15 years of reuse experience in evolutionary prototyping for the defense industry (preprint)

Pierre Laborde[1], Steven Costiou[2], Éric Le Pors[1], and Alain Plantec[3]

[1] THALES Defense Mission Systems France - Établissement de Brest, 10 Avenue de la 1ère DFL, 29200 Brest, France
{pierre.laborde,eric.lepors}@fr.thalesgroup.com
[2] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL
steven.costiou@inria.fr
[3] Univ. Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
alain.plantec@univ-brest.fr

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 report three scenarios of reuse that we encounter regularly in our prototyping activity.

Keywords: Evolutionary prototyping · Component reuse · Traits · Pharo

1 Introduction

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

At Thales Defense Mission Systems (DMS), the Human-Machine Interface (HMI) industrial prototyping activities are an important part of the software production process. HMI industrial prototyping is the building of software prototypes as close as possible to real products from the HMI point of view (graphics and ergonomics). Using prototypes, we evaluate software HMI design and experiment complete use-cases 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, Thales use component-oriented programming [17, 11]. Component-based architectures bring the necessary modularity to enable reuse and evolution of prototypes.

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). This framework is today the main tool that we use for evolutionary prototyping, for which we report and discuss 15 years of reuse experience and the difficulties we face today (Section 4).

2 Reuse in prototyping for the defense industry

Exploring design ideas through Concept prototyping [4, 12] is one of the main activities at Thales DMS. Prototypes are developed to help communicate concepts to users, demonstrate the HMI usability and exhibit potential problems. Moreover, prototypes are developed and maintained until they meet users expectations regarding not only the HMI, but also the main business functionalities. Some of our prototypes implement complete business behavior and fulfill functional requirements. Thus, they are also kind of evolutionary prototypes [12]. 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[4] 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.

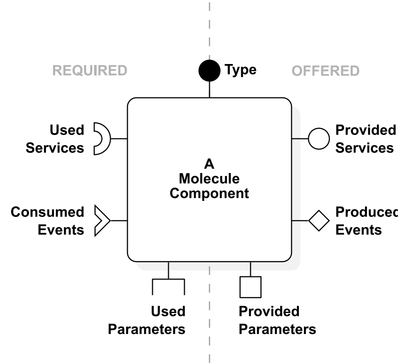

Fig. 1: Public view of a Molecule component.

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 the first motivation for using components. This is particularly true for Graphical User Interfaces (GUI) components that we call Panels. It is well known that GUIs are expensive to implement. Our panels expose a stable public interface and implement a standard contract. We therefore directly reuse our panels in several prototypes (Figure 2). 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

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

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 2, 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 sub-system (*e.g.*, a geographic view with layering, filtering, selection...). Prototypes (re)use and deploy instances of sub-systems 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 sub-systems for future reuse.

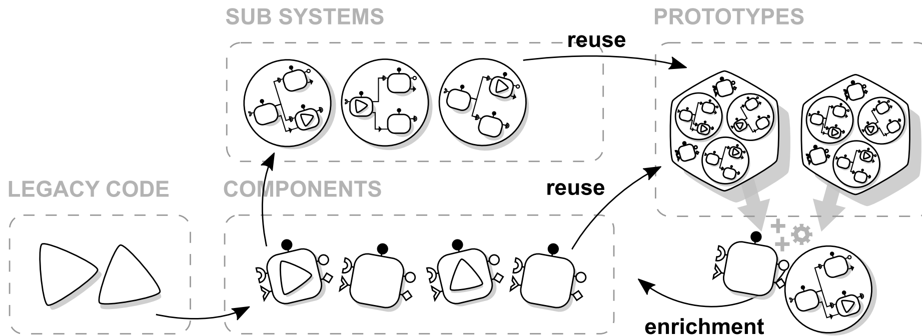

Fig. 2: From components to prototypes: a reuse chain.

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, 19].

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 sub-systems 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 3). We evolve the related components implementation without changing GUI clients nor legacy and business software.

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

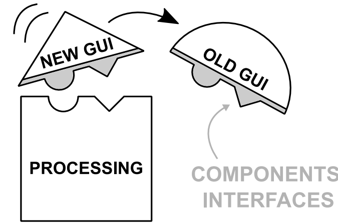

Fig. 3: Components to ease evolution, for example: changing GUIs.

systems' design models using dedicated methods like Arcadia [20] 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 evaluate prototypes based on these models. The software engineering team develops final products.

The prototyping team is composed of 5 to 10 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 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.

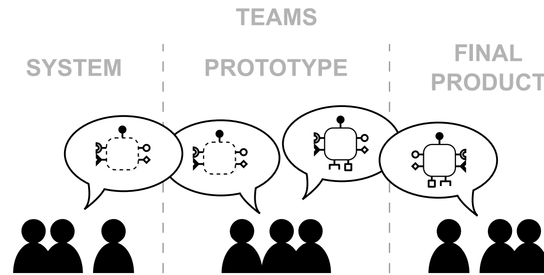

Fig. 4: Engineering impact: design traceability and co-working;

3 Component-oriented programming with Molecule

Since 2005, Thales DMS use Smalltalk for prototyping. The 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, 15]. Thales adopted component-based development to ease software reuse and separation of concerns. Since 2016, we use Pharo [7] with the Molecule component-oriented programming framework. Thales developed Molecule to capitalize on its experience in component-based development. This section briefly describes how we program components with Molecule.

The Molecule framework relies on Traits [16, 6, 18] to implement component Types. A Trait is an independent set of methods with their implementation and requirements (methods and variables). Classes using a Trait automatically benefit from these methods, and must define that Trait's requirements. A Trait can be composed of multiple other traits. In this case, a class using this composed Trait benefits from all the methods provided by the Traits composition. A Trait provides orthogonal behavior to all classes using that Trait, regardless of their class inheritance hierarchies.

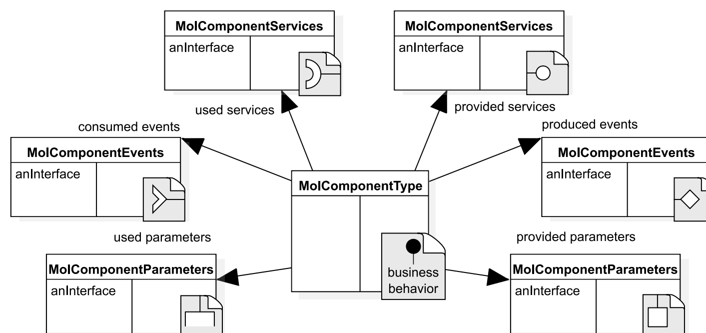

Fig. 5: Component contract implementation in Molecule.

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

Molecule provides syntactic sugar to directly implement components by inheriting from the MolAbstractComponentImpl 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

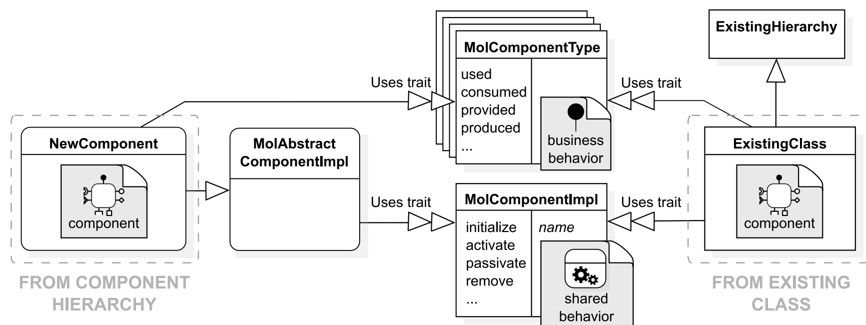

Fig. 6: Two ways to implement a component.

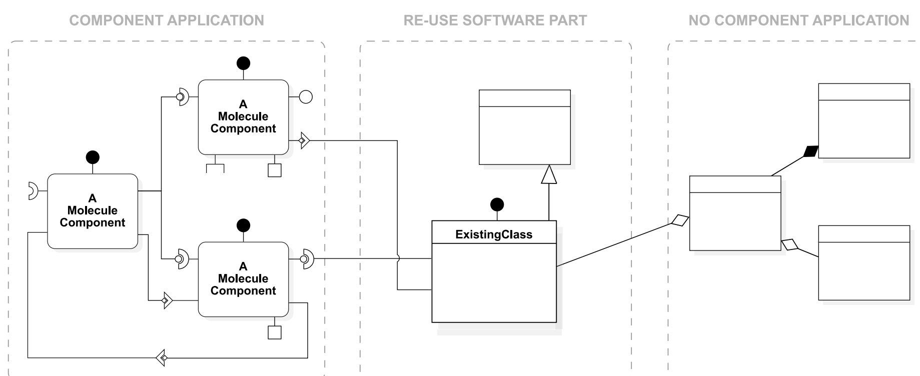

Fig. 7: We use augmented classes in component and non-component applications.

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). This code adaptation ensures that the augmented class provides the correct level of information in regards with the requirements of the connected components.

4 Reuse scenarios and difficulties at Thales DMS

In this section, we describe three reuse scenarios that we regularly and successfully deal with during our prototyping activity. While we learnt how to cope with these reuse scenarios, we still encounter many difficulties that we discuss.

4.1 Common reuse scenarios in the prototyping activity

While prototyping, we frequently deal with three reuse scenarios: long-term reuse of legacy code, reuse of non-component frameworks and reuse of component business interfaces. The story of *Prototype X*[5] is a successful illustration of the

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

application of these scenarios. This prototype is an HMI for touch tables with completely new ergonomics and multi-user functionalities, that we built 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, for a total of 47447 methods. We only wrote 22 new components and 35 classes. We reused 603 components and 991 classes from 5 existing sub-systems (Table 1). Reused components represent 89.47% of the prototype's components and reused classes represent 58.16% of the prototype's classes[6]. This use-case is representative of fast-written prototypes we build, with multiple iterations over two years.

*Sub-System (the * means reuse)*	*Components*	*%*	*Classes*	*%*
Prototype X (new code)	22	3.26	35	2.05
Survey mission system*	37	5.49	71	4.17
Mission technical system*	332	49.26	236	13.85
Command-Control architecture*	26	3.86	34	2.00
Scenarios and simulation*	53	7.86	127	7.45
Tactical views and models*	155	23.00	523	30.69
Prototyping tools & models (framework)	49	7.27	678	39.79
Total	674	100	1704	100

Table 1: Component and class reuse in a prototype from 2019: 603 reused components (89.47% of the prototype's components) and 991 reused classes (58.16% of the non-component prototype's classes).

Long-term reuse of legacy code. This scenario is the most common case of reuse in our prototyping activity. For each new prototype, we systematically need to reuse functionalities from previous prototypes or from legacy code. Before we based all our architectures on reusable components (2005-2013), we used to reuse code by manually performing copy/paste of hundred of classes from prototype to prototype. Since then, component-based architectures help us to reuse legacy code from previous prototypes in new prototypes.

For example, 10 years ago, we took an old prototype using an old GUI technology. As this prototype was component-based, we were able to redesign the HMI with another 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 3). Components forming this business layer became particular sub-systems reused in many other prototypes. Today, 10 years later, we continue to reuse and enrich these sub-systems in our new prototypes. Typically, these sub-systems simulate complex hardware and software systems such as radars, sensors, detection and communication equipment, etc.

Another example of frequently reused sub-system is the tactical view[7]. This view is a recurrent requirement throughout our different prototypes. We reuse the same tactical view sub-system from prototype to prototype since 10 years.

[6] We do not count the prototyping framework's code as reuse.

[7] Geographical business objects display on a map

Reuse of non-component frameworks in component architectures. In this scenario, we need to reuse non-component code in our component-based architectures. Typically, we rely on open-source frameworks in our prototypes: UI models, graphics engines, visualization engines, etc. These frameworks are object-oriented, so we augment their classes with Molecule to reuse them directly in our component-based prototypes.

We start by inheriting from the framework classes we want to reuse. We then apply the Molecule component Traits to the inherited classes to augment them with component behavior. As illustrated by Figure 7, these classes become usable transparently by both standard applications and our component systems.

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 sub-system to support another graphics engine. Therefore, we augment classes of graphical elements (views) as components to directly connect them to our component sub-systems. This allows us to add new graphics technologies as display backends with very few adaptations.

Reuse of component business interfaces. In this scenario, we reuse components' business contract to replace component implementations transparently (as in Figure 5). Interactions between components are expressed 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.

To change the database, we implemented a new component for a new backend and we reused the Type Traits for high-level data access. We were then able to transparently switch the old component by the new one in the prototype.

4.2 Discussion

The example of Table 1 is a nice success story of software reuse. We built a new prototype from existing sub-systems, while developing additional code only for new aspects of this prototype. Software reuse brings direct benefits to our prototyping activity, but there are difficulties that we do not overcome yet.

Benefits of reuse for evolutionary prototyping. We apply systematic reuse [13] 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 sub-systems (*e.g.,* the

tactical view, or a radar simulator) will not change over many years while being systematically reused in many prototypes. These sub-systems 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 [14, 2]. The most reported benefits are increased quality [2], increased productivity [14, 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.

Smalltalk to explore reuse opportunities. While building and evolving prototypes, we rely on Smalltalk's live and exploratory nature [9, 15] to find and to understand components to 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. 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 [10]. 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 sub-systems: we cannot explore everything lively, and the amount of necessary knowledge to choose the right components is oversized. To overcome this difficulty, we needs to study means to filter meaningful components ahead of time, before live experiments.

Pitfalls of reusing everything. We observe that we tend to reuse everything. The example from Table 1 is a typical illustration of massive reuse, although successful. However, we still lack rigorous means of evaluation to determine if we should reuse a component [10], if we should implement a new component, and in this latter case if we should make this component reusable. Making components reusable has a cost but in practice some of them are never reused. In this case, the cost of systematic reuse exceeds the cost of ad-hoc development.

While applying massive reuse, we do not allow enough time for documenting our prototypes, their components and their sub-systems. We therefore lack the knowledge to choose which component or sub-system to reuse, nor how to reuse them. Typically, building the prototype from Table 1 requires an extensive knowledge of existing sub-systems, 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 sub-systems for reuse.

In addition, this expertise is shared among developers but mostly depend on a few experts. This is a problem for transmitting knowledge to new people, and a serious concern in the long term. If the few 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 [13, 3], are clearly a weakness and a slowdown in our development process.

Finally, some of our sub-systems implementations are too focused on the realization of business behavior, and mix business and technical concerns. To reuse technical parts of such sub-system, we have to reuse the complete sub-system. In this case, we lack perspective while implementing and architecturing components into sub-systems to avoid this pitfall.

5 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 sub-systems 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. 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. After 15 years, we can highlight a beneficial rate of component reuse. 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.

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. 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)
11. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on software engineering **33**(10), 709–724 (2007)
12. Lidwell, W., Holden, K., Butler, J.: Universal Principles of Design. Rockport Publishers (2010)
13. Lynex, A., Layzell, P.J.: Organisational considerations for software reuse. Annals of Software Engineering **5**(1), 105–124 (1998)
14. Mohagheghi, P., Conradi, R.: Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering **12**(5) (2007)
15. Rein, P., Taeumel, M., Hirschfeld, R.: Towards exploratory software design environments for the multi-disciplinary team. In: Design Thinking Research, pp. 229–247. Springer (2019)
16. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: European Conference on Object-Oriented Programming. Springer (2003)
17. Szyperski, C., Bosch, J., Weck, W.: Component-oriented programming. In: European Conference on Object-Oriented Programming. pp. 184–192. Springer (1999)
18. Tesone, P., Ducasse, S., Polito, G., Fabresse, L., Bouraqadi, N.: A new modular implementation for stateful traits. Science of Computer Programming (2020)

19. 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
20. Voirin, J.L.: Model-based System and Architecture Engineering with the Arcadia Method. Elsevier (2017)