Collectors

Steven Costiou, Mickael Kerboeuf, Alain Plantec, Marcus Denker

Seeking and Finding Objects at Runtime with Collectors

S. Costiou
Lab-STICC, UMR CNRS 6285
Université de Brest, France, France
costiou@univ-brest.fr

M. Kerboeuf
Lab-STICC, UMR CNRS 6285
Université de Brest, France, France
kerboeuf@univ-brest.fr

A. Plantec
Lab-STICC, UMR CNRS 6285
Université de Brest, France, France
plantec@univ-brest.fr

M. Denker
RMoD, Inria Lille, UMR CNRS 9189, CRIStAL
Université de Lille, France
marcus.denker@inria.fr

ABSTRACT

Observing and modifying object-oriented programs often means interacting with objects. At runtime, it can be a complex task to identify those objects due to the live state of the program. Some objects may exist for only a very limited period of time, others can be hardly reachable because they are never stored in variables. To address this problem we present the Collectors. They are dedicated objects which can collect objects of interest at runtime and present them to the developer. Collectors are non-intrusive, removable code instrumentations. They can be dynamically specified and injected at runtime. They expose an API to allow their specification and the access to the collected objects. In this paper, we present an implementation of Collectors in Pharo, a Smalltalk dialect. We enrich the Pharo programming and debugging environment with tools that support the Collectors *API*. We illustrate the use of these *API* and tools through the collection and the logging of specific objects in a running *IOT* application.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software prototyping*; Object oriented frameworks; Integrated and visual development environments; Object oriented development;

KEYWORDS

Object Oriented Debugging, Debugging, Object Tracking

ACM Reference Format:
S. Costiou, M. Kerboeuf, A. Plantec, and M. Denker. 2018. Seeking and Finding Objects at Runtime with Collectors. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Interacting with running programs is interesting for many reasons, for example monitoring applications, reverse engineering, debugging... For such interaction, the first abstraction we can look for is the source code, that we can modify to see changes of behavior and get feedback. But at runtime, we may want to interact with abstractions closer to our programming paradigm. In object-oriented programming we would look for objects, as they are the main entities that structure our programs. Behavior is expressed through the classes of these objects, in which methods are defined. Within the control flow of these methods, we will find other objects referenced by instance variables or temporary variables. It is tempting to adopt a pure *object-centric* view [14]. We would see these variables in the source code as objects that we can interact with. In the end, we would deal with objects instead of using exclusively source code.

From this point of view, imagine a developer who is debugging a running program, and is looking to its source code. One of the objects in particular is of interest and should be, for example, logged somewhere. The developer could look for instances of this object's class in the running program – which would be a very tedious task if performed manually. He/She could also look where this class is instantiated in the code, then dig into the running program's structure and try to find objects of interest. This can also be very tedious if multiple objects of different classes should be targeted, and even impossible for objects with very short lifespans if the program is still running. Moreover, it becomes a bit more complex in dynamically typed systems, like Smalltalk, Python or Ruby. In programs written in these languages, there is no guarantee that a reference in the code will always be of the same type [11]. The developer could also try to locate positions of interest in the code, *e.g.* after a temporary variable is assigned, and insert code that somehow mark the object, so that it can easily be retrieved while the program is executing. This could be very intrusive, especially if the inserted code should check for conditions before marking the object. It also makes it unpractical: What if the developer wants to change the condition? How to ensure that the base code is not polluted by manual instrumentations that could be forgotten by the developer?

To help the developer interact with objects from the variables in the source code, we introduce the Collectors. Collectors are first-class objects that target entities in the source code. Objects referenced by these targets will be collected at runtime, and presented to the developer. Collectors provide a non-intrusive way of gathering objects during the program execution and to interact with them. They are available through an API that provides means to:

(1) Dynamically specify in the control flow which object must be collected

<Programming'18> Companion, April 9–12, 2018, Nice, France
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(2) Scope the collection of objects to specific conditions or to specific entities (classes or objects)
(3) Provide references to the collected objects and means to interact with them at runtime

The contribution of this paper is the description of the Collectors solution and its *API*. On top of this API, we describe and show an implementation of the Collectors tool-set in Pharo Smalltalk [1]. It provides integrated support for the *object-centric* views and interactions brought by Collectors. It is completed by an illustration of object collection and logging in a running, remote *IOT* application.

The paper is organized as follows: Section 2 explains the problem through a simple example. Section 3 introduces Collectors. Section 4 demonstrates Collectors through a simple *IOT* application and Section 5 discusses the Collectors solution and its limitations. Sections 6 explores other possibilities and related work. We conclude in Section 7 with working directions and a summary of the Collectors proposition.

2 THE PROBLEM

We consider the case of running object-oriented programs, for which the developer needs to interact with specific objects. This need may come from various reasons, for example debugging. A developer may want to find objects of interest, *e.g.* to change their behavior or to log their state. Looking for these objects can be a problem because this need appears at runtime. A first research could provide some results, but it may need to narrow down this search to find more specific objects. That is often the case while debugging, when exploring a runtime behavior to find bugs. Furthermore, the objects found may have been changed in some way by the developer, *e.g.* through behavior adaptation. In that case they must be referenced somewhere if the developer wants to revert these changes in the future. In the following, we describe and emphasize this problem through an example of code written in a Smalltalk fashion.

2.1 The *IOT* application example

Let us imagine an *IOT* application on a device with a temperature sensor and a light sensor. The application has a main class *IotApp* which runs a loop. Every second in this loop, the *IotApp* reads the temperature and the light from the sensors. The *GenericSensor* class models common behaviors and interface of all sensors. It implements a method *#readChannel:*, which queries the sensor and returns an interpreted physical sensor read. As shown in Listing 1, this method takes an object as a parameter.

A physical sensor read is performed by an instance of class *SensorRead*, in the *#read:* method. This method reads a physical address, interprets the raw value if necessary, then returns the value. In the code above, *channel* is declared as a temporary variable (first line). At the end of the method, the ^ symbol returns the result of the sensor read. The *GenericSensor* is specialized into two subclasses: *TemperatureSensor* and *LightSensor*. These classes model temperature and light sensors that use the *#readChannel:* method from their superclass. Instances of these class are referenced by two instance variables in the *IotApp* class: *temperatureSensor* and *lightSensor*.

Listing 1: The GenericSensor class

```
Class GenericSensor>>readChannel: index
  | channel |
  channel := self channelOfIndex: index.
  ^ SensorRead new read: channel
```

2.2 The problem

This application is running on a remote device. Let us suppose that we have a remote access to the application. We would like to log the state of specific objects, so we need to find those objects. Pharo Smalltalk [1] for example provides reflective capabilities and tools to interact with running programs. It allows the developer to find references to objects. We could send the message *#allInstances* to a class to recover all instances of this class. Then it would be possible to find one of these instances, to inspect its structure, its relations, its state and find the objects we are interested in. It seems granted that any object could be found and modified at runtime. However, we expose below three cases for which introspection capabilities do not suffice to find objects of interest in a running program: *the limited lifespan object* case, *the "anonymous" object* case and *the multiple instances* case.

The limited lifespan object case. We call *"limited lifespan objects"* entities that are non-permanent objects or states, that exist only for a very limited amount of time before being garbage collected. It makes it impossible for the developer to find those objects by inspecting the state of another object at runtime. Typically, these objects are referenced by temporary variables, objects instantiated within a local scope, and instance variables of these latter objects. This case is illustrated in Listing 1 where the *channel* temporary variable never exist long enough to be inspected. We can also see in Listing 1 that the lifespan of instances of *SensorRead*, and by extension their instance variables, are scoped by the *#readChannel:* method execution.

The "anonymous" object case . Such *"anonymous"* objects are instantiated objects to which cascaded messages are sent and the execution result is returned without storing the object in any temporary state. This is the case in Listing 1, where an instance of *SensorRead* is created and never stored. There is no variable to inspect, and the developer would have to halt the system and inspect the stack to find the object.

The multiple instances case . In this application there could be other sensors than the temperature sensor. All these sensors would be modeled by subclassing the *GenericSensor* class, and all objects would share the *#readChannel:* method from Listing 1. Modifying the method will impact all objects sharing it. But when probing the system, we may be interested only by the *SensorRead* objects created by instances of *TemperatureSensor*, and not all existing instances using the method throughout the execution of the application.

2.3 Why it is not trivial

Imagine that the developer wants to do the following: Each time a *SensorRead* object is created by the temperature sensor, then its state is logged in a console. He/She will be facing the three problematic cases described above. First, dynamically inserting logging instructions in the *#readChannel:* method where the objects are created (Listing 1) will not work, as a modification of this method will impact all instances of classes using it and not only the temperature sensor. Second, the *SensorRead* objects have limited lifespans in this method. Getting them manually by inspecting the program is excluded. Finally, those objects are never stored so that there are no explicit references to them. There are no practical ways to find them with a debugger, or without complex instrumentation of the code. Such instrumentation would imply inserting variables and adapting the code to provide references to objects. Moreover, these operations must not alter the original behavior of the method.

It becomes more complex if the wanted object must satisfy a specific condition, for example: Each time a *SensorRead* object created by the temperature sensor has a value lower than zero then its state is logged in a console. In addition, this condition could dynamically change if the developer wants to narrow it down. And if at some point during execution, the modified behavior of these objects is not necessary anymore, all behavior changes must be reverted. The code we presented as an example is very simple, and yet we can illustrate three cases where it is not trivial to find specific objects at runtime for a simple case of logging.

3 COLLECTORS

In this Section we present the **Collectors**, that allow the developer to dynamically collect objects from the control flow and to interact with them. They provide references to collected objects, so that the developer can freely manipulate them. Collectors are composed of:

- A programming interface to specify which object to collect in the control flow and how to interact with them
- A set of tools to help viewing the source code as objects references with which we can interact

Objects that can be collected are (1) objects referenced by temporary variables, (2) by instance variables or (3) receivers of specific messages in the control flow. These kind of collection can be performed either through the Collectors tools or through their programming interface. Collectors can be specified either before or during runtime. The following subsections describe the Collectors tools, API, model and its implementation for object-oriented languages.

3.1 Collectors model

Collectors are first-class objects. They can be installed and released at runtime through their API. Their purpose is to collect references of objects at strategic places in the control flow of methods. How this collection is done depends on how it is specified for each collector. One collector can collect either:

- Objects referenced by a temporary variable, each time it is accessed
- Objects referenced by a slot, each time it is accessed
- Objects receiving a specific message

For a given collector, object collection is performed each time the collection condition is met within the specified control flow. An object collection is permanent, unless the collector is released by the developer, or the object is garbage collected or the collector specification changes. In that latter case, all objects are released before the collection can start again. It is the case if the developer adds a condition or a scope to the collector.

Collectors cannot be composed themselves, *i.e.* they can target only one of the items enumerated above. A Collector collecting an instance variable cannot collect temporary variables. However it is possible to compose a global object collection strategy by creating multiple collectors.

Objects are released if reclaimed by the memory handler (garbage collector, manual release...). Collectors only hold weak references of objects, and never interfere with the program memory management. Interaction is only possible during the object lifetime. Similarly, a single object cannot be collected more than once. If a collector attempts to collect an object already collected, then the reference that it holds is not duplicated.

3.2 Collectors API

In the following we describe the Collectors programming interface in a Smalltalk fashion. The *Collector* class provides the global interface to define collectors, while its instances provide means to interact with the collected objects. The API can be used both at development time and at runtime. Collectors specification and modification is dynamic, provided the host language features means to interact with the running program.

Specifying which entity to target. A collector can collect either (1) objects referenced by a temporary variable, (2) by an instance variable or (3a, 3b) by the receiver of a message. The result of each API call returns a collector object. It is stored in a global collection and can be accessed at any time. In the following, the words preceded by a *'#'* character are parameters passed to the API. The *class* keyword means these methods are static class methods.

```
dummyMethod
1  | temp col |
2  temp := Time now.
3  col := OrderedCollection new.
4  col add: temp.                    ->collect!
5  ivar := col copy.
6  10 timesRepeat: [ temp := Time now.
7        ivar add: temp ]
```

Figure 1: Collecting objects referenced by a temporary variable in a method.

(1) Collecting objects referenced by a temporary variable, within a single method. It is performed each time the instance variable is accessed (read/write).

```
Collector class>>
    collectTemporary: #tempName
    in: #methodName
    fromClass: #className.
```

Figure 1 shows how objects referenced by a temporary variable named *temp* are collected in a dummy method. Each time the temporary variable is accessed, the object it references is collected (line 2, 6 and 7). The collection is performed after the variable has been read or written. On a write access, the object is collected after the value is stored in the variable. In that case, it is that new value that is collected (lines 2 and 6). If the variable accessed references an object that has already been collected, the object is not collected twice. We can see this case at line 6 and 7), where a collection is attempted two times in a row for the same object stored in *temp*.

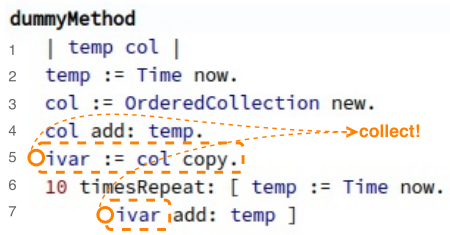

Figure 2: Collecting objects referenced by an instance variable.

(2) Collecting objects referenced by an instance variable. It is performed each time the instance variable is accessed (read/write).

```
Collector class>>
    collectInstVar: #ivarName
    fromClass: #className.
```

Collecting an instance variable follows the same pattern as the temporary variable collector. All using methods of the target instance variable will trigger object collection after a read or a write to this variable. This collection pattern is illustrated in Figure 2, where the *ivar* instance variable is collected.

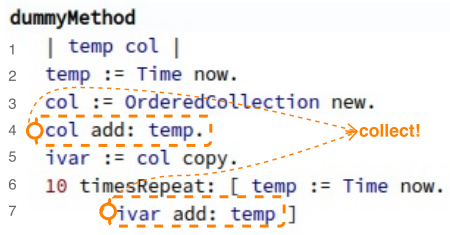

Figure 3: Collecting objects to which the *#add:* message is sent in a method.

In that case, the *ivar* instance variable will also be collected each time another method accesses it.

(3a) Collecting objects to which a message is sent, within a single method. Every object receiving the message is collected.

```
Collector class>>
    collectReceiversOf: #messageName
    in: #methodName
    fromClass: #className.
```

Figure 3 shows the collection of objects receiving an *#add:* message. Like the instance and temporary variable collectors, the collection is performed after the message is received by the object. All objects receiving the specified message within the method body are collected when the code executes, regardless of their class (lines 4 and 7).

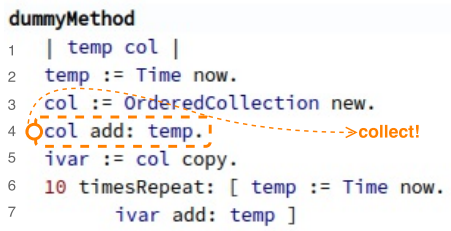

Figure 4: Collecting objects to which the *#add:* message is sent at a specific position in the code of a method.

(3b) Collecting objects to which a message is sent, within a single method. Only a target object receiving the message within a single statement is collected. This collector is specified by passing an abstract syntax tree node (#anAstNode) representing a message send. This interface is less practical to use as is, because the developer has to dig into the AST of a method to find the appropriate AST node.

```
Collector class>>
    collectReceiverOfNode: #anAstNode.
```

Figure 4 shows the collection of objects receiving a message at a specific place in the control-flow. Other objects receiving the same message but elsewhere in the control-flow are not collected. Object collection is also performed after the message is received, and regardless of the receiver's class.

Accessing and releasing collectors. All collectors present in the system can be retrieved at once, and the developer can browse this list and select a particular collector. Collected objects are stored in weak references. They can be retrieved and enumerated. Finally, the developer can release the collector. It will stop collecting objects and be removed from the Collectors global collection.

```
collectors := Collector allCollectors. "Get all collectors"
collector := collectors first. "Access first collector"
objects := collector collectedObjects. "Collected objects"
collector release "Release collector"
```

Scoping a collector. A collector can be scoped to a single object, if the developer can provide a reference to this object. In that case, object collection is active for the specified object only:

```
collector scopeToObject: anObject.
```

Multiple entities can compose a scope for a given collector. They can be classes or individual objects. Object collection is performed only if one of the scoping entities executes a statement for which a collector is defined.

Defining actions for collected objects. The user can specify an action to be performed each time an object is collected or released. There is only one possible action for the release and for the collection. It takes the form of a block closure with two parameters, namely the collected object and the collector that collected this object:

```
collector onCollectDo: aBlock.
collector onReleaseDo: aBlock
```

As an example, we can configure a collector to print each collected object in the *Transcript* console of Pharo:

```
collector onCollectDo:
  [:collectedObject :collector|
    Transcript crShow: collectedObject printString].
```

Conditioning object collection. A collector can be configured to collect only objects satisfying a specific condition. The condition is evaluated each time a collector attempts to collect an object. The condition takes the form of a block with one or two parameters, namely the object or the object and its context:

```
collector condition: aBlock
```

For example, we can condition the object collection to only collect objects with a specific state:

```
collector condition:
  [:object :context| object class hasSlotNamed: #someState].
```

3.3 Collectors Tools

Collectors provide tools to help the developer specify object collection and interaction. All the tools allow dynamic interaction with the source code representing objects. Object collection can be specified before or at runtime, on a program running locally or on a remote device (*e.g.* an IOT object). The tools presented below are from a Pharo Smalltalk [1] implementation of Collectors.

3.3.1 The control-flow object selector. The developer can ask for object collection through a new contextual menu, which is integrated in the code browser of Pharo. In any method that is browsed by the developer it is possible to define a collector. This menu allows the collection of objects references by instance and temporary variables, and to which a message is sent.

Figure 6 shows the contextual menu spawned in a method from the code browser. In this example, the receiver of the *#read:* message will be collected. This object collection is made through the control flow. Only the object receiving this message at this particular place will be collected.

Other objects receiving the same message within the same method but in another statement will not be collected. The source code displayed in native code browsing tools is not altered by collectors.

3.3.2 The Collectors browser. The Collectors browser provides the developer with a view on all defined collectors. From this view, one can explore collected objects, define actions to perform at collection and at release time, express conditions for object collection and explore entities to which the collector is scoped.

This is illustrated in Figure 5. The browser shows the list of active collectors (A). Collectors can be renamed through a contextual menu to help the developer identify their meaning. Collectors specifications are shown in (B).

For example, in Figure 5, we see that the object that will be collected is the receiver of the *#read:* message after it has been sent. The part of the code representing this object is automatically highlighted by the browser. The collector specification is the only aspect of the collector that cannot be changed dynamically.

To collect objects from another part of the control flow, a new collector must be defined. In (C) we can see the collect and release actions. These actions are performed at collect or at release time. This feature can be used to perform direct actions on collected objects, or to pass them to another mechanism (*e.g.* an adaptation mechanism).

Actions can be defined and changed dynamically, by coding them in the browser and saving them through the contextual menu. The defined actions are applied to the next object collection or release, but not to already collected or released objects. (D) shows a condition for object collection. Objects are collected only if the condition is met. By default the condition is disabled. Like the collect/release actions, the condition can be dynamically changed through the browser editor and only apply for the next object collection. Finally, in (E), we can see entities to which the selected collector has been scoped.

For instance, the collector named *Receiver of #read:* is scoped to an object *aTemperatureSensor* instance of a *TemperatureSensor* class. It means that object collection will be performed only if this object goes through the control flow highlighted in the specification pane (*i.e.* in (B)). This scoping is dynamic and can evolve at runtime, and multiple entities can scope the collector (classes and objects). The browser allows the user to browse these entities through a contextual menu.

3.3.3 Scoping collectors to objects. Collectors add a new pane to the native Pharo inspectors. Upon inspection of an object, it is possible to see the defined collectors in the program. A contextual menu allows to scope the collection operation to the current inspected entity. Once scoped, a collector will perform object collection only if the entity to which it is scoped goes through the adequate control flow. An example is shown in Figure 7. We can see an inspector on an instance of a *TemperatureSensor* class. In the collectors pane we can see all defined collectors. We could, for example, scope the collector defined in Figure 6 to the inspected temperature sensor. This collector will then collect objects only when the temperature sensor object executes the *#readChannel:* method.

3.4 Implementation

We implemented Collectors with Pharo [1] and we rely on the Reflectivity layer [12] to inject non-intrusive annotations on abstract syntax tree of methods. These annotations give us access to reifications of the running program, which are instance or temporary

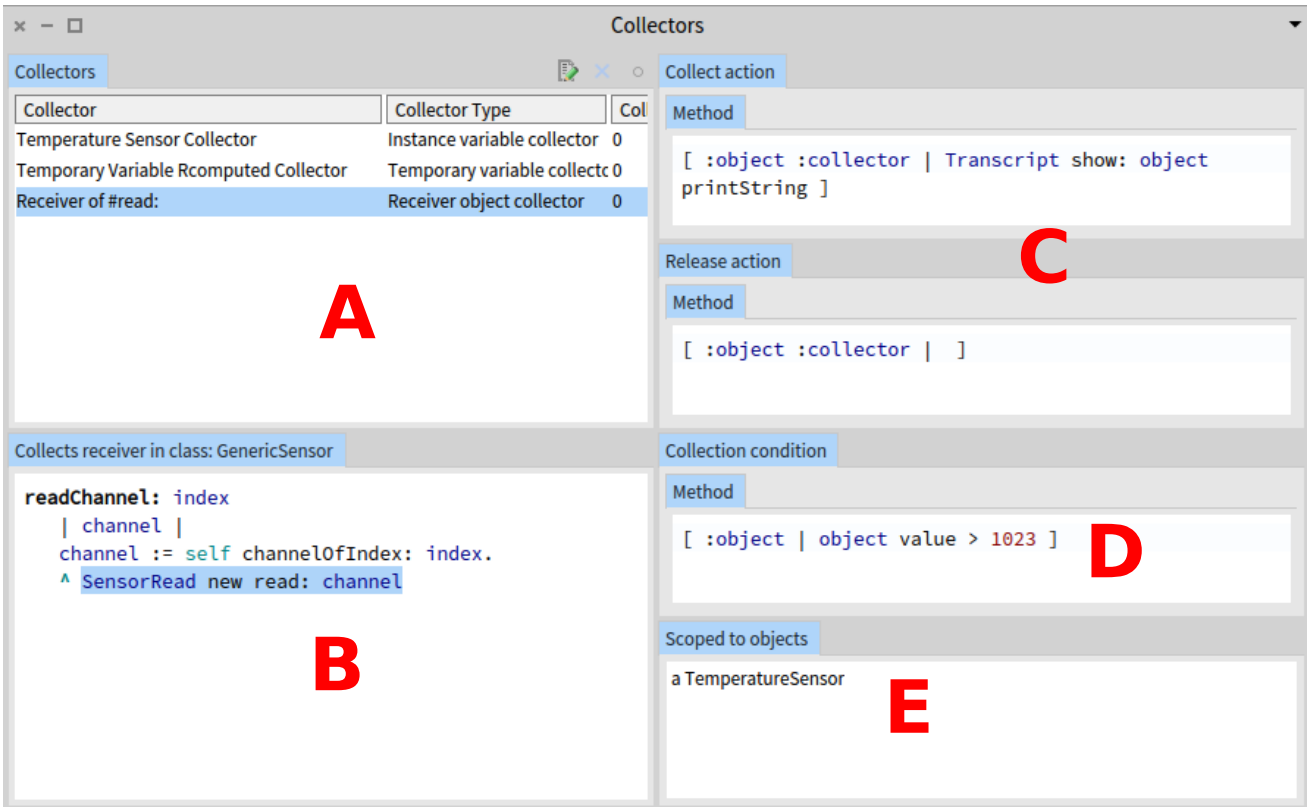

Figure 5: The Collectors browser: (A) collectors list, (B) collectors specification, (C) actions on collection and release of objects, (D) condition for object collection and (E) the entities to which object collection is scoped

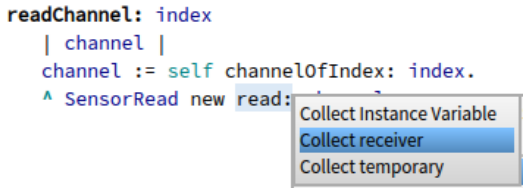

Figure 6: Defining collectors from source code

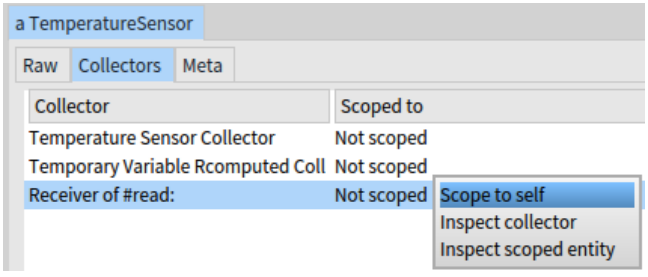

Figure 7: Scoping object collection to a given object

variables and receivers of messages. As we dynamically access these reifications at runtime, we provide them to the associated collector to gather objects of interest.

Methods for which Collectors are specified are recompiled for the collection behavior to be active. However that is transparent to the user, and to the environment tools. There are no visible instrumentations when the user looks at the code. Collectors instrumentations do not interfere with the original behavior of the code. A downside is that if the user manually modifies an instrumented method, the original behavior is restored and the collection behavior is removed.

Alternative to Reflectivity, Aspect oriented techniques could serve as another back-end for a Collectors implementation. We made experiments of such implementation with PHANtom [3]. This implementation is partial and scoped to the collection of objects receiving a given message.

4 EXAMPLE ON AN *IOT* APPLICATION

In the following, we demonstrate the use of Collectors in an *IOT* application. This example features a simple case of debugging where a developer wants to add log on specific objects in the running system. The difficulty, as explained in Section 2, is to identify those objects and to get references to them for interaction. Results and demonstration of the experiment are available online.[1]

[1] https://kloum.io/costiou/collectors

4.1 The Sensor Monitoring Application usecase

As an illustration, we used the Sensor Monitoring Application use-case [9]. It is a single threaded Pharo *IOT* application deployed on a Raspberry Pi[2] in which there is an unpredictable bug at runtime. The bug comes from an erratic sensor read that may happen at any time during the execution of the program. From the original use-case we slightly changed the setup to fit our example, which now uses two sensors (temperature and light). We deployed the code from Section 2 as the main sensor access in the application. As shown in Figure 8, the developer is remotely connected to the application through Telepharo, the Pharo remote debugger. All operations described below are executed at runtime.

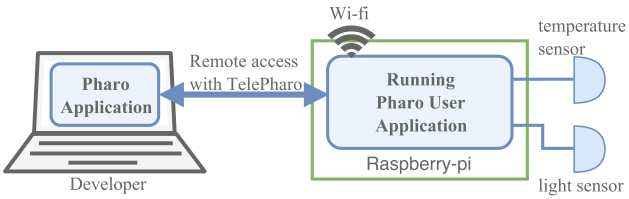

Figure 8: The Sensor Monitoring App

4.2 The unanticipated logging operation

While debugging the Sensor Monitoring Application, we suspect a problem in the temperature sensor readings. So we would like to log *SensorRead* objects in a console when their values become erratic. We would also like to scope this logging to *SensorRead* objects created by the temperature sensor, and not all of them that may be created by other entities in the system. As there is no way to predict when the erratic values may appear it is impossible to know when to start logging. We also cannot distinguish sensor readings created by the temperature sensor from the ones created by others sensors. That is because all sensors share the same method *#readChannel:* from their super class, as described in Section 2. It is complicated to know which are the objects that should be investigated.

4.3 Collecting and adapting objects

In the following, we describe how we used the Collectors API in the IOT App to collect sensor readings objects and to log them into a console. We programmatically use the API, but all of the operations below can be performed in an interactive way using the tools described in Section 3.3.

Collecting all SensorRead objects. We want all instances of *SensorRead* to be logged in a console. They are instantiated in the *#readChannel:* method from the class *GenericSensor* (see Listing 1). To collect them, we have to ask for the receivers of the *#read:* message which is sent just after their creation:

```
collector := Collector collectReceiversOf: #read: in: #readChannel:
    fromClass: #GenericSensor.
```

The collector will collect all objects receiving a *#read:* message in the specified method. Fortunately there is only one receiver of this message in the control flow (Listing 1). If there were many, and if we wanted to collect only one of the receivers in the control flow, we could have used the contextual menu in the class browser to specify this receiver (Figure 6).

Once it has been defined, the collector starts collecting objects. It can be accessed through a local variable if it has been stored by the user, or through the Collectors interface *#allCollectors*. We can then specify a collect action, which will print the sensor reads in the console:

```
collector onCollectDo:
    [:object :collector| Transcript crShow: object printString]
```

Instead of logging, we could apply more complex operations on the collected objects (*e.g.* behavioral adaptation). In that case, we should also specify a release action which will ensure that the released objects are not affected by the collector anymore. However that is not the case in this example.

Collecting only for the temperature sensor. Once specified, the collect action logs every object at collection time. We can scope the collector and its action to the temperature sensor object, which is the sole instance of the *TemperatureSensor* class. Once scoped, the collector will only collect and log sensor reads instantiated by this object:

```
collector scopeToObject: temperatureSensor
```

Note that finding this *temperatureSensor* object can be subject to the same problems from Section 2.2. We are able to find it using the remote inspector of TelePharo, by inspecting the states of the main running application (*i.e.* the sole instance of the *IotApp* class).

Scoping the collection to a specific condition. All temperature sensor readings are not that interesting, as most of them are accurate. We are interested in readings with erratic values, to log their full state and to understand what is wrong. Thus, we specify a condition for object collection. Only sensor readings with values outside of the reading bounds will be collected:

```
collector condition: [:object| object value > 1023
                                or:[object value < 0]]
```

5 DISCUSSION

We successfully used Collectors in a simple use-case of debugging to log the state of specific objects. Collectors were remotely defined in the IOT App from the developer's computer. Objects of interest were collected at runtime on the IOT object, on which was running the app. Condition checks and collect time actions (*i.e.* the logging) were also performed on the IOT object. The resulting traces were dynamically displayed on the developer's computer when objects were collected. Only very specific objects were collected and printed.

In regards to the original use-case [9], Collectors provided dynamic and non-intrusive instrumentations to observe the runtime. We use remote debuggers as a support to remotely control Collectors and get feedback from the collected objects. We thus avoid direct code changes injection, and give the developer flexibility on how to interact with objects of interest. Specifically, collectors can be dynamically scoped to the control flow of specific objects.

[2] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Collection can be subject to conditions, and the developer can perform custom actions at collect time. Furthermore, a collector can be released when it is no longer needed. This ensures that the original program stops collecting objects, and thus stops performing actions at collect time.

The logging demonstration in this paper is only meant to keep the focus on the Collectors main purpose. There could be many applications, from logging and debugging to behavior adaptation of objects. For example, we successfully reproduced the same experiment with Collectors but with an adaptation mechanism [2]. Collectors provided objects to the adaptation mechanism, which applied per-instance logging behavior adaptation to them. In a way, Collectors were a mean of scoping the adaptations to a restricted and dynamic set of objects. In that case, Collectors could be considered as an explicit selection mechanism for *Group-Based Adaptation* [13].

However, the current implementation and model are limited to one thread. Collectors are also inefficient for methods on the stack with running loops. For now, a collector will only be activated once the loop finished and once the method is re-executed. How Collectors affect performances remains to be studied, and how the solution can scale in bigger programs has yet to be evaluated. There are also no fail safe mechanism if an error occurs in the collect action. For now it is the responsibility of the developer to handle errors. However, error handling does not prevent the developer to jeopardize the state of the collected objects. It is not sufficient to guarantee that objects are preserved throughout the developer's actions.

The object focus of Collectors also has the drawback that everything is expressed from and for the object. Other solutions, for example aspects [7], can provide access to a larger context of execution. That particular point is part of a future extension of the Collectors.

6 RELATED WORK

In the following, we discuss the possibilities of acquiring objects through runtime modifications, through the control flow or through direct object references.

One important aspect is the ability to inject removable code instrumentations without manually modifying the code. For Collectors, this comes from the implementation layer, Reflectivity [12]. This is why we also discuss related work from an implementation perspective.

Easing the chase for objects through code update. The example code we showed in Section 2.1 could be improved in a way that it could solve part of or all of the problems we described.

It could be done through refactoring [4], but it would change the structure of the program – which may not be desirable or even possible at runtime.

Furthermore, it might not solve everything in larger, more complex programs. Intrusive techniques, like dynamic software update [15], may also change the program's structure and make it difficult to revert instrumentations.

Control Flow Object Collection. Selecting objects from the control flow means that somehow code has to be dynamically injected to get references. This code would also need to be later removed, when the target objects are no longer of interest. Such technique is used for example in Context Oriented Programming [5] or in *Group-Based Adaptation* [13].

Objects can be explicitly selected by manually inserting code, if the developer already knows in which part of the code these objects can be found. They can also be implicitly selected, when they automatically become part of an active context. Because Collectors are based on Reflectivity[12], they do not require manual insertion or removing of instrumentation. Instead, object collection is explicit and the developer has to define it in the control flow – although it is implemented as hidden instrumentations of the *AST*.

Aspects [7] can provide access to objects at key points in the control flow and trigger method calls on them. However and as far as we know, Aspects mainly provide code as the main interacting abstraction. By themselves, Aspects are a solution on top of which the Collectors could be implemented.

Edit Transactions [10] embeds live modifications into change sets that can be dynamically turned on and off. One could manually design object selection in the control flow of the program, and (de)activate them when needed. It could provide a dynamic object collection at runtime, with live feedback and view on the collected objects. The code instrumentation to gather the objects has to be manually written into the control flown and the developer always sees the instrumentations to collect objects as source code modifications. It is another way of seeing objects from a running program, from a *code-centric* oriented perspective.

Object references. Remote debuggers can provide entry points in programs, *e.g.* to halt the system and open a view on the current stack from which the user can search for references. It has the disadvantage to stop the execution of the inspected thread. Pharo [1] provides Telepharo[3], that has a remote debugger, a remote code browser and a remote scripting interface. With such dynamic tool, it is possible to remotely query for all instances of a given class, and then try to find objects of interest.

However it is highly unpractical, as there are no guarantee about the number of objects it would return. It could be impossible to find a particular object among thousands of references. It could also miss objects or provide false positives because of *ducktyping*, if variables in the code do not always refer to objects of the same types [11]. Also, this technique would not allow to get reliable references on temporaries variables or on objects with a very short lifespan.

Listing references of objects of interest can be an interesting alternative. For example the Chisel framework [6] provides an object store that can reference objects of interest by name. These objects must be manually named, and necessitate the user to profile the application to identify and name them. This kind of solution is also particularly resource consuming, as every object in the program must be logged in a database for inspection by the user.

Implicit selection mechanisms in *Group-Based Adaptation* [13] can automatically find and manage collections of objects satisfying specific conditions. Objects can enter or leave a group, depending on selection criteria. For example Reactive Object Queries [8] allow the developer to automatically create and maintain views of objects. Objects are selected through queries. Depending on its state an

[3] https://github.com/dionisiydk/TelePharo

object can enter or leave a view. The semantics of Collectors is different because they do not select subsets of existing objects, but try to collect all the marked entities in the control flow.

Although Collectors can evaluate conditions for object collection, these conditions are never reevaluated for collected objects. Furthermore, Collectors explicitly target objects from the control flow instead of querying potential objects from the entire object space.

7 CONCLUSION

In this paper, we described three problems of finding objects at runtime, namely:

(1) *The limited lifespan object* problem,
(2) *The "anonymous" object* problem,
(3) *The multiple instances* problem.

To address these problems we presented the Collectors. They allow to target objects of interest at runtime from variables in the source code. These objects are gathered and presented to the developer for interaction. We presented the Collectors API and tools, which provide *object-centric* views of variables from the control flow.

We experimented Collectors on a use-case of remote debugging in an *IOT* application. We used the Collectors on a runtime example featuring the aforementioned problems. We were able to gather objects by dynamically targeting them in the source code, and to log them into a console. This would have been difficult without a possibility to collect these objects at runtime.

We concluded by a discussion on Collectors and their application. We drafted future works that should be done to evaluate further the solution. We plan a more thorough investigation of the Collectors impact on the overall performance of a runtime. We also plan to investigate how to collect objects in multi-threaded applications. Finally, and in regards to the studied related work, it would be interesting to combine Collectors with features from other solutions. For example, access to execution context could be added to the Collectors specifications, or a finer management of the collected objects collections.

ACKNOWLEDGMENTS

We would like to thanks the anonymous reviewers and the participants of the *PX/18 Writer's Workshop* for their very interesting and constructive discussion and feedback.

REFERENCES

[1] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. 2010. *Pharo by example.* Lulu. com.
[2] Steven Costiou, Mickael Kerboeuf, Glenn Cavarle, and Alain Plantec. 2017. Lub: A pattern for fine grained behavior adaptation at runtime. *Science of Computer Programming* (2017). https://doi.org/10.1016/j.scico.2017.09.006
[3] Johan Fabry and Daniel Galdames. 2014. PHANtom: a modern aspect language for Pharo Smalltalk. *Software: Practice and Experience* 44, 4 (2014), 393–412.
[4] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.
[5] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *Journal of Object technology* 7, 3 (2008).
[6] John Keeney. 2004. *Completely unanticipated dynamic adaptation of software.* Ph.D. Dissertation. University of Dublin.
[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. *ECOOP'97—Object-oriented programming* (1997), 220–242.
[8] Stefan Lehmann, Tim Felgentreff, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2016. Reactive Object Queries. In *Constrained and Reactive Objects Workshop (CROW).*
[9] Matteo Marra, Elisa Gonzalez Boix, Steven Costiou, Mickaël Kerboeuf, Alain Plantec, Guillermo Polito, and Stéphane Ducasse. 2017. Debugging Cyber-Physical Systems with Pharo. In *International Workshop on Smalltalk Technology IWST'17.*
[10] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *CoRR* abs/1703.10862 (2017). arXiv:1703.10862 http://arxiv.org/abs/1703.10862
[11] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. 2017. It's duck (typing) season!. In *Proceedings of the 25th International Conference on Program Comprehension.* IEEE Press, 312–315.
[12] Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. 2009. Model-centric, context-aware software adaptation. In *Software Engineering for Self-Adaptive Systems.* Springer, 128–145.
[13] Patrick Rein, Stefan Ramson, Jens Lincke, Tim Felgentreff, and Robert Hirschfeld. 2017. Group-Based Behavior Adaptation Mechanisms in Object-Oriented Systems. *IEEE Software* 34, 6 (2017), 78–82.
[14] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 485–495.
[15] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2013. Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming* 78, 5 (2013), 481–498.