Reuse in component-based prototyping: an industrial experience report from 15 years of reuse

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

Noname manuscript No.
(will be inserted by the editor)

Reuse in component-based prototyping: an industrial experience report from 15 years of reuse

Pierre Laborde* · Steven Costiou · Éric Le Pors · Alain Plantec

the date of receipt and acceptance should be inserted later

Abstract At Thales Defense Mission Systems, software products first go through an industrial prototyping phase. We elaborate evolutionary prototypes which implement complete business behavior and fulfill functional requirements. We elaborate and evolve our solutions directly with end-users who act as stake-holders in the products' design. Prototypes also serve as models for the final products development. Because software products in the defense industry are developed over many years, this prototyping phase is crucial. Therefore, reusing software is a high-stakes issue in our activities. Component-oriented development helps us to foster reuse throughout the life cycle of our products. The work presented in this paper stems from 15 years of experience in developing prototypes for the defense industry. We directly reuse component implementations to build new prototypes from existing ones. We reuse component interfaces transparently in multiple prototypes, whatever the underlying implementation solutions. This kind of reuse spans prototypes and final products which are deployed on different execution platforms. We reuse non-component legacy software that we integrate in our component architectures. In this case, we seamlessly augment standard classes with component behavior, while preserving legacy code. In this paper, we present our component programming framework with a focus on component reuse in the context of evolutionary prototyping. We study our prototypes composition through data we extracted from our code repository. On these last 15 years we observe that, in average, our prototypes are composed of 80% reused components. Today, some of our prototypes can be constituted of more than 96% reused components. In this context, we report three reuse scenarios we regularly encounter in our prototyping activity, and that motivates our engineering efforts towards reusable components.

Keywords Evolutionary prototyping · Component reuse · Traits · Pharo

*Pierre Laborde (corresponding author) · Éric Le Pors
THALES Defense Mission Systems France - Établissement de Brest, 10 Avenue de la 1ère DFL, 29200 Brest, France
E-mail: {pierre.laborde,eric.lepors}@fr.thalesgroup.com

Steven Costiou
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France
E-mail: steven.costiou@inria.fr

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

1 Introduction

In the defense industry, software systems are designed, developed and evolved over many years. It is important to evaluate and to adjust systems' design ahead of time before starting long and costly developments.

At Thales Defense Mission Systems (DMS), the Human-Machine Interface (HMI) industrial prototyping activities are an important part of the software production process. We build complete software systems from business to technical requirements in order to evaluate and validate the end-user HMI (graphics and ergonomics). We make these prototypes as close as possible to real products, and we use these prototypes to evaluate software HMI design and experiment complete use-cases with end-users. This enables early and strong feedback loops between developers and end-users. 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.

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, we have been using a component based software engineering approach [19,13] since 15 years. Prototypes we delivered over this period are, in average, composed of 80% reused components. Today, some of the prototypes we build are composed of more than 96% reused components. In our prototyping activity, component-based architectures bring the necessary modularity to enable reuse and facilitates the evolution of prototypes.

We extend our previous work based on one industrial prototype [10] with the study of 23 additional prototypes resulting from 15 years of prototyping. We complete our analysis and discussions with these new data. Overall, we observe that, in our industrial context:

- Through component-oriented programming, we benefit from high rates of reuse;
- Building reusable components does not guarantee that these components will actually be reused;
- Reuse expertise depends on a few experts, and this limits reuse opportunities.

In this paper, we present our requirements for reusable software in our industrial context. We describe how component-oriented programming helps us fulfill these requirements (Section 2). We present how we implement components for modular and reusable software architectures with the *Molecule* component-oriented programming framework (Section 3). We present our research method and objectives (Section 4), then we report the three reuse scenarios we regularly encounter in our activity (Section 5). We present reuse data for 24 industrial prototypes (Section 6). Finally, we discuss 15 years of reuse experience and the difficulties we face today (Section 7).

2 Reuse in prototyping for the defense industry

Exploring design ideas through Concept prototyping [4, 14] is one of the main activities at Thales DMS. We develop prototypes to communicate concepts to users, demonstrate the HMI usability and exhibit potential problems [10]. Moreover, we develop and maintain prototypes until they meet users expectations regarding not only the HMI, but also the main business functionalities. In some of our prototypes, we implement complete business behavior and fulfill functional requirements. Thus, they are also kind of evolutionary prototypes [14]. Thales DMS engineers maintain a lot of them since 15 years.

Maintaining evolutionary prototypes is a very expensive activity and Thales struggled with evolution issues. Building a final product is also an expensive and a tedious task, and engineers must take advantage of the prototyping activity. Furthermore, the prototyping and the final product teams have to fully understand each others' designs. This poses a knowledge sharing issue.

We use component-oriented programming to build and to maintain evolutionary prototypes, and to share knowledge between prototyping and industrialization activities. In this context, this section introduces briefly the component model we use in prototyping, and the overall benefits of component-oriented programming that we observed in our development process.

2.1 The Molecule component model

Our component model is close to the light-weight CCM [1]. We use Molecule, a Thales open-source implementation[1] of the light-weight CCM.

A Molecule component implements a contract defined by its Type (Figure 1). A contract consists in a set of services that the component provides, a set of services that the component uses, a set of events that the component may emit, and a set of events that the component is able to consume. The Type implements the services that the component provides and that are callable by other components, and defines the events that the component produces. Other components use its *Provided Services* interface through their *Used Services* interface. Other components listen to the component's *Produced Events* interface through their *Consumed Events* interface. Components subscribe and unsubscribe to event interfaces to start and stop receiving events. Parameters are specific and are not present in the CCM. We use parameters to control components' state, and only once when initializing components.

2.2 Components as reusable modules

At the beginning of a project, we benefit from direct reuse of previous prototypes' components. Through composition of existing components, we reduce the time and efforts to get to the first versions of a new prototype.

Being able to build reusable and homogeneous software architectures was our first motivation for using

[1] https://github.com/OpenSmock/molecule

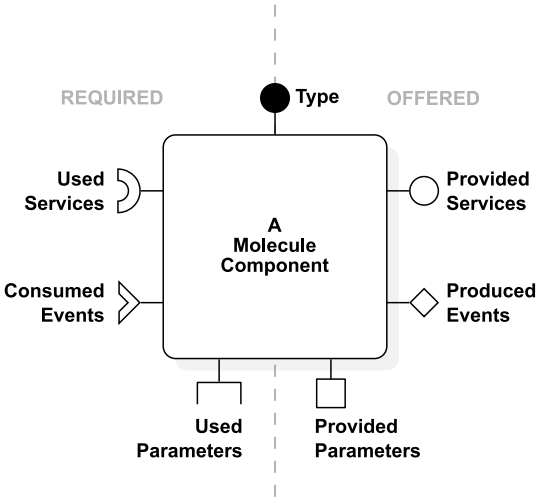

Fig. 1 Public view of a Molecule component.

components. This is particularly true for Graphical User Interfaces (GUI) components that we call Panels. Our panels expose a stable public interface and implement a standard contract. We therefore directly reuse our panels in several prototypes. The reused parts include not only the contract but also its implementation provided by the Type. This kind of reuse is now possible because we capitalized a sufficient quantity of components. After a decade, we benefit from a virtuous cycle where a finished prototype is kept in a prototype repository and some of its component set can be reused later for another prototype. This virtuous cycle tends to minimise the number of prototypes that we need to implement from scratch.

In Figure 3, we show our prototyping reuse chain. From previous legacy code, we build standardized and reusable components, defined by their contracts. When a component assembly covers a technical or a business requirement, it becomes a subsystem (*e.g.*, a geographic view with layering, filtering, selection...). Prototypes (re)use and deploy instances of subsystems and/or of single components. Each time we build a new prototype, we identify new functionalities or reusable bricks and we integrate them back into the repository of existing components or subsystems for future reuse.

2.3 Components to ease evolution

Change and evolution of software is always an issue. Switching to another technology implies modifying the existing software code to integrate that technology. This kind of change hinders our ability to reuse software to evolve our prototypes. For example, GUI technologies are often changed, because they evolve, they become obsolete or they stop being maintained. A new GUI framework will expose different interfaces and trigger different events. This forces developers to rewrite the same GUI in different technologies over the years [8, 21].

Without components, this kind of change forces us to change how GUIs interact with legacy or business software. It implies adaptation of such software, which will impact all using projects, or force us to maintain different versions of the adapted code. This is not desirable as, *e.g.*, we have legacy subsystems in use since a decade by multiple maintained prototypes. Because our prototypes are composed of components, in case of such a change, only a well identified part of the system has to be adapted (Figure 2). We evolve the related components implementation without changing GUI clients nor legacy and business software.

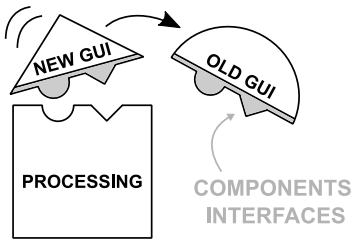

Fig. 2 Components to ease evolution, for example: changing GUIs.

2.4 Engineering impact: design traceability and co-working

Thales DMS' design process involves different and specific teams that work together throughout the life cycle of a product. The system engineering team builds systems' design models using dedicated methods like Arcadia [22] and modeling tools like Capella [5]. System engineers model interactions between hardware entities, software entities, users, etc. These models are produced early in the design of a product. The prototyping team develops and evaluates prototypes based on these models. The software engineering team develops final products.

The prototyping team is composed of 5 to 12 engineers. For each product, there are 10 times more system and software engineers working with the prototyping team. The descriptive properties of components models (interfaces, interaction models...) ease communication between different teams working with different technologies (Figure 4). For example, when building a GUI panel, the system team uses components to specify interactions between end-users and the panel. From these specifications, the prototyping team builds a first version of the panel to evaluate these interactions with

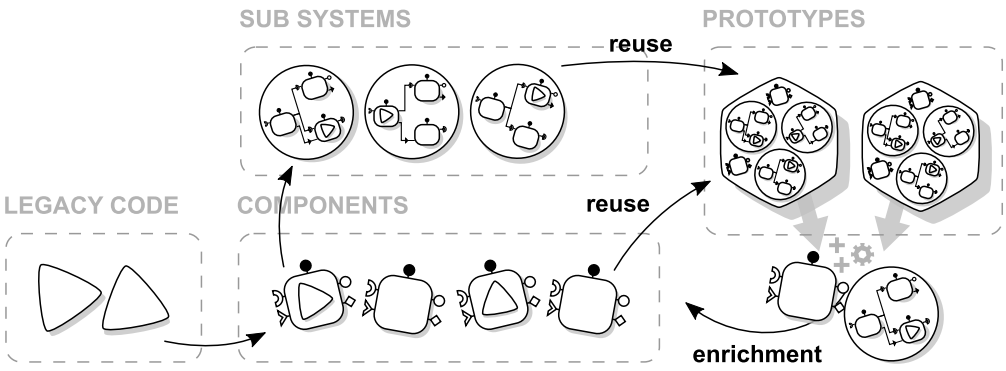

Fig. 3 From components to prototypes: a reuse chain.

end-users. Finally, the software engineering team builds a stable and robust version of the panel, connected to the real product's environment.

Using a common vocabulary also favors reuse, as components can be inserted and (re)used in an architecture solely based on their contract, without having to master implementation details.

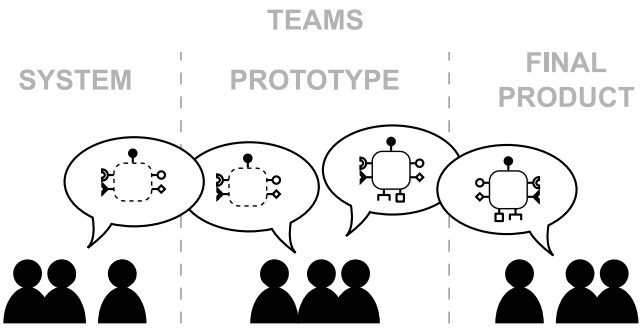

Fig. 4 Engineering impact: design traceability and coworking.

3 Component-oriented programming with Molecule

Since 2005, Thales DMS use Smalltalk for prototyping. Our main motivations behind the choice of Smalltalk is the ability to quickly design and program complex prototypes, and the capabilities to lively change a design in the front of customers [9,10,17]. Since 2016, we use Pharo [7] with the Molecule component-oriented programming framework. We developed Molecule to capitalize on our experience in component-based development. In this section, we briefly describe how we program components with Molecule.

The Molecule framework relies on Traits [6,18,20] to implement component Types. A Trait is an independent set of methods (like a Java interface) and variables. A Trait provides implemented methods and explicit implementation requirements (or both). Classes using that Trait automatically acquire all implemented methods from the Trait, and developers have to implement all methods specified as required by the Trait. Those classes can also redefine some of the implemented methods from the Traits they use.

A Trait can be composed of multiple other traits. In this case, a class using this composed Trait acquires all implemented methods provided by the Traits composition, and developers have to implement all requirements coming from the Traits composition. Traits provide orthogonal behavior to all classes using them, regardless of their class inheritance hierarchies.

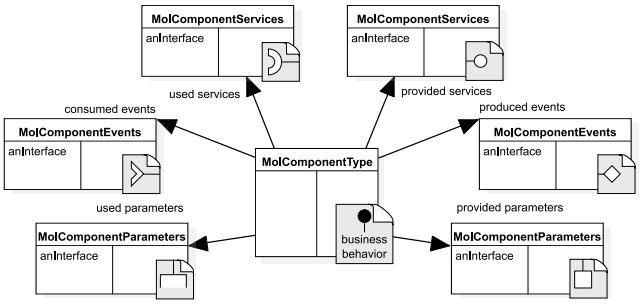

Fig. 5 Component contract implementation in Molecule.

In Molecule, we define the elements of a component's contract (services, events, parameters) as a set of Traits (Figure 5). A component Type aggregates theses traits, and is itself defined as a Trait.

Molecule provides a dedicated Trait MolComponentImpl, which implements cross-cutting behavior shared by all components (*e.g.*, components' life-cycle management). Implementing a component consists in defining a class that (1) uses the MolComponentImpl Trait to obtain component shared behavior and (2) uses a Type Trait (MolComponentType) implementing the component's business behavior (Figure 6).

The direct benefit of this approach is that it is possible for any existing class to become a component. This existing class is then turned as (or augmented as) a

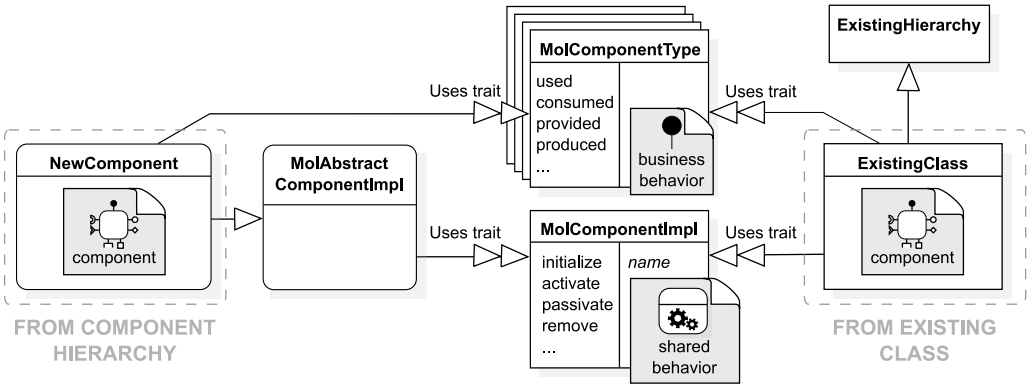

Fig. 6 Two ways to implement a component.

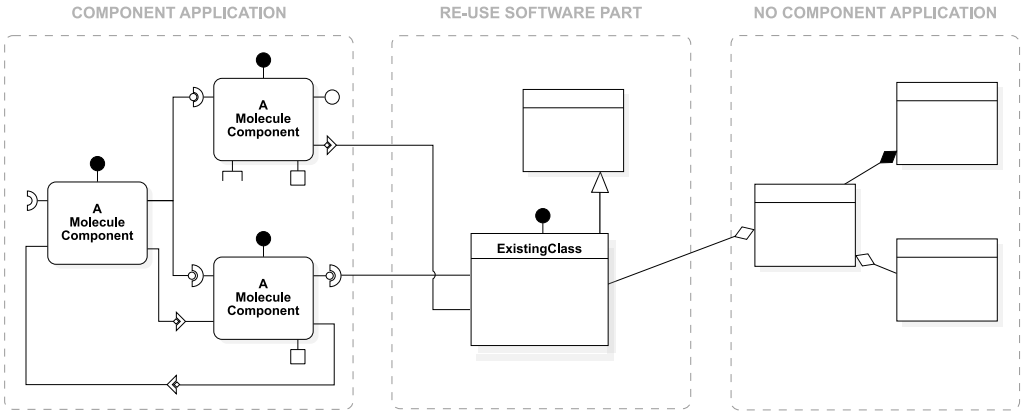

Fig. 7 We use augmented classes in component and non-component applications.

component, and becomes usable in a Molecule component application while remaining fully compatible with non-component applications (Figure 7).

As an example, this how we reuse non-component classes from open-source frameworks in a component-based prototype:

1. First, we inherit from the framework classes we want to reuse,
2. second, we apply the Molecule component Traits to the inherited classes to augment them with component behavior.

These augmented non-component classes then become reusable transparently by both standard applications and by our component systems.

Molecule provides syntactic sugar to directly implement components by inheriting from the MolAbstract-ComponentImpl class (Figure 6). In that case, we only need to satisfy (2) by using a Type Trait for our component.

The evolution of components is driven by Traits. For example, to provide new services, an augmented class will use additional Traits defining these services. The integration of the Traits implementing the component contract with the original class may require some code adaptation. Typically, when we reuse classes from legacy software as components (*e.g.*, a radar class), we have to adapt how the component's Type (*i.e.*, it's business behavior) interacts with the class API (*e.g.*, by converting the radar's accuracy from feet to meters [10]). This code adaptation ensures that the augmented class provides the correct level of information in regards with the requirements of the connected components.

The technical details about how we use Molecule to implement reuse in our prototypes is outside the scope of this paper. However, we described examples illustrating our industrial practice in another paper [11], as well as live demonstrations[2], tutorials[3], and a full example application[4].

4 Research method and objectives

The last 15 years, we spent considerable efforts in building practices and tools (described in the above sections) to make our code reusable. We wanted to know what

[2] https://github.com/OpenSmock/ReuseTale
[3] https://github.com/OpenSmock/Molecule
[4] https://github.com/OpenSmock/LittleSweetHome

was the real impact of our reuse practices on our prototypes. Today, and informally, in our day-to-day work we have the impression to systematically and massively reuse our components and subsystems to build new prototypes. Does the component-based structure of the prototypes we published over the last 15 years really reflects this informal feedback?

To answer our question, we studied the list of prototypes we built and published in our code repository since 2007. We defined a set of metrics to understand the code composition of these prototypes. Especially, we aimed at observing how much code is and is not reused over the years.

Definitions. We first give general definitions on which we rely to define our metrics:

- *Reuse.* For a given prototype, we consider that a class or a component has been reused if it is part of an identified subsystem that has been reused. We consider that a subsystem is reused each time it is included in the code of a prototype we published in our code repository.
- *Developer.* A developer involved in a given prototype is a developer with at least one commit in that prototype code repository.

Metrics. The list of metrics we defined to study our reuse practices is the following:

- *Number of components in time by subsystem*[5]. we counted the number of components composing a subsystem at each release date of that subsystem.
- *Reuse count of subsystems.* We counted in how much prototypes each subsystem is used.
- *Reuse Rate (per prototype) of classes and components:* To obtain the reuse rate of components, we counted them in each subsystem composing a prototype and divided that count by the total number of components composing the prototype. We applied the same method for classes.
- *New classes and components written specifically for a given prototype*[6]. We subtracted, for each prototype, the count of reused components form the total number of components of that prototype. We applied the same method for classes.
- *First publication of each subsystem.* It is the date of the oldest publication of each subsystem in our code repository.
- *First time a subsystem has been reused in a prototype.* It is the oldest publication date of a prototype reusing a given subsystem.
- *The number and identification of developers per prototype.* We identified and counted all developers involved in each prototype development. Two of the authors of this paper are Thales DMS experts included in this metric.

Metrics computation and data extraction. To compute these metrics, we extracted the raw data from 23 prototypes. 20 of these prototypes are versioned under a configuration management infrastructure, namely our *code repository*. We implemented scripts to query our code repository[7] and automatically retrieve the raw data from which we computed the metrics presented in this paper. 3 of these prototypes dated from before 2011 (prototypes A, B and E), before we started using a configuration management tool. The source code of these prototypes is only archived locally on disk. To recover data from these prototypes, we used *ad-hoc* source code analysis on the last published version of each prototype. No data were manually extracted. Raw data, computed metrics are available online[8] along with artifacts to reproduce metrics computation from the raw data.

We collected the following data:

- The number of classes, components and subsystems in each prototype,
- the list of subsystems in each prototype,
- the list of developers involved in a prototype development,
- the publication date of each subsystem version, and of each prototype.

5 Common reuse scenarios in prototyping

We systematically practice reuse for building our prototypes, using our Molecule framework and its methodology. We identified three different scenarios: (1) long-term reuse of legacy code, in which we reuse whole components' code in our component architectures, (2) reuse of non-component frameworks, in which we reuse classes that we augment as components usable in our component architectures, and (3) reuse of component business interfaces, in which we reuse components' business contracts to transparently interchange component implementations without touching the component architectures. We present theses three scenarios based on the reuse data we extracted from our code repository.

5.1 Long-term reuse of legacy code

Reusing legacy code is the most common case of reuse in our prototyping activity. For each new prototype, we

[5] This gives a view of how subsystems evolve in time in terms of components

[6] *I.e.*, the non-reused code rate per prototype

[7] The structure of our repository is described in Annex A

[8] https://github.com/OpenSmock/ReuseTale

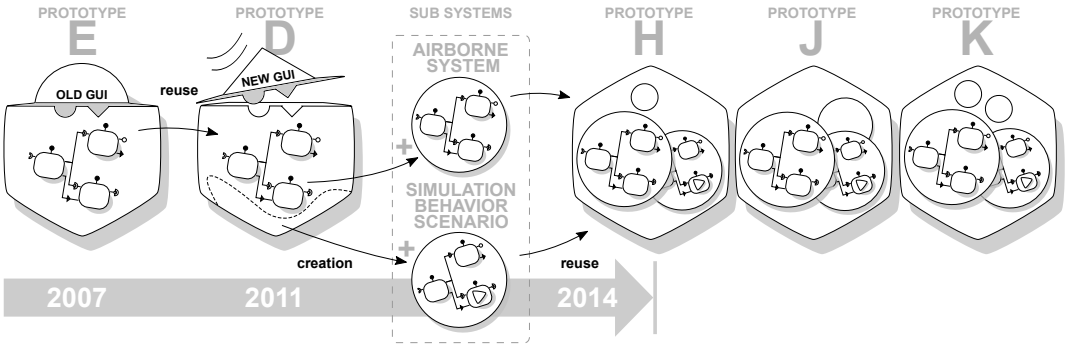

Fig. 8 From prototypes to reusable subsystems.

systematically need to reuse functionalities from previous prototypes or from legacy code.

Table 1 and Figure 8 show a reuse story that started with the Prototype *E* (implemented in 2007). Prototype *E* was using an old GUI technology. As this prototype was component-based, we were able to redesign the HMI with another GUI technology without modifying the component architecture. The complete business layer of this old prototype continued to work transparently with the new HMI design (as in Figure 2). From *E* to *D*, the main evolution concerns the *Tactical Model and Geographical View* sub-system that has been deeply redesigned. We simplified the code of *E*, replaced the GUI, and *E* became the standalone prototype *D*. The new GUI code was transferred to the *Prototyping Tools* sub-systems and therefore became reusable.

While building *D*, we actually isolated a completely reconfigureable prototype. In 2013, from *D*, we extracted two sub-systems: the *Airborne System* and the *Simulation Behavior Scenario* subsystems. These subsystems have been made reconfigurable (through configuration files) and then fully reused in the three prototypes *H*, *J* and *K*. Table 1 shows that these two sybsystems remain relatively stable, mainly in term of components. In addition, we can see that prototypes *H*, *J* and *K* are exclusively composed by their sub-systems. They contain no additional components and no additional classes. Since then, the *Airborne System* and the *Simulation Behavior Scenario* sub-systems have been reused respectively in 7 and 10 prototypes.

	Components	*Classes*
Prototype E	552	89
Tactical Model and Geo. View	221	290
Prototyping tools	0	7
Component framework	3	11
Total	*776*	*397*
Prototype D	301	66
Tactical Model and Geo. View	41	434
Prototyping tools	6	447
Component framework	3	11
Total	*50*	*892*
Prototype H	0	0
Airborn System	217	82
Simulation/behavior scenario	25	72
Tactical Model and Geo. View	150	481
Prototyping tools	21	591
Component framework	3	11
Total	*416*	*1237*
Prototype J	0	0
Airborn System	233	85
Simulation/behavior scenario	30	81
Tactical Model and Geo. View	146	476
Prototyping tools	21	593
Component framework	3	11
Total	*433*	*1246*
Prototype K	0	0
Airborn System	237	142
Simulation/behavior scenario	27	73
Tactical Model and Geo. View	154	513
Prototyping tools	50	607
Component framework	3	11
Total	*471*	*1346*

Table 1 Subsystem composition of five prototypes from 2007 to 2017. The first row of each prototype shows the prototype name with the number of additional components and classes developed specifically for that prototype. Below each prototype are shown their subsystems' components and classes.

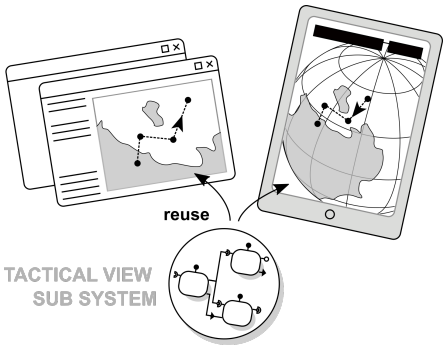

Fig. 9 Reuse of *Tactical Model and Geographical View* subsystem on different platforms with different display technologies.

Another concrete example of a massively reused legacy subsystem is the *Tactical Model and Geographical View*. This subsystem provides a geographical business objects display on a map. It is a recurrent require-

ment throughout our different prototypes. In particular, this subsystem must be reusable in different technological contexts and platforms (Figure 9). Figure 10 shows the evolution of the *Tactical Model and Geographical View* over 10 years, in term of numbers of components. We can see that from around 2013, this subsystem is relatively stable compared to the previous years. Since then, this sub-system encounters significantly less bugs and we consider it as legacy. Figure 11 shows that we reused this sub-system in 22 prototypes out of 24. This is one of the most successful reuse cases of our prototyping activity.

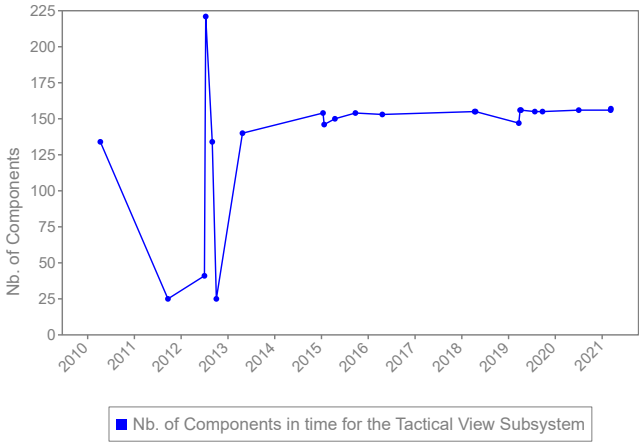

Fig. 10 Evolution of the *Tactical Model and Geographical View* sub-system from 2010 to 2021.

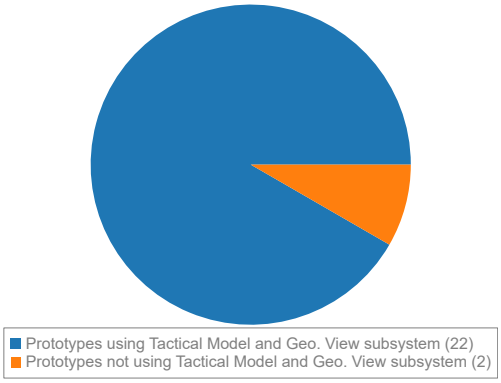

Fig. 11 Reuse of the *Tactical Model and Geographical View* subsystem in prototypes, from 2010 to 2021.

5.2 Reuse of non-component frameworks in component architectures

We encounter this scenario each time we need to reuse non-component code (*i.e.*, classes) in our component-based architectures. We rely on many object-oriented open-source frameworks in our prototypes: UI models, graphics engines, visualization engines, etc. As presented in section 3, with Molecule we can directly reuse regular classes as components. We augment classes from open-source frameworks to reuse them in our component-based prototypes (Figure 13).

For example, we regularly need to reuse graphical, non-component elements from open-source graphics engine. This happens when we want to extend a component subsystem to support another graphics engine. Therefore, we augment classes of graphical elements from the new graphics engine as components to directly connect them to our component subsystems. This allows us to add new graphics technologies as display backends with very few adaptations.

5.3 Reuse of component business interfaces

In this scenario, we reuse components' business contract to replace component implementations transparently (as in Figure 5). We express interactions between components through component contracts, which are reusable for different components implementations. Components exposing the same contract are seamlessly interchangeable.

For example, we had a prototype for which we needed to migrate the database system to a new database backend. The database access (*i.e.*, database requests) was implemented in a dedicated component. High-level data access (*i.e.*, data requests) was defined in that component Type (*i.e.*, its Type Traits). Other components communicated with the database through the contract defined by the Type, *i.e.*, to request data without knowing about the database access details.

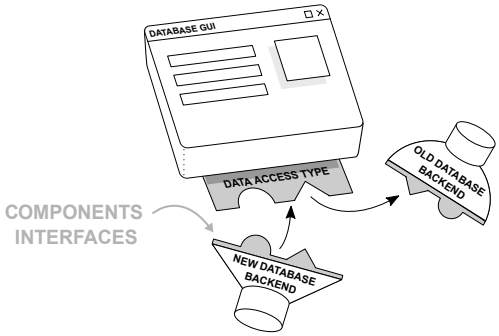

Fig. 12 Transparently switching a component implementation by reusing that component's business interface.

To change the database, we implemented a new component for a new backend and we reused the Type

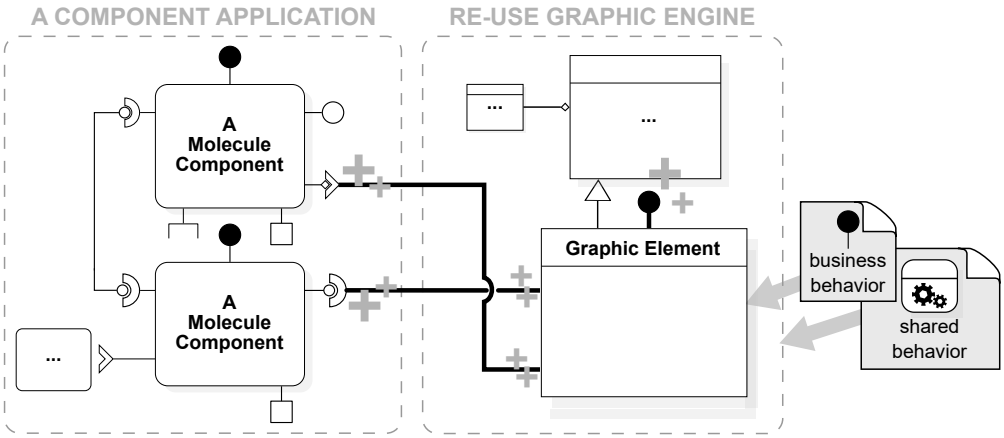

Fig. 13 Reusing graphical, non-component classes from open-source graphics engine into a component application.

Traits for high-level data access. We were then able to transparently switch the old component by the new one in the prototype (Figure 12).

6 A summary of 15 years of reuse in industrial prototyping

In this section, we summarize the impact of component-oriented programming and the systematic organization of these components in reusable subsystems on our prototyping activity. From our repositories, we extracted reuse data of 24 real industrial prototypes that we built and used from 2007 to 2021[9]. We describe the detailed data of a recent prototype to illustrate how massive reuse is now possible when we build new prototypes.

Overall, we learn that our reuse methodology (Sections 2 and 3) allowed us to build prototypes with an average component reuse rate of 80%, while today's prototypes can be constituted of more than 96% reused components. From an industrial and methodological point of view, we consider that we successfully built and setup a reuse environment in our prototyping activity and that we benefit from it directly. We discuss possible threats to validity in the discussion that follows (Section 7.5).

A reuse overview over 15 years. Table 2 provides an overview of reuse rates in 24 of our prototypes, spanning 15 years. In average, the 24 prototypes are composed of 80% reused components and 92% reused classes. This is the first time we observe this data since we deploy systematic efforts to make our components and the subsystems they form reusable.

[9] The data extraction methodology is detailed in Appendix A.

Detailed statistics of a prototype with massive subsystem reuse. Table 3 shows detailed reuse statistics of a recent prototype from 2019, named *Prototype U*[10]. The story of *Prototype U* is a successful illustration of the application of the scenarios we described in Section 5. This prototype is an HMI for touch tables with completely new ergonomics and multi-user functionalities, that we delivered in 2019. We did not have any functional code implementing touch table ergonomics with multiple users at the same time. However, most of the business and tooling behavior already existed in previous prototypes. Overall, the complete prototype is composed of 674 components and 1704 additional classes. We only wrote 22 new components and 35 classes. We reused 652 components and 1704 classes from 5 existing subsystems. Reused components represent 96.7% of the prototype's components and reused classes represent 97.94% of the prototype's classes. This use-case is representative of fast-written prototypes we build, with multiple iterations over two years.

7 Discussion and lesson learned

The example of Table 3 is a nice success story of software reuse. We built a new prototype from existing subsystems, while developing additional code only for new aspects of this prototype. Software reuse brings direct benefits to our prototyping activity, but the reasons of

[10] For confidentiality reasons, we cannot give the real name of this prototype.

	Components	*Classes*
Average new code	69.92 (19.8%)	99.63 (8.0%)
Average reused code	282.71 (80.2%)	1152.05 (92.0%)
Average code	352.63	1251.67

Table 2 Average code (components and classes) per prototype over 15 years. Values are rounded to hundredths.

*Subsystem (the * means reuse)*	*Components*	*%*	*Classes*	*%*
Prototype U (new code)	22	3.26	35	2.05
Maritim Survey System*	37	5.49	71	4.17
Airborn System*	332	49.26	236	13.85
Command & Control (C2) architecture *	26	3.86	34	2.00
Generic simulation and behavior scenario*	53	7.86	127	7.45
Tactical Model and Geographical View*	155	23.00	523	30.69
Prototyping tools*	44	6.53	664	38.97
Component framework*	5	0.74	14	0.82
Total	674	100	1704	100

Table 3 Component and class reuse in a prototype from 2019: 652 reused components (96.7% of the prototype's components) and 1669 reused classes (97.94% of the non-component prototype's classes).

these benefits must be put in perspective in regards with this activity. In addition, there are difficulties that we do not overcome yet and that we discuss.

7.1 Reuse in the Thales' prototyping activity

We present very positive results in terms of reusing software components. However, these results must be analyzed in the context of our prototyping activity. This activity often involves adaptations of implementations for the evaluation or demonstration of innovative or new technologies. On the other hand, the functional domain remains relatively stable. From this perspective, it is not surprising that a component-based approach allows for a good level of reuse even though the private implementation of components evolves. It is not surprising either that the level of reuse remains stable as long as components' interfaces related to functional needs remain stable.

Globally, these results tend to confirm our choice of a component-based approach for prototyping. They also stress the importance of interfaces and functional contracts. The correct definition of contracts is the result of a maturation obtained through continuous reuse. Indeed, from reuse to reuse, the interface of a component evolves less and less until it becomes stable.

7.2 Benefits of reuse for evolutionary prototyping

We apply systematic reuse [15] of software to reduce the time and cost of building, maintaining and evolving prototypes. The ability to reuse (parts of) previous prototypes, legacy software and non-component code within component architectures enables fast building of new prototypes. We can build a base prototype for a new project in a few days. From there, we experiment ideas, enrich that base, then evolve and maintain the resulting prototype over years. Some subsystems (*e.g.,* a geographical view, or a radar simulator) will not change over many years while being systematically reused in many prototypes. These subsystems became more stable with time, and today we trust that we can reuse them while maintaining software quality. Typically, we encounter less bugs in older and frequently reused components than in more recent components.

Similar benefits of software reuse in industrial contexts are reported in the literature [16,2]. The most reported benefits are increased quality [2], increased productivity [16,2], reduced development cost and time and a lower defect rate [2]. While reflecting on 15 years of reuse, we observe these same benefits throughout our different prototypes. Another commonly reported benefit of reuse is a shorter time to market [2]. In our case, prototypes are not destined to end up in the market. Rather than a shorter time to market per se, we speak of a shorter time to prototype evolution. The purpose of our prototypes is to live and evolve from end-users feedback and evaluation. Fast prototype evolution is therefore a valuable benefit. Over the years, this benefit has been plebicited by our customers and this encouraged us to push it further this way.

7.3 Smalltalk to explore reuse opportunities

While building and evolving prototypes, we rely on Smalltalk's live and exploratory nature [9,17] to find and understand components we shall reuse. To select which component to reuse, we lively explore and experiment components from our repository. For a given requirement, we study components' interfaces, we start and connect them, observe how they behave, and dynamically explore how they can be reused. To discriminate components providing similar business interfaces, we also evaluate orthogonal properties such as quality of service and performance. We choose which component to reuse from these experiments, and with time we know from this empirical experience which components fit specific (re)use-cases.

This strategy of components selection seems common in practice [12]. Using Smalltalk provides a live and dynamic perspective, that improves this strategy's output. However after many years, relying on this sole strategy became less effective. We have today too much

components and subsystems: we cannot explore everything lively, and the amount of necessary knowledge to choose the right components is oversized.

To overcome this difficulty, we plan to study means to filter meaningful components before live experiments. We will explore recent advances in Traits [20] that we include in our current architecture (Section 3). Because Traits define components contracts, we should be able to explore all users of a given Trait, *i.e.*, all implementors of a given contract.

7.4 Pitfalls of reusing everything

We would ideally like to reuse everything we can to speed-up our development process. However, making systems reusable and reusing them for real is not that straightforward. In addition, our prototyping activity aims at producing quick functional results. This tends to discourage us to take the necessary step back to reflect on how to make things reusable. In this section, we describe and analyze the difficulties we encounter after 15 years of reuse.

When to make subsystems reusable? Table 4 shows the reuse details of our subsystems. For each subsystem, we see the year of the first published version, the year of the first time the subsystem was reused in another prototype, and the number of prototypes in which the subsystem was reused. 7 of our 12 subsystems were reused at least 4 times since their creation. 3 subsystems are reused in almost each prototype since 2007.

However, 4 subsystems are less successful in being reused. For instance, subsystems 9 and 11 were preemptively made reusable even though they were not in use in any prototype. We believed at the time that these subsystems would be reused many times, and we invested a development effort into their reusability. Subsystem 9 was published as a reusable subsystem in 2011, then reused once in 2012. Since then, 9 years later, we never reused that subsystem. Subsystem 11 was first published in 2013, and we reused it only once in 2019 (one reuse over 8 years as of today). Subsystems 6 and 10 were reused only 2 times in 10 years. The reason is that we still lack rigorous means of evaluation to determine if we should reuse a component [12], if we should implement a new component, and in this latter case if we should make this component reusable. Making components (and subsystems) reusable has a cost but in practice some of them are rarely reused. In these cases, the cost of systematic reuse exceeds the cost of ad-hoc development.

Subsystem 12 is an exception: it was created in 2018 and reused only once in 2019. This subsystem is too young to determine if the choice to make it reusable was a good reuse investment.

Subsystem	*1^{st} v.*	*1^{st} reuse*	*Reuse*
1 Maritim Survey System	2015	2018	4
2 Airborn System	2011	2015	7
3 C2 architecture	2013	2018	8
4 Simulation/behavior scn.	2012	2015	10
5 Tactical Model/Geo. View	2007	2010	22
6 Connectivity and network	2012	2019	2
7 Prototyping tools	2007	2011	23
8 Component framework	2007	2010	24
9 Connexion with Java (JNI)	2011	2012	1
10 Graph Model Views	2011	2012	2
11 Ship Identification DB	2013	2019	1
12 Behavior Analysis IA	2018	2019	1

Table 4 Reuse detail of subsystems: the year of their first publication (1^{st} column), the year of their first reuse in a prototype (2^{nd} column) and the number of reuses in prototypes (last column).

Lack of documentation and concentration of expertise. While applying massive reuse, we do not allow enough time for documenting our prototypes, their components and their subsystems. We therefore lack the knowledge to choose which component or subsystem to reuse, nor how to reuse them. Typically, building the prototype from Table 3 requires an extensive knowledge of existing subsystems, how to integrate them into the new prototype architecture and what is missing to realize the new prototype. Consequently, it is difficult to select components and subsystems for reuse.

In addition, this expertise is shared among developers but mostly depend on a few experts. This is a problem we experience for transmitting knowledge to new people joining our team, which is a serious concern in the long term. If our reuse experts leave, knowledge will be lost and reuse possibilities will be jeopardized as well as all its benefits. Knowledge and retrieval of reusable components in the context of a project, while in our case not directly inhibiting reuse [15,3], are clearly a weakness and a slowdown in our development process.

Figure 14 shows the number of active developers for the last 24 prototypes we built from 2007 to 2021. This data only shows which developer participated to the development of a given prototype, without details about the level of involvement of that developer. We see that two developers (that we call *experts*) participated to the development of almost all prototypes, while all others participated to significantly less prototypes. We find that it directly reflects our day-to-day experience, as these experts are the one struggling with the transmission of knowledge to new team members.

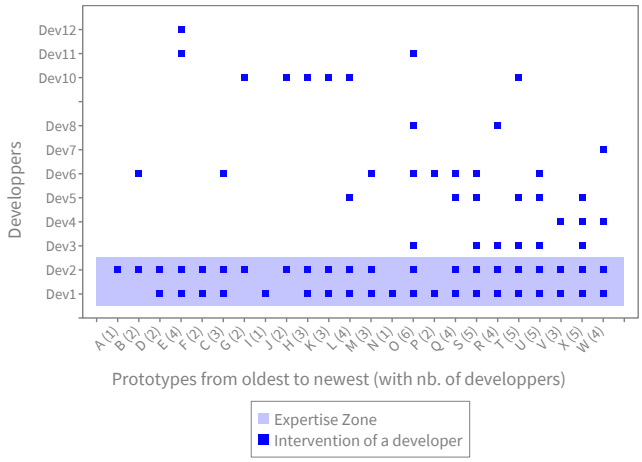

Fig. 14 Active developers per prototype from 2007 to 2021.

Granularity of reusable subsystems. Some of our subsystems implementations are too focused on the realization of business behavior, and mix business and technical concerns. To reuse technical parts of such subsystem, we have to reuse the complete subsystem. In this case, we lack perspective while implementing and architecturing components into subsystems.

7.5 Threats to validity

The main threat to validity comes from the non-public aspect of our software artifacts (our prototypes). First, we made public the extracted data, its extraction method (Appendix A), and all code transforming this data into tables and graphs (in a reproducible form[11]). However, the code repository and the software from which the data was extracted from remains confidential. Second, automatic data extraction was only possible for prototypes delivered after 2012, the year when we started using a configuration management system. Data from before 2012 was recovered through *ad-hoc* scripting on the archived source code.

Another problem is that we only have data concerning the effective reuse of classes, components and subsystems in our prototypes. As such it highlights how much reuse we achieve in our prototyping activity, but not directly the benefits that we report. We do not have clear data (nor metrics) at our disposal to rigorously evaluate these benefits. These benefits are therefore to be taken as an informal feedback from our developers' experience. Putting them in relation with our code reuse rate is not in the scope of this paper.

To reason about the impact of reuse in our activity, we rely on Tables 2 and 3. These tables present respectively the average reuse rates of all our prototypes over 15 years, and detailed reuse statistics of one particular recent prototype. Presenting average reuse rates over 15 years may be biased, as we cannot see how the amount of reused components and subsystems evolved over the years. However, the detailed statistics we present concern a recent prototype. This tends to show that, today, reusing components allows us to create new prototypes with minimal additions. It would be interesting to see and to analyze detailed reuse statistics of the 24 prototypes we delivered since 2007, but that is out of the scope of this paper.

Finally, how we count the contribution of developers does not take into account the detail of each developer commit in the code repository. We only extracted from our repository the tags of developers who worked on a prototype without extracting their real contribution. This raises a number of questions about the role of experts. Did they participate actively during all the development of each prototype? If not, did they only participate at the beginning to distribute knowledge and start the project? Or did they sparsely participate at key points to control the development? These questions are legitimate and would be interesting to explore if we could extract more data. However, the presented data exhibiting a concentration of expertise on two developers seems to reflects these developers' feelings regarding the difficulty of sharing knowledge.

8 Conclusion

At Thales DMS, we elaborate evolutionary prototypes that we refine until they meet end-users functional and ergonomics requirements. We then evolve and maintain these prototypes for years. Prototypes serve as realistic demonstrators used as the main input when building and evolving final products. This process is expensive, and we heavily reuse software to minimize development costs and to maximize the quality of our prototypes.

To support the reuse of software, we use component-based development. Since 15 years, we develop reusable components using Smalltalk, and we reuse these components in our prototypes. Today, Thales capitalized this experience in Molecule, a component framework built in Pharo Smalltalk, with which we build our new prototypes. We organize our components and subsystems in a repository. We reuse elements from this repository to build new prototypes, and each time we enrich the repository back with new reusable elements. Building a realistic demonstrator based on these reusable elements is industrially efficient. Our prototyping team builds prototypes with reduced development costs and time, while focusing on functional concepts and ergonomics.

[11] https://github.com/OpenSmock/ReuseTale

Long-time reused elements expose less bugs and are trusted while reused in new prototypes.

We presented our reuse scenarios: long-term reuse of legacy code, reuse of non-component frameworks and reuse of component business interfaces. We studied 24 industrial prototypes we built over 15 years with component-oriented programming. We observe an average reuse rate of 80% for components, while today's prototypes can be constituted of more than 96% reused components.

We also reported our reuse difficulties. We struggle with our too large amount of reusable elements, and the necessary knowledge required to exploit them efficiently. It is more and more difficult to identify reusable elements to build prototypes.

To improve how we select reusable elements, we plan to study how to exploit preliminary systems models to allow for the early identification of reusable elements. We also plan to conduct larger scale empirical studies at Thales to better understand how we applied reuse over the years. This will help us to take a step back and to reflect more on our reuse processes in order to improve our prototyping activity and to standardize our reuse practice in all Thales.

Acknowledgements 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. We also wanted to thank Vincent Verbeque who created the ancestor of our current component framework and who helped introduce the component approach that we now use.

Declaration

Funding

- The authors did not receive specific support from any organization for the submitted work, except their usual salaries from their respective employers.
- No funding was received to assist with the preparation of this manuscript.
- No funding was received for conducting this study.
- No funds, grants, or other support was received.

Conflicts of interest

- The authors have no relevant financial or non-financial interests to disclose.
- The authors have no conflicts of interest to declare that are relevant to the content of this article.
- All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.
- The authors have no financial or proprietary interests in any material discussed in this article.

Availability of data and material

All the data presented in this paper, the original raw data it was extracted from as well as the methods and algorithms used to transform this raw data are publicly available. The data transformation code is available in an executable form, allowing for the reproduction of the presented data directly from the original raw data. These materials are available at https://github.com/OpenSmock/ReuseTale

Code availability

The raw data on which we base our article was extracted from proprietary industrial software developed during the last 15 years at Thales DMS, France. This code remains proprietary and is not available for public disclosure. Nonetheless, in the article we describe the precise methodology we used to extract the raw data from the proprietary, non-disclosable code.

Ethics approval

N/A

Consent to participate

N/A

Consent for publication

N/A

References

1. Corba component model specification. https://www.omg.org/spec/CCM/4.0/PDF, accessed: july 7th, 2020
2. Barros-Justo, J.L., Pinciroli, F., Matalonga, S., Martínez-Araujo, N.: What software reuse benefits have been transferred to the industry? a systematic mapping study. Information and Software Technology **103** (2018)
3. Bauer, V.M.: Analysing and supporting software reuse in practice. Ph.D. thesis, Technische Universität München (2016)
4. Bernstein, L.: Foreword: Importance of software prototyping. Journal of Systems Integration **6**(1-2), 9–14 (1996)
5. Bonnet, S., Voirin, J.L., Exertier, D., Normand, V.: Not (strictly) relying on sysml for mbse: Language, tooling and development perspectives: The arcadia/capella rationale. In: 2016 Annual IEEE Systems Conference (SysCon). IEEE (2016)
6. 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)
7. 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
8. Dutriez, C., Verhaeghe, B., Derras, M.: Switching of gui framework: the case from spec to spec 2. In: Proceedings of the 14th Edition of the International Workshop on Smalltalk Technologies. Cologne, Germany (2019)
9. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of squeak, a practical smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 318–326. OOPSLA '97, Association for Computing Machinery, New York, NY, USA (1997)

10. Laborde, P., Costiou, S., Le Pors, É., Plantec, A.: 15 years of reuse experience in evolutionary prototyping for the defense industry. In: International Conference on Software and Software Reuse. pp. 87–99. Springer (2020)
11. Laborde, P., Costiou, S., Plantec, A., Le Pors, E.: Molecule: live prototyping with component-oriented programming. In: International Workshop on Smalltalk Technologies - IWST 2020 (Nov 2020), https://hal.inria.fr/hal-02966704
12. Land, R., Sundmark, D., Lüders, F., Krasteva, I., Causevic, A.: Reuse with software components-a survey of industrial state of practice. In: International Conference on Software Reuse. pp. 150–159. Springer (2009)
13. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on software engineering **33**(10), 709–724 (2007)
14. Lidwell, W., Holden, K., Butler, J.: Universal Principles of Design. Rockport Publishers (2010)
15. Lynex, A., Layzell, P.J.: Organisational considerations for software reuse. Annals of Software Engineering **5**(1), 105–124 (1998)
16. Mohagheghi, P., Conradi, R.: Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering **12**(5) (2007)
17. Rein, P., Taeumel, M., Hirschfeld, R.: Towards exploratory software design environments for the multidisciplinary team. In: Design Thinking Research, pp. 229–247. Springer (2019)
18. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: European Conference on Object-Oriented Programming. Springer (2003)
19. Szyperski, C., Bosch, J., Weck, W.: Component-oriented programming. In: European Conference on Object-Oriented Programming. pp. 184–192. Springer (1999)
20. Tesone, P., Ducasse, S., Polito, G., Fabresse, L., Bouraqadi, N.: A new modular implementation for stateful traits. Science of Computer Programming (2020)
21. Verhaeghe, B., Etien, A., Anquetil, N., Seriai, A., Deruelle, L., Ducasse, S., Derras, M.: Gui migration using mde from gwt to angular 6: An industrial case. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou, China (2019), https://hal.inria.fr/hal-02019015
22. Voirin, J.L.: Model-based System and Architecture Engineering with the Arcadia Method. Elsevier (2017)

A Repository project structure and data extraction algorithms

A.1 The project structure

Since we were programming in Smalltalk, we chose the Store[12] infrastructure to hold and version our code. Our research results and discussions are given according to data that we automatically extract through queries in the code repository. This Section presents the data structure, and how projects are organized in the repository.

Since we used Store vocabulary to implement the extraction scripts, we first introduces that vocabulary before presenting the scripts we implemented to extract the data.

[12] Store is a commercial product of *Cincom*: www.cincomsmalltalk.com/main/community/product-portal/store-repository/

Figure 15 shows a meta-model for the data set that we extract from the code repository. First class objects are presented with white squares.

The root elements of a project are instances of *Package*. A package owns a set of *Behavior* instances (its classes and traits). A package also owns a set of methods but each method is specified either in a class or in a trait. The Store infrastructure manages *Pundles* and *Bundles*. a *Pundle* is a package and a *Bundle* is the top container of a project. It is made of Pundles and of other Bundles (it doesn't directly own any source code). Each Bundle and Pundle manage its prerequisites through a dependency list.

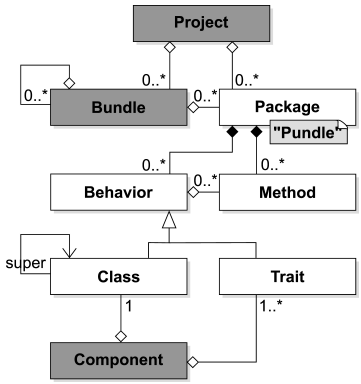

Fig. 15 The data meta-model

Figure 16 shows the structure of our projects in the repository. There are two types of projects : prototypes and subsystems. The top-level element is a prototype project. A project consists of several packages. Store doesn't consider the concept of subsystem. Subsystems are composed of one or several identified packages that are self contained and reusable.

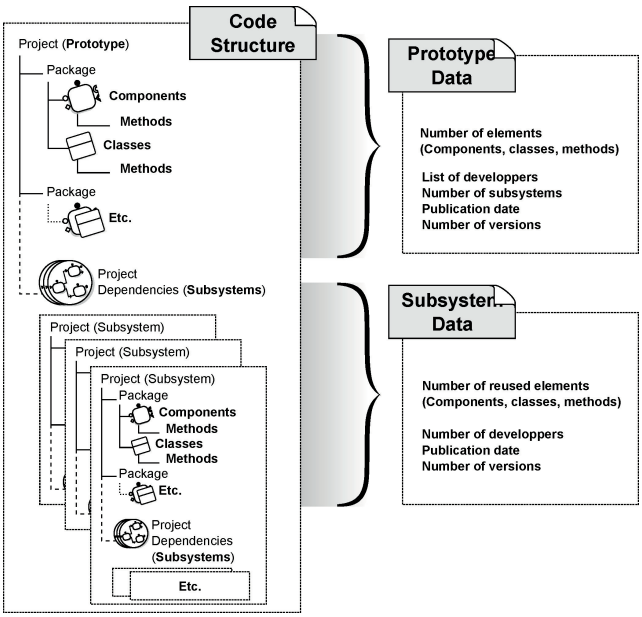

Fig. 16 The project repository structure

A.1.1 Data extraction algorithms

Our data extraction algorithms are implemented by using the main five methods described in the listings below. All algorithms querying a project use its name as input parameter (*i.e.*, *aPundleName*). In Figure 16, we consider Pundles and Bundles as containers that we simply call Packages.

For readers unfamiliar with Smalltalk, here is a comparison with Java-like syntax:

- Message-sends use spaces: *pundle definedClasses* is equivalent to *pundle.definedClasses()*.
- Arguments are specified by colons, without parenthesis: *self getPundleByName: aPundleName* is equivalent to *this.getPundleByName(aPundleName)*.
- The up arrow ↑ is the return keyword: ↑ *allComponents* is equivalent to *return allComponents*.
- Square brackets [ ] delimit lexical closures.

Extracting the number of non-overwritten components from a project:
This method returns the number of Components that are defined in a Pundle. The Pundle is requested from his name (*getPundleByName*). Then, all components from the Pundle defined classes are selected (*includesBehavior* : *Component*). The resulting collection is stored in *allComponents*. The size of this collection is finally returned.

```
numberOfDefinedComponentsInPundle: aPundleName
  | pundle allComponents |
  pundle := self getPundleByName: aPundleName.
  allComponents := pundle definedClasses
    select:[ :e | e value includesBehavior: Component].
  ↑ allComponents size
```

Extracting the number of methods of a project:
This method returns the number of methods specified in a Pundle. The Pundle is requested from his name (*getPundleByName*). Then, *allMethods* is assigned with a collection containing all the specified methods of the Pundle (as a result of the *allMethods* message sent to the Pundle). The size of this collection is finally returned.

```
numberOfMethodsInPundle: aPundleName
  |  pundle allMethods |
  pundle := self getPundleByName: aPundleName.
  allMethods := pundle allMethods.
  ↑allMethods size
```

Extracting the number of published versions of a project:
This method returns all the published versions of a Pundle from the repository. The Pundle is requested from his name (*getPundleByName*). Then the current repository session is fetched and assigned to *session*. It represents the remote connection to the repository. The returned versions are those of the Pundle which are stored in the repository (as a result of the *allStoreVersionsIn* : message which is sent to the Pundle with the session passed as argument).

```
getStoreVersionsForPundle: aPundleName
  | pundle session versions |
  pundle := self getPundle: aPundleName.
  session := StoreLoginFactory currentStoreSession.
  versions := pundle allStoreVersionsIn: session.
  ↑versions
```

Extracting the developers who have published a project:
This method build a set of developer names who have published at least one version among a list of Bundles (a list of projects). The set contains only unique developer names.

```
numberOfDeveloppersInStoreBundles: manyStoreBundles
  | set |
  set := Set new.
  manyStoreBundles do:[ :e | set add: e username ].
  ↑set
```

Extracting the number of published versions of a project:
This method count the number of versions from a list of Bundles (a list of projects).

```
numberOfVersionsInStoreBundles: manyStoreBundles
  ↑manyStoreBundles keys size
```