Live Application Programming in the Defense Industry with the Molecule Component Framework

Pierre Laborde, Yann Le Goff, Éric Le Pors, Alain Plantec, Steven Costiou

Live Application Programming in the Defense Industry with the Molecule Component Framework

Pierre Laborde[a,*], Yann Le Goff[a,b], Éric Le Pors[a], Alain Plantec[b], Steven Costiou[c,*]

[a] *THALES Defense Mission Systems France,*
[b] *Univ Brest, CNRS, Lab-STICC, CS 93837, 6 avenue Le Gorgeu, 29238 Brest Cedex 3, France,*
[c] *Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL,*

Abstract

At Thales Defense Mission Systems (DMS), software products first go through an industrial prototyping phase. Prototypes are serious applications that we evaluate with our end-users during demonstrations. 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 combined 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, an open-source implementation of the Lightweight CORBA Component Model in Pharo. We use Molecule to build HMI systems prototypes, and 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.

Molecule is an industrial contribution to Smalltalk, as it capitalizes 20 years of usage and maturation in our prototyping activity. The Molecule framework and tools are now mature, and we started building end-user software used in production at Thales DMS. We present two such end-user software and analyze their component architecture, that are representative of how we (learnt to) build HMI prototypes. Finally, we analyze our technological decisions with regards to the benefits we sought for our industrial activity.

Keywords: Live Prototyping, Components, LwCCM, Pharo

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 our HMI with end users through realistic scenarios during demonstrations that can last for hours. This enables early and strong feedback loops to fulfill users requirements. Using prototypes, we evaluate software HMI design: usability, efficiency and satisfaction. This allows us to put users in a situation as if they were looking for the final product and to be able to identify design problems upstream. Thanks to prototypes, we can anticipate architectural needs and problems before we start developing real products. We leverage these experiences to develop reusable components and component subsystems [1]. This approach enables us to create new prototypes faster, while trusting components reused across the years. The prototyping activity is followed by an industrialization phase, in which we build final products based on prototypes' evaluations and feedbacks.

Because we build and rebuild prototypes, we need means to reuse code from existing pieces of software. We need to evolve parts of prototypes without changing how these parts interact with the rest of the software. To foster such modular architectures, Thales use component-oriented programming [2, 3, 4, 5]. Component-based architectures bring modularity by easing components reuse and evolution. The dynamic aspects of prototyping also 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 running program 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.

We started our prototyping activity in 2004, at which time we had to make technological choices (languages, architectures, frameworks) to satisfy our constraints. At the

*Corresponding authors
Email addresses: pierre.laborde@fr.thalesgroup.com (Pierre Laborde), yann.le-goff@thalesgroup.com (Yann Le Goff), eric.lepors@fr.thalesgroup.com (Éric Le Pors), alain.plantec@univ-brest.fr (Alain Plantec), steven.costiou@inria.fr (Steven Costiou)

time (and still as of today), Smalltalk dialects were (and still are) powerful technologies for live programming. Not only is it possible to modify a running Smalltalk program, but most Smalltalk dialects include a comprehensive development environment that enables seamless live modifications and immediate feedback on the running program. We therefore chose Smalltalk as a language and technology for our prototyping actvity. We chose the *Lightweight CORBA Component Model* (LwCCM) [6, 7] for architecturing our prototypes. This choice was guided by 1) the well-known properties of components for building modular software and 2) the need to match the architectures of the industrial software built from our prototypes in our clients' technologies (that are usually not Smalltalk).

However, at the time, there was no LwCCM implementation for Smalltalk. We therefore developed our own LwCCM in Smalltalk, and since then we used it intensively for building HMI systems prototypes and evaluating these prototypes with end-users. The implementation matured over the years, allowing us to capitalize on a large number of reusable software components [1]. This paper is an industrial contribution to Smalltalk, based on 20 years of development and usage of a LwCCM Smalltalk implementation. We present *Molecule*, our latest open-source LwCCM framework implemented in *Pharo* [8].

In a previous paper [9], we presented an overview of Molecule, our component framework that introduces a clear separation of concerns to our prototype implementations and facilitates updates at runtime. We illustrated Molecule through examples showcasing the framework's main features. We briefly described the motivation behind Molecule, which stems from our intensive live prototyping activities to develop complex systems. In this paper, we extend our previous work with the following new contributions:

1. We elaborate on our motivation (Sections 2 and 3), placing it precisely in the context of live prototyping for the defense industry. We reflect on Thales DMS' decision to adopt the LwCCM 20 years ago and analyze the original benefits we sought for our industrial activities (Section 7).
2. We provide more descriptive and illustrative code examples to improve the understanding of the modular and dynamic aspects of Molecule architectures (Section 5). These examples are representative of production scenarios for which we use Molecule.
3. We analyze the architectures of two end-user applications developed since [9] and deployed in production at Thales DMS (Sections 6). These architectures and their designs reflect how we build and design industrial prototypes after 20 years of practice.

The paper is therefore organized as follows. We describe our industrial context and methodology for HMI prototypes evaluation in Section 2. We describe our design requirements and choices in Section 3, which led to the implementation of the Molecule framework in Section 4. We illustrate Molecule's usage through examples in Section 5. In Section 6 we present and analyze the component architecture of two real applications we built with Molecule. Finally, we discuss the choice of component based architecture in the context of live prototyping in Section 7 and conclude the paper in Section 8.

2. Motivation: dynamic design exploration and reusable software components

At Thales DMS we build software for the defense industry. This software undergo long and costly processes of design, development and validation, especially their HMI. Once development and validation are complete, it is extremely costly to change the design, for example, if the final users have problems using the HMI [10].. Therefore we base our HMI design decisions on ergonomic evaluations that we conduct prior to the software development phase.

To conduct these evaluations we develop prototypes that are as close as possible to end-products. We then design simulations based on real defense missions scenarios [11], and we put end-users to work in these simulations during events we call *demonstrations*. Demonstrations can last hours without interruption (like a real mission), during which users use the prototype like they would use the real system during a genuine defense mission. Therefore demonstrations need to be as close as possible to real conditions, to improve the quality of the evaluation, the relevance of users' feedback and make sure that we address the right ergonomic issues.

However, end users from the defense industry (such as military people) have limited availability due to the nature of their work. They participate to a demonstration during a few days then wait for years until they obtain the final product. These few days are our sole opportunity to gather feedback from end users and to improve, fix or change our HMI designs. Therefore we need to capture as much feedback as possible from end users during demonstrations. Under this constraint, implementing demonstrations is difficult due to the following challenges.

Dynamic Design Exploration. During demonstrations, end users provide us with feedback, ideas and requests. Because these suggestions change the HMI design, we need to evaluate the new system after its implementation. We cannot wait for another demonstration because often we never see the users again. For every user suggestion, we have to immediately modify the design and its implementation, and submit the new system to users to explore the variations they proposed.

Users also encounter bugs that block them in their simulated mission. In that case we have to fix the bug to make sure the simulation can continue, and we have to fix it as fast as possible to minimize its impact on the ongoing simulation.

In both cases, suggestions and bugs depend on interaction scenarios performed by users. Such scenario depends on the current state of the simulation, which can accumulate hours of data. Indeed, a demonstration scenario may last hours, and users perform lots of actions and configurations. Complex user interactions inputs are hard to reproduce, *e.g.*, mouse events sequences or touch finger gestures. Hence, we cannot simply interrupt the software system, implement the changes and restart the system:

1. We would lose all simulation data and state, and we cannot guarantee that we could reproduce the exact sequence of user interactions that lead to that state,
2. we would lose our users, as there is little chance that they would accept to replay hours-long scenarios to experiment variations.

Due to these reasons, we need to implement changes (users' suggestions and bug fixes) dynamically at run time while minimizing disruptions to the ongoing simulation.

Reusable software components. As we learn during demonstrations with end-users, the architecture as well as the implementation of the simulation system's software change many times. Thus, a prototype implementation is expected to evolve often and quickly during an evaluation. However, after many demonstrations over the years we developed different flavors and variations of software elements. When users request change, it is common that the variations they request are already implemented but not integrated to the current simulation where it has a running variant. We need to be able to reuse previously developed software parts to replace parts of the running simulation software.

Furthermore, our final products that are ultimately delivered to end users are component-based architectures. Industrially speaking, we have the constraint that our prototypes and final products share similar architecture principles [12]. This guarantees equivalent levels of services, and precise evaluations of development costs of final products. Hence, an architecture design in the prototype must be reusable in the final product and conversely.

3. Live prototyping in the defense industry

In our demonstrations with end users, we dynamically change the running software to experiment new solutions without restarting the entire simulation. We modify the software by intercheanging existing code components, or by writing and injecting new code (*e.g.*, bug fixes) using *live programming* methodologies and techniques [13, 14, 15, 16, 17]. To users, these changes are transparent: they do not lose their configurations, their data nor the state of the running scenario. Ideally, they do not even notice that changes are being made.

We call this activity *live prototyping*. We summarize below its basic principles and how we chose the implementation technologies in our industrial context.

3.1. Basic principles 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 demonstrations. The running prototype is halted on the fly, its code is updated in front of our end-users and then resumed without stopping the running simulation. 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.

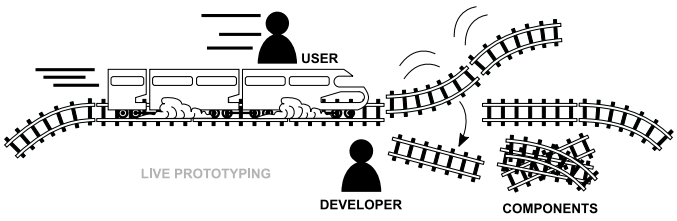

Figure 1: Live prototyping with Components

We see in Figure 1 that 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, already existing one. It is possible that users request changes that are totally new, or that require a modification of existing code. Developers then have to write the new changes and integrate it in existing code. In both cases (reused code and new code), developers rely on the capabilities of a live programming environment to apply changes at run time.

In case of a bug, the simulation system might stop. The live programming environment catches the bug and opens a debugger. Depending on the demonstration scenarios and the simulation's constraints, the debugger may open either on a user's station (*e.g.*, if only one user is affected) or in a remote control location to hide code edition. Developers use the debugger to inspect the simulation state and understand the bug. They fix it by editing code from the debugger, and proceed the execution that continues. While the simulation is halted when the debugger is opened, it resumes transparently after the bug is fixed and the debugger is closed.

3.2. Enabling live prototyping: an industrial context

Since 2004, the Thales DMS prototyping team uses *Smalltalk* [18] as a technical foundation for prototyping. Our motivation behind the choice of Smalltalk is its dynamic capabilities which enable live code changes, and thus to experiment design variations, directly in the front of our end users. Live code modification gives us the ability to understand constraints in a given execution context (simulation state, user interactions, recorded metrics), and to experiment code alternative without losing that context.

For reuse aspects, we chose to focus on component-based architectures, specifically the Lightweight CORBA Component Model [19] (or LwCCM). At the time, the choice of components was straightforward. Component architecture are reconfigureable, and component interfaces allow us to interchange different components sharing the same interface. Components also helped in clearly separating the code portions that are updated on the fly from the legacy code that must stay unchanged [20]. Finally, the industrial teams already worked with components, and we had to build similar architectures so that they could study our prototypes and easily benefit (and reuse) what we learnt from demonstrations. We chose the LwCCM specifically because industrial teams were using the Corba Component Model, and the LwCCM was a simpler compatible model – our prototyping activity did not require all the complexity of the industrial component architectures.

However at the time, 20 years ago, there were very few component frameworks implemented in Smalltalk (such as [21]). There was no publicly available implementation of the LwCCM in Smalltalk, and therefore we had to build one.

From 2004 to 2016, we used the *VisualWorks*[1] Smalltalk system to develop our LwCCM framework and run our demonstrations. Since 2016, we also integrate Pharo [8] Smalltalk as a live-programming system. We developed Molecule, the open source version of our component-oriented programming framework for Pharo. The rest of this paper describes Molecule, how we develop with Molecule and how we design software architectures with Molecule.

4. Molecule overview

Molecule is an implementation of the Lightweight CORBA Component Model [19]. 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.

Molecule is based on Traits [22, 23, 24] to define component contracts and to define interfaces' behavior. The dynamic aspects of Traits in Pharo enables run-time redefinition and changes of component architectures.

The rest of this section briefly presents what is a Molecule component and how it is dynamically managed.

4.1. Molecule, a component framework

Similarly to the LwCCM, 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*).

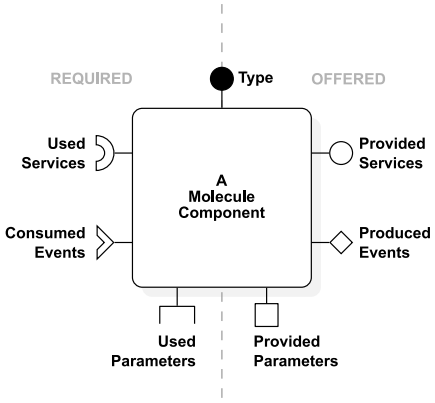

Figure 2: Public view of a Molecule component.

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. LwCCM does not define Parameters, but instead allows for direct access to public attributes. In order to establish an architectural similarity between our component model and the one used in software production, we introduced the *Provided Parameters* as an interface to explicitly define how the state of a component can be initialized. The *provided services* and *Produced events* pertain to functional aspects of components, while the *Provided Parameters* relate to behavioral aspects (textite.g., threads priorities, the number of elements in a component cache, the sort algorithm used, etc.). The *Provided Parameters* do not allow to change what the component is doing but how it does it. Other components can use this capability through their *Used Parameters* interface.

A Molecule component definition is based on Traits [22, 23, 24]. The Type, as well as the services, the events and the parameters parts are all defined as Traits. A Molecule component therefore is an instance of a standard class which uses Molecule traits.

We based Molecule on traits for two main purposes. First, because component interfaces (contracts) are defined as traits we can interchange component instances on the fly during program execution (Figure 1). Second, we can modify a component contract (statically or dynamically at run time) by modifying the trait that implement that contract. All component instances using the modified trait are automatically updated (Figure 3).

[1] http://www.cincomsmalltalk.com/main/products/visualworks/ — accessed February, 23d, 2024

4.2. 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. Remember that during a demonstration, we need to be able to switch from one component to another as quickly as possible to benefit from a live demonstration flow.

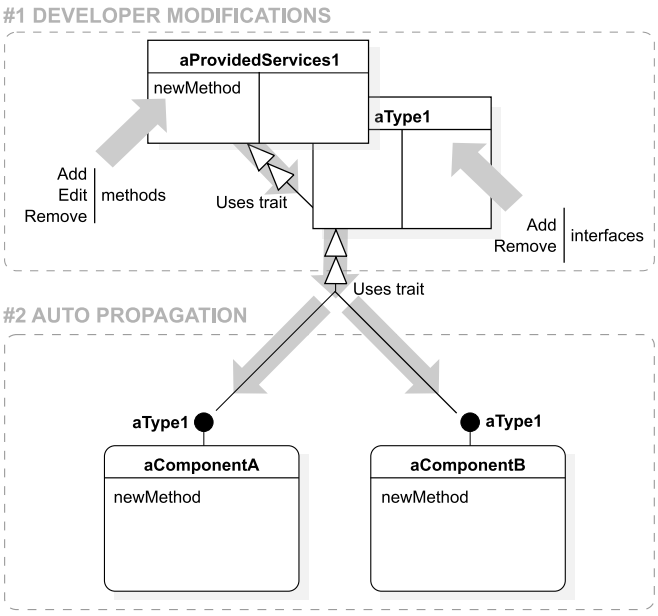

Figure 3: Components automatic modifications using Traits.

Upon the modification of a Type, an additional mechanism automatically implements changes into component classes using this Type (Figure 3). The addition or removal of Traits (services, events or parameters) in a Type is automatically detected and all classes using the updated Type are themselves updated accordingly.

4.3. Run-time management of components

All components are managed by a singleton `ComponentManager` object. It maintains the list of component instances currently alive in the system. 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.

4.4. The component life cycle

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 4). 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 components that have been associated with it during the *Initialized* state. After this subscription step, the component is able to receive and to react accordingly to any of its consumed events.

When a component is paused, it switches to the *Passivated* state. The component unsubscribes from 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 unnecessary resources, 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 a component's lifecycle with the example of a GUI window handled by 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, 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.

5. Molecule by example

In this section, we give an overview of Molecule through an example of a geographical position location application (Figure 5). The application is composed of a cartographic view which displays the geographical position of a user. The position is surrounded by a circle which represents the precision of the location. In the bottom of the application window, the user can select a location system. When a

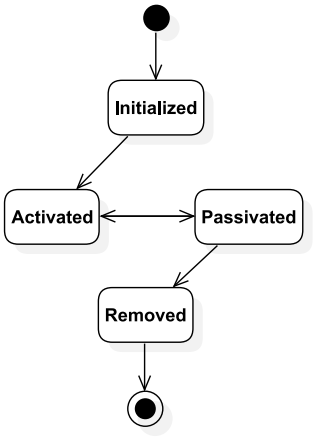

Figure 4: Molecule components life-cycle.

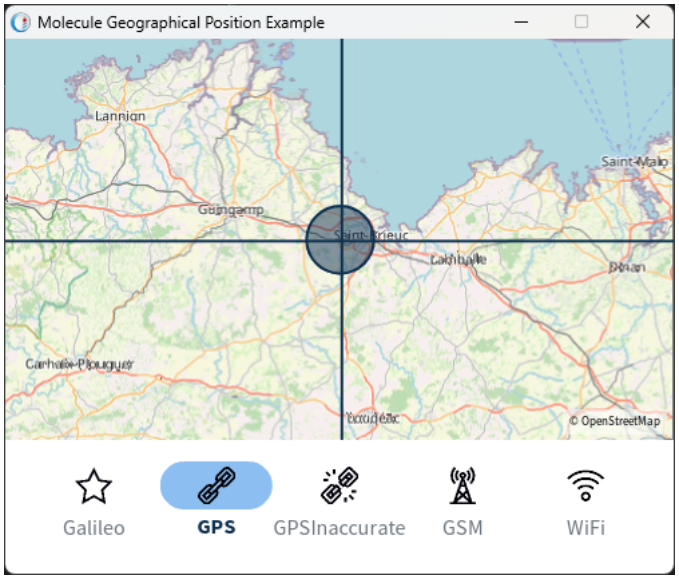

Figure 5: Snapshot of the Molecule Geographical Application Example.

system is selected, the previous one is stopped and the new one is started. The new geographical position is updated on the cartography view. The geographical position and precision can change depending on the performance of the selected location system. This example is extracted from a publicly accessible project [2]. The procedure to install and reproduce the example is available on the project page.

As depicted in Figure 6, in the first part, we program a component application that connects to a geolocation equipment (for example a *GPS* chip hardware) and displays the position and the precision (accuracy data) on a user interface (for example a map).

In the second part, we reuse an existing class that connects to a *Galileo* geolocation equipment in our Molecule application as a component. However, this second class is not a component, and we augment this class with component behavior to reuse it as a component. Finally, we dynamically switch between the GPS component and the Galileo component without stopping the application.

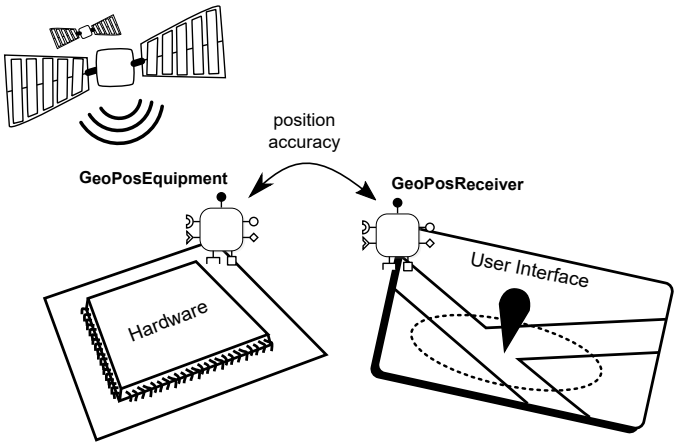

Figure 6: Geographical Position Application component overview.

5.1. *The map and the geolocation equipment example*

In this example, we create two Molecule components defined by 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 to provide a geographical position from a GPS chip. 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 syntactic sugar directly using the default Molecule Trait which injects component shared behavior into classes. The component Type is implemented through a Trait. In our example `GeoPosEquipmentType` is the Type Trait and `GNSSGPS` is the component class which uses it. GNSS is the generic term to describe a Global Navigation Satellite System like GPS or Galileo. In this step we implemented the GPS one.

```
MolAbstractComponentImpl subclass: #GNSSGPS
    uses: GeoPosEquipmentType.
```

Listing 1: Creation of `GNSSGPS` component class.

We define the `GeoPosEquipmentType` 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.

```
Trait named: #GeoPosEquipmentType uses:
    MolComponentType.
GeoPosEquipmentType class>>providedComponentServices
    <componentContract>
```

[2] https://github.com/OpenSmock/Molecule-Geographical-Position-Example

```
    ↑{ GeoPosEquipmentServices }
GeoPosEquipmentType class>>producedComponentEvents
    <componentContract>
    ↑{ GeoPosEquipmentEvents }
```

Listing 2: Declaring the `GeoPosEquipmentType` Type with one provided services interface and one produced events interface.

The method `providedComponentServices` (or `producedComponentEvents` for events) returns 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, `GeoPosEquipmentServices` and `GeoPosEquipmentEvents` (Listing 3). In these traits, we add methods that the `GeoPosEquipmentType` component Type has to implement.

```
Trait named: #GeoPosEquipmentEvents.
GeoPosEquipmentEvents>>currentPositionChanged:
    aGeoPosition
    "Notify geographical position when changed"
Trait named: #GeoPosEquipmentServices.
GeoPosEquipmentServices>>getAccuracyRadiusInMeters
    "Return the accuracy of the geographical position
     depending equipment precision. For example the
     quality of the signal, the quantity of connected
     satellites, etc. can impact this value"
```

Listing 3: Defining the `GeoPosEquipmentType` Type content.

We create a second component to display a geographical position on a map. The map displays a circle around the position whose radius represents the accuracy of the position. The second component requires services and events from an existing geographical position equipment component. In Listing 4, we add a new component class `GeoPosMapReceiver` (lines 1-2) with a new `GeoPosReceiverType` Type (line 3). The Type `GeoPosReceiverType` requires to consume services of the `GeoPosEquipmentServices` interface (lines 4-6) and to use events of the `GeoPosEquipmentEvents` interface (lines 7-9). Note that these interfaces are already defined and used by our first component `GNSSGPS`.

```
1 MolAbstractComponentImpl subclass:
      #GeoPosMapReceiver
2     uses: GeoPosReceiverType.
3 Trait named: #GeoPosReceiverType uses:
      MolComponentType.
4 GeoPosReceiverType class>>usedComponentServices
5     <componentContract>
6     ↑{ GeoPosEquipmentServices }
7 GeoPosReceiverType class>>consumedComponentEvents
8     <componentContract>
9     ↑{ GeoPosEquipmentEvents }
```

Listing 4: Creating `GeoPosMapReceiver` component class that uses `GeoPosReceiverType` Type with one used services interface and one consumed events interface.

As explained in Section 4.2, some methods are automatically generated to plug the services in the component class. Declaring the uses of `GeoPosEquipmentServices` automatically generates the corresponding accessor `getGeoPosEquipmentServices` which allows the `GeoPosMapReceiver` component to call the services methods. Declaring the consumption of `GeoPosEquipmentEvents` automatically generates a skeleton implementation of the interface in the component class `GeoPosMapReceiver` (Listing 5). Developers have to complete generated methods with their application code.

```
GeoPosMapReceiver>>currentPositionChanged:
    aGeoPosition
  | radius |
  "Get the accuracy value"
  radius := self getGeoPosEquipmentServices
    getAccuracyRadiusInMeters.
  "Display a circle on the map arround the updated
    geographical position"
  self updatePositionCircleOnMap: aGeoPosition radius:
    radius.
```

Listing 5: Consumed event: the method signature was generated (name and parameters) and the method body was completed by developers.

A component is expected to implement a particular method for each state introduced in Section 4.4. These methods are called when the component is switched to the corresponding state. The method corresponding to the *Activated* state 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. The `GeoPosMapReceiver` component subscribes to the default event provider. When `GeoPosMapReceiver` is activated, it subscribes to the `GeoPosEquipmentEvents` event provider. After the subscription, `GeoPosMapReceiver` receives the position change events to update its view accordingly.

```
GeoPosMapReceiver>>componentActivate
  self getGeoPosEquipmentEventsSubscriber subscribe: self
```

Listing 6: Subscription of `GeoPosMapReceiver` receives to `GeoPosEquipmentEvents`.

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

```
GNSSGPS start.
GeoPosMapReceiver start.
```

Listing 7: Start the components.

Note that `start` is a syntaxic sugar method. This method instantiate and activate a component as a default instance.

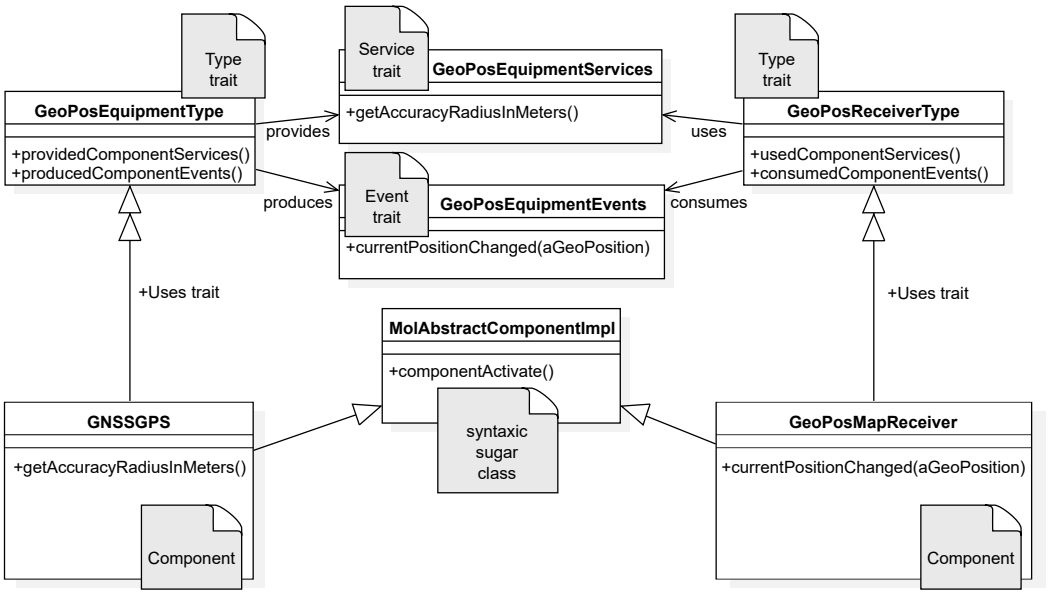

Figure 7: The map and the geolocation equipment example

5.2. *Reusing existing code as Molecule components*

Imagine that we want to reuse an open-source library that implements the support for the Galileo chip that we use. We want to reuse a class from this existing library in our application to add the capability to use the chip.

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 the architecture less understandable and harder to maintain.

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 `GNSSGalileo`. This class is originally used as described in (Listing 8). Developers must instantiate this class, then interact with its instances to use the Galileo chip.

```
1 driver := GNSSGalileo new.
2 "Use the library API to connect to the chip and get the
      accuracy"
3 driver connect.
4 accuracy := driver accuracy.
```

Listing 8: A `GNSSGalileo` class providing accuracy of the geolocation.

To use the `GNSSGalileo` class as a `GeoPosEquipmentType` Type component, we augment that class with Molecule component behavior and with the `GeoPosEquipmentType` Type (Listing 9). First, we

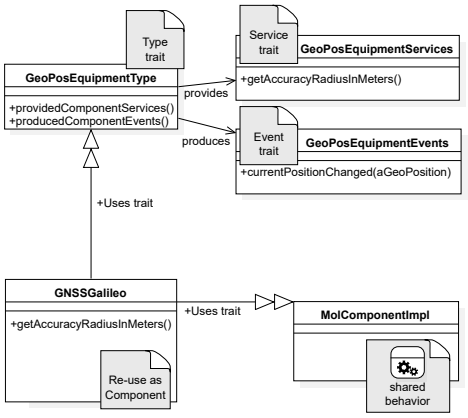

Figure 8: Reusing existing code as Molecule components.

add the Molecule component interface (*i.e.*, a Trait) `MolComponentImpl` to the `GNSSGalileo` class. Any class that implements this Trait is usable as a Molecule component. Then, we affect the `GeoPosEquipmentType` Type (also a Trait) to the `GNSSGalileo` class to make it a component.

The `MolComponentImpl` interface and `GeoPosEquipmentType` Type are implemented as Traits. Therefore the `GNSSGalileo` class is augmented just by declaring the use of these Traits (line 2). Traits automatically adds to the target class the code for implementing interfaces.

```
1 Object subclass GNSSGalileo
2     uses: {MolComponentImpl + GeoPosEquipmentType}
```

Listing 9: Augmenting the `GNSSGalileo` class with Molecule component behavior and set its Type.

After augmenting the `GNSSGalileo` class with com-

ponent behavior, we make sure the class implements the component contract in accordance with the `GeoPosEquipmentType` 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 delegation from the existing library to the `GeoPosEquipmentServices`

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

```
1 GNSSGalileo>>getAccuracyRadiusInMeters
2     "This GNSSGalileo class use meters units in this
      implementation, just write a pass—through"
3     ↑self accuracy
```

Listing 10: Reusing and extending the `GNSSGalileo` implementation in the Molecule component interface.

Finally, we instantiate this class as a Molecule component and we swap at run time the previous implementation `GNSSGPS` with our new implementation `GNSSGalileo` (Listing 11). We first stop the `GNSSGPS` component (line 2). At this moment, all other components subscribed to `GNSSGPS` cease to receive its events because their connection was severed. We then start the new component `GNSSGalileo` (line 4). Molecule automatically detects that `GNSSGalileo` implements the same contract as `GNSSGPS` (*i.e.*, `GeoPosEquipmentType`), and connects the new `GNSSGalileo` to all components requesting the `GeoPosEquipmentType` contract.

```
1 "Removing old GeoPosEquipmentType Type
      implementation from first part of the example"
2 GNSSGPS stop.
3 "Instantiating new GeoPosEquipmentType Type
      implementation with our re—use class as a component
      class"
4 GNSSGalileo start.
```

Listing 11: Dynamically swapping components.

Swapping components does not require restarting the entire application and can be performed at run time. In this example, the swap is illustrated in Figure 9. The GPS component is stopped, and the Galileo component is started. This swap remains transparent to the map component, which continues to display a received position and retrieve precision data through component events and services. This serves as a practical example of live prototyping, as depicted in Figure 1.

This second part of the example shows how to manage components using a new component that is not started.

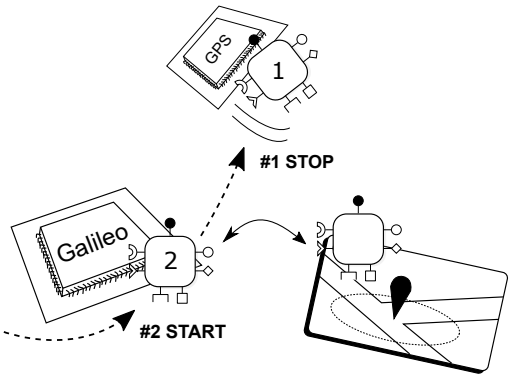

Figure 9: Swapping components at run time.

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.*, for example in multi-threaded applications). `GeoPosReceiverType` cannot get `GeoPosEquipmentServices` services between `GNSSGPS` and `GNSSGalileo` swap. In that case, Molecule automatically returns a default object that corresponds to a non-existing services provider: `MolNotFoundServicesProvider`. Components must handle this object when they require a service from another component and manage the case when a required component is not started.

We write the management case when a required component is not ready to request a service (Listing 12). To do that we use the method `isNotFoundServices` of a component services Trait to test if the required services is available.

```
1 services := self getGeoPosEquipmentServices.
2 services isNotFoundServices
3     ifTrue:["Services are not available here"]
4     ifFalse:["Services are available here"].
```

Listing 12: Case of not available services

6. Beyond prototyping: Industrial Software with Molecule

After many years of prototyping [1], Molecule is now stable and mature. Today, we build and sell industrial software based on Molecule and on the experience we gained over the years in architecturing component-based software.

In the following, we present two examples to illustrate the typical design of all component-based software in our industrial activity (Section 6.1). How we designed these examples is representative of how we design industrial prototypes. We then perform a brief analysis of these software architecture composition (Section 6.2) to compare them with the architecture composition of our prototypes [1].

6.1. Real-World Examples

In this section we present two real software we built with Molecule: the `Support Wizard` and *gRoom*. We use

these examples to illustrate how we design our component architectures.

6.1.1. The `Support Wizard`

Support Wizard is a real application with a rich user-interface. It is designed and built using Molecule. Figure 10 shows a snapshot of the Support Wizard. This application was first built and used as a show-casing tool at the *Paris Air Show 2023* to present Thales services to customers [3]. It allows us to simulate, given a customer, what would happen in the future depending on different scenarios built from the customer situation. The Support Wizard demonstration had so much success that it became a full application. Business employees use it every day to understand customers' specific problems and to build the best possible solutions and proposition for each customer.

Component architecture. The Support Wizard is structured into two layers of components: the back end management (simulation and data) and the front end management (GUI). The back end components are deterministically organized, started and used. Once started, these back end components do not change during the execution of the application. The front end components can be non-deterministically organized, started and used. During execution, some front end components can be stopped, restarted, or swapped with other components. For example, different *Area infos* can be displayed on and off when the user clicks on a graphical building on the `World map`. This implementation represents the principles and structure of component-based software architectures employed at Thales since many years.

Augmenting graphics classes to components. To illustrate how we use Molecule components in the Support Wizard, let us dive into an example focused on the front end. The application allows users to open panels that present description of Thales products. These panels (not illustrated here) are larger than the *Area infos* and can be displayed as an overlays above the entire GUI in Figure 10. These panels are graphical parts of the application, with text, illustrations, widgets and a background. Panels classes come from the *Bloc* graphical library[4]. Graphic designers designed and built the content of these panels with Bloc, and we wanted to reuse these nice-looking panels as Molecule components to integrate them into the Support Wizard architecture. We augmented these classes into Molecule components with a `PanelComponent` type. We built a `PanelManager` component to control panel components visibility (opening and closing).

Dynamic component switching. When the user clicks on a user-interface button to display the description of a product, the panel manager opens the corresponding product description panel. Each product description panel is dynamically started and connected to other components by the panel manager, and displayed by the user-interface. The panel manager contains different algorithms that process different situations. For example, if a previous panel was already opened for a product, the panel manager closes it before opening a new one. The panel manager can also save a stack of previously opened panels to restore them when requested by the user.

The Support Wizard application contains many kinds of panels components providing supplementary functionality, not only product description. All of these panels are started and stopped dynamically in the same way, regardless of their nature and contents. This is possible because the panel manager manipulates panel components from a Molecule component perspective, as any other Molecule component in the application. This pattern allows us to dynamically create, update and remove panels without impacting others parts of the application.

6.1.2. `GRoom`

GRoom is an office space allocation software designed for employees. It is collaborative and connected to a centralized database. The application offers a rapid overview of all offices and their availability on a specific date or during a designated period, presented through a map (Figure 11). Thales DMS used gRoom during the COVID-19 pandemic to ensure the continuation of critical activities supporting state entities while ensuring the health and safety of employees required to work on site. It was used to schedule several thousand offices per week for about 2500 employees across several cities [5].

GRoom is designed and built with Molecule. It marked the first instance of using Molecule for an end-user application rather than just a prototype for the defense industry. Building and using gRoom provided valuable feedback on Molecule's maturity for constructing real software applications.

Component architecture. Figure 11 displays a snapshot of gRoom, where we have annotated all elements that are either components or managed by components. Components are structured into two layers: the back end management (database and authentication) and the front end management (GUI). All theses components are deterministically organized, started and used. We chose this deterministic architecture to save time during development, and to quickly deliver a minimal functional application. Because everything was statically anticipated, it was easier to write tests as all possible components and component interaction combinations were known in advance.

Components as development skeletons. Molecule allowed us to developp this application in a short time with a small

[3] https://www.youtube.com/watch?v=zoRx704rLCI
[4] https://github.com/pharo-graphics/Bloc
[5] https://pharo.org/success/Groom.html

Figure 10: Support Wizard application main window. Each annotated section of the GUI represents a `Bloc` class augmented as a component.

team. It took two weeks for three developers working remotely during the first lockdown in France. First, all the team defined the component architecture and all its contracts (Component Types, Services, Events and Parameters) as empty shells. Second, each team member chose on what component implementation to work on, according to their skills and personal preferences. Developpers then implemented their assigned components and a single global merge was done after a few days. The application launched successfully on the first attempt. After this first launch, we did not perform any corrective phase. Contracts helped us to define clear and isolated architectural component blocks that each developer could implement and test separately.

Choosing components to deploy. To illustrate how we use Molecule components in gRoom, let us dive into an example focused on the user rights management. The user action manager is a component that allows users to perform actions according to their rights. It is a facade that exposes all application actions of the GUI to the users. For example, it allows users to consult the employees list or to edit someone else's planning.

At startup, before starting the main GUI, users connect with a user name and a password. The application connects to the database to get the correct access level for the connected user. The application then starts a user action component according to that access level, *i.e.*, *read only* or *administrator*. Each component of the GUI uses that component when executing actions or displaying data. Depending on the component implementation, the access level is complete or limited. Typically, a standard user will only have a *read only* access and cannot create an office desk in a plan.

GUI components expose all their actions through the `UserAction` type. This type is implemented for the read-only and the administrator rights. Read-only and administrator (read and write) rights have distinct implementations, but they communicate in the exact same way through the `UserAction` type requested by the GUI components. It is the chosen implementation of the `UserAction` type that determines the actual rights of the connected user (Figure 12).

6.2. Software analysis

We have an intense prototyping activity, for which we massively rely on component-based architectures and component reuse. In [1], we presented architectural data of 24 industrial prototypes over 15 years of component-based development with Molecule (our current standard) and its ancestors. In that work, we described our development procedures and strategies to favor reuse. We massively reused components through different prototypes, speeding-up our development time while ensuring the quality of the most reused components. On average, 80% of the components of our prototypes came from a component database that we built over the years [1].

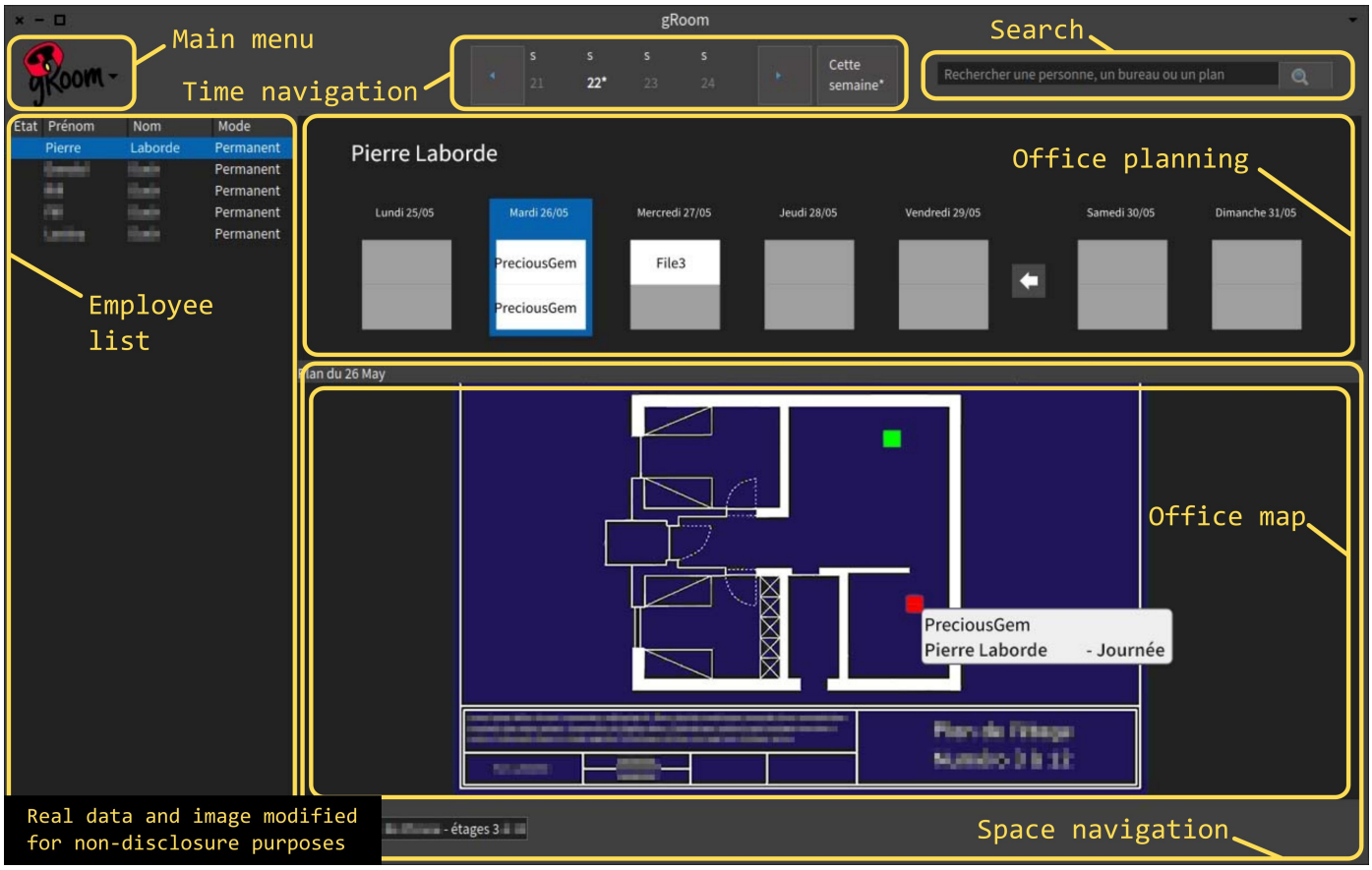

Figure 11: `GRoom` application main window. Each annotated section of the GUI represents components or elements managed by components. The sample data is copyrighted by © Disney Corporation.

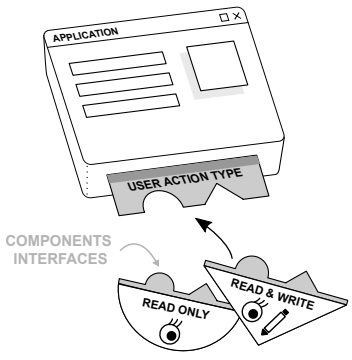

Figure 12: When the user logs in, a right-management component implementation is selected based on the user profile.

The Support Wizard and gRoom are increment compared to [1] and are also different. They represent application software destined to end users and not prototypes for live demonstration and evaluation. This is a new type of software for us, that is orthogonal to our activity in the defense industry. We do not reuse our previous codebase of components and sub-systems, but we build new components specifically for these applications. This shows empirically the versatility of Molecule, and that Molecule is not restricted to developing prototypes. Table 1 shows the architectural elements statistics of the wizard and gRoom.

For the Support Wizard, 43% of classes are components and 90% of traits are component contracts. Most component are not native components but classes from the Bloc library augmented as components. The wizard has more components (or type implementations) than types. The reason is that there are multiple implementation for a given type, each type providing a set of services, events and parameters. With this design, we can statically choose which type implementation we use when starting the software (*i.e.*, configure our software) and dynamically switch implementations at run time (*i.e.*, adapt our software such as described in Section 5). The Support Wizard back end just handles a data model, including a contracts manager, a services manager, etc. The startup components of the back end never change during run time as they are all necessary and utilized. The front end components however can change on-demand based on how the user decides to use the GUI. There may be different graphical views displayed depending on what the user chooses to show or hide, open or close menus, etc. Therefore, not all front end components are always necessary and they are started and stopped as needed.

For gRoom, 25% of classes are components and 100% of traits are component contracts. Most components are na-

tive components, developed specifically for gRoom. There is an equal number of types and implementations, *i.e.*, each type has its own implementation. The explanation of this 1-1 type to implementation mapping is two-fold. First, the gRoom component architecture has a static configuration that is set at startup and never changes during run time. We therefore did not need to dynamically interchange components, and type reuse was not a constraint. Second, gRoom is the first end-user application we built, and it was different from our prototyping activity where type reuse came from legacy component sub-systems. When building the Support Wizard, we needed to interchange components at run time and therefore learnt to reuse types in that kind of application.

We did not made the components of these two applications reusable, *i.e.*, they are not in our component database and they are only meant to work in these respective applications. Making components (*i.e.*, the types and their implementations) reusable across software is difficult and costly. We only do that when we can anticipate sufficient reuse of the components which would justify that cost, and sometimes we fail [1]. That decision may come if we require similar or identical components in future software projects, being prototypes or end-user applications.

	Support Wizard	*gRoom*
General		
Total Classes	65	53
Total Traits	21	32
Molecule components		
Native components	6	12
Augmented classes	22	1
Total	*28*	*13*
Molecule interfaces		
Types	6	13
Services	7	12
Events	6	6
Parameters	0	1
Total	*19*	*32*
Non-Molecule entities		
Classes	37	40
Traits	2	0
Total	*39*	*40*

Table 1: The gRoom application (2020) and the Support Wizard application (2023) architectures composition.

7. Discussion

In this section, we will explain why Thales chose to build and use a component model for its industrial software architectures and why Molecule, used in prototyping phase, is based on the Lightweight Component Model.

The benefits of component architectures. The purpose of components is reuse. Component-based development allows for a reduction in the cost of maintenance and development [25]. In terms of reusability, Molecule provides the same well-known architectural and reuse benefits of software components [2, 3, 4, 5]. An important part of live prototyping is putting blocks together to build a part or adapt software in front of a customer as quickly as possible. To assemble blocks we need an explicit model of the interfaces and this is what components specifically enable. In [26], an interface is defined as a way to describe the interactions (and the conditions of these interactions) with other components or the environment. In our business the inherent ability of components to be connected via a clearly defined interface is a fundamental advantage.

The choice of the Lightweight CORBA Component Model. Thales developed its own CORBA Component Model (CCM) compliant framework to take advantage of reusability and adaptability. Using CCM was motivated by a pragmatic approach because *CORBA Distributed System* was used in the context of services exchanges between our distributed software and because CORBA was well known by Thales architects.

The prototyping activity takes place upstream of the industrialization phase of a system at the end of which the final system can be delivered to the customer. To minimize the cost of the industrialization phase, it was necessary to build our prototypes on an architectural base that was reproducible and naturally intelligible during the industrialization phase. This is the main motivation that led us to the development of Molecule.

Prototyping with Molecule helps us to anticipate architectural needs and problems before development of real products begin [27]. As described in [6], the goal of the LwCCM is to simplify as much as possible the use of CCM for embedded applications and to avoid presenting the user with multiple ways to do the same thing. Thus, applications build with LwCCM remains compatible with *standard* CCM based components. Indeed, according to LwCCM, a component model still provides interfaces and events. Moreover, LwCCM has the essential properties of the CORBA Component Model: component life-cycle, and tools for managing and deploying components. Relying on LwCCM for prototyping allowed us to minimize development costs of prototypes. Today if we had to choose again, we might not use LwCCM because CORBA is no longer as widely used in software development at Thales. However, the development model it implies remains current and still meets our needs regarding live prototyping.

Run-time reconfiguration and modification. Run-time adaptation in component systems has been well studied. For instance, and non-exhaustively, there is work on dynamic component modification [28, 21, 29], with safety [30] and consistency [31] concerns. These systems generally only provide run-time modification of component-related entities 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 issues in third party libraries. Smalltalk-based component models implementations [21, 32, 33] enable run-time modification of both component and non-component code thanks to Smalltalk's reflective properties. Thus, the Molecule implementation provides dynamic, unanticipated run-time changes that allow us to dynamically modify our prototypes in front of customers and as far as we know, Molecule is the only of Smalltalk LwCCM implementation.

8. 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 products are based on components, 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 live adaptation of our prototypes is also facilitated by the component paradigm. In this paper we presented Molecule, a Lightweight CORBA Component Model implementation based that we have developed and matured over 20 years in this context. We presented an overview of Molecule and illustrated its usage through examples. Molecule is now mature, and we started developing end-user software deployed and used in production at Thales DMS.

In the future, we will improve Molecule in three areas regarding component specification, asynchronous event handling and distributed management of components.

1. Users must be able to specify components in a standard way. We will give the possibility of using a standard syntax, *e.g.*, *Component Implementation Definition Language* (CIDL) [19]. Adding a CIDL will facilitate interoperability of specifications with other component systems beyond the Pharo environment.
2. Currently, Molecule does not support asynchronous event handling, leaving developers to manually handle these types of events. This manual handling can cause system freezes and performance slowdowns when processing events, especially if calculations take place after receiving such messages. This is challenging as this implies consistent modifications of remote and distributed run-time architectures.
3. Nowadays, many applications are implemented in a distributed manner on a network. As a consequence our prototypes tend to involve multiple users on collaborative tasks on interconnected workstations. This increases the cost of prototyping and adds difficulties in the context of live prototyping. Thus, the possibility of distributing components over the network must be implemented.

References

[1] P. Laborde, S. Costiou, É. Le Pors, A. Plantec, Reuse in component-based prototyping: an industrial experience report from 15 years of reuse, Innovations in Systems and Software Engineering 18 (1) (2022) 155–169.

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

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

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

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

[6] J. K. Olivier Hachet, F. Pilhofer (2002). [link].
URL https://staff.info.unamur.be/ven/CIS/OMG/lightweight%20component%20model.pdf

[7] Corba Components Package, Corba Components and Scripting, http://www.omg.org/technology/corba/corba3releaseinfo.htm.
URL http://www.omg.org/technology/corba/corba3releaseinfo.htm

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

[9] P. Laborde, S. Costiou, A. Plantec, E. Le Pors, Molecule: live prototyping with component-oriented programming, in: International Workshop on Smalltalk Technologies - IWST 2020, 2020.
URL https://hal.inria.fr/hal-02966704

[10] P. Laborde, E. Le Pors, Y. Le Goff, A. Plantec, S. Costiou, Evaluations ergonomiques d'interfaces humain-machines pour la défense : un exemple de systeme de surveillance maritime, IHM'24 - 35e Conference Internationale Francophone sur l'Interaction Humain-Machine (2024).

[11] C. Letondal, P.-Y. Pillain, E. Verdurand, D. Prun, O. Grisvard, Of models, rationales and prototypes: Studying designer needs in an airborne maritime surveillance drawing tool to support audio communication, in: HCI 2014, 28th BCS International Conference on Human-Computer Interaction, 2014, pp. pp–8.

[12] O. Grisvard, C. Huntzinger, M. Le Berre, V. Verbeque, Cost-efficient design and production of flexible and re-usable near real-time tactical human-machine interfaces, in: Embedded Real Time Software and Systems (ERTS2008), 2008.

[13] D. Ungar, H. Lieberman, C. Fry, Debugging and the experience of immediacy, Communications of the ACM 40 (4) (1997) 38–43.

[14] C. M. Hancock, Real-time programming and the big ideas of computational literacy, Ph.D. thesis, Massachusetts Institute of Technology (2003).

[15] S. McDirmid, Usable live programming, in: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, 2013, pp. 53–62.

[16] S. Burckhardt, M. Fahndrich, P. De Halleux, S. McDirmid, M. Moskal, N. Tillmann, J. Kato, It's alive! continuous feedback in ui programming, in: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 95–104.

[17] A. Aguiar, A. Restivo, F. F. Correia, H. S. Ferreira, J. P. Dias, Live software development: Tightening the feedback loops, in:

Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming, 2019, pp. 1–6.

[18] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983.
URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

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

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

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

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

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

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

[25] Y. Kermarrec, Approches et expérimentations autour des composants - applications aux composants logiciels, aux objets d'apprentissages et aux services distribués (2005) 7.

[26] I. Crnkovic, M. P. H. Larsson, Editors Building Reliable Component-Based Software Systems, Artech House, 2002.

[27] P. Laborde, S. Costiou, E. Le Pors, A. Plantec, 15 years of reuse experience in evolutionary prototyping for the defense industry (2020).
URL https://inria.hal.science/hal-02966691

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

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

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

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

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

[33] P. Spacek, C. Dony, C. Tibermacine, 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, 2014, pp. 13–22.