Molecule: live prototyping with component-oriented programming

Pierre Laborde, Steven Costiou, Alain Plantec, Eric Le Pors

Molecule: live prototyping with component-oriented programming

Pierre Laborde
THALES Defense Mission Systems France, 10 Avenue de la 1ère DFL, 29200 Brest, France
pierre.laborde@fr.thalesgroup.com

Steven Costiou
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
steven.costiou@inria.fr

Alain Plantec
Univ. Bretagne Occidentale, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
alain.plantec@univ-brest.fr

Eric Le Pors
THALES Defense Mission Systems France, 10 Avenue de la 1ère DFL, 29200 Brest, France
eric.lepors@fr.thalesgroup.com

Abstract

At Thales Defense Mission Systems, software products first go through an industrial prototyping phase. Prototyping are serious applications we experiment with our end-users during workshops. End-users have a central role in the design process of our products. They often ask for software modifications during demonstrations to experiment new ideas or to focus the existing design on their needs.

In this paper, we present how we combine Smalltalk's live-programming capabilities with software component models to obtain flexible and modular software designs in our context of live prototyping. We present Molecule, a Trait-based Lightweight Corba Component Model implementation in Pharo. Molecule components are standard Pharo classes using exclusively Traits to become software components.

We benefit from the dynamic run-time modification capabilities of Pharo during demonstrations with our end-users, where we explore software designs in a lively way.

Keywords: Live Prototyping, Components, LCCM, Traits, Pharo

ACM Reference Format:
Pierre Laborde, Steven Costiou, Alain Plantec, and Eric Le Pors. 2020. Molecule: live prototyping with component-oriented programming. In *IWST20: International Workshop on Smalltalk Technologies, September 29th and 30th, 2020, Novi Sad, Serbia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

IWST20, September 29th and 30th, 2020, Novi Sad, Serbia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

At Thales Defense Mission Systems, the Human-Machine Interface (HMI) industrial prototyping activities are an important part of the software production process. HMI industrial prototyping is the building of software prototypes as close as possible to real products from the HMI point of view (graphics and ergonomics). Using prototypes, we evaluate software HMI design and experiment complete use-cases provided by end-users. This enables early and strong feedback loops to fulfill users requirements. Using prototypes, we anticipate architectural needs and problems before development of real products begin. The prototyping activity is followed by an industrialization phase, in which we build final products based on prototypes' evaluations and feedback.

The dynamic aspects of prototyping requires the capability to modify code at run time. Evaluations of prototypes take hours and end-users are rarely available for review meetings. During a prototype evaluation bugs may appear, and end-users may request live modifications of the prototype to experiment ideas. In such cases, it is important to efficiently benefit from direct feedback. We cannot stop the program and lose hours of evaluation. We need to modify and to debug our prototypes without restarting everything.

In this paper, we present *Molecule*, an open-source component-oriented programming framework for the building of modular software architectures. Molecule has been implemented in Pharo [5] to favor changes at run time, during demonstrations in front of our end-users. Molecule features a component model based on the Light-weight CORBA Component Model (CCM) specification [1]. Molecule is based on Traits [4, 11, 16] to define component contracts and to define interfaces' behavior. The dynamic aspects of Traits in Pharo allows us to dynamically redefine and change component architectures at run time. We are thus able to experiment changes and ideas with end-users during demonstrations: this is what we call live prototyping.

In this paper, we present the following contributions:

- We exhibit the benefits of Pharo in an industrial context where we use live prototyping intensively to elaborate complex systems,
- an overview of Molecule, our Trait-based component framework which brings a clear separation of concerns to our prototypes implementation and facilitates updates at run time.

The paper is organized as follows. We explain our requirements for live prototyping and how components help for the elaboration of complex prototypes in Section 2. We present an overview of Molecule in Section 3, and illustrate its usage through examples in Section 4. We study related work in Section 5 and conclude the paper in Section 6.

2 Live prototyping in the defense industry

In the defense industry, we obtain high value feedback from prototypes demonstrations with end-users. During a demonstration, we need to quickly and efficiently capture users' needs to elaborate Graphical User Interface (GUI) prototypes. This work cannot be done remotely. It requires presence, observation and feelings from users in front of their application. However, there are strong constraints from our end-users' availability:

- We might only see end-users during a few days once a year for a prototype demonstration,
- during demonstrations, users always have requests and ideas they want to experiment,
- we need to take advantage of the users' presence to experiment those ideas and benefit from live feedback loops.

Therefore, a strong requirement of our prototyping activity is the ability to change pieces of software at run time to adapt prototypes according to the end-users's feedback. Thus, a prototype implementation is expected to evolve often and quickly during a demonstration. As we learn during our meetings with end-users, the architecture as well as the implementation of the software change many times.

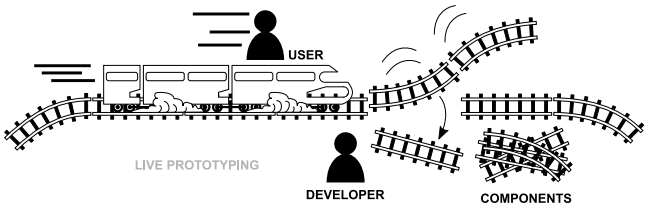

Figure 1. Live prototyping with Components

2.1 The importance of live prototyping

Imagine a moving train (Figure 1) and imagine that workers are changing the rails path without stopping the train. In practice, this is what we expect during workshops and evaluations: the running prototype is halted on the fly, its code is updated in front of our end-users and then restarted without stopping the running demonstration. This is particularly important when we need to choose one solution among several design candidates. In that case, we evaluate alternatives by changing the prototype code on-the-fly.

Live code modification gives us the ability to understand constraints in a given context, and to experiment code alternative without losing this context. Indeed, a demonstration scenario may last hours, and users perform lots of actions and configurations. If we have to restart demonstrations each time we change the software, we lose the execution context, the scenarios' data and the measured metrics. Complex user interactions inputs are hard to reproduce, *e.g.*, mouse events sequences or touch finger gestures. We cannot guarantee that users will do the same actions again and that they will reproduce the same execution context. In addition, end-users' availability is limited and we cannot afford to replay hours-long scenarios to experiment variations.

When live prototyping with end-users, we dynamically change the running software to experiment new solutions without restarting the entire evaluation scenario and losing measured metrics. To users, these changes are done transparently: they do not lose their configurations, their data nor the state of the running scenario.

2.2 Enabling live prototyping: an industrial context

Since 15 years, the Thales Defense Mission Systems prototyping team uses Smalltalk for prototyping. The motivation behind the choice of Smalltalk is its dynamic capabilities which enable live changes of a design in the front of customers. From 2005 to 2016, the *VisualWorks*[1] system was used. Since 2016, we integrate also Pharo [5] as a programming environment solution.

However, these technical capabilities for live programming are not enough in our context. Let us come back to the illustration of Figure 1. Developers are changing preassembled rails. They do not have to build them on the fly with wood and steel. They dynamically switch one code portion implementing a particular functional solution by another in few editing steps. Similarly, we need to clearly separate the code portions that are updated on the fly from the legacy code that must stay unchanged [8].

Our final products are component-based architectures, and we need prototypes and final products to share similar architecture principles. This guarantees equivalent levels of services, and precise evaluations of development costs of final products. Therefore, we also decided to base the prototyping step on component-oriented development to bring consistency in the whole product life-cycle from the prototyping to the final product. In this context, we developed

[1] http://www.cincomsmalltalk.com/main/products/visualworks/ — accessed July, 27th, 2020

Molecule, our component-oriented programming framework for Pharo. The rest of this Paper describes Molecule.

3 Molecule overview

Molecule is a Light-weight implementation of the Corba Component Model (LCCM) [1]. It allows for the specification of components as in the Corba standard: provided and used services, produced and consumed events. However, Molecule components are only specified and instantiated locally. They are not exchanged nor shared through a standard object bus. The rest of this section briefly presents what is a Molecule component and how it is dynamically managed.

3.1 Molecule, a component framework

Similarly to the LCCM, a component's business contract is exposed through its component *Type* (Figure 2). The Type specifies what a component has to offer to others components (namely, *provided services* and *produced events*) and what that component requires from others components (namely, *used services* and *consumed events*).

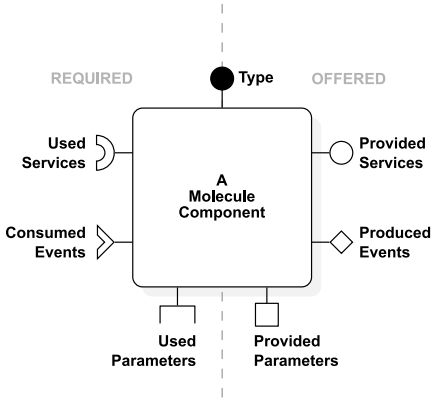

Figure 2. Public view of a Molecule component.

Thus, the main role of the Type is to implement the services that the component provides and that are callable by other components. Other components use this interface through their *Used Services* interface. *Produced Events* represent the receivable events interface of the component. Other components listen to this interface through their *Consumed Events* interface. They subscribe and unsubscribe to their event interface to start and stop receiving notifications. Parameters are used to control the component's state. Parameters can only be used once at the component's initialization. The light-weight CCM model does not define Parameters, but instead it allows direct access to public attributes. We introduced the *Provided Parameters* as an interface to explicitly define how the state of a component can be initialized. Other components use this interface through their *Used Parameters* interface.

A Molecule component definition is based on Traits [4, 11, 16]. The Type, as well as the services, the events and the parameters parts are all defined as Traits. A Molecule component is an instance of a standard class which uses Molecule traits.

3.2 Live update of components

When a Type is modified, a mechanism automatically updates the component classes implementing this Type. A Molecule Type is defined by a set of related Traits. A Molecule component is made of a class which uses a Type Trait. A Type can be modified either by modifying the Trait aggregation or the Traits themselves.

For example, adding a particular method in a service Trait implies updating the related methods in the class that is using the component Type. In that case, some methods have to be implemented by hand to plug in the services, and then to turn the component into a fully usable one at run time. Recall that during a demonstration, switching from one component to another must be doable as fast as possible to benefit from a live demonstration flow.

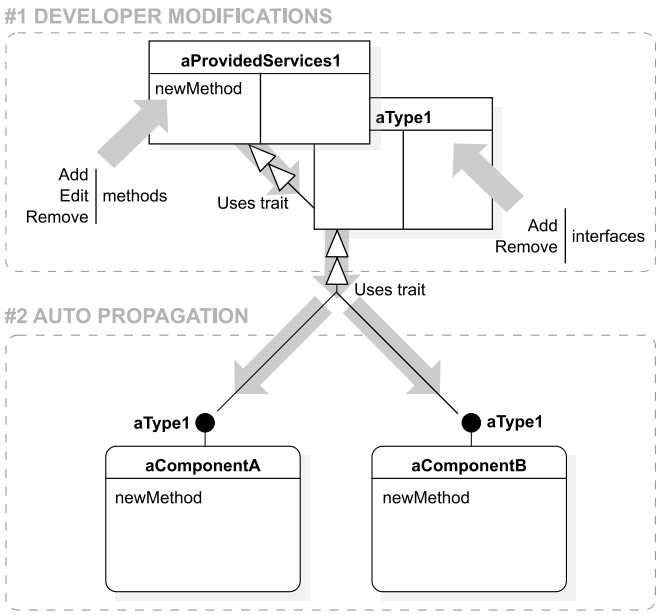

Figure 3. Components automatic modifications using Traits.

Upon the modification of a Type, an additional mechanism automatically implements changes into component classes using this Type (Figure 3). Adding or removing Traits (services, events or parameters) in a Type is automatically detected and all classes using the updated Type are themselves updated accordingly. This mechanism is implemented using the Pragma [3] infrastructure of Pharo.

3.3 Run-time management of components

All components are managed by the ComponentManager object. It maintains the list of component instances currently alive in the system. It is currently handled as a singleton. The ComponentManager class implements an API to instantiate and to remove each component, to associate them, to connect events, etc. This API is used to manage each component life cycle programmatically.

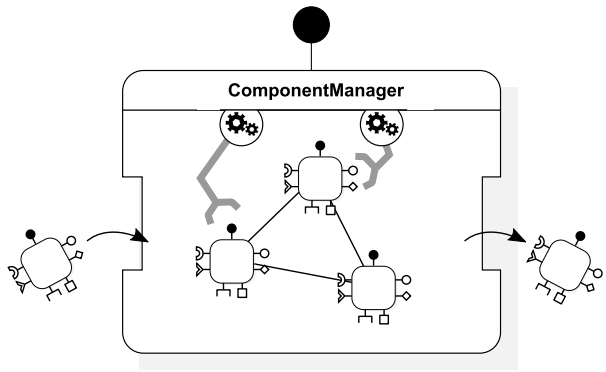

Figure 4. The ComponentManager manage all components.

3.4 The component states

The activity of a component depends on contextual constraints such as the availability of a resource, the physical state of hardware elements, etc. To manage consumed resources accordingly, the life-cycle of a component has four possible states: *Initialized*, *Activated*, *Passivated* and *Removed* (Figure 5). After its initialization, a component can switch from an *Activated* state to a *Passivated* state and conversely. When the life-cycle of a component is over, then it switches to the *Removed* state.

Let us details each state of a component life-cycle. When a component is switched to the *Initialized* state, it is configured through its provided parameters. If a component depends on another component through its interfaces (used services, consumed events or used parameters), these components are associated during this state.

The *Activated* state is the nominal state of a component. When a component is switched to this state, it subscribes to each consumed events that are produced by the components that have been associated with it during the *Initialized* state. After this subscription step, the component is able to receive and react accordingly to any of its consumed events.

When a component is paused, it switches to the *Passivated* state. Then, the component unsubscribes to its subscribed events and all its required resources are set in waiting mode. As an example, a hardware can be set in its sleeping mode, it can also be asked to free its Graphics Processing Unit memory. The idea behind this state is to avoid consuming resources if not needed, and to be able to switch back as quickly as possible to the *Activated* state.

The terminal state of a component is the *Removed* state. When a component switches to this state, all of it resources are released. The ComponentManager removes that component from its list of alive components.

Let us illustrate the use of these states with the example of a GUI window handled as a component. First, the window is instantiated by the component. Then the component state switches to *Initialized*. When the window is displayed on the desktop, the component's state switches to *Activated*. When the window is reduced and its icon is stored into a task-bar, then the component switches to the *Passivated* state. As the window is only reduced, it can be re-opened very quickly. Finally, when the user closes the window. The component is first switched to the *Passivated*, then to the *Removed* state.

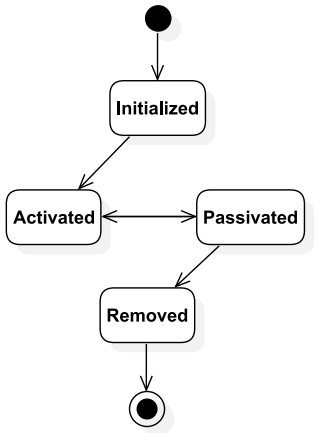

Figure 5. Molecule components life-cycle.

4 Molecule by example

In this section, we give an overview of Molecule through two examples. As depicted in Figure 6, in the first example, we program a component application that connects to a Global Positioning System (GPS) hardware and displays the GPS data on a view map. In the second example, we reuse an existing non-component class in our Molecule application. To do so, we augment this class with component behavior.

4.1 The GPS map and data provider example

In this example, we create two Molecule components with their Types. We show how we define the components events and services and how these components interact with each other (Figure 7).

We create a first component, its job is to manage and provide datas from a GPS hardware (as the current geographic position). Listing 1 shows an example of component implementation. To develop a new component from scratch, developers subclass the MolAbstractComponentImpl abstract class. In fact, MolAbstractComponentImpl is a syntactic sugar directly using the default Molecule trait which injects

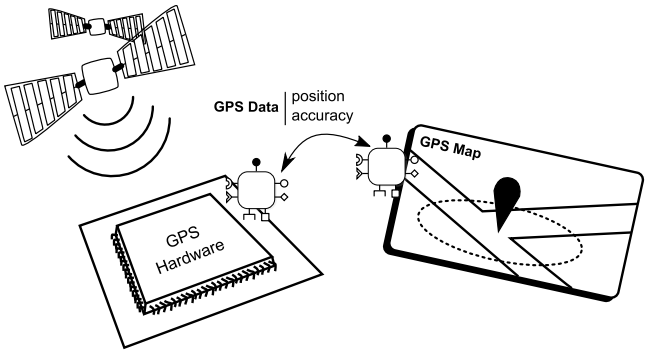

Figure 6. Examples with a GPS application.

component shared behavior into classes. The component Type is implemented through a Trait. In our example GPSData is the Type Trait and GPSDataImpl is the component class which uses it.

```
1 MolAbstractComponentImpl subclass: #GPSDataImpl
2   uses: GPSData.
```

Listing 1. Creation of GPSDataImpl component class.

We define the GPSData Type as a Trait which itself uses the Molecule base Type MolComponentType (Listing 2). This base Type provides the skeleton methods providedComponentServices and producedComponentEvents. Developers have to complete these methods by hand to declare the provided services and the produced events.

```
1 Trait named: #GPSData uses: MolComponentType.
2 GPSData class>>providedComponentServices
3   <componentContract>
4   ↑{ GPSDataServices }
5 GPSData class>>producedComponentEvents
6   <componentContract>
7   ↑{ GPSDataEvents }
```

Listing 2. Declaring the GPSData Type with one provided services interface and one produced events interface.

The method providedComponentServices (or producedComponentEvents for events) return the array of Traits that implement the specific provided services (or produced events for events). Specific services and events are implemented in their own trait, namely, GPSDataServices and GPSDataEvents (Listing 3). In these traits, we add methods that the GPSData component Type has to implement.

```
1 Trait named: #GPSDataEvents.
2 GPSDataEvents>>currentPositionChanged: aGeoPosition
3   "Notify the current geographic position of the GPS
      receiver when changed"
4 Trait named: #GPSDataServices.
5 GPSDataServices>>getAccuracyRadiusInMeters
6   "Get and return the accuracy of the GPS depending
      quality of signal and quantity of connected satellites"
```

Listing 3. Defining the GPSData Type content.

We create a second component to display a GPS position on a map. The map displays a circle around the position whose radius represents the accuracy of the GPS. The second component needs to be connected with an existing GPS data provider component. In Listing 4, we add a new component class GPSMapImpl (lines 1-2) with a new GPSMap Type (line 3). The Type GPSMap requires to consume services of the GPSDataServices interface (lines 4-6) and to use events of the GPSDataEvents interface (lines 7-9). Note that these interfaces are already defined and used by our first component GPSDataImpl.

```
1 MolAbstractComponentImpl subclass: #GPSMapImpl
2   uses: GPSMap.
3 Trait named: #GPSMap uses: MolComponentType.
4 GPSMap class>>usedComponentServices
5   <componentContract>
6   ↑{ GPSDataServices }
7 GPSMap class>>consumedComponentEvents
8   <componentContract>
9   ↑{ GPSDataEvents }
```

Listing 4. Creating GPSMapImpl component class that uses GPSMap Type with one used services interface and one consumed events interface.

As explained in Section 3.2, some methods are automatically generated to plug the services in the component class. Declaring the uses of GPSDataServices automatically generates the corresponding accessor getGPSDataServices which allows the GPSMapImpl component to call the services methods. Declaring the consumption of GPSDataEvents automatically generates a skeleton implementation of the interface in the component class GPSMapImpl (Listing 5). Developers have to complete generated methods with their application code.

```
1 GPSMapImpl>>currentPositionChanged: aGeoPosition
2   | radius
3   radius := self getGPSDataServices
      getAccuracyRadiusInMeters.
4   "Display a circle on the ma |p view at the current
      position"
5   self updatePositionCircleOnMap: aGeoPosition radius:
      radius.
```

Listing 5. Consumed event generated method.

A component is expected to implement a particular method for each state introduced in Section 3.4. These methods are called when the component is switched to the corresponding state. In our example, regarding the *Activated* state, the corresponding method is componentActivate. This method is completed by hand with the subscription code shown in Listing 6 to enable events reception management.

In this example, the subscriber does not specify which instance of the event provider is required. In that case, the GPSMapImpl component subscribes to the first available event provider. When GPSMapImpl is activated, it subscribes to the GPSDataEvents event provider. After the subscription, a GPSMapImpl is able to receive the position change events to update its view accordingly.

```
1 GPSMapImpl>>componentActivate
2   self getGPSDataEventsSubscriber subscribe: self
```

Listing 6. Subscription of GPSMapImpl receives to GPSDataEvents.

As shown in Listing 7, components instances are created and activated by the ComponentManager which has the responsibility to connect them each other lazily.

```
1 MolComponentManager
2   instanciateComponent: GPSMapImpl.
3 MolComponentManager
4   activateComponent: GPSMapImpl.
```

Listing 7. Instanciating and activated components.

4.2 Reusing existing code as Molecule components

Imagine that we want to reuse an open-source library that implements the driver for the GPS hardware that we use. We want to reuse a class from this existing implementation in our application to add the capability to use this GPS hardware.

This class is not a Molecule component, and does not share the same class hierarchy as Molecule components. Therefore this class does not answer the Molecule component's interface, and cannot be reused directly as a component. To manually plug this class into a Molecule component, we have to write glue code for the component to use the API of this class. This requires an additional effort to write non-functional code, which introduces noise in the application code. This makes such architecture less to understandable and to maintainable.

With Molecule, we reuse any existing class by augmenting that class with component behavior (Figure 8). This class becomes seamlessly usable as a component in a Molecule architecture.

Imagine that we want to reuse a class GPSHardware. This class is originally used as described in (Listing 8). Developers must instantiate this class, then interact with its instances to use the GPS hardware.

```
1 driver := GPSHardware new.
2 driver connect.
3 accuracy := driver accuracy.
```

Listing 8. A GPSHardware class providing the accuracy of the GPS.

To use the GPSHardware class as a GPSData Type component, we augment that class with Molecule component behavior and with the GPSData Type (Listing 9). First, we add the Molecule component interface MolComponentImpl to the GPSHardware class. Any class that implements this interface is usable as a Molecule component. Then, we affect the GPSData Type component to the GPSHardware class.

The MolComponentImpl interface and GPSData Type are implemented as Traits. Therefore the GPSHardware class is augmented just by declaring the use of these Traits (line 2). Traits automatically brings the code for implementing interfaces.

```
1 Object subclass GPSHardware
2   uses: {MolComponentImpl + GPSData}
```

Listing 9. Augmenting the GPSHardware class with Molecule component behavior and set its Type.

After augmenting the GPSHardware class with component behavior, we make sure the class implements the component contract in accordance with the GPSData Type. By default, these methods are automatically generated by Molecule. In this example (Listing 9) as we reuse an existing class, we adapt the generated interface to fit our needs:

- We implement a call to the behavior provided by the class,
- we implement in addition a conversion from the feet to meters unit.

Because of the interface enforced to every Molecule component, we are able to directly reuse and specialize an existing class. We only write domain code to reuse existing behavior into the component architecture and avoid writing non-functional code.

```
1 GPSHardware>>getAccuracyRadiusInMeters
2   "GPSHardware use imperial units, need to convert into
    meters"
3   ↑self accuracy feetToMeters
```

Listing 10. Reusing and extending the GPSHardware implementation in the Molecule component interface.

Finally, we instantiate this class as a Molecule component and we swap the previous implementation GPSDataImpl with our new implementation GPSHardware (Listing 11). We remove the current component from the application (lines 2-3) and replace it with the new component (lines 5-6).

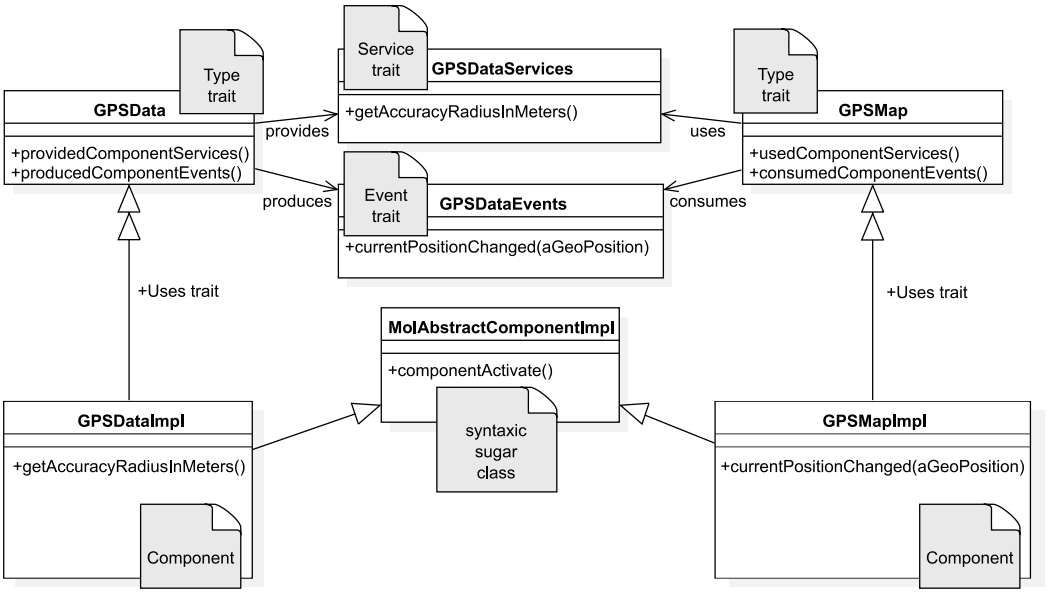

Figure 7. The GPS map and data provider example

```
"Removing old GPSData Type implementation from
    example 1"
MolComponentManager
  removeComponent: GPSDataImpl.
"Instanciating new GPSData Type implementation with
    our re−use class as a component class"
MolComponentManager
  instanciateComponent: GPSHardware.
```

Listing 11. Dynamically swapping components implementation.

Swapping components does not require to restart the whole application, and can be performed at run time. Between the removal and replacement of a component, it is possible that other components continue to access the services of the removed component (*e.g.*, in multi-threaded applications). In that case, Molecule automatically returns a default instance that corresponds to a non-existing provider. Components implementations must handle this default instance when they require a service from another component.

5 Related Work

Molecule provides the same well-known architectural and reuse benefits of software components systems [7, 13–15]. To focus on the live aspect of prototyping, we study related work on run-time software modification.

Run-time adaptation in component systems has been well studied. For instance, and non-exhaustively, there is work on dynamic component modification [6, 9, 18], with safety [17] and consistency [10] concerns. These systems generally only provide run-time modification of component-related entities

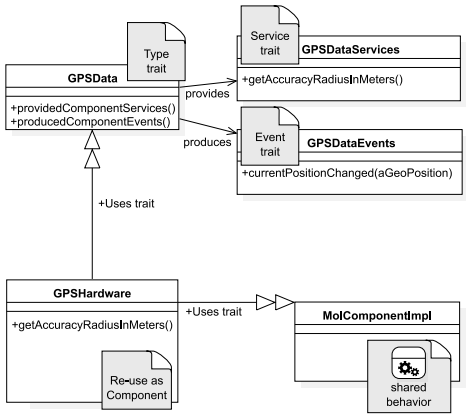

Figure 8. Reusing existing code as Molecule components.

such as components, component ports or interconnections. However, while designing and demonstrating prototypes we also require to modify code outside components, *e.g.*, to fix bugs in third party libraries.

Smalltalk-based component models implementations [2, 6, 12] enable run-time modification of both component and non-component code thanks to Smalltalk's reflective properties. Such implementations provide the same dynamic, unanticipated run-time modification capabilities as the Molecule Pharo implementation.

As far as we know, Molecule is the only implementation of LCCM relying on Traits.

6 Conclusion

Developing industrial prototypes as close as possible to final products is tedious. At Thales Defense Mission Systems, we

build such prototypes before a final version is implemented by another team. Our prototypes are evaluated not only through their HMI but also through complete functional use cases. In order to fulfill the requirements and the actual end-users expectations, we evaluate prototypes with end-users and adapt them lively according to their feedback. Thus, we use Pharo to implement our prototypes because of its dynamic capabilities. Because our final product are component oriented, we use this paradigm for the implementation of our Pharo prototypes. Regarding the life-cycle of a product, the direct benefit is the traceability between the prototypes components and the final products components. Moreover, the implementation and the lively adaptation of our prototypes is also facilitated by the component orientation. In this paper we presented Molecule, a light-weight CORBA Component Model implementation based on Traits that we have developed in this context. We presented an overview of Molecule and illustrated its usage through a simple example.

More and more, our prototypes involve multiple users on collaborative tasks on interconnected workstations. We will upgrade the capacity of Molecule to build distributed component architectures. In the future, we want to use Molecule in distributed architectures but keeping all its live prototyping capabilities. This is challenging as this implies consistent modifications of remote and distributed run-time architectures.

Acknowledgments

We thanks Thales Defense Mission Systems for their continuous support and for believing in the powers of live prototyping, and the Thales technical board who authorized us to publish our work in the open source world. We thanks also Nolwenn Fournier and Camille Delloye for their participation to the elaboration of Molecule.

References

[1] Corba component model specification. https://www.omg.org/spec/CCM/4.0/PDF, accessed: july 7th, 2020

[2] Bouraqadi, N., Fabresse, L.: Clic: a component model symbiotic with smalltalk. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 114–119 (2009)

[3] Ducasse, S., Miranda, E., Plantec, A.: Pragmas: Literal Messages as Powerful Method Annotations. In: International Workshop on Smalltalk Technologies - IWST 2016. Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, Prague, Czech Republic (Aug 2016). https://doi.org/10.1145/2991041.2991050, https://hal.inria.fr/hal-01353592

[4] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) **28**(2), 331–388 (2006)

[5] Ducasse, S., Zagidulin, D., Hess, N., written by A. Black, D.C.O., Ducasse, S., Nierstrasz, O., with D. Cassou, D.P., Denker, M.: Pharo by Example 5. Square Bracket Associates (2017), http://books.pharo.org

[6] Fabresse, L., Dony, C., Huchard, M.: Unanticipated connection of components based on their state changes notifications. In: EECC: Evaluation and Evolution of Component Composition (2006)

[7] Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on software engineering **33**(10), 709–724 (2007)

[8] Panunzio, M., Vardanega, T.: A component-based process with separation of concerns for the development of embedded real-time software systems. Journal of Systems and Software **96**, 105 – 121 (2014). https://doi.org/https://doi.org/10.1016/j.jss.2014.05.076, http://www.sciencedirect.com/science/article/pii/S0164121214001381

[9] Piechnick, C., Richly, S., Götz, S., Wilke, C., Aßmann, U.: Using role-based composition to support unanticipated, dynamic adaptation-smart application grids. Proceedings of ADAPTIVE pp. 93–102 (2012)

[10] Rudametkin Ivey, W.A.: Robusta: une approche pour la construction d'applications dynamiques. Ph.D. thesis, Grenoble (2013)

[11] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: European Conference on Object-Oriented Programming. pp. 248–274. Springer (2003)

[12] Spacek, P., Dony, C., Tibermacine, C.: A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In: Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering. pp. 13–22 (2014)

[13] Szyperski, C.: Component technology-what, where, and how? In: 25th International Conference on Software Engineering, 2003. Proceedings. pp. 684–693. IEEE (2003)

[14] Szyperski, C., Bosch, J., Weck, W.: Component-oriented programming. In: European Conference on Object-Oriented Programming. pp. 184–192. Springer (1999)

[15] Szyperski, C., Gruntz, D., Murer, S.: Component software: beyond object-oriented programming. Pearson Education (2002)

[16] Tesone, P., Ducasse, S., Polito, G., Fabresse, L., Bouraqadi, N.: A new modular implementation for stateful traits. Science of Computer Programming (2020)

[17] Vandewoude, Y.: Dynamically updating component-oriented systems (2007)

[18] Vandewoude, Y., Berbers, Y.: Supporting run-time evolution in seescoa. Journal of Integrated Design and Process Science **8**(1), 77–89 (2004)