A Selection of Development Processes, Tools, and Methods for Organizations that Share a Software Framework between Internal Projects

Ciprian I. Paduraru
Department of Computing Science, University of Bucharest, Bucharest, Romania
Electronic Arts, Bucharest, Romania

Keywords: Framework, Shared, Software Development, Collaboration, Object Oriented Programming.

Abstract: One of the ways organizations are saving development costs nowadays is to share code between internal projects. Shared frameworks with highly reusable components are usually desired, but their development and maintenance processes usually generate important challenges. This paper describes development processes and methodologies that can be used to reduce costs in developing and maintaining this kind of shared framework inside an organization considering distributed development and collaboration between teams which have limited resources. Technical aspects for providing extensibility, components reusing and tools that assist the process of integration, release, and development are also presented. The work is sustained by the experiments and best practices taken from the development of such a shared framework inside a real organization.

1 INTRODUCTION

One of the main solutions used inside organizations to reduce development costs is to improve the reusability of the code across internal projects. This can be done by creating a common framework base code with highly reusable components and have different development teams inside organization reuse them. Because the capacity and resources are always limited, the team behind framework would quickly become a bottleneck when individual teams require changes or new features for the existing framework components. One of the solutions to eliminate this bottleneck is to create a distributed development for the framework's source code, where internal teams developing projects using the framework can also contribute to it. Another aspect that needs to be mentioned is that projects can be at different stages in their lifecycle (e.g. one project might be in its final stages, while another in a prototyping phase) and this could dramatically influence the time needed to do changes over framework's code. Considering this, the conclusion is often that the framework repository needs to be branched in each development team repository (Figure 1), let them do the changes at the needed pace, and after changes are done, processes that do code sync should start.

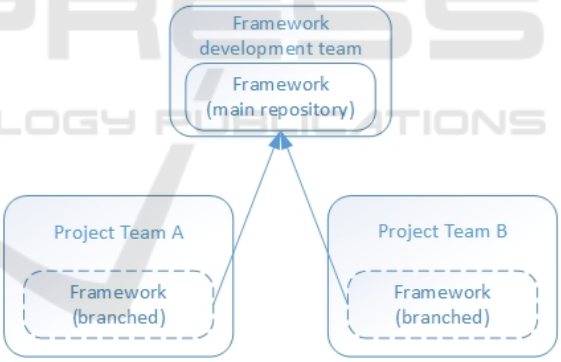

Figure 1: Development teams sharing the framework (i.e. projects inside an organization) are branching the framework code to make changes at their own pace.

The main roles of the framework development team are to maintain a clear architecture of its repository, integrate the changes made by different projects over its branched repository, make code reusable among projects (e.g. Team A could implement a feature on top of the framework that can be reused by Team B), and at the same time be pro-active by developing new features that prepares the organization for the future. In this paper, by *divergence* we mean a piece of code that differs in a framework branched repository (hold by one of the teams) compared to the main branch of the framework. The purpose of the code sync processes is to make sure that divergences are as low as possible. In the rest of the paper, we'll denote the main framework development team with

Paduraru, C.
A Selection of Development Processes, Tools, and Methods for Organizations that Share a Software Framework between Internal Projects.
DOI: 10.5220/0006426602940301
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 294-301
ISBN: 978-989-758-262-2

framework team while the teams branching and using the framework are called *project teams*.

The contribution of this paper is to analyze the development process in the use case mentioned above, with the clear target of reducing costs, and more specifically:

- Suggest a workflow collaboration protocol among teams that works under limited resources and aims at reducing the bottlenecks.
- Investigation of current object-oriented practices that help reducing divergences and promote code reusability.
- Presentation of tools and protocols that can assist the framework's maintenance process.

This paper is structured as follows. Related work in the field is presented in Section 2. A suggestion for a collaboration and communication model is described in Section 3. Applications of object-oriented programming paradigms, principles, and patterns that helps code reusability and aims at reducing divergences, together with tools that assist the maintenance and integration process are presented in Section 4. A release methodology that could help the integration process is described in Section 5. Finally, Section 6 gives conclusion and future work ideas. Evaluation and comparison between methods are presented at the end of Sections 3,4,5 since they discuss aspects from different views. The data used for evaluating the methods considered in this paper was collected manually at certain points of the framework's lifetime (approximately 8 years in a real organization), by storing statistics from internal tools and metadata about the methods used at each point.

2 RELATED WORK

There are many papers discussing high-level or low-level aspects of framework development, but only a selection of them are mentioned in this section, mainly because of the limited space. Papers and content discussing object oriented programming principles that are useful for framework development are presented mostly in Section 4, together with how we used those techniques in our experiments.

In (Mattsson and Bosch, 1997) and (Bosch et al., 2000) authors are describing possible problems and solutions that stem when having more than one framework to reuse and integrate. It is a problem that intersects with the one discussed in this paper since development teams occasionally develop small or medium sized components that can be shared by the rest of the teams inside an organization. Their paper discusses object-oriented techniques such as inversion of control and adapters, which are being reused in our approach. A design decision tool that can simplify the design decision process in a framework by writing code for the design patterns employed is described in (MacDonald et al., 2009). The target of the paper is to start with a good design framework, while our target is to continuously design, adapt and evolve the framework with a focus on reducing overall costs. Metrics that can be used for structural stability of framework architecture are given in (Jagdish, 2000) and (Sant'Anna et al., 2007).
High-level ideas about object oriented framework development are sketched in (Fontoura et al., 2000). Authors are naming the points of flexibility of a framework, *hot spots* while the points of immutability, frozen spots' and show how to glue code from different domains of applications to the framework. From this point of view, our approach tries to get into more details and analyze general object-oriented programming principles and how they fit into a framework architecture.

From a distributed development perspective, (Spichkova and Schmidt, 2015), presents a formal framework for analyzing, structuring and optimizing requirements that come from different countries and organization. The process can be reused in our workflow since we deal with a similar one when integrating the divergences from development teams' repositories back to the main repository. Open source software communities (OSS) have similar targets with the ones described in this paper: a collaboration model and code reuse methods targeted to cut costs of developments. In (Yuan-Hsin Tung, 2014) and (Haefliger et al., 2008), authors are analyzing different frameworks and strategies for code reuse in OSS. To calculate the costs, the open source reuse processes are split into three stages: search, integration, and maintenance. The solutions proposed in their paper are reused and extended in our usage by going deeper into different problems and aspects to optimize the development costs even more.

3 COLLABORATION MODEL

Because of the limited resources in the framework team (i.e. development capacity over a period of time), requests that are coming from various projects using the framework product could easily create important delays in terms of delivery. Even if the core team would know better how to implement or fix various features, to reduce bottlenecks such a model

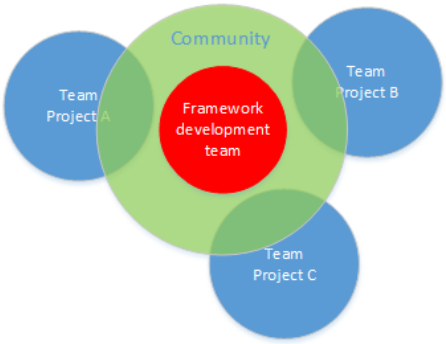

Figure 2: Communities and collaboration between projects and framework team.

needs work distribution. Distributed development of a shared framework needs an efficient collaboration protocol between the internal projects using constantly the product and the main team that holds its repository and releases. First, one efficient way that worked in our organization is to establish communities: people from each project were assigned to hold the communication with the framework team in term of issues or needed features (Figure 2). More, the community can be split into domains of knowledge (e.g. systems, databases, web design, etc.). To make sure that projects' interests are considered in the product strategy and development activities of the framework team, the community can have constant meetings to discuss the needs of each project (e.g. defects in existing implementation or new features needed).

In the distributed model, project teams can implement both new features or fix issues with the existing ones. The key however in distributed development is the correct coordination and having the framework team implementing a technical design before starting the collaboration on tasks. The process when resources are not enough to implement the requested tasks on its own by the deadline is depicted in Figure 3. When a project requests a new feature or fixes to the Framework team the request is analyzed by them, creating first a technical design document (TDD) for implementation and testing. If the Framework team doesn't have resources (time) to finish the request in an acceptable time (specified by the Project team), then the TDD is sent to back to the Project team and they can start the implementation. There is even the possibility of a mix (both team working at the same time) since the TDD usually contains the work breakdown and tasks' estimations. After rounds of code reviews and evaluations, the code can be merged back to the Framework team. What differs from the GitHub and other OSS adopted solutions, is that instead of forking/branching the code and working on its own, the collaboration model proposed here has a

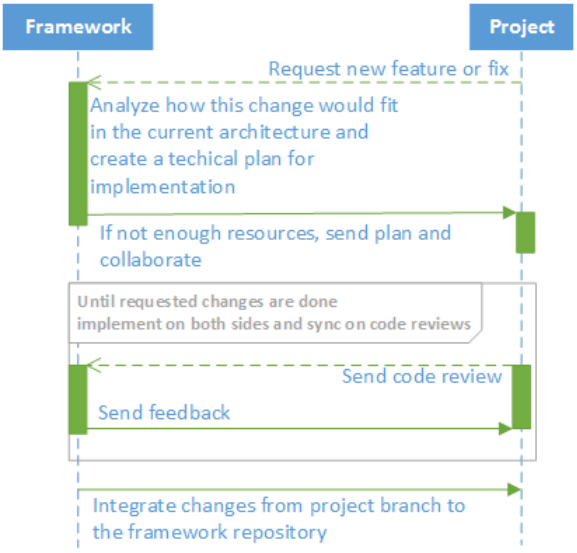

Figure 3: Protocol for distributed task coordination when framework team doesn't have time to finish the requested work in time.

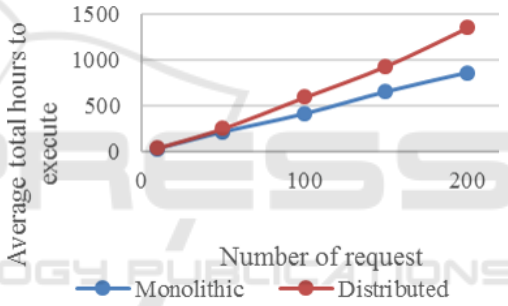

Figure 4: Average total hours per employee to execute different requests entered in the framework's database.

better rate of time-saving since once a development team starts an implementation in its own branch, it has a technical design adopted by specialists on the field. Also, this workflow for requesting changes in the framework's code is automated by some internal tools.

To compare between monolithic (i.e. requests handled only by the framework development team) versus distributed implementation (the collaboration model mentioned above), we recorded the time taken to implement a different number of requests in both models and how much the deadline per request was exceeded on average. Each request was considered to have a deadline set by the initial requester which translates to the time when that request is needed to avoid bottlenecks for the project needing it (of course, the deadline could be current time of request creation). As expected (and shown in Figure 4), the total cost in hours per employee is increased in the distributed development, mainly because of the

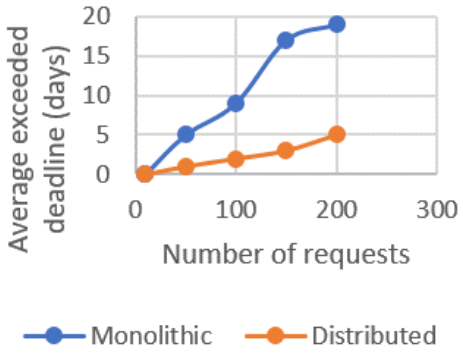

Figure 5: Average exceeded deadline per request (days) by their total number at once in the database.

overhead of code reviews and sync between different teams (Figure 3). The real advantage of the distributed development comes when comparing the average exceeded deadline (Figure 5) because the requests' deadlines usually affect the project delivery time and in the end, the organization's business.

The requests considered could be both defects reported or feature requests (i.e. different time to implement), but the comparison still holds since their percent distribution was similar over time. A possible different solution would be to scale the capacity of the framework team (in terms of employees per domain), but similar to distributed programming concepts, a static allocation of resources could easily create unused resources. The results captured in Figures 4, 5 (and later in Section 4.6 evaluation) implied data captured automatically from our project management tools during approximatively 8 years of continuous development. There were on average 6-7 projects working in this environment, with a team size of 15-20 software engineers per each project team and 10 on the framework development team (each software engineer being expected to work on average 8 hours a day). The projects' average length was of 2-3 years and involved entertainment software applications.

4 TECHNICAL ASPECTS FOR REDUCING DIVERGENCES AND MAKE CUSTOM CODE REUSABLE BETWEEN DEVELOPMENT TEAMS

The global task of reducing code divergences generates some technical challenges that must be well addressed to reduce costs. A simple example situation is presented in Figure 6, where N development teams needed to modify the functionality of a core framework function (i.e. either its interface or its semantics).

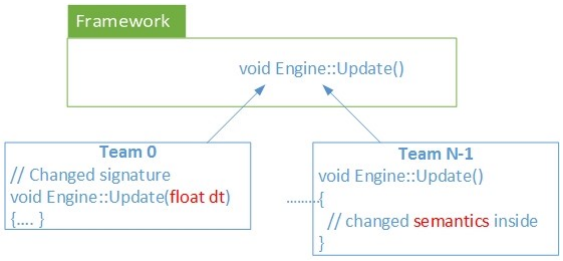

Figure 6: Showing teams changing one of the core functions of the framework inside their local branched depot.

Consider the following costs:

- $C1$ = Cost to analyze divergence for a single team
- $C2$ = Cost to create a plan to unify divergences into framework code and implement the needed changes.
- $C3$ = cost to integrate back unified code into each team depot, and N teams using the framework, then the total overhead is: $O = N * C1 + C2 + N * C3$.

The ideal overhead cost would be 0, but it is difficult to imagine an ideal architecture of modules and components that would allow extensibility without any code divergence. Divergences need to be taken back from teams' depots to the framework depot as soon as possible otherwise individual costs C1, C2, C3 can grow exponentially. This order of magnitude comes from the fact that the entire components hierarchy is a graph of dependencies and instead of integrating back changes from local nodes, software engineers would have to consider a set of logical paths with nodes that have divergences. This process of observing, refactoring then integration back was named in our workflow *harvesting*. The rest of this section presents high-level ideas on how to identify divergences and solutions for increasing modularity, extensibility and testability of the software in the presented use case.

4.1 Identifying Divergences

It is important to focus the effort of adding extensibility where it matters most. To do this we must find the spots with the highest divergence first and refactor them for extensibility. Being a difficult process to do by hand, a solution is to build a tool that can analyze the framework source code in projects' depots against the base version of the framework repository. In our implementation, the tool compares file by file and outputs metrics that looks like in Figure 7. The tree-view control returned as output can be used to identify the number of lines changed per file, class, function and

Area of divergence	Team A	Team B	Team C
⊟EngineUpdate.cpp	273	351	7
⊟class Updater	215	321	5
Updater::init	14	317	2
⊞class Feedback	58	19	6
⊞Physics.cpp	14	177	0

Figure 7: Showing output of the divergence tool execution: the number of lines changed per file, class, and function in a tree view, for each project using the framework.

project. This can be a good hint for software engineers where to focus their efforts and refactor things first.

When analyzing the changes that each team has, for instance on a class, it makes sense to have some comments that describe the change and the author of it. A change causing divergence can spread through various files. To promote divergences documentation, a plugin for the source control tool used by projects' depots can be created. In our case, this tool was not allowing any code commit that changed the framework's code without divergence tags and its proper format. This documentation was of a great help for our use-cases because the software engineer that needed to analyze, converge and integrate back changes could understand better the purpose and areas (files, components) affected by the change. Without these, it would be difficult to understand the purpose and how changes are spread through the entire code.

4.2 Base Code Modularity and Extensibility Principles

In Software engineering, the basic design principle of modularity implies the decomposition of a software architecture into modules characterized by high cohesion and low coupling. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change (Larman, 2005). The base design principles to get a modular and easy to extend framework code are known as *SOLID* ((Myers, 1978) and (Cesare and Wilde, 2014)). The usage of these principles proven to be essential in any refactoring decision. Framework's components should have interfaces and ideally, the aggregation and communication between them should be done using interfaces instead of concrete objects' instantiations, i.e. avoiding hard-dependencies. This would let users inject dependencies dynamically and allow them to reuse components between different projects while keeping the framework code intact. The concept is known in theory as Dependency inversion principle or dependency injection (Martin, 2000), (Schwarz et al., 2012). The intent behind is to have both higher-level modules (in our case this is the framework code) and lower level modules (usually customized by development teams) depend on common interfaces. An example can be seen in listing code below and Figure 8. Both client and framework know the interface of these components at compile time and does not depend on any concrete implementation. In our implementation, a repository with common components was created to facilitate the reuse of concrete components between projects. This way, when a project needs a certain functionality it checks first if something from the common components repository can be reused or customized. When finishing the implementation of a customized component, a sync with the community would decide if their component worths added to the common components depot. This sync implies regular meetings (online or live) between the members of the community (Figure 3) or open discussion on communication channels (currently our solution is to create Slack channels www.Slack.com - per development area, project or whatever make sense).

```
void Framework::Main::update(float deltaTime)
{
 // Internal call, not overridden by client code
 startLogging();

 // Gather all jobs first
 const Job* jobs =
     m_aiComp->gatherJobs(deltaTime,nullptr);
 mPhysicsComponent->gatherJobs(deltaTime, jobs);
 mAnimationComponent->gatherJobs(deltaTime, jobs);

 // Run the tasks graph
 runScheduler(); // internal framework call

 // Call after update functionality
 mAnimationComponent->postUpdate(deltaTime);
 mPhysicsComponent->postUpdate(deltaTime);
 sendNetworkUpdate(); // Internal framework call
}
%\end{verbatim}
```

4.3 Solutions to Keep Clean Interfaces in Framework's Components

Interface segregation principle (Martin, 2002) ensures that clients won't have to inherit and use interfaces with functions that are not needed by them. This could happen frequently after a bad refactoring and can cause difficulties in understanding the new interface and what piece of code projects should provide when implementing their custom concrete object. The

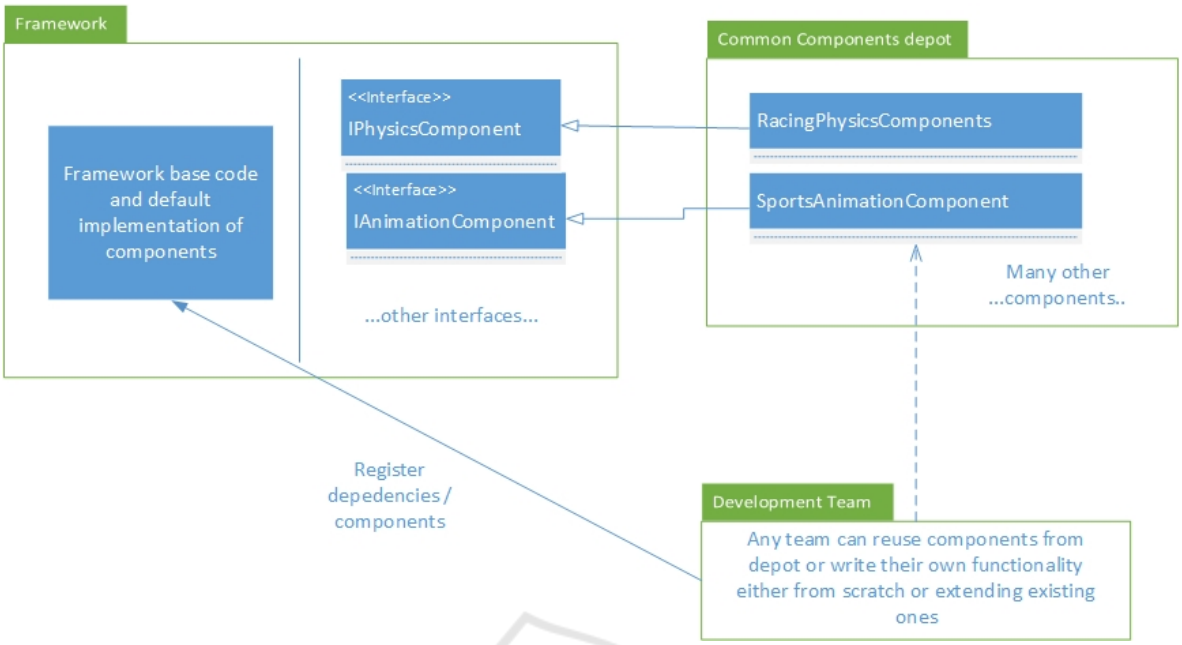

Figure 8: An example where framework uses components that are injected by client code. Projects can use or contribute to the components in the Common Components depot.

need to provide more boilerplate code just to make things compile or work together is known in the literature as "fat implementation" (Martin, 2002). The refactoring process must decide at some point if one interface should be split in two (or even more) interfaces or concrete implementers are requested to implement all operations in there. Instead of modifying the core code with adding data or methods to existing classes, an approach that works better sometimes is to use proxy, adapter or mediator patterns (Gamma et al., 1994), thus not changing the interfaces or core functionality at all.

These are used to make the framework's code compile and semantically compatible with new or modified API in the client project. This is valid especially if changes requested are specific to a single development team. Since these can remove code divergences, it might hide reusable work done by different teams. In our implementation, a set of comment tags is used by the teams to suggest that they are using an interface in an interesting way from a reusability point of view. At each code commit with such a tag, the source control plugin sends a message to a database with suggestions. This database is analyzed from time to time (manually, in this moment) and checked if it worth doing a convergence and make changes reusable between teams

4.4 Testing Components

The techniques mentioned above in this section that suggest how to work with interfaces or inject dynamically new functionality without affecting framework code are highly compatible with the process of unit testing and mocking components. Having mocks and unit tests for each framework component can reduce testing and integration costs. The reason is that when the process of harvesting and integrating back changes from clients to framework repository starts, ideally clients' code affecting framework's components should be bug free. To ensure that the local correctness of the code is maintained, the code commit workflows should perform automatic testing over a database of tests that is continuously updated. More, after integration is finished, unit testing of individual framework component in its main depot must take place because local correctness doesn't imply correctness after integration. Without unit testing in place for the framework components on both sides (projects and framework's depots), the code taken back might contain bugs that would require integration back and forth before fixing all problems. This would induce important costs on the overall process. Design by contract and its follow-ups suggested in (Meyer, 1997) and (McNeile, 2010) represent important topics to apply in the discussed use-cases. On short, functions should have pre and post conditions (implemented in many common programming languages used these days by asserts on input and output). Derived classes overriding functions must have the post conditions as least as strong as the overridden function in the base class. This ensures that Liskov substitution principle

((Liskov and Wing, 1994) and (Leavens et al., 2000)) holds for the client customized objects.

4.5 Evaluation

Before each release, the framework team should check divergences and take back changes that make sense from the framework branched repository of each project. In Table 1, we compare the average time needed for this process considering the initial moment when there was no comment rule on framework code changing, versus the version when we enforced the rules through automatic scripts that checked every code commit. The results show that the comments using the description of the change and author were very efficient in terms of cutting costs, integration being approximately 50% faster due to a quicker collaboration between the author of the change and integration software engineer, and overall an easier way to understand the change and put it into a context.

Table 1: Average harvesting and convergence time (average hours for a single software engineer).

Initial version (no code comment rules)	Enabling automatic code commit checks
125	67

SOLID principles were strongly suggested and somehow enforced between teams by using code reviews. When working on a project team, the change affects only that team but when you share a component with N teams, then someone's work will impact everybody. The divergence tool provided an easier way to check the divergence hot spots and consider them for refactor. This is shown in Table 2. The comparison was made between the version where a software engineer had to manually compare branches using source compare tools, and the version with the divergence tool enabled. Not only that the divergence metric is more exact, but the results show, as expected, an improved overall cost in identifying the hot spots.

Table 2: Average time (hours for a single software engineer) to analyze the divergence hot spots (and priorities for extensibility refactoring). There were on average 6-7 projects sharing the framework.

Manual repository compare	Using divergence tool
21	4

5 RELEASE AND PACKAGE UPDATES

As mentioned in Sections 1 and 4, aside from continuous integration, maintenance and sharing code between projects, the framework team has another two important roles:

- Refactor the existing code and provide more flexibility in areas that are subject to divergence.
- Implement new features that prepare the framework used inside the organization for the future.

All these changes are performed in the main repository of the framework team (Figure 1) and at certain moments of time, a new framework package is released. This package must be integrated by the clients to take advantage of an improved framework code base and new features. Integrations are always costly especially for the client who needs to reconsider pieces of code that were interacting with framework code that has recently changed. Efforts must be put in making the integration process easier. The process of refactoring existing code in the framework can be split into two parts: announce deprecation (through compile warnings) in release N, then in release $N+T$ (in our organization T was 1, but it might adapt by the frequency of releases) make changes mandatory by removing deprecated functionality at all. This way, not only that teams sharing the framework have time to allocate resources and plan for an upgrade, but they could also take position if the changes are not good for their project. Also, at each release of the framework, the main development team responsible for releases can provide a collection of scripts to assist the integration of the new version inside projects. For instance, if a core function doesn't need one of its parameters anymore, a script can iterate over all client code and automate this change. In our organization, by creating scripts for automatizing the integration process as much as a machine can do straightforward, and enabling the deprecation technique, the integration time of a new framework release was reduced by approximately 30%, as shown in Table 3.

Table 3: Average integration time (average hours for a single software engineer) to a new framework release. Projects were allowed to integrate at their own pace, in the table we show integration cost at each 4 weeks (every framework release) or 8 weeks (every other iteration). Integration at every iteration scales better because divergences can add and integration complexity grows (as mentioned in Section 4).

Integration cycle	Integration time before	Integration time after
4 weeks	117	79
8 weels	253	172

6 CONCLUSION AND FUTURE WORK

This paper presented a collection of processes, tools and development methodologies that can help organizations to improve the distributed development and integration costs of a framework shared by internal project teams and which is subject to frequent changes. Many of the things discussed in this paper could be improved in the future. In the first place, a more formal and detailed metrics system which evaluates the framework processes against costs should be developed. At each modification and adoption of processes, metrics could show if the organization is stepping in the right direction or not. Apart from the object-oriented techniques for extensibility and modularity, which will probably always remain an area of improvement, another aspect that might worth more efforts in the future is the automatism of the integration processes (e.g. using artificial intelligence mechanism to provide automatic smart agents that can do integrations with automatic code reviews from human software engineers).

REFERENCES

Bosch, J., Molin, P., Mattsson, M., and Bengtsson, P. (2000). Object-oriented framework-based software development: problems and experiences. In *ACM Computing Surveys, volume 32, number 1*.

Cesare and Wilde, E. (2014). Why is the web loosely coupled? a multi-faceted metric for service design. In *Proceedings, WEBIST 2014, Barcelona*.

Fontoura, M., Braga, C., de Moura, L. M., and Lucena, C. (2000). Using domain specific languages to instantiate object-oriented frameworks. In *IEEE Proceedings - Software, volume 147, number 4, pp. 109-116*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns, second edition, Template method, pp. 325330*. Addison-Wesley, pp. 325-330.

Haefliger, S., von Krogh, G., and Sebastian, S. (2008). Code reuse in open source software. In *Management Science Journal, Volume 54, no 1, pp 180-193*.

Jagdish, B. (2000). Evaluating framework architecture structural stability. In *ACM Computing Surveys, volume 32, number 1*.

Larman, C. (2005). *Applying UML and Patterns An Introduction to Object-Oriented Analysis and Design and Iterative Development 3rd edition*. Prentice Hall.

Leavens, G., Dhara, K., and Krishna, K. D. (2000). *Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems*.

Liskov, B. and Wing, J. M. (1994). A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems, volume 16, number 6, pages 1811-1841*.

MacDonald, S., Tan, K., Schaeffer, J., and Szafron, D. (2009). Deferring design pattern decisions and automating. In *ACM Transactions on Programming Languages and Systems, volume 31, number 3*.

Martin, R. (2000). *Design Principles and Design Patterns*. objectmentor.com.

Martin, R. (2002). *Agile Software Development: Principles, Patterns and Practices*. Pearson Education.

Mattsson, M. and Bosch, J. (1997). Framework composition: Problems, causes and solutions. In *Proceedings of 23rd International Conference on Technology of Object-Oriented Languages and Systems*.

McNeile, A. (2010). Framework composition: Problems, causes and solutions. In *A framework for the semantics of behavioral contracts, In Proceedings BM-FA '10. ACM, New York, NY, USA*.

Meyer, B. (1997). *Object-Oriented Software Construction, second edition*. Pretience Hall.

Myers, G. J. (1978). *Composite/Structured Design*. Van Nostrand Reinhold.

Sant'Anna, C., Figueiredo, E., Garcia, A. F., and Pereira, J. (2007). On the modularity of software architectures: Concern-driven measurement framework. In *Proceedings of Software Architecture, First European Conference, ECSA 2007, Spain, pages 207-224*.

Schwarz, N., Lungu, M., and Nierstrasz, O. (2012). Seuss: Decoupling responsibilities from static methods for fine-grained configurability. In *Journal of Object Technology, Volume 11, no. 1, pp. 3:1-23*.

Spichkova, M. and Schmidt, H. (2015). Requirements engineering aspects of a geographically distributed architecture. In *Proceedings of the ENASE 2015, Barcelona, pp. 276-281*.

Yuan-Hsin Tung, Chih-Ju Chuang, H.-L. S. (2014). Framework of code reuse in open source software. In *n Proceedings of APNOMS 2014*.