which maintains a fixed temperature. When a temperature sensor publishes a new value, the context `Thermostat` should compare this value with the desired temperature, and accordingly turn the fan on or off. Since `FanOnOff` and `Temperature` resources (action and source, respectively) are defined separately, the application developer only needs to provide implementations of the context and controller components, as declared in Fig. 3.

```
1 context TgtTemp as Integer {
2     when required }
3 context Thermostat as Boolean {
4     when provided Temperature
5       get TgtTemp maybe publish }
6 controller FanController {
7     when provided Thermostat do FanOnOff }
```

Figure 3. DiaSpec declarations of the Thermostat application.

- The *TgtTemp* context component should return the desired temperature as an integer value. It is activated by pull (*c.f.* `when required`).
- The *Thermostat* context component is triggered whenever a new value is published by the `Temperature` source (`when provided` *Temperature*), then requests the value of the *TgtTemp* context component (`get` *TgtTemp*) and maybe pushes a new value (`maybe publish`).
- The *FanController* is triggered by *Thermostat* (`when provided` *Thermostat*) and controls the *FanOnOff* action accordingly.

These declarations should be translated into restrictions and obligations for the application developer, to be enforced by the programming framework.

2.3 Requirements, instantiated for DiaSpec

The goal of the framework, as our chosen solution to enforcing the requirements, is to support the developer (*guidance*), and ensure certain behaviours (*restriction*). Each of the requirements introduced previously is therefore instantiated as programming obligations, restrictions or support, which the framework should provide. *Obligations* are where the developer should be forced to do something, *e.g.,* implementing a given component, and *restrictions* are when we want to ensure certain properties, *e.g.,* the developer not accessing private user data. *Support* is where the developer should be aided, *e.g.,* by being provided with a specialised API.

Req1 *Resource declarations* list the permissions for each component. They imply restrictions, since unlisted resources may not be used.

- Restriction: each component should only have access to the resources explicitly granted. For example, the *Thermostat* context should only be able to query the *TgtTemp* component, no others.
- Obligation: all declared components must be implemented.

Req2 In our example, components are to be triggered if and only if their activation condition is met, *i.e.,* they must be *reactive*. *Interaction contracts*, combined with the reactive style, make the *data flow* explicit.

- Restrictions: all components are to be strictly reactive. In particular, the developer should not be able to activate components by arbitrarily broadcasting or pulling values, except via the framework.
- Support: the notification of subscribed components with fresh values should be transparent to the developer. For example, when the *Temperature* resource publishes a new value, the *Thermostat* context should always be called.

Req3 *Programming support* should be provided to the developer based on the declarations. This might include prompting for certain components' implementation, or providing an easy, tailored programming API. In DiaSuite, the API calls for accessing resources are only made available to the components authorised to use them.

Req4 The framework should *verify* that the implementation conforms to the declarations. This includes resource usage, data flow constraints, and any other specified aspects of an application's behaviour. Ideally, they are done as early (statically) as possible, giving the developer early warning and the user a better experience (no crashing applications).

Next, we will show the translation of these requirements into concrete programming artefacts in the form of a framework. It should not result in an unreasonable overhead for the developer. Furthermore, the framework should provide support for the developer to fulfil these obligations. We remark, however, that a difficult development process for the framework maintainer is accepted, since we assume it only needs to be implemented once. We will also not focus on issues like maintainability.

3. From Requirements to Implementation

Now that we have established the requirements for our case study, we present our implementation of a framework ensuring them.

We will be using two very different languages: a mainstream object-oriented, strongly-typed language (Java), and a dynamically typed functional language (Racket [10]).[3] Usually dynamic languages are seen as providing the programmer with less guidance, but more freedom and flexibility. Consequently, the large contrast between these languages should substantially differentiate the implementations. We conjecture that if we had used a strongly typed language for the functional implementation, the temptation might be too great to follow a similar approach to the object-oriented version. For example, we might approach extension using subtyping simply because that is the way we are familiar with.

We will see that even in a dynamic setting, we can meaningfully leverage the aforementioned declarations: we can use them to provide mechanisms which guarantee the application conforms to the declarations. Also, the API can be

[3] The code for both prototypes will be available online, from `http://phoenix.inria.fr/`.

specialised to each specific application, lowering the development cost.

In Sec. 3.1, we deal with the object-oriented prototype, and in Sec. 3.2 we present the functional prototype. Both sections are structured as follows: a *general description* of the design of the prototype framework, the *translation* of the DiaSpec declarations into programming language concepts, an *illustration* of a context translated into supporting code which ensures the requirements, and finally an *evaluation* of the prototype with respect to the requirements.

3.1 Java prototype

General design. As we see in Fig. 4, each component declaration is translated into an abstract class, which the developer must implement. By providing an abstract method with a type corresponding to the declaration, the developer is required to provide well-typed implementations. The implementation of a component will be placed in this method, which gets access to the allowed resources via specialised arguments.

Translation of the DiaSpec declarations. This is a sketch of how to translate various constructs into Java programming artefacts. We follow the grammar given in Fig. 2.

Introduction of a statement declaring a context or controller C results in an abstract class `AbstractC`, with an abstract method corresponding to the interaction contract. The type of the context partially determines the Java return type of the function the developer must provide. In the case of controllers, the type of the abstract method is always `void`, since these only actuate but are not supposed to return values. The abstract method's arguments are further determined by the interaction contract, which consists of an activation condition, a data requirement (or action, for controllers) and the publication specification.

Activation conditions.

`when provided x` This causes the abstract method to be named `onComponentProvided` where *`Component`* is the name of x. The first argument to this abstract method will be of the type of x. For example, if the contract is `when provided Temp`, the abstract method becomes `onTempProvided(int temp, ...)`, since *`Temperature`* provides an integer.

`when required` This causes the abstract method to be named `whenRequired`, without the first value-argument.

Data requirements. The optional *data requirement* produces a tailored proxy, to facilitate and regulate access to resources.

`get, do x` Adds an inner class to the `AbstractC` class, the proxy. An instance is passed to the `whenRequired` or `on..Provided` method, so that the developer has managed access to the resource. For example, if we encounter `get TgtTemp` as a data requirement, the inner class `TgtTempProxy` is created, and the context's abstract method is given an extra argument `TgtTempProxy discover`. Controllers' actions are handled the same way.

`get nothing` No extra argument.

Publication requirements. Finally, the *publication requirement* determines the return type of the abstract method. The framework manages the notification of subscribed components when the method returns.

`always publish` Equal to the context's type.

`maybe publish` The context's type T wrapped in an option type, *i.e.,* `Maybe<T>`.

Illustration with the `Thermostat` context. We will now detail how a developer would implement the *`Thermostat`* context, and which mechanisms come into play. The developer's code is presented in Fig. 5.

```
1  public class Thermostat extends AbstractThermostat {
2    @Override protected Maybe<Boolean>
3        onTempProvided(int temperatureValue,
4                       TgtTempProxy discover) {
5          int desired = discover.temperature();
6          if (temperatureValue > desired) {
7              return new Just<Boolean>(true);
8          } else {
9              return new Nothing<Boolean>();
10         }}}
```

Figure 5. Example implementation corresponding to *`Thermostat`*.

Here we see that the developer implements the single method, `onTempProvided`. This corresponds to the entry in the declarations `when provided Temp`, and is called by the framework if the temperature sensor publishes a fresh value. We see that the developer chooses to publish only if the threshold is reached, which is in accordance with the declaration `maybe publish` from Fig. 3. The type, `Boolean`, corresponds to the type of the context as declared. Furthermore, the developer has easy access to the data requirement *`TgtTemp`* via the second method argument.

Instead of generating an abstract class to be implemented, we could skip generation and use generics: this strategy could require a developer to provide a class which inherits from *e.g.,* `Context<Boolean>`. This is feasible, but we would lose the descriptive power of the generated method names, and the simplicity of the interface the developer must implement. This technique of producing human-readable method names is sometimes called a fluent interface [11]. This would not be possible, since we customise the name of the accessor method, in this case `temperature()`. Therefore, our generative approach allows a greater level of guidance for the developer than we would be able to achieve using only generics.

To better explain the abstract class, it is shown in abbreviated form in Fig. 6. The abstract method `onTempProvided` is

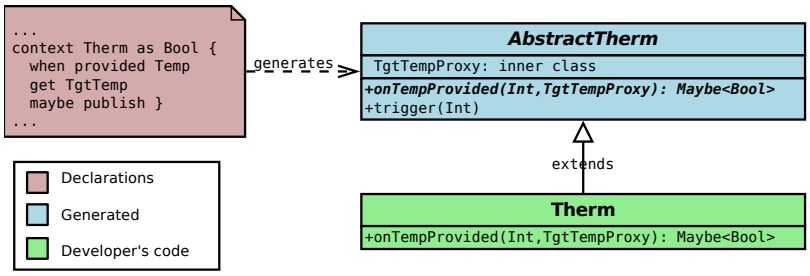

Figure 4. Schematic of the design of the Java prototype.

hidden for brevity. We will explain the proxy object `TgtTempProxy`, which allows indirect access to the data requirement, *`TgtTemp`*.

```
1  abstract public class AbstractThermostat
2        extends Component<Integer> {
3  ...
4      void trigger(Integer value) {
5          TgtTempProxy proxy = new TgtTempProxy();
6          proxy.setAccessible(true);
7          Maybe<Boolean> v = onTempProvided(value, proxy);
8          proxy.setAccessible(false);
9          if (v instanceof Just)
10             notify(((Just<Boolean>) v).just_value); }
11     protected class TgtTempProxy {
12         protected TgtTempProxy() { }
13         private boolean isAccessible = false;
14         private void setAccessible(boolean isAccessible) {
15             this.isAccessible = isAccessible; }
16         public Integer tgtTemp() {
17           if (isAccessible)
18             return runner.getTgtTemp().onRequireTgtTemp();
19           throw new RuntimeException("Access forbidden!");
20       }}}
```

Figure 6. An excerpt from the `TgtTempProxy` inner class.

The `TgtTempProxy` argument comes from the declaration `get TgtTemp`. By using Java access modifiers (`protected`, `private`), and a run-time check, we ensure that the developer can only access *`TgtTemp`* while the framework is polling *`Thermostat`*. Otherwise, it fails with an access violation, modelled as a run-time exception. This proxy is intended to provide restriction, but also a more simple API. The extra layer of abstraction also aids in keeping the API stable, should the resource's implementation change. A simpler approach would be passing the concrete value representing the resource's result, but our approach prevents the resource being unnecessarily polled pre-emptively, in case the developer decides not to use it. Note that a developer might try to subclass this proxy, and then instantiate it, but because of the run-time check, this would still not afford them access to the other components.

To ensure all components are implemented exactly once, we also generate an abstract class called `AbstractRunner`, to take care of linking the implementation classes to their declarations. It uses similar techniques to ensure that the developer provides the right components. It defines abstract methods like `getThermostatInstance()`, `getTgtTempInstance()`, *etc.* for a developer to implement, returning an instance of their class implementing that particular component. Since this class also provides the framework's `run()` method, the developer is forced to provide all the component bindings.

```
1  public class Runner extends AbstractRunner {
2      @Override
3      public AbstractThermostat getThermostatInstance() {
4          return new Thermostat();    }
5  ...
6  }
```

Figure 7. Example implementation's deployment description.

Now the framework's main method can simply query all sources and notify all subscribed contexts, then notify any components which are subscribed to them, *etc.* The framework's main method is able to do this thanks to another code snippet in the components' generated abstract classes, the `trigger()` method in `Component` (shown in Fig. 6). This generated glue code calls the method with the specialised name (in this case `onTempProvided()`) whenever the framework's main method needs to trigger it. This way we can use a generic layer of code to execute any scenario, only generating a small amount of glue code which dispatches the generic calls. Note that the type of the argument to `trigger()` is determined by the type argument to the `Component` abstract class.

We might improve our prototype by making use of Java annotations. By defining our own annotations such as `@Thermostat` to mark which class is the implementation of which declared component, we could further ease development. Using the annotations system, we could generate the above-mentioned abstract classes as needed and automatically tie the implementations together instead of requiring the developer to extend the `AbstractRunner` class (since the only purpose behind that action is to associate components with their implementations coherently). We consider this an optional improvement to what we have presented here, since functionally it would be identical, but would save the developer a few lines of code. Note that the guarantees provided would still be identical: we already check most properties statically, and this would not improve using annotations.

Evaluation of the prototype. Reflecting on the requirements from Sec. 2.1, we see that our prototype forces the implementation to conform to the declarations provided.

Req1 Resource access is strictly controlled by the framework, and is only possible via the generated proxy classes which are given to the developer's code as function arguments. Additionally, there are checks in place ensuring that each access is authorised.

Req2 The framework takes care of polling sources and publishing values, therefore the control flow is totally managed by the run-time environment. The only way to use the framework is by calling the `run()` method, which is only available on the implementation of `AbstractRunner`. This necessitates providing well-typed and well-formed implementations of all the declared components.

Req3 From the developer's point of view, implementation is extremely simple: the API is concise and specialised (*i.e.,* the arguments passed to the implementation, nothing else), publication is transparently taken care of by the framework, and there is no way to forget to provide an implementation for some component, because of the static checks.

Req4 All the properties can be checked at compile-time, except for the access to data requirements. This property is checked dynamically, every time access is attempted. We imagine that this could alternatively have been solved by using a Java extension with a more expressive ownership type system [1], but we decided remain with native Java.

3.2 Racket prototype

We will now look at the functional language prototype. Our aim is to provide the same level of support and constraint as in the previous prototype. This section will follow the same layout as previously, except that we will commence with a general description of Racket.

About Racket. Racket [10] is a dynamically typed functional language, a descendant of Scheme. It has powerful syntax transformers (*i.e.,* macros [15]), supports easily creating language extensions or even entire languages as libraries [23], which may have full use of Racket's features. It also has a recently-introduced advanced module system [9], supporting submodules and arbitrarily many phases (*i.e.,* macro transformer stages). Finally, it includes a library of function contracts [5], checked at run-time. Contracts are a language extension to annotate functions with arbitrary checks on input and output, with a system that usually blames the misbehaving component correctly. An example of a contract is `(-> int? bool?)`, which denotes that a function must take an integer and produce a Boolean.

General approach. Our approach makes heavy use of the language extension capabilities of Racket, generating specialised macros from the declarations. The language extension feature enables passing the entire contents of a module to a syntax transformer. This way the transformer has absolute control over the syntax, and as well as the available language features for that module.

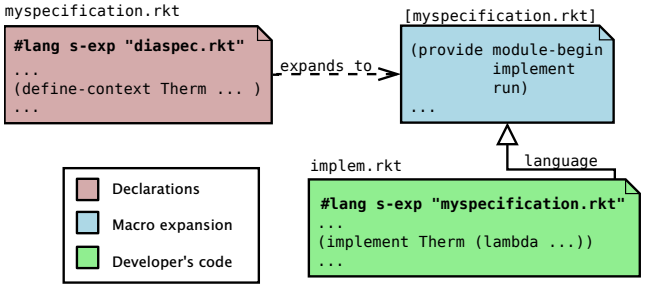

Figure 8. The Racket implementation. The developer provides declarations. These are transformed into a language for the implementation. The `implement` macro has cases for each declared component.

Fig. 8 illustrates the general design of our prototype. We have implemented the framework to provide a language for the declarations. This language provides the keywords `define-context` and `define-controller` which are used to specify the system. It only allows these declaration-keywords as top-level terms.

Next, the declarations are transformed into an application-tailored language, which the developer uses to provide the implementation. This language extension provides a keyword `implement` for implementation of the components. It has cases for each of the declared components, such as `Thermostat`. This provides maximal ease for the developer and control from the framework's point of view.

Translation of the DiaSpec declarations. Here we give a general outline of the syntax that each declaration written in a Racket-DiaSpec module will be translated into, and how the `implement` macro works.

Declaring a context or controller C adds a case to the `implement` macro. Now, a developer can use `(implement` C f`)` to provide a lambda function f which will be bound as the implementation of C. Not just any function f may be provided, as the arguments to `implement` are subject to a specific contract. Just like the Java abstract method, the function contract depends on the interaction contract.

Activation conditions. This defines the first argument to the function f.

`when provided` *`x`* Argument of type of x, *i.e.,* `(-> x? ...)`.

`when required` Argument omitted – the context was activated by pull.

Data requirements. The next argument is determined by the data requirement. It is a closure which will provide dynamically checked, proxy access to the relevant component. This

makes it convenient for a developer to query a resource, as well as regulating access. Actions for controllers are handled the same way.

`get` *`x`* The contract of the closure becomes `(-> t?)` where t is the output type of x. The full contract so far is therefore `(-> ... (-> t?) ...)`.

`do` *`x`* The contract of the closure becomes `(-> t? void?)` where t is the input type of x. The full contract so far is therefore `(-> ... (-> t? void?) void?)`.

`get nothing` Argument omitted.

Publication requirements. Finally, the *publication requirement* determines the last arguments to the function contract, corresponding to the return type of the context. Publishing is treated using continuation functions; these never return control to the developer's code.

`always publish` One continuation function, *i.e.,* the final contract becomes `(-> ... (-> bool? void?) void?)`.

`maybe publish` Here, we pass two continuations to f, for publish and no-publish. The publish continuation has the contract `(-> t? void?)` with t the output type of C. It is provided by the framework at run time, and allows the developer freedom over whether they choose to publish. If the developer chooses not to publish, they use the no-publish continuation, which stops the evaluation. The contract therefore becomes `(-> ... (-> t? void?) (-> void?) void?)`.

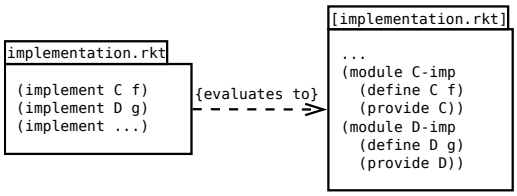

Figure 9. Separation of components using modules. The developer's code (left) is transformed (right). Now, C cannot access D, because of scoping.

Furthermore, the `implement` macro wraps f in its own submodule (see Fig. 9). Thanks to Racket's advanced submodule system, we can dynamically add submodules to the implementation module, which do not have access to the surrounding code, but merely export their implementations for use in the top-level module. Finally, the implementation module will also be checked to contain exactly one `(implement` C `...)` term for each declared C.

Illustration with the `Thermostat` context. Ignoring superficial differences, Fig. 10 is a transliteration of the declarations in Fig. 3. The developer indicates that the language to use is the one provided by `diaspec.rkt`, which provides the macros `define-context`, *etc.* This is done using the

```
1 #lang s-exp "diaspec.rkt"
2 ((define-context TgtTemp Integer
3   [when-required get nothing])
4  (define-context Thermostat Boolean
5   [when-provided Temp get TgtTemp maybe_publish]))
6 ((define-controller FanSpeed
7   [when-provided Thermostat do FanOnOff]))
```

Figure 10. The complete declarations of the Thermostat system, as used in the Racket prototype.

`#lang s-exp "diaspec.rkt"` tag. When Racket encounters this annotation at the top of a file, its entire contents are passed as a syntax term to `module-begin` in `diaspec.rkt`.

The macros we have implemented transform the description, generating `implement` and `begin-module` macros accordingly. This way the declarations module itself becomes a new Racket language. Each declaration of a component, using `(define-context Thermostat...)`, creates a corresponding binding, for use with the `implement` macro. To illustrate, the implementation of *`Thermostat`* is shown in Fig. 11. The other implementations have been omitted from the figure.

The developer simply provides a lambda term for each component. When the implementation module is loaded, the `begin-module` macro checks that all `(implement` C f`)` terms are present.

```
1 #lang s-exp "thermspec.rkt"
2 (implement Thermostat
3   (lambda (temp getTgtTemp publish nopublish)
4     (if (< (getTgtTemp) temp) (publish #t)
5         (nopublish))))
6 ...
```

Figure 11. The implementation of the `Thermostat` context.

After the developer provides a conforming implementation for each component, the module can be loaded, and the complete system can be executed. To understand the runtime system, and how the requirements previously detailed are enforced, we will look at the syntax produced by the macros.

As mentioned previously, when the developer uses the `implement` keyword, it is expanded to a submodule with a contract, as in Fig. 12.

After expansion, a submodule `Thermostat-module` is introduced. The advantage of using a submodule is to prevent its body having access to identifiers outside its lexical scope. This way each implementation is bound to a name, all of which are lexically inaccessible to the developer. Access to other components is only possible via the condoned proxy-closures the framework provides. There are ways a malicious developer might get around this restriction, for example by importing a module which has shared mutable state. Context A and B could both import module M, and use it as a communication channel. However, since the `require` key-

```
1 (module thermimpl "thermspec.rkt"
2   (module Thermostat-module racket
3     (define/contract Thermostat-impl
4       (-> int? (-> int?) (-> bool? void?) (-> void?) void?)
5       (lambda (temp getTgtTemp publish nopublish)
6         (if (< (getTgtTemp) temp) (publish #t)
7             (nopublish))))
8     (provide Thermostat-impl))
9   ...)
```

Figure 12. The developer's code snippet is transformed into a submodule. This snippet is the expansion of Fig. 11.

word in Racket is only allowed at the top-level of a module, the isolation should be sufficient. We could also imagine the `implement` macro inspecting the function for "reasonable" terms, like arithmetic or Boolean operators. This might be done if communication integrity were critical, but such static analyses are beyond the scope of our case study.

Next, the provided code snippet is bound to an identifier using `define/contract`, which is the Racket way to attach a behavioural contract to a function [5]. This is a dynamic check which, in this case, dictates that the first argument should be an integer (with `int?`, because of the activation condition). The second argument should be a closure which evaluates to an integer (with `(-> int?)`; it is the data requirement from the declaration `get TgtTemp`). The third argument is a function from `bool` to `void`, modelling the publication continuation (corresponding to the declaration `maybe publish`). If the developer decides not to publish a new value, they use the `nopublish` continuation. We would add run-time checks that the publication is happening at a valid moment (*i.e.,* only while the framework passes control to the component), and only once, in these continuations. This is to prevent a malicious developer saving a reference to the publish term, and publishing at some time other than permitted by the declarations, for example.

By providing the data requirement closures, we avoid giving the developer direct access to other components. The framework proxies the call to the required components. This way, we can be sure that only legal requests are allowed to be made. Alternatively, a keyword could have been introduced for this, facilitating a static verification that *e.g.,* a resource is only polled once. Finally, we see how the developer obtains and uses the provided values and publishes the computed value.

The last remaining step is collecting the implementation terms provided by the developer, allowing the framework to run the implemented system. In the generated language's `module-begin` macro, static checks are done, verifying that all the specified components are implemented exactly once. All the implementations from submodules like `Thermostat-module` are then assembled in a convenience function called `run`, which the developer may call to execute the system.

Evaluation of the prototype. Reflecting on which language mechanisms have been used to implement the various requirements, and how well the requirements are met, we summarise as follows.

Req1 Resource usage is strictly controlled by the framework. The developer's code is wrapped in separate submodules and only given access to resources via carefully checked proxy closures. This ensures that resource access must proceed via the framework.

Req2 By providing continuations, which proxy and check publications, we can be sure that the developer cannot influence the control flow. Because the developer must provide a function as the implementation term, we can also assume that a module with state cannot be `required` to illicitly communicate between components, for example. Combined with submodule scoping this ensures deep separation of components. The framework fully manages the control flow.

Req3 The implementation is simplified by the novel use of a tailored language extension. The developer is provided with helpful error messages if an implementation is omitted, and the API merely consists of the allowed resources being passed to the implementation as function arguments.

Req4 The shape of the implementation is verified statically, but the types a developer provides are dynamically checked using contracts.

All the properties resulting from the DiaSpec specifications are checked and enforced. We are able to ensure the same level of restriction as the Java prototype, and have managed to give the developer a clear and concise way of implementing the components. There is no complex API to communicate with other components, and the verification is mostly static. Just as in Java, checking whether data requirements are being legally accessed is performed at run-time. Furthermore, the types of values are checked using Racket contracts, also at run-time. If we switched to Typed Racket [23], the checks would be static just as in Java. Using Typed Racket would be trivial – it amounts to changing the language to `typed/racket` instead of `racket` in Fig. 12, and translating the contract-syntax (`-> ...`) into equivalent Typed Racket syntax.

We might later consider adding a safe way for developers to specify which libraries they would like to use, because currently only basic Racket is available (*c.f.* the module declaration in Fig. 12, `racket` is the submodule language). It might be unfortunate that we disallow importing of modules if a developer desires a particular library. It would be easy to provide our own `require`-like keyword, although this might allow aforementioned illicit communication (breaking **Req2**).

It is not essential to use the language extension feature of Racket, instead generating macros a developer could freely use without imposing a DSL. Defining a language extension, however, allows full control over the implementation: we can require that only `implement` terms appear at the module level, or that the declarations only consists of uses of `define-context`.

We could have also completely refrained from using macros, simply manipulating plain data structures – clearly a non-embedded DSL. This would have another disadvantage (apart from the inconvenience of not having specialised syntax), namely that the checks for coherence of the implementation, existence of the required implementations, *etc.* would all be dynamic. Because we value early warnings for the developer, our solution where the developer's mistakes cause the syntax check to fail is advantageous.

4. Principles

Here we will try to synthesise principles for how to implement the general requirements relevant to open platforms outlined in Sec. 2.1, in a spectrum of programming languages. We try to generalise the case study to framework design and how requirements translate to programming features in general, to help with understanding and building such frameworks.

Treating declarations statically or dynamically influences restriction and guidance. We have seen that controlling access to resources, or even more generally speaking, enforcing a certain control flow is essential to the open platform domain. This need can be fulfilled by declaration-driven frameworks. The declarations need to be given a semantics: for each requirement and for each resource the framework developer may choose to handle the restrictions either statically or dynamically, depending on the sensitivity of the resource. It is important to note that it is not necessary to choose one approach globally.

As with type systems, if a static approach is chosen, an advantage is early warning if a developer performs an illegal operation, but with the cost of less accurate specifications (*e.g.,* not all declared resources are necessarily used). Consider the case study, where a developer cannot access resources illegally, without triggering compile-time errors. If a dynamic approach is chosen, an advantages is a high degree of accuracy regarding which resources are used, and when, but with the cost of receiving late warnings about incorrect code (in the form of run-time exceptions, for example).

For example, in iOS the *a posteriori* dynamically-handled resource access controls are the most accurate: the user can choose to allow or disallow on a per-resource basis, if and only if access is requested. This still does not give the user a clear view on what happens with sensitive data, though. It might be that at the same time as a legitimate access to sensitive data, the application caches it for later exfiltration to a malicious 3rd party.

This is in contrast to the Android model, which is still dynamically checked but even less favourable: even if a particular resource remains unused for a particular session it still has to be accepted by the user at install time (causing vulnerabilities, *e.g.,* advertisement libraries abusing their embedding into over-privileged applications [24]). In DiaSpec the permissions are also fixed once for all sessions, but this is used as an advantage: the development can be eased by specialising the API per application, and giving the developer warnings about misuse of resources at compile time. The user also has more insight, because permissions are not global to an application as they are in Android, making the declarations more meaningful.

A recent version[4] of Android added the possibility of allowing an application to run with reduced/partial privileges (*e.g.,* GPS accessible but contact list forbidden), indicating that the old permission model was not meeting users' needs. This in turn suggests that the original design choices were ad-hoc, with implicit principles – now that there is a large user base feedback is appearing on how this situation should be improved. We hope that this work can clarify the possible design space for such systems, and allow a better understanding of the choices of static or dynamic enforcing of particular rules, and which advantages and trade-offs this implies.

Viability of enforcing requirements is independent of programming paradigm. We have some computations which are specific to our declaration-driven approach, such as checks whether specific resources may be used at given times. Depending on whether we choose to encode the semantics statically or dynamically we get differing types of support and restriction, as shown in the previous principle.

We note that the choice between statically and dynamically handling the declarations is orthogonal to whether the host language is statically typed or dynamically typed. In fact, in general, a type system is not even a prerequisite to being able to realise all the requirements introduced in the case study. This is evidenced by the fact that in both the Java and Racket prototypes we manage to implement identical guarantees.

All that is necessary is a programming environment with at least one stage before run-time which enables processing of the static semantics of the declarations. Note that this is a less strong requirement than saying we need a programming language which supports compile-time stages. Consider the Racket example, where we make no use of static features or a type system, but implement everything using syntax transformers, which can simply be seen as extra compilation phases. A type system can be seen as a limited-expressiveness programming stage too, which is where the properties are verified in the Java prototype.

If we have (or can implement, whether by a declaration compiler or syntax transformers) stages, the place to handle

[4] As of version 4.3, Android contains a hidden permissions screen to allow or disallow resources per-application and per-resource [17].

the enforcing of requirements and obligations arising from the declarations, is in the stage(s) before run-time. In our case study, we used code generation plus the type system for the Java framework, and macro expansion for the Racket implementation for achieving this kind of checking. This shows that widely varying techniques can be used to implement stages. Therefore, a strong and/or static type system is not required, even if static enforcing is desired.

5. Discussion

In this section we discuss the lessons that we have learned from the implementation of the prototypes in Java and Racket.

We have developed a system where meaningful declarations regarding shape and behaviour of an application are used to provide a programming environment which actively ensures these properties are met, while at the same time reducing development effort for the application developer. We do not claim that our technique of generating a Java framework or using Racket's language extensions is the best engineering approach to implement a tailored framework, rather we argue that tailored frameworks can deliver great advantages.

We emphasise that from both the developers' and users' point of view, both implementations are equivalent in terms of features and guarantees. This is a convincing argument for the fact that a strong type system certainly is no prerequisite for a property-ensuring framework. Most of the verification is static, even in Racket, because of our use of syntax transformers. This enables giving errors and warnings already at syntax-checking time. However, we are fortunate in that Racket is a very extensible and expressive language, which might give an optimistic impression of what is easily realisable in other perhaps less expressive host languages.

A strong case for using declaration-driven tailored frameworks. There is therefore a strong case for using tailored frameworks, whether one is working in an object oriented or functional domain. They potentially provide a way to turn declarations from a contemplative document into something verifiable, and which can provide the user of the application with valuable information on the behaviour of an application. Aside from that, we see that from the developer's point of view, implementing an application with a specialised framework is much less laborious than using a general purpose framework: in our example, only the implementations for the components are necessary. All communication, deployment, *etc.* is taken care of by the framework – which is possible because the framework has detailed information about the intended structure and behaviour of the app.

Therefore, we believe that this new generation of frameworks can give us fundamental advantages – even in a functional programming setting. Even if traditional, general-purpose frameworks do provide a notion of restrictions, the available declarations (*e.g.,* the Manifest file in Android) are usually not used to ease development. This seems like a missed opportunity.

We also find that the implementations in both Java and Racket are natural: extending classes is the traditional way of using an object-oriented framework, as is using domain-specific languages to solve problems in a Lisp-like language. Except that, even while maintaining the traditional approaches, it is very feasible to go beyond the status quo and enforce properties, give the user insight into the usage of their resources, and assist the developer. We would argue, though, that the Racket prototype, from the developers' point of view, probably is even more natural. There, the tailored language makes is very simple to provide an implementation. In Java, on the other hand, there is the problem of the relatively unwieldy proxy classes, plus the deployment boilerplate code which might be a stumbling block to application developers. We conjecture that this is because functional languages with higher-order functions allow a more natural encoding of inversion of control concepts than is possible in Java.[5]

Related work

Our work asks a different question than has been posed before: we attempt to take a step back and analyse the design of declaration-driven frameworks, where they have usually been engineering solutions to specific problems such as containment of sensitive data. However, there are certain publications that should be mentioned, even if they do not aim to answer the same questions.

DiaSuite, Yesod. This work was inspired by DiaSuite [2, 3]. Therefore, it most closely resembles the prototype framework. However, the articles related to DiaSuite explain the theory of interaction contracts, and the idea of a generated tailored framework which guides and supports the developer, but the reflection on design space as well as what the requirements are for implementing such a system have not been addressed. Furthermore, the discussion about DiaSuite's design is exclusively in the context of Java. We therefore regard our work as an overview of the principles implicitly motivating previous work on DiaSuite, as well as a generalisation to a language-agnostic approach.

Yesod [21] is a web framework in Haskell which makes similar use of declarations to tailor the framework per application, and then to guide the developer, and statically verify the implementation. In the Yesod documentation, a reflection on the design space and the potential benefits of the use of declarations is also not made – it is not clear if the declarations are being optimally leveraged.

TouchDevelop. Regarding frameworks which support open platforms, we find many approaches attempting to restrict sensitive data usage, and give the user more insight. For

[5] Starting from version 8, the Java language includes lambda functions [18]. They might provide a more elegant way of modelling the proxied polling access to resources, much like we do in Racket.

example, Xiao *et al.* [25] provide a domain-specific programming language based on TouchDevelop[6] to facilitate static analysis, per-resource permissions, and showing a user what the potential flow of information is (*e.g.,* from camera to Facebook). This is different to our approach since a developer has to learn a new language, whereas we are able to achieve meaningful and fine-grained restrictions while allowing a programmer to use their familiar general-purpose programming language (allowing freedom to choose IDE, libraries, tools, *etc.*).

Non-invasive static analysis. Much other work exists [6, 12, 16] looking into static analysis of existing code without requiring changes to the platform. Usually this work is motivated by privacy and safety concerns. We believe this is potentially less accurate since most static analyses have to choose between a trade-off of false positives and false negatives.

6. Conclusions

Considering our research question, which dealt with being able to encode constraints arising from declarations into concrete programming features, we have shown that strong guarantees can be built into a wide spectrum of programming paradigms, from object-oriented to functional. We have also demonstrated that very little is required from the target programming language in terms of static typing or other features, but that these techniques should be possible in any language which supports pre-run-time stages. If there is no such support, this can be modelled using a generative approach.

Furthermore, we believe that our prototypes constitute strong evidence that declaration-driven frameworks have a lot to offer all the stakeholders in the context of open platforms. They facilitate development, increase possibilities of confining sensitive data, and promote reuse through abstraction.

To the best of our knowledge, such frameworks are not yet widespread in the functional world, and we hope that this work will stimulate research towards the possibilities and advantages for functional programmers provided by this technique, as well as establishing good practices for developing such frameworks.

References

[1] N. Cameron and J. Noble. Encoding ownership types in java. In *Objects, Models, Components, Patterns*, pages 271–290. Springer, 2010.

[2] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985852. URL `http://doi.acm.org/10.1145/1985793.1985852`.

[3] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Trans. Software Eng.*, 38(6):1445–1463, 2012.

[4] J. L. Dave Mark. *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

[5] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *ACM SIGPLAN Notices*, volume 46, pages 215–226. ACM, 2011.

[6] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang. A static assurance analysis of android applications. *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[7] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997. ISSN 0001-0782. doi: 10.1145/262793.262798. URL `http://doi.acm.org/10.1145/262793.262798`.

[8] J. Feiler. *How to Do Everything: Facebook Applications*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 2008. ISBN 0071549676, 9780071549677.

[9] M. Flatt. Submodules in racket: you want it when, again? In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 13–22. ACM, 2013.

[10] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[11] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[12] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013.

[13] R. N. Horspool and N. Tillmann. *TouchDevelop: Programming on the Go*. The Expert's Voice. Apress, 3rd edition, 2013. ISBN 978-1-4302-6136-0. available at `https://www.touchdevelop.com/docs/book`.

[14] R. E. Johnson. Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[15] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[16] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[17] Michael Lee on ZDNet.com. Hidden Android feature allows users to fine tune app permissions. Online, `http:`

[6] TouchDevelop [13] is an application creation environment allowing developers to write scripts for mobile devices and publish them in an app store for users to install. It offers an imperative, statically-typed language. Xiao *et al.* have developed a static information flow analysis, as well as a modified run-time which allows individual resources to be replaced by anonymised values.

//www.zdnet.com/hidden-android-feature-allows-users-to-fine-tune-app-permissions-7000018944/, 2013. Accessed: 2014-03-23.

[18] Oracle. "Lambda Expressions", in Learning the Java Language, Mar. 2014. URL http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[19] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *Software, IEEE*, 4(1):6–16, 1987.

[20] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly, Beijing, 2009. ISBN 978-0-596-52147-9.

[21] M. Snoyman. *Developing Web Applications with Haskell and Yesod - Safety-Driven Web Development*. O'Reilly, 2012. ISBN 978-1-449-31697-6.

[22] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[23] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[24] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[25] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. User-aware privacy control via extended static-information-flow analysis. In M. Goedicke, T. Menzies, and M. Saeki, editors, *ASE*, pages 80–89. ACM, 2012. ISBN 978-1-4503-1204-2.