Type-driven Cross-Programming for Android and LEGO Mindstorms Interoperability

Alvise Spanò[1], Agostino Cortesi[1], Giulio Zausa[1]

Università Ca' Foscari, Via Torino, 153, 30172 Venezia VE
`alvise.spano@unive.it, cortesi@unive.it, mail.zausa.giulio@gmail.com`

Abstract. We present Legodroid, a Java library for Android that allows cross-programming LEGO Mindstorms through an Android device to exploit its extra computational capabilities in a seamless way. From a programmer's perspective, the paradigm it suggests for programming the EV3 is straightforward and resembles a standard *main* function in the likes of `leJOS`, which natively runs on the EV3 side though. Moreover, the library imposes type-driven coding patterns for interacting with motors and sensors, which guide developers in writing correct code with less runtime errors thanks to a rigid discipline over types. This is particularly effective in Android, whose component-based pattern complicates coding of traditional long-running algorithms for robots. Compared to `leJOS`, Legodroid users reported shorter bugfixing times and a more accessible paradigm for programming the robot, which had a positive impact on how much resources could be put in writing smarter algorithms and sophisticate interactions.

Keywords: Android, Java, LEGO Mindstorms, EV3, type-driven development, type soundness, design patterns

1 Introduction

LEGO Mindstorms is an educational platform including an SDK for programming the main control unit - namely the EV3 brick - by using RoboLab, a visual language meant to learn coding and problem solving [38]. Alternative ways of programming the brick in a more traditional way exist as well:

(a) Using `leJOS`: a Java-based SDK ported from the LEGO NXT Kit [35] offering classes and methods for accessing EV3 motors and sensor in an object-oriented fashion; programs run directly on the EV3 device.
(b) Sending *commands* from a remote device: the EV3 brick runs an interpreter of instructions, constantly listening to incoming wifi[1] or bluetooth connections and processing requests; requests are structured streams of bytes, formatted according to the *EV3 Communication Developer Kit* specification [36].

[1] At the time of writing, Legodroid does not support wifi connections, as the bluetooth counterpart is preferable in most cases. A `WifiConnection` class is expected by design though and will be added in a future update.

(c) Flashing the brick ROM with a new custom firmware is an option for those willing to take over the system and reprogram it from scratch.

While option (a) produces programs running on the EV3 brick, with its limited performance[2], option (b) unleashes the computational power of external connected devices hosting and running the program. Legodroid facilitates the latter approach: it consists in a Java library for Android with a strongly-typed API for programming the EV3 brick from a remote device in a type-driven disciplined way. Application business logic can be implemented on the Android side: interaction with the brick is seamless, allowing the programmer to design complex algorithms that apparently run on the robot but actually don't. Benefits of this Android-based approach include:

- computing power: Android devices, from mobile phones to tablets, have a much more powerful CPU than the EV3;
- development environment: Android Studio [19] is an advanced IDE, offering debugger, code analyzers and other powerful tools for developing apps in a comfortable way;
- third party technologies: the whole Android SDK is at your fingertips, including a powerful UI, background services and components; plus, many third party libraries are available for Android.

Legodroid introduces a new way for programming LEGO Mindstorms compared to other APIs, by making the application Android-centric but not only: it imposes a sound style, where error-prone coding habits are discouraged by a disciplined use of types. Additionally, it offers a straightforward paradigm for programming the robot: regular Java code packed into one callback representing the robot *main* function; within its scope users are allowed to access to the sensors and motors connected to the brick.

The library is mainly addressed to two kinds of audience:

(1) juniors willing to learn coding: we argue that type-driven programming, despite being a methodology mostly explored by a niche of advanced programmers, effectively aids in teaching coding and problem solving even at the basic level, thanks to the educational power of a strict discipline over types;
(2) experienced developers willing to explore new ways of programming Android and improve their programming skills through a sophisticate use of types in Java.

We put the library to the test by developing 17 different apps controlling LEGO Mindstorms robots for different purposes. In section 4 of this paper we show the outcome of this experiment, confirming that type-driven programming reduces bug fixing time and favours quick deployment of stable applications, allowing programmers to focus on the business logic of the application and on the algorithms.

[2] The EV3 CPU is a 300 MHz TI Sitara AM1808 (ARM926EJ-S Core) with 64 MB of RAM.

1.1 Type-driven Development

Type-driven programming wants to bring the type-sound coding discipline coming from the world of functional languages to the world of mainstream application development, where traditionally less care is put on types and static safety. Literature on the matter is scarce and mostly of industry origin, type-driven programming being more of a programming methodology than an actual scientific achievement; still, we believe its teachings and benefits are particularly meaningful in an era where dynamic languages have become mainstream and widely used for writing small as well as big software, training generations of coders against a disciplined use of types.

Also known as *typeful programming*, its principles have been foreseen decades ago by a handful of knowledgeable computer scientists [7], albeit it has quite never broken through despite the advancements of technology, arguably because of its difficulty [29]. Type-driven development relies on a accurate type design and compile-time validation of code, as opposed to writing down unthoughtful algorithms dealing with untyped or barely typed data. The basic idea behind it is that "a strong type system can not only prevent errors, but also guide you and provide feedback in your design process" [31], which admittedly requires a deep understanding of types and how to exploit the compiler as a tool for validating code.

Recently, interest in type-driven development has increased: the Haskell and F# communities have been promoting the benefits of *designing with types* for years [41], showing how writing programs with the Hindley-Milner type system [10] shifts the emphasis towards type design and improves the programmer's understanding of the static properties of software [32]. Even more advanced languages based on dependent type systems such as Idris [5] brought type-driven development to new heights [6], by conducting the programmer to the correct implementation in a quasi-mechanical way, putting the basis for a form of *assisted* programming guided by rich type information.

Although not all programmers can realistically learn dependent types [43], we believe that any programmer could be trained in respecting the basic type-driven programming principles to some extent, even without dependent types or complex In this paper we claim that the fundamental principles of type-driven programming can be ported to mainstream languages like Java, and any programmer, at any level of skill, can benefit of it at the cost of learning the core concepts of functional programming.

1.2 Core Principles

The type-driven approach is particularly recommended when writing libraries. Libraries impose styles, patterns [13] and discipline to programmers designing applications and are often responsible for the quality of the outcome in terms of code maintainability, scalability and safety. A modern library should carefully find a balance between two opposite characteristics:

Flexibility. Exported functions must be generic and cover a wide range of scenarios, exploiting forms of polymorphism;

Type safety. Operations must be constrained to certain data types, guiding the programmer in writing correct code.

We synthesize a handful of type-driven qualitative principles that have been put on test in designing and implementing Legodroid.

I Make code more general via higher-order functions [14]. Custom behaviours can be formulated via callbacks instead of overriding methods in sub-classes, which leads to a greater use of parametric polymorphism in place of subtyping. The higher-order function approach, obviously originating from the functional world, has been incrementally adopted by mainstream languages in recent years and is nowadays not unaccepted from the object-oriented programming community as it used to be in the past [23]. It fits better the immutable programming style, reducing *statefulness* in code and thus errors due to state invalidity [42].

II Never allow the programmer declare uninitialised variables, but rather emulate the functional let-binding by constructing objects in a valid state. Nullness checking is crucial: adding Java annotations `@NotNull` and `@Nullable`, combined with an aggressive use of the `final` qualifier, raises code quality in a measurable way [8]. This has an impact on how classes and constructors are designed: avoiding no-argument constructors discourages the "create empty and populate with setters"-sort of approach, which in turn discourages unneeded mutable data [9].

III Reduce side effects to the minimum [20]. Arguably, most mutable data structures in imperative programs are involuntary so, since mutability is the default condition for variables and fields in most mainstream languages. Overuse of assignments is a common source of runtime bugs in presence of concurrent code, for instance, and in general contributes to the proliferation of runtime errors, whereas programming with immutable data tends to lift errors up to the type level.
Manipulating immutable data types, moreover, does not make code slower: this is a common misconception that happens to be true less often than not, since most OO languages implement call-by-reference parameter passing and data is never copied unless explicitly cloned [2]. Quite the opposite, this is another point in favour of immutability, as modifying function parameters is another error-prone practice that compilers today discourage.

IV Use strong types even for intermediate result. A wise use of types and generics [4] can literally drive the programmer to the correct solution, by rendering unwanted chains of function calls impossible due to type mismatches. Control flow reflects data flow; and data is ultimately validated by types [46].

Despite these may seem in contradiction with classic OOP practices and advises, it has been observed that object-orientation *adds* abstractions, in the form of heavyweight type names that represent operations, whereas the functional approach *subtracts* abstractions thanks to anonymous functions, i.e. lambda expressions [22]. Abstractions are useful and powerful in many cases, but overly complicated class hierarchies are hard to understand: for example, reducing the number of heavyweight types representing callbacks relieves the programmer of memorizing dozens of type names that represent functions somehow, and could therefore be anonymized. This has another advantage in terms of design: it reduces the number of classes and interfaces to those cases modelling actual data types, which aids in separating data from behaviour [2].

2 Library Breakdown

In this section we describe the architecture of Legodroid in terms of types, API design and patterns. We motivate aspects of the type-driven design in particular. Before delving into it, a few words on the notation we are using in this paper.

Notation 21 (Subtyping)**.** *We write `T` $\preceq$ `S`, where `T` and `S` are types, for indicating that `T` is a* subtype *of `S`; its symmetric counterpart `S` $\succeq$ `T` holds as well.*

Notation 22 (Qualified Names)**.** *Method names are suffixed by brackets, e.g. `run()` is a bare method name, whereas the notation `EV3.run()` specifies the class it belongs to.*

Legodroid is designed with 3 layers of API. Each layer strictly wraps the underlying one and supports extensions.

Low level API. It deals with serialization and byte-level manipulation of commands for communicating with the EV3 brick according to the *EV3 Communication Developer Kit* specification [36]. The `comm` sub-package, detailed in section 2.2, contains the `Bytecode` class, aimed at building commands by appending op-codes and manipulating parameters at the byte level in a straightforward way. Users willing to extend the library with new commands can limit use of such low-level primitives to small self-contained methods.

Mid level API. Class `Api`[3] provides the core primitives for interacting with EV3, such as reading SI or PCT values from a sensor. The *EV3 Firmware Development Kit* defines these as half-baked data types translating, respectively, into `float` and `short` in Java. Extending the library at this level means to add new methods implementing EV3 instructions that are currently unsupported by Legodroid, manipulating arrays of floats or short according to the specification in section 4 of [37].

High level API. The `Api` class offers a family of *getter* methods constructing strong-typed handles to sensors and motors defined in the `plugs` package. Such handles exhibit methods performing high-level operations over sensors and motors and are distinct classes within the `plugs` sub-package. Extending the library at this level means to extend the `Api` class with new methods

[3] We refer to the `EV3.Api` nested static class as `Api` for brevity.

constructing new handles, which provide the methods implementing new commands for the brick in the same way as classes in `plugs` do.

The reason why the `Api` class includes two layers of API is subtle: from a user perspective the high-level methods and handles dealing with sensors and motors should be enough for most situations. Mid-level methods like `getSiValue()`, `getPercentValue()` and `execAsync()` are not enough, in number, to justify the architectural overhead of an additional class. Users willing to implement new high-level methods have all they need at their fingertips.

We now delve into the details of each package of Legodroid. This is not a replacement for the documentation of the library but rather the explanation of how its major feature impact programming in a *typeful* way.

2.1 `legodroid.lib.util`

Package `legodroid.lib.comm` contains general utilities such as the definition of functional interfaces [27] for supporting older versions of Android preceding Java 8; and a `Prelude` class containing miscellaneous utility functions.

Interfaces `Function` and `Consumer` reproduce `java.util.Function` and `java.util.Consumer` as defined in the JDK 8+, providing compatible functional interfaces working with Android API 21, which does not include Java 8 features. A more formal way for describing such functional interfaces would be using arrow types, assuming that ∅ represents the `unit` type.

$$\begin{aligned}\texttt{Function<A, B>} &\equiv \texttt{A} \to \texttt{B}\\ \texttt{Consumer<T>} &\equiv \texttt{T} \to \varnothing\\ \texttt{Runnable} &\equiv \varnothing \to \varnothing\end{aligned}$$

`ThrowingConsumer<T, E>` extends the functional interface `Consumer<T>` adding an extra type parameter $\texttt{E} \preceq \texttt{Throwable}$ that statically tracks the exception possibly thrown by the `Consumer` callback through a constrained generic. The Java `throws` declaration in method signatures can be modelled by a special arrow type where the exception is annotated:

$$\begin{aligned}\texttt{ThrowingFunction<A, B, E} \preceq \texttt{Throwable>} &\equiv \texttt{A} \to \texttt{B} \bigtriangleup \texttt{E}\\ \texttt{ThrowingConsumer<T, E} \preceq \texttt{Throwable>} &\equiv \texttt{T} \to \varnothing \bigtriangleup \texttt{E}\\ \texttt{ThrowingRunnable<E} \preceq \texttt{Throwable>} &\equiv \varnothing \to \varnothing \bigtriangleup \texttt{E}\end{aligned}$$

In order to make these interfaces compatible with their inherited parent functional interfaces, an additional `callThrows()` method is defined which adds the `throws E` declaration, whereas the original `call()` method is overridden and traps any exception raised by `callThrows()` by converting it to a non-checked `RuntimeException`. This allows for the best of both worlds: either executing the callback knowingly expecting an exception or totally trapping it, type-wise.

Finally, class `Prelude` is just a container for utility functions, among which `trap()` is arguably the most useful: `trap()` picks a function and executes it within a try-catch block trapping any exception possibly raising from it. Overloaded versions for different functional interfaces exist as well, the behaviour emerging clearly from the following functional type signatures:

$$\begin{aligned}
&\texttt{trapFunction} : \forall\alpha\ \beta\ (\gamma \preceq \texttt{Throwable}).\ (\alpha \to \beta \bigtriangleup \gamma) \to \alpha \to \beta \bigtriangleup \varnothing \\
&\texttt{trapConsumer} : \forall\alpha\ (\beta \preceq \texttt{Throwable}).\ (\alpha \to \varnothing \bigtriangleup \beta) \to \alpha \to \varnothing \bigtriangleup \varnothing \\
&\texttt{trapRunnable} : \forall\alpha \preceq \texttt{Throwable}.\ (\varnothing \to \varnothing \bigtriangleup \alpha) \to \varnothing \to \varnothing \bigtriangleup \varnothing
\end{aligned}$$

Mind that these arrow-based type representations are not meant to be accurate, but rather to display how functional manipulation occurs in a more readable way. Java does not provide arrow types in its type system [33]; there is no `unit` type, being `void` only a keyword for expressing methods with no return statement and not a type constructor [26]; also, currying is technically possible but not as straightforward as in functional languages, where the application syntax models the lambda-calculus term for application [1]. These are complications that ultimately make the actual implementation different from the clean type signature.

2.2 `legodroid.lib.comm`

Package `legodroid.lib.comm` in figure 1 shows the architecture of types related to the communication facilities provided by the library. Channels represent the basic abstractions offering communication primitives: a `Channel` can send a `Command` and receive a `Reply`, both of which are subclasses of `Packet`. Low-level communication with EV3 is based on exchanging data as untyped byte arrays formatted according to the official specification defined by LEGO in the *EV3 Communication Development Kit* [36]: *direct commands* sent by the client and consequent replies coming from EV3 require a byte-per-byte encoding, which includes a header followed by a an extra sequence of bytes carrying the custom content of each request; the header consists of fixed byte fields such as the length of the packet, the sequence number, the command type, the attached data etc. Class `Const` binds all C-style preprocessor symbols defined in the official header files as static numeric constant fields in Java, mostly used by the `Bytecode` class for serializing commands.

The `Connection<C>` interface represents the contract for constructing channels of type `C`, where $\texttt{C} \preceq \texttt{Channel}$, essentially implementing a *typed factory* pattern tracking the type information associated to the result type, as opposed to the old-fashioned, classic factory pattern which is often considered an obsolete unsound way of constructing objects [12]. Interestingly, `Connection` is equivalent to the `java.util.Supplier` interface defined by JDK 8+, making it a functional interface *de facto*, albeit with type `C` being upper-bounded to type `Channel`. We can, in other words, write the following type equation:

$$\texttt{Connection<C} \preceq \texttt{Channel>} \equiv \texttt{Supplier<C>} \equiv \varnothing \to \texttt{C} \bigtriangleup \varnothing$$

Fig. 1. UML Class Diagram of package `legodroid.lib.comm`

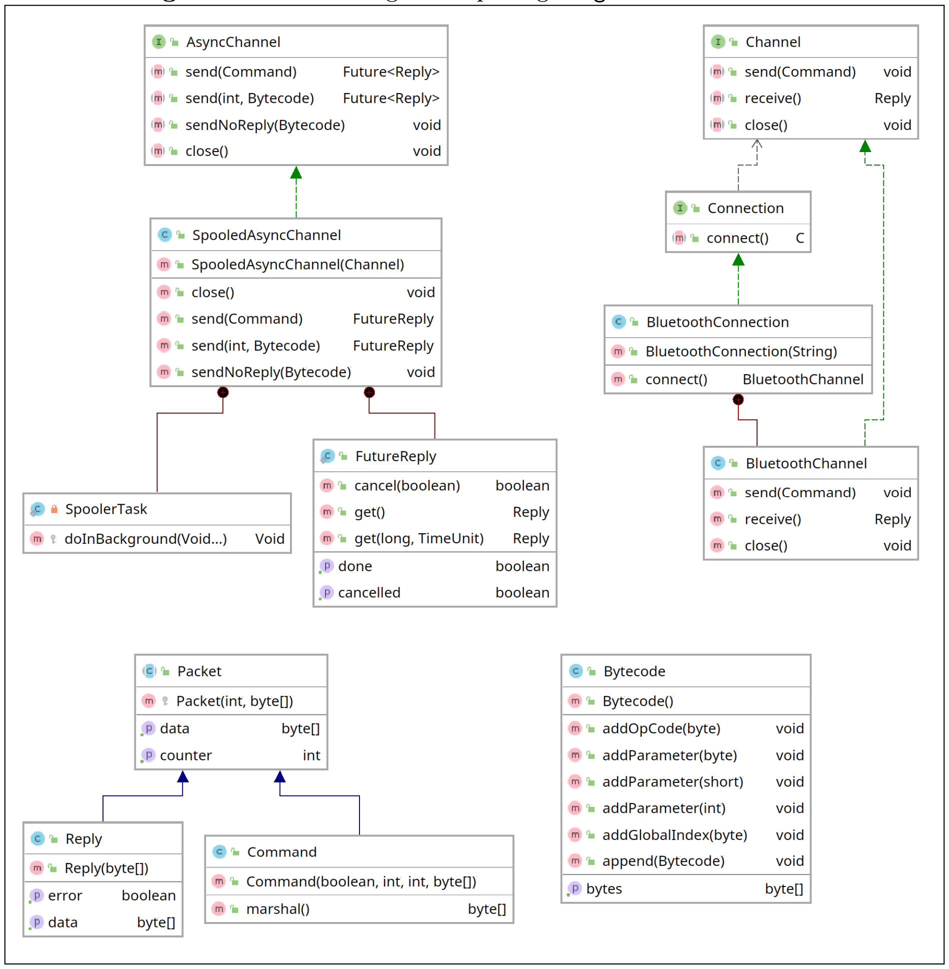
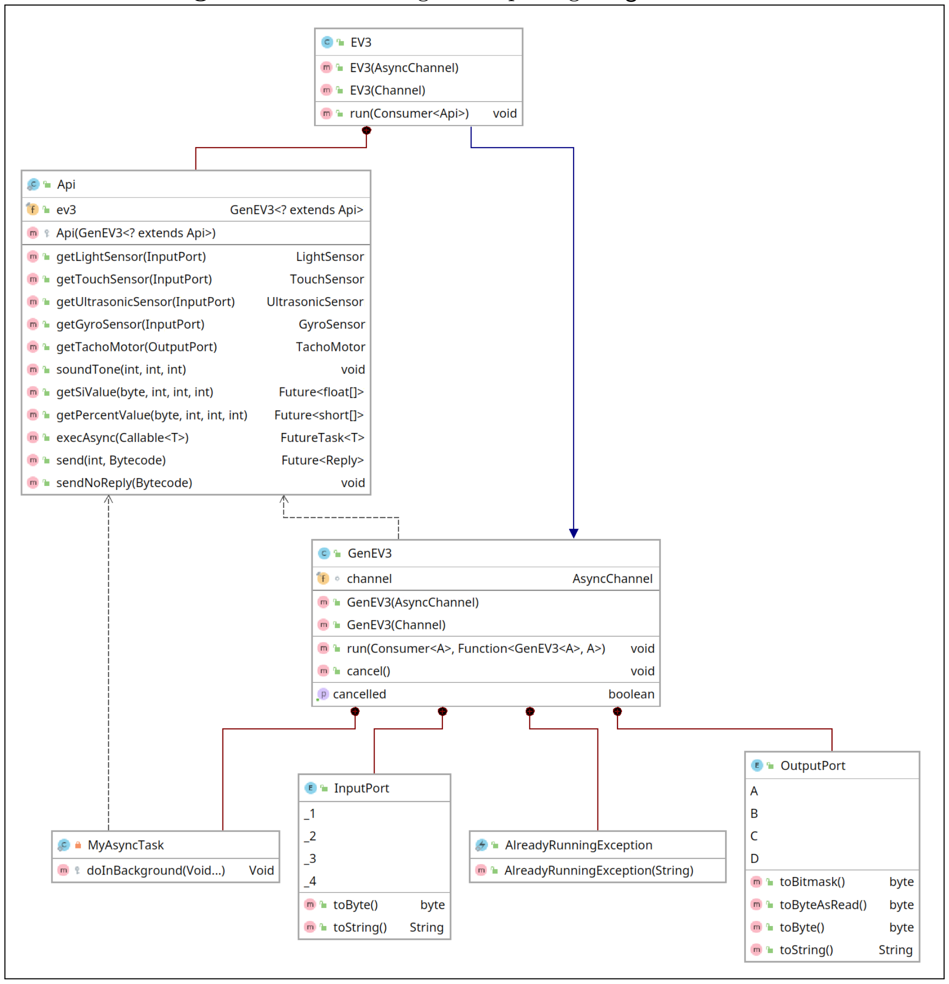


Fig. 2. UML Class Diagram of package `legodroid.lib`


EV3
EV3(AsyncChannel)
EV3(Channel)
run(Consumer<Api>) void
Api
ev3 GenEV3<? extends Api>
Api(GenEV3<? extends Api>)
getLightSensor(InputPort) LightSensor
getTouchSensor(InputPort) TouchSensor
getUltrasonicSensor(InputPort) UltrasonicSensor
getGyroSensor(InputPort) GyroSensor
getTachoMotor(OutputPort) TachoMotor
soundTone(int, int, int) void
getSiValue(byte, int, int, int) Future<float[]>
getPercentValue(byte, int, int, int) Future<short[]>
execAsync(Callable<T>) FutureTask<T>
send(int, Bytecode) Future<Reply>
sendNoReply(Bytecode) void
GenEV3
channel AsyncChannel
GenEV3(AsyncChannel)
GenEV3(Channel)
run(Consumer<A>, Function<GenEV3<A>, A>) void
cancel() void
cancelled boolean
MyAsyncTask
doInBackground(Void...) Void
InputPort
_1
_2
_3
_4
toByte() byte
toString() String
AlreadyRunningException
AlreadyRunningException(String)
OutputPort
A
B
C
D
toBitmask() byte
toByteAsRead() byte
toByte() byte
toString() String

2.3 `legodroid.lib`

The root package of the library contains the main classes for programming with Legodroid, as shown in figure 2. Class `EV3` stands at its core and exhibits most of the type-driven practices. An instance of type `EV3` represents a physical instance of the EV3 brick and can basically do one thing: executing a callback as if it was the *main* function for that brick, running on the Android device as a standalone thread which constantly communicates with the brick in a transparent way.

The `run()` method takes a function $\texttt{Api} \to \varnothing$ (or a `Consumer<Api>` function object) as argument and executes it within an Android `AsyncTask` that traps and logs any unexpected exception. It also behaves like a singleton, granting only one callback is running at any given time on the brick. Perhaps surprisingly, the `EV3` class does not provide any method for interacting with sensors and motors: the `Api` class does; and objects of type `Api` cannot be freely constructed - this is a crucial design point of the whole library. An object of type `Api` is passed to the `Consumer<Api>` callback being passed to the `EV3.run()` method by the programmer as the robot *main* function: no other legal way of obtaining an object of type `Api` exists, which is a strong type-driven principle that discourages error-prone programming approaches. Pattern 3.2 in section 3 shows its details.

2.4 Concurrency in the Safe Way

In Legodroid concurrent computations occur often and are wrapped by *future* computations [40], also known as *promises* [21]. The implementations extends Android `FutureTask` [17], which gets lazily evaluated as the `get()` method is invoked and caches the result for subsequent calls. In class `Api`, method `execAsync()` belongs to the mid level API and is the basic primitive for performing automatic future computations in a type-safe way. It executes the callback argument of type `Callable<T>`[4] with the default Android single-thread serial executor [16] and returns a `FutureTask<T>`, i.e. a delayed computation over a value of type `T`. The type of `execAsync()` can be formally described with the following polymorphic type scheme [24]:

$$\texttt{exeAsync} : \forall\alpha.\ (\varnothing \to \alpha) \to \texttt{Future<}\alpha\texttt{>}$$

Thanks to Java 8+ lambdas [34] [39] wrapping any code block into a no-argument closures is syntactically convenient, that is why only callbacks of type `Callable<T>` are supported. All methods communicating with the EV3 brick, such as `getPercentValue()`, internally call `execAsync()` for decoding the data contained in the `Reply` at the byte level.

This mechanism for processing replies to command requests is related to how `SpooledAsyncChannel` works, converting a synchronous `Channel` into an asynchronous `AsyncChannel` by spawning an `AsyncTask` in the background which

[4] The functional interface `Callable<T>` represents functions with no arguments and a result of type `T`, in the same way as `Supplier<T>` does, whose functional type can be written as $\varnothing \to \texttt{T}$.

acts as a spooler service, hence `SpooledAsyncChannel` : `Channel` $\rightarrow$ `AsyncChannel`. The background thread constantly reads from the underlying synchronous channel and dispatches incoming replies to the right *owner*; where an owner is a `FutureReply` object pushed into a synchronized queue at the moment the original command has been sent. When a reply is received, the spooler checks its sequence number and notifies the relevant `FutureReply` object, triggering its computation - such computations are those created by `execAsync()`.

The decoding of replies is performed asynchronously by dozens of short-living threads belonging to the Android thread pool: any input operation, from reading sensors to moving motors, is wrapped within a future computation - in other words, the whole high level API returns objects of type `Future<`τ`>`, for some type τ. From the user's perspective, blocking calls to `Future.get()` are required to retrieve any result; subsequent calls do not trigger the computation again, enabling a form of lazy evaluation [25]. Virtually, by postponing `Future.get()` calls to the very point where the result is needed may lead to minor performance gains due to massive concurrency, depending on the level of support from the Dalvik virtual machine [11] for fine-grained future computations [45] [44].

Admittedly, other systems in literature use a more sophisticate approach to programming with futures, for instance by generating transparent proxies for delayed computations [30], whereas Legodroid proposes a lighter-weight library based solution. Our point is: how realistically useful is a solution relying on external tools, analyzers and code generators that time makes obsolete and incompatible with the ever-changing technology underlying Java and its world? A pure library surely cannot yield the same results, but we believe it is a good compromise of usability and safety that arguably has a better chance to survive the test of time.

2.5 Generalized EV3

Class `EV3` is actually a subclass of the more general `GenEV3<A>` parametric class, where `A` $\preceq$ `Api`. This mixes generics with subtyping, object-orientation with generic programming, in order to achieve extensibility and type safety at the same time [3]. Nested static types `InputPort`, `OutputPort` and `Api` are defined within `EV3` rather than `GenEV3` for simplicity and name brevity, hiding the superclass to casual users. `EV3` instantiates the type parameter `A` with the `Api` concrete type, which is enough for most applications; programmers willing to extend the `Api` class with additional custom methods are allowed do so by applying the subclass as type argument to `GenEV3<A>`. The sample below shows how:

```java
public class MainActivity {
    static class MyApi extends EV3.Api {
        MyApi(GenEV3<? extends EV3.Api> ev3) {
            super(ev3);
        }
        public void myAdditionalCommand() {
            // implementation
        }
    }

    protected void onCreate(Bundle b) {
        BluetoothConnection conn =
            new BluetoothConnection("MyEV3Brick");
        BluetoothChannel ch = conn.connect();
        GenEV3<MyApi> ev3 = new GenEV3<>(ch);
        ev3.run(this::legoMain, MyApi::new)));
    }

    private void legoMain(MyApi api) {
        api.myAdditionalCommand();
    }
}
```

Since the `legoMain()` function passed to `GenEV3.run()` picks a parameter of type `MyApi`, calls to additional methods are allowed without any cast or undisciplined pattern.

This generalization comes at the cost of providing one extra argument to `GenEV3.run()`: a function of type `GenEV3<A>` $\rightarrow$ `A`, required for constructing the object of type `A` to be passed to the *lego main*. Such second argument could be omitted if the Java type system supported *constructor constraints*: type parameter `A` could have been constrained to be *constructible* from an object of type `GenEV3<A>`. Alternatively, lambdas can replace this missing feature at the cost of explicitly passing a constructor reference, like `MyApi::new` in the example [28]. This is also a preferable solution than a factory, which would otherwise require additional classes and verbosity [12].

2.6 `legodroid.lib.plugs`

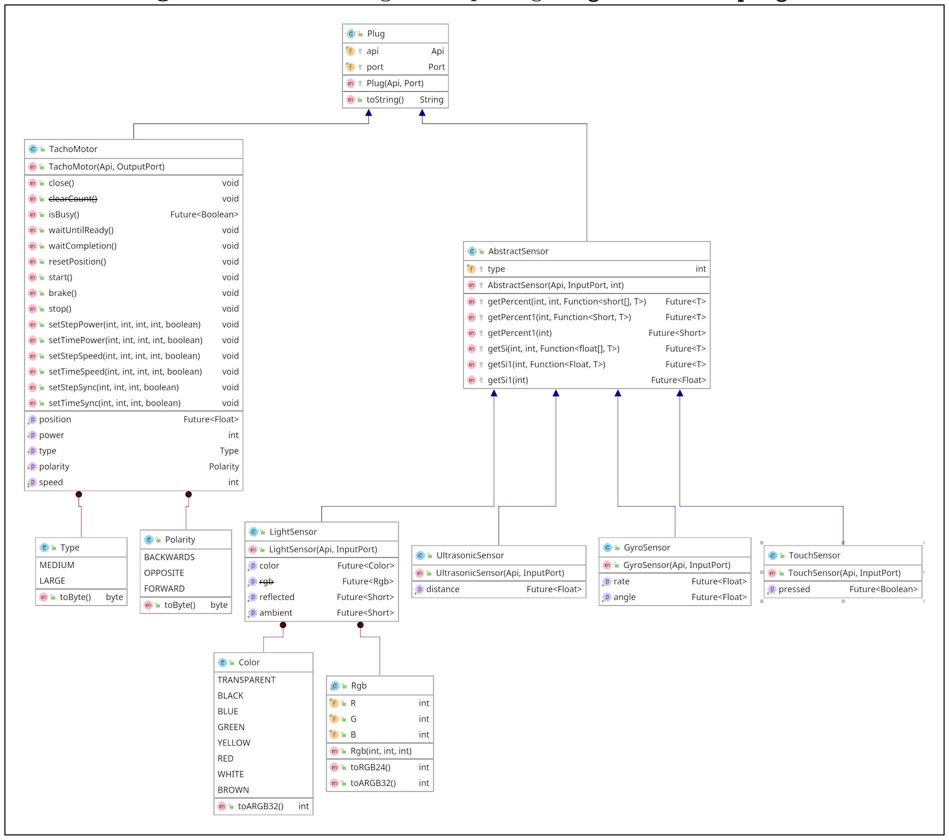
Figure 3 shows the classes representing sensors and motors. Each one exposes methods for reading sensors (`GyroSensor`, `LightSensor`, `TouchSensor`) or moving motors (`TachoMotor`) in a typed way:

1. ports are distinct enum types (`EV3.OutputPort`, `EV3.InputPort`);
2. minor flags representing motor polarity and type[5] are enum types as well;
3. all sensor and motor classes inherit from a common superclass `Plug<P>` where `P` represents the port type: this makes subclasses instantiate the generic `P` with some concrete type at inheritance time;
4. sensors inherit from a common abstract class `AbstractSensor`: protected methods `getPercent()`, `getPercent1()`, `getSi()` and `getSi1()` are commodities for quickly implementing actual sensor subclasses.

   The classes included in this package behave as *handles* for accessing LEGO accessories, such as sensors and motors, connected to the I/O ports. As explained in section 2, extending the library for supporting new accessories requires extending the `Plug` abstract class (or even `AbstractSensor`) and implementing the relevant communication primitives in function of the mid-level methods mentioned above.

[5] Refer to the documentation of EV3 firmware instructions (op-codes `opOutput_Polarity` and `opOutput_Set_Type`) in section 4.9 of [37].

Fig. 3. UML Class Diagram of package `legodroid.lib.plugs`



3 Type-driven Patterns

This section describes in detail the most interesting type-driven programming patterns used in Legodroid. These are the result of combining design patterns existing in literature and in the industry with the formal and sound approach of the functional language scientific community.

3.1 Objects as Evidences

Using objects as *evidences* for constructing other objects is a type-driven practice any strongly typed language can benefit of.

Pattern 31 (Objects as Evidences)**.** *In order to ensure that a given set of operations O becomes available only after some state S_k has been reached within a sequence of increasingly mutating states S_1 .. S_n such that $1 \leq k \leq n$ and $n > 0$, the following pattern can be followed:*

- *operations O can be translated into methods of a stateless object of type* `O`*;*
- *each state S_i can be translated into an object of type $\mathtt{S}_i$ for $i \in [1, n]$:*
  - *each object $\mathtt{S}_i$ holds the information for the state S_i;*
  - *an object of type $\mathtt{S}_i$, for $i > 1$, can only be constructed by providing an argument of type $\mathtt{S}_{i-1}$, i.e. the previous state;*
  - *the initial state S_0, implemented by an object of type $\mathtt{S}_0$, can be constructed explicitly from scratch;*
- *objects of type* `O` *can only be constructed given an argument of type $\mathtt{S}_k$.*

Each object constructor formally behaves like a function $\mathtt{S}_i : S_{i-1} \to S_i$, where only S_0 is initially given and no alternative way of instantiation exists for the remaining states.

In Legodroid this type-driven practice emerges from the type architecture:

- in order to access EV3 motors and sensors, the programmer needs an object of type `Api`;
- an object of type `Api` can only be obtained as argument passed to the callback of type `Consumer<Api>` the programmer passes to the `EV3.run()` method;
- an object of type `EV3`, which represents a physical EV3 brick connected to the Android device, requires an object of type `AsyncChannel`, which provides the I/O primitives for asynchronous communication between Android and the brick itself;
- `AsyncChannel`'s can be created given a synchronous `Channel` by using the `SpooledAsyncChannel` class, which can be seen as a function of type `Channel` $\to$ `AsyncChannel`;
- a `Channel` can be created by calling `Connection.connect()`: class `BluetoothConnection` implementing `Connection<BluetoothChannel>` behaves like a *factory* for producing objects of type `BluetoothChannel`, which are synchronous channels;

– a `Connection` requires the target peer in order to be constructed, e.g. `BluetoothConnection` requires the name the EV3 brick physically uses for pairing with Bluetooth, whereas `WifiConnection` requires the IP address of the brick.

Each step in this chain represents a state S_i in our generalized pattern 31: the initial state S_0 here is a string constant and subsequent states $S_1..S_3$ are represented by each object in the flow:

$$\begin{tikzcd}
S_0 \arrow[d] & = & \texttt{String} \arrow[d] \\
S_1 \arrow[d] & = & \texttt{Connection} \arrow[d] \\
S_2 \arrow[d] & = & \texttt{Channel} \arrow[d] \\
S_3 \arrow[d] & = & \texttt{EV3} \arrow[d] \\
S_3(\lambda O.\ M) & = & \texttt{EV3.run((Api api) -> } M\texttt{)}
\end{tikzcd}$$

The final step is different: we do not consider it as a further state S_4, but rather as a dictionary of operations O that requires state S_3 and is eventually passed to the *lego main* block M. In Java the chain above is implemented by the following code:

```
String name = "MyEV3BrickName";
Connection conn = new BluetoothConnection(name);
Channel ch = conn.connect();
AsyncChannel ach = new SpooledAsyncChannel(ch);
EV3 ev3 = new EV3(ach);
ev3.run((Api api) -> /* lambda body M */);
```

The last line introduces the next pattern.

3.2 Limiting Access to a Resource

Let us assume a general scenario where a given resource R has to be accessible in a restricted environment, a mere binding does not meet the robust discipline we are looking for, as once an object has been bound to a variable name there is no mechanism for unbinding it from the environment. Callbacks and lambdas are an effective tool for controlling the scope of a resource R, since the application of the argument R is performed and controlled by its owner.

Pattern 32 (Lambdas for Scoped Access)**.** *Users willing to access a resource R must provide a callback f by either defining a lambda, an anonymous class or a functional object*[6] *parametric over the resource R. The owner applies R to f and can control what happens before and after the function application.*

The following example shows a minimal implementation of this general pattern (not an excerpt of Legodroid):

[6] In Java 8+ all the mentioned language constructs are equivalent type-wise.

```java
public class FunProxy<R> {
    private R resource;
    public FunProxy(R resource) {
        this.resource = resource;
    }
    public <T> T perform(Function<R, T> f) {
        // do something before
        T result = f.apply(resource);
        // do something after
        return result;
    }
}
```

Admittedly, this pattern does not prevent the user from saving the pointer to the protected resource for later use outside of its controlled scope. The Java compiler detects the simplest scenario and forbids assignments to plain variables captured by a closure:

```java
String copy;    // target pointer in closure
FunProxy<String> p = new FunProxy<>("MyString");
p.perform((res) -> { copy = res; })  // illegal!
```

However, a well-known workaround exists: putting the copy inside the cell 0 of a `final` array of size 1 [15]. We believe such a syntactic overhead discourages most programmers though.

This pattern has a number of applications, ranging from synchronizing access to a resource by locking and unlocking a mutex, to trapping exceptions by surrounding the application with a try-catch block (`Prelude.trap()` in package `legodroid.lib.util` is an example). Even without performing any operation, the callback alone is useful: assume the dictionary of operations O described in pattern 31 is our resource R, then we can merge the two patterns for restricting the use of `Api` objects to the scope of a callback.

4 Experimental Results

Legodroid is open source and available on GitHub at the following URL:
`https://github.com/alvisespano/Legodroid`

The repository includes an Android Studio project with two modules: the library and a demo app showing the main patterns and features.

We extensively tested the library with undergraduate students of the Software Engineering course[7] Over 100 students divided into small teams of 3-5 people produced 17 Android apps performing complex interactions with LEGO Mindstorms devices. The LEGO physical devices created by each team ranged from wheeled machines capable of processing the environment via sensors and avoiding obstacles, to printers capable of moving a pen up and down on a scrolling paper, rendering an input image with dots, to automatic equation solvers printing each reduction step on a paper.

[7] Bachelor degree in Computer Science, year 2018-19, at DAIS, Università Ca' Foscari Venezia.

Teams using Legodroid were 7 out of 17; the remaining teams either used leJOS (8 teams) or flashed the brick with a custom firmware (2 teams).

Anyhow all teams had to interact with the EV3 from an Android device in a non-trivial way. Apps were supposed to exploit the additional computational power of Android, though depending on how teams designed their system this requirement has been more or less fulfilled. A few teams wrote entire algorithms with leJOS, reducing the Android contribution to a remote UI; others exploited the Android mobile, moving robot logic to the Android code to varying degrees.

We observed that teams using Legodroid reported a smoother development experience compared to those working with leJOS. The type-driven patterns offered by the library were positively received by all teams. Advanced coding practices have not been entirely understood by the majority, which is reasonable for a junior developer, though this did not prevent them to take full advantage of those patterns. This is a strong point in favour of type-driven programming: types impose such a strict, yet clear, discipline that any possible misuse of the API is rejected by the compiler, putting the development process on a rail that forcefully led teams to the correct implementation. This sense of guidance and safety has been particularly appreciated by our teams and arguably counterbalances the lower understanding due to type complexity.

The comfortable programming style allowed teams to put more effort into smarter algorithms and advanced interaction. The same did not apply to leJOS users, who had a harder time implementing their applications, even though leJOS wraps low level operations as much as Legodroid does, offering comparable levels of abstraction. Android makes the difference here: its programming pattern requires extra care when managing object references due to complications arisen from activity life-cycle [18]. This impacted those apps interacting with leJOS: despite the longer development time, teams using leJOS could not finalize their code, presenting runtime bugs and weird behaviours at different levels.

Teams reported their appreciation for how Legodroid immutable data types can be constructed and reconstructed safely, in a stateless fashion that fits how Android transitions through different construction/destruction phases. Positive feedback was given to the *lego main* control flow, which is straight, in the likes of leJOS native code and unlike the respective Android side, which consisted in fragmented code spread throghout many callbacks. The event-driven style was often used, which is less than ideal for encoding long algorithms and introduce bugs that could arguably be avoided by a type-driven discipline.

5 Conclusions

We presented Legodroid, an Android library for interacting with LEGO Mindstorms devices through type-driven programming patterns that guide the user into writing robust code. This cross-programming practice has been put on test by a number of junior teams designing and implementing several original LEGO systems exploiting the Android platform capabilities in a non-trivial way. Teams using Legodroid reported a smoother coding experience compared to teams using alternate solutions such as `leJOS`, as our library imposes a type-driven style that enhances the development process in various ways, making sophisticate interaction between the mobile device and the robot seamless and sound.

References

1. Hendrik Pieter Barendregt. Functional programming and lambda calculus. In *Formal models and semantics*, pages 321–363. Elsevier, 1990.
2. Joshua Bloch. *Effective java*. Pearson Education India, 2016.
3. Gilad Bracha. Generics in the java programming language, 2004.
4. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *Acm sigplan notices*, 33(10):183–200, 1998.
5. Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN Notices*, volume 48, pages 133–144. ACM, 2013.
6. Edwin Brady. *Type-driven Development With Idris*. Manning, 2016.
7. Luca Cardelli. Typeful programming. *Formal description of programming concepts*, pages 431–507, 1991.
8. Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.
9. Patrice Chalin, Perry R James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software*, 2(6):515–531, 2008.
10. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, volume 82, pages 207–212, 1982.
11. David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4(8), 2010.
12. Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.
14. Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2006.
15. Brian Goetz. *State of the Lambda 4th Edition*, 2011. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html`.
16. Google Inc. *Android Executor Interface Documentation*, 2012. `https://developer.android.com/reference/java/util/concurrent/Executor`.
17. Google Inc. *Android FutureTask Class Documentation*, 2012. `https://developer.android.com/reference/java/util/concurrent/FutureTask`.
18. Google Inc. *Understand the Activity Lifecycle (Android documentation)*, 2014. `https://developer.android.com/guide/components/activities/activity-lifecycle`.
19. Google Inc. *Android Studio download page*, 2019. `https://developer.android.com/studio/`.

20. Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *European Conference on Object-Oriented Programming*, pages 520–545. Springer, 2009.
21. Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. *URL: http://docs. scala-lang. org/overviews/core/futures. html*, 2012.
22. Justin Kestelyn. *How Apache Spark, Scala, and Functional Programming Made Hard Problems Easy at Barclays*, 2015. `https://blog.cloudera.com/blog/2015/08/`.
23. John McClean. *Lambdas are not functional programming*, 2018. `https://medium.com/@johnmcclean/lambdas-are-not-functional-programming-63533ce2eb74`.
24. Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, pages 217–228. Springer, 1984.
25. Dung Nguyen and Stephen B Wong. Design patterns for lazy evaluation. In *ACM SIGCSE Bulletin*, volume 32, pages 21–25. ACM, 2000.
26. Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *POPL*, 1997.
27. Oracle Corp. *JDK 8 Documentation: Package java.util.function*, 2010. `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`.
28. Oracle Corp. *Method References (Java Documentation)*, 2017. `https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html`.
29. Tomas Petricek. *Type-First Development*, 2015. `http://tomasp.net/blog/type-first-development.aspx/`.
30. Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 206–223, New York, NY, USA, 2004. ACM.
31. Mark Seemann. *Type Driven Development*, 2015. `https://blog.ploeh.dk/2015/08/10/type-driven-development/`.
32. Mark Seemann. *Type Driven Development*, 2015. `https://blog.ploeh.dk/2016/02/10/types-properties-software/`.
33. Anton Setzer. Java as a functional programming language. In *International Workshop on Types for Proofs and Programs*, pages 279–298. Springer, 2002.
34. Venkat Subramaniam. *Functional programming in Java: harnessing the power of Java 8 Lambda expressions.* Pragmatic Bookshelf, 2014.
35. The Lego Group. *LEGO Mindstorms NXT Education Kit*, 2011. `https://www.generationrobots.com/media/Lego-Mindstorms-NXT-Education-Kit.pdf`.
36. The Lego Group. *EV3 Communication Developer Kit*, 2013. `https://le-www-live-s.legocdn.com/sc/media/files/ev3-developer-kit/lego%20mindstorms%20ev3%20communication%20developer%20kit-f691e7ad1e0c28a4cfb0835993d76ae3.pdf`.
37. The Lego Group. *EV3 Firmware Developer Kit*, 2013. `https://le-www-live-s.legocdn.com/sc/media/files/ev3-developer-kit/lego%20mindstorms%20ev3%20firmware%20developer%20kit-7be073548547d99f7df59ddfd57c0088.pdf`.
38. The Lego Group. *ROBOLAB Reference Guide*, 2013. `http://www.legoengineering.com/robolab-programming-references/`.
39. Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming.* " O'Reilly Media, Inc.", 2014.

40. Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 439–453, New York, NY, USA, 2005. ACM.
41. Scott Wlaschin. *Designing with Types*, 2013. `https://fsharpforfunandprofit.com/series/designing-with-types.html`.
42. Scott Wlaschin. *Designing with Types: Making illegal states unrepresentable*, 2013. `https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable`.
43. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.
44. Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.
45. Lingli Zhang, Chandra Krintz, and Sunil Soman. Efficient support of fine-grained futures in java. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*. Citeseer, 2006.
46. Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, et al. Object and reference immutability using java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 75–84. ACM, 2007.