Processes and Practices for Quality Scientific Software Projects

Veit Hoffmann[a], Horst Lichter[a], Alexander Nyßen[b]

[a]*Research Group Software Construction, RWTH Aachen University, Aachen, Germany*
[b]*itemis AG, Lünen, Germany*

Abstract

Nowadays modern software development processes are well established and are one of the mayor success factors for software projects. However, many software projects in scientific organisations have serious quality issues since they lack a feasable development process. In this paper we discuss a feature based development process for scientific software projects that addresses the specific characteristics of scientific software. Moreover, we introduce a set of best practices for the infrastructure and management of such projects.

Keywords: software development process, best practice, scientific software

1. Introduction

Software tools are a vital prerequisite for scientific research. They can be used to demonstrate the relevance of research results and their applicability in practice. Software tools enable to conduct case studies in scientific as well as in industrial environments. Furthermore they extend the addressees of research results and are often the root for industrial or open source software development projects.

Although the importance of software tools for research is well accepted in the scientific community the quality of scientific software is seldom adequate. Scientific software is often error prone, doesn't implement the intended functionality or is only usable by the developers since either the documentation is missing or outdated.

In our experience quality problems of scientific software are mostly caused by an inadequate software development process and an insufficient development infrastructure. Many scientific software projects don't define a dedicated development process or implement a process that doesn't cope with the specific constraints of scientific software development. However, this induces various problems in the management of the software and in the collaboration of the developers. Those problems often lead to severe quality issues in the software.

The remainder of this paper is structured as follows. In chapter 2 we characterize scientific software projects and delineate them from other development projects. In chapter

Email addresses: vhoff@swc.rwth-aachen.de (Veit Hoffmann), lichter@swc.rwth-aachen.de (Horst Lichter), alexander.nyssen@itemis.de (Alexander Nyßen)

3 we briefly describe ViPER, a scientific software development project that inspired most of the results presented in this paper. Chapter 4 defines a set of goals for an adequate scientific software development process for medium to large size, long living projects. In chapter 5 we introduce a process frame that enables those defined goals. Afterwards we introduce best practices for our process frame in chapter 6. Finally chapter 7 situates our best practice approach with some related research before we give some concluding remarks in chapter 8.

2. Characteristics of Scientific Software Projects

Scientific software projects are very diverse. They vary in size, lifespan, application domain and used base technology. Therefore it is impossible to define one single fixed development process that fits for all those projects. Anyhow, all scientific software projects can be distinguished by two major characteristics that clearly separate them from other development projects like open source or commercial development.

1. **Scientific software projects are embedded in one or more research projects.** This has various impacts to the software and its underlining process, because research directions may change based on obtained results, research opportunities or funding. This usually shifts the focus of the software and often induces changes of the requirements and architecture. Additionally not every research produces the expected results. Empirical studies may uncover misconceptions and flaws in earlier research. This often also impacts the software since empirical results may show that methods or technologies supported by the software are not feasible. Thus scientific software projects have to deal with dead-ends and rollbacks regularly.
2. **Scientific software projects are preformed with heavy student involvement.** Main parts of the functionality of many scientific software projects are developed in students' theses or with the help of student developers. This causes two special problems.
   First, students are usually inexperienced. Many students have never been in real world development projects. They often lack software engineering know-how and have no experience with tools, languages, libraries or frameworks that are used in the project.
   Second, students just take part in the development of the software for a very limited period of time. Students typically join the development team just for their thesis and are seldom a part of the development team for more than nine months. Additionally students often only contribute to a very limited part of the project, namely the specific feature that is being developed in the context of their thesis. Thus students often don't have overview over the project as a whole and have little interest in the overall project success.

Any development process aiming to support and guide scientific software projects has to consider these special characteristics. In the following however we concentrate on medium to large size, long lining development projects, like SESAM [20], Fujaba [7] or MontiCore [11]. I.e., we primariliy consider scientific development projects with a livespan of at least several years and at least 5-10 developers. Those projects typically create software, that should be used in multiple contexts, by several different kinds of stakeholders, especially those who are not directly involved in the development process. Although several of

the described best practices are also useful in smaller projects the development process described here is not adequate for small prototype projects that don't have to address documentation or architectural quality because it enforces different quality assurance measures that are unnecessary in those situations.

3. Viper - A Scientific Software Development Case Study

In the following we briefly describe the ViPER project that we run for several years. The essence and the lessons learned of this project are the central experience basis to propose a process frame for scientific software projects.

ViPER (Visual Tooling Platform for Model-Based Engineering) [21] is a tooling platform to leverage model-based engineering. It is based on Eclipse [6] technology and offers support for UML-based visual modeling, UML-based ANSI-C code generation, extended support for editing and simulating of detailed narrative use case descriptions, as well as built-in dedicated methodical support for the MeDUSA-Method [12].

Having started in July 2004 as a mere experiment to evaluate Eclipse related EMF and GEF technologies in the form of a simple UML state machine diagram editor, which was bundled into a single plug-in and accounted for about 3800 lines of source code, ViPER has grown into a rich and extensive tooling environment in the following years. It up to now bundles round 50 plug-ins and subsumes more than 200,000 lines of code, incorporating the contributions of 18 developers (scientific staff as well as students) over the years. In the recent years some parts of ViPER have been promoted to the Eclipse frameworks or to separate open source projects.

As ViPER grew the development infrastructure and the process management measures had to be adopted several times and nowadays ViPER is developed with a defined development process that relies on independent features, build on a common platform and a common development infrastructure.

The development infrastructure of ViPER consists of a CVS-Version-Management Repository [5] and a Bugzilla-Issue-Tracker [4], that are connected by a SCM-Bug [19] integration. Additionally ViPER maintains a dedicated build server for release engineering and quality assurance. This build server runs an ANT based Eclipse product build that creates releases on a nightly basis. Apart from the assembly of the ViPER-Product each build includes the validation of the source artifacts with checkstyle rules and the creation and execution of a regression test suite to assure functionality and conformance.

Inspired by Scrum [18] the ViPER project defines a lightweight sprint oriented planning process and uses Bugzilla as planning tool. Every sprint is connected to a target milestone in the Bugzilla system and all intended functionality is filed as enhancement requests against this milestone.

Since the ViPER development team is quite small the ViPER process relies on direct communication and the coordination of different features is done in weekly management meetings. Those management meetings are the core activity of the ViPER process and must therefore be attended by all project members. They are used for planning, the coordination of feature projects and the maintenance of the platform. They regularly last for about 1,5 hours. Every team meeting is started with a short status report from every project member. Afterwards team meetings include a management part, where sprints and scoping sessions are planned or development tasks are assigned. The meetings often end with an open discussion session where specific problems are discussed.

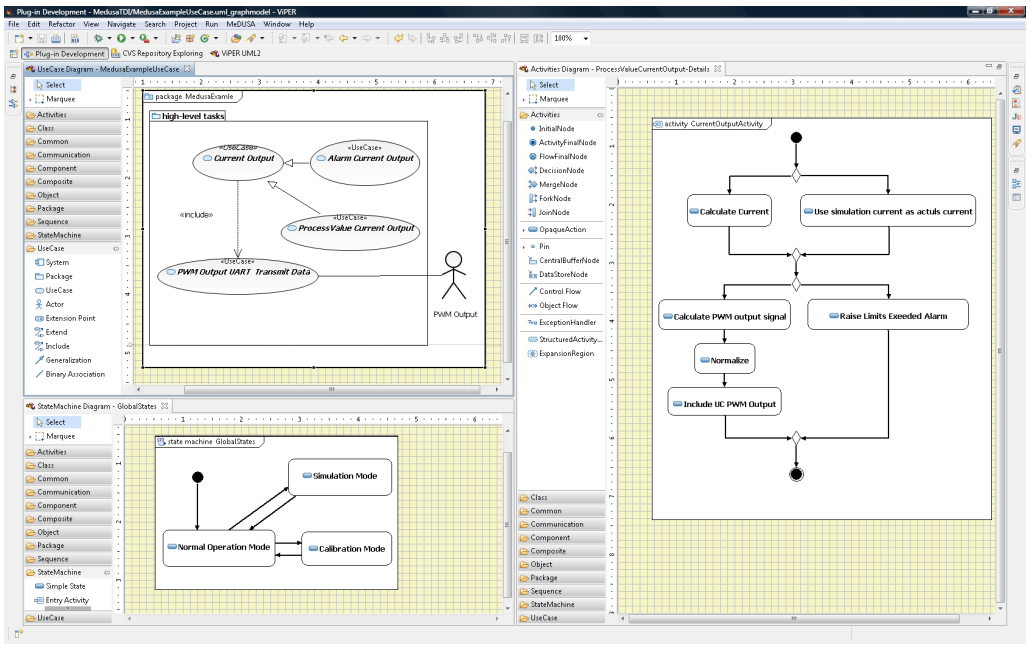

Figure 1: ViPER Screenshot

Currently ViPER is managed in a well established process that supports its efficient development. Although it is lightweight enough to enable an easy setup of new feature projects it includes measures to assure integration and quality. Moreover parts of the ViPER process have been successfully transferred to several other scientific development projects. In the next sections we present the essence of the ViPER process in the form of an extensible process frame and a set of best practices for scientific development projects.

4. Objectives of a Scientific Software Development Process

As stated before we primarily target long living scientific software projects that should be used by several different kinds of stakeholders. The primary objective of a development process for this kind of scientific software projects is to enable the development of software in sufficient quality. For most scientific software projects a beta-quality level is sufficient, since it is neither possible nor necessary to develop the software to full commercial quality. Anyhow, since scientific software must be usable in case studies and field research they have to provide a minimal level of usability and robustness. Apart from implementing the intended features correctly the software must be documented and robust against simple misuses like invalid input data.

Additionally the architecture is a core concern for many scientific software projects, since the software is often enhanced by new features or restructured due to new requirements. Thus the maintenance of sustainable software architecture is a vital task for a scientific software development process.

To achieve those objectives the process must especially address the specific characteristics of scientific software projects presented in chapter 2. This implies that a feasible development process must provide support for agile, evolutionary development, which is induced by the volatile development. Changes caused by shifted tool usage and the impact of requirement changes to the architecture must be manageable, too. Furthermore the software has to be rolled back to defined baselines after reaching a dead-end in the project.

Additionally a development process must address the specific human factors in scientific software projects, which means it must enable students to produce high quality results and to contribute to the project. Thus a development process must define clear rules and provide measures to transfer existing and new know-how. Producing high quality results however demands a lot of discipline especially from inexperienced developers like students. Therefore the development process has to provide team measures to raise the involvement of the participating students and to raise their interest in the project success.

5. A Process Frame for Scientific Software Projects

As denoted before scientific software projects have two specific characteristics that affect their lifecycle and development processes. First, they undergo continuous changes since they evolve alongside the theoretical research. Second, most of the functionality is developed in different thesis works performed by students.

In this section we describe an iterative, incremental and evolutionary process frame for scientific development projects that explicitly addresses those specific characteristics. This process frame should be enhanced with best practices described in the later sections of this paper.

Scientific software projects resemble open source projects in many ways. They are evolutionary, open, focused on an extensible infrastructure to react on usage shifts, suffer from quick changes in the development team and mainly develop functionality in dedicated subprojects. Therefore we have adopted many ideas from open source development projects for the proposed process frame and the best practices.

Our process frame reflects the overall project organization approach of incremental development projects based on feature development. A feature may be defined as a set of related functions that are implemented together to realize a specific goal. Furthermore it explicitly considers the systematic development of reusable features, which we call platform.

To maintain a stable feasible architecture and manage the development of the various features, the process frame consists of three separate but interconnected sub-processes: a platform-process, a set of feature-processes and an underlying coordination and collaboration process (see Figure 2). We will explain these processes in the following.

5.1. Platform Process

The objective of the platform process (similar to project line development) is to ensure that a reusable platform (called the platform feature) is developed and maintained continuously. The platform process is initialized by the platform team at the start of a scientific development project and is performed until project end. It manages the

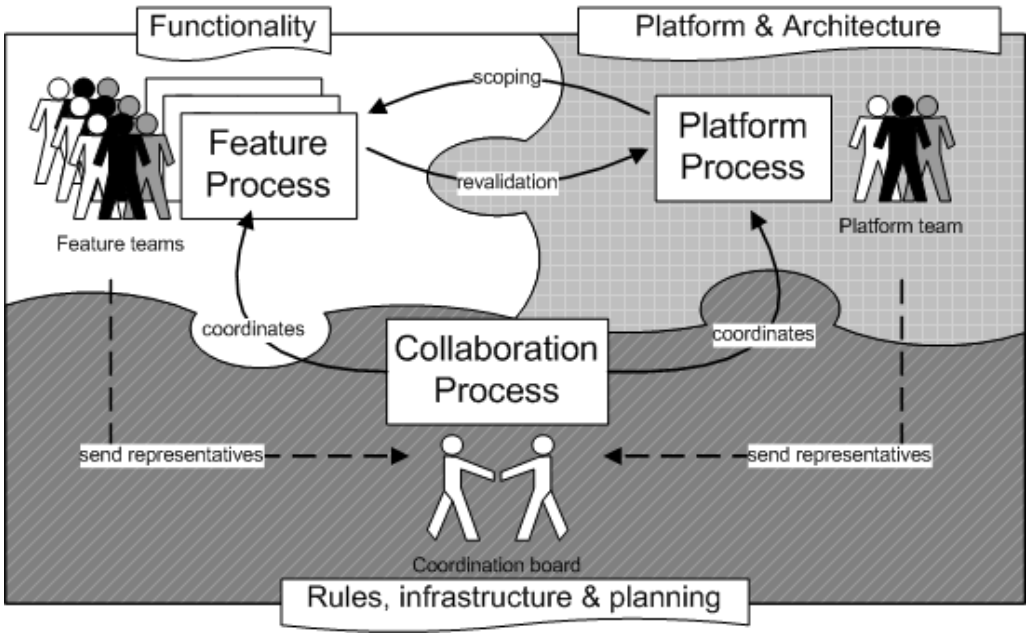

Figure 2: Dependencies between the sub-processes

platform feature, a single feature that contains two different kinds of information. First, it contains the software's common code platform (e.g. reusable functionality that is used as a basis for feature development). Second, it manages the software architecture. It defines an architectural pattern for the entire software and a common scheme for the definition of interfaces between features. Additionally it introduces a set of tasks for the maintenance and quality assurance of the software architecture, e.g. scoping, revalidation or measurement tasks. Because the platform-feature is the core of the development and the platform process is performed continuously, the platform team should consist of experienced developers that are responsible for the project and accompany the project for a longer time. Thus, in most cases the platform team is formed by PhD students, post-docs or assistant professors.

5.2. Feature-Process

As the main functionality of the software is developed in feature projects, a new feature process is initialized for each feature development. Because every feature process is performed by an autonomous feature team, several feature processes can be run in parallel. The single feature processes may vary broadly, since the feasibility of a development process for a feature is dependent on the type of the feature, the abilities of the team, the schedule and the workload. But it has to be ensured that each feature process includes an internal planning for the feature, a requirements engineering session, the definition of the feature's architecture as well as the feature's implementation, integration and documentation.

5.3. Coordination and Collaboration-Process

Finally, the coordination and collaboration process manages and coordinates the evolutionary feature based development of the software. It defines all process standards, i.e., process rules and documentation standards and templates for the platform and all feature processes. Furthermore, it defines and establishes the development infrastructure, constituting of e.g., configuration and change management, release engineering infrastructure and development environment. Moreover, the coordination and collaboration process determines the project's global time schedule, i.e., it defines synchronization points for the platform and all feature projects. Finally, it nails down global management tasks for the coordination of development teams. The coordination and collaboration process is typically managed by a coordination board that is created out of developers from the feature and platform teams.

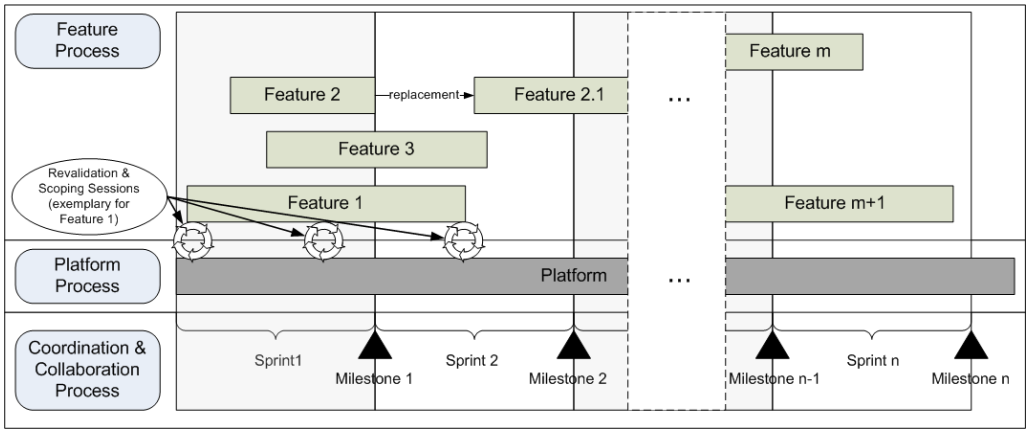

Figure 3: Project snapshot showing sub processes on the timeline

Although each feature process may define its own specific development lifecycle, all feature processes as well as the platform process must adhere to the coordination and collaboration process. They must apply the defined development infrastructure and the documentation must conform to the templates. Additionally the coordination and collaboration process defines the project pulse, i.e., it defines the releases and corresponding milestones that are obligatory for all feature processes.

The presented process frame on the one hand enables to efficiently develop features in dedicated feature processes, since feature processes are lightweight an rely on a common infrastructure. On the other hand a global, sustainable platform can be developed and maintained. Thus a set of management practices must be established to coordinate the decentralized sub processes of the presented process frame. A set of best practices addressing especially the most pressing management challenges is presented in the following section.

6. Best Practices

In the following we present a couple of best practices that we recommend for scientific software projects. These best practices are often adaptations of agile or open source

techniques and are mostly associated to the platform-process or to the coordination and collaboration process.

6.1. Release and Version Management

Apply a sprint based Release Management

Industrial strength release planning is typically inadequate for scientific software projects where features can quickly emerge and disappear again, because it is too heavyweight and inflexible. However, the delivery of the features and their integration in the platform needs to be managed. We therefore recommend a lightweight release planning similar to sprint planning in Scrum [18]. We recommend sprints of about three months. Although this is longer that typical Scrum sprints we have very positive experiences. First, the development speed is often slower compared to commercial projects because often developers (students) are inexperienced and the feature-teams are very small. Second, we recommend that each milestone should include contributions from every active feature-project. This positively raises the acceptance of milestones as the most important delivery target. But this can only be achieved if the sprints are not too short. At the end of each sprint a release is build which should contain all completed features as well as those features under development that have reached a stable state.

Use an integrated Change Management and Version Control System

Scientific projects are performed in a decentralized way and evolve evolutionary. Therefore the management of changes and their impact to existing project artifacts is a crucial task. This demands a close integration of the version and change management system. Every bug or enhancement request should be handled by a ticket of the change management system and every change to a project artifact must be associated to a respective ticket. This enables to keep track of changes and to trace changes back to project artifacts. Moreover all artifacts (including source code, documentation, test cases and examples) should be under version control. This has three mayor benefits. First, storing all kinds of documents in a version control system reduces loss of important information since even less important information is stored. Second, the actuality of non-code artifacts, like requirements or architecture definitions, is raised significantly. Third, monitoring the impact of changes to all affected artifacts is supported. Additionally we recommend to version all artifacts of a single feature project together.

Implement all new features against the head revision

Each new feature should be developed against the head revision of adjacent feature projects and the platform project. This demands some discipline from the developers, since changes to the interface of one feature may directly affect others. Thus a stable interface design and continuous communication are needed. We experienced only little communication overhead if the interfaces are defined upfront and described explicitly. Besides, the integration effort of components developed in parallel is reduced. Additionally an immediate integration of features under development has positive effects on the team spirit, since developers get a quick feedback about their results.

Use an automated build and test system

An automated build system should be used to assure the integration of all features including those that are under development. We recommend a dedicated build process performing nightly integration builds to assure the compilability and integration of the components. Additionally the build system should create release builds on a regular basis. We recommend one release alongside each sprint. A release should always be preceded by several release candidates used for manual testing.

Furthermore we recommend enriching the build system with automated regression tests and static analyses performed after each successful build. This is especially important since many developers only work on fragments of the system functionality and don't test the integration with other fragments or features. Although the integration of a dedicated quality assurance infrastructure in the build system needs quite a lot of setup effort, it is worthwhile since deficiencies and problems in the source code are detected very early.

Provide an update infrastructure

Scientific software often changes and evolves quickly. Therefore an update infrastructure should be provided once the software is released, e.g., for field studies. An operative update infrastructure improves the user's acceptance to use the respective software, since they can experience the evolution of the software and keep up to date with only little effort. Therefore a single available source for updates and software releases should be established and users should be informed about new updates.

6.2. Quality assurance

Create regression tests

The platform project and its feature projects change rapidly because requirements are changed or new requirements emerge. Moreover functionality is moved from feature projects to the platform or vice versa as a result of scoping sessions or performed refactorings. This demands to run regression tests as often as possible to assure the stability of the software. We recommend implementing black box regression tests for the platform and every feature. Moreover the coverage of those tests should be measured and additional tests should be added if the coverage is insufficient.

Perform a quality assurance phase for each feature project

Feature projects should plan a feature freeze and a quality assurance phase at the end of their development (often called endgame). The endgame should assure three important quality aspects. First, it must assure that the required functionality is implemented and that the feature is correctly integrated in the platform. Second, it should assure that the feature's architecture conforms to the platform's architectural rules. Third, the endgame should assure that sufficient documentation is available. Since feature projects are typically performed in the context of students' thesis works and students are often unavailable after the end of the thesis a rigorous endgame is vital for the overall project success. Thus any information that is not handed over is lost and can only be recovered with very high effort. Additionally we recommend that the platform team performs a platform scoping session as part of the endgame together with the feature team (see best practices of platform management).

Measure the architectural quality on a regular basis

The architecture is a core concern for most scientific software projects. To maintain a sustainable architecture in the dynamic environment of scientific software projects the conformance of feature implementations to the architecture definition in the platform must be measured and checked regularly. Nowadays, there are several tools to measure and assess software architectures [9] [10] that can be used. The results of those measurements should be discussed in team sessions to sensitize the developers for the architectural demands of the platform and thus to get less violations in upcoming sprints or feature projects.

Explicitly define the interfaces of each feature

The interfaces of every feature should be explicitly defined and access to any feature should only be allowed through those interfaces. Thus the impact of changes of a feature to other features is reduced and manageable. The adherence to this rule should be checked automatically alongside the quality measures performed by the build system.

Define templates and rules for project artifacts

A template and a respective set of rules for all kinds of project artifacts (code and non-code) should be defined. Templates increase the readability of documents and ease identification of changes if a version management system is used. We recommend to use style rules defining formatting and encoding for all code artifacts and to check those rules by an automatic checker.

6.3. Platform Management

Perform platform scoping sessions on a regular basis

The platform contains reusable code that should be used by all feature projects. Its main purpose is to prevent feature projects from developing the same functionality several times. However, most of the functionality is developed in feature projects including reusable functionality that should be part of the platform. Therefore special platform scoping sessions should be performed regularly to transfer reusable parts from feature projects to the platform. A scoping session is a joint session of a feature team and the platform team and has two purposes. First, potential platform candidates useful for other features should be identified. Second, each candidate should undergo a detailed scoping analysis. This analysis has to define a migration strategy for the candidate, i.e., a set of refactoring and generalization steps that are necessary to integrate the candidate with the platform. Every migration strategy should afterwards be planned as a normal task in a sprint.

Perform refactoring analysis before developing a new feature

Whenever a new feature should be developed the first step is to evaluate the current platform architecture to ensure that the platform architecture is still adequate to implement the new feature. If this is not the case, a set of refactoring steps have to be identified to improve the architecture and to enable the integration of the new feature. These refactorings are performed (typically by the platform team) before the new feature project is started. This results in a stable and maintainable architecture and minimizes the integration effort of new features.

Check for obsolete features on a regular basis

Since resources in scientific software projects are short no effort should be wasted to develop or maintain features that are no longer needed. Thus it is necessary to check for features that are no longer needed regularly. A feature may become obsolete for three reasons. Either because the usage of the tool has changed or it was replaced by a new feature or it can be replaced by functionality imported from e.g., open source projects. We recommend to check for obsolete features at least every time a sprint is planned. For each obsolete feature the platform team has to define a decommission strategy. It defines necessary steps how to migrate those features depending on an obsolete feature to the replacing one. The implementation of the decommission strategy should be planned as a normal task in the upcoming sprint, similar to a migration strategy.

6.4. Team Management

Perform pair programming sessions with new developers

Inexperienced developers should at first do some pair programming session together with an experienced developer (often a member of the platform team). This is highly accepted and has three mayor benefits. First, it enables the new developer to bridge the technological gap. Second, the experienced developer can transfer knowledge about rules and standards to the new team member easily. Third, the integration of new team members to the development team is facilitated, since they get to know other team members better.

Provide an infrastructure for knowledge documentation

Every project should provide an infrastructure to document and transfer knowledge, e.g., an open wiki or a blog. This is extremely useful if the project is using a lot of open source technologies and frameworks where the documentation is scattered in the Web. Those systems serve as a starting point for developers to search for documentation, tips and advices.

Perform regular team sessions

The project team should meet on a regular basis. Those meetings are a central practice in our experience and have four mayor purposes. First, they are a management and planning instrument. In the team meetings every developer should present the status of the current work and the next upcoming development steps, to keep track of the overall project status. Additionally new sprints should be planned in the meetings. This raises the developers' commitment to deliver in time, because they are involved in the planning. Second, team meetings are a discussion platform for specific problems of the developers which has the following benefits. First, whenever a developer is stuck with a problem the team may help him to find a feasible solution and second the discussions can be used to spread technical as well as process knowledge. The third purpose of team meetings is to maintain the platform architecture. Therefore the results of architectural measurements should be discussed, architectural decisions should be made and scoping sessions should be planed. The last and maybe the most important purpose of the team meetings is to create a team spirit and raise the developers' involvement in the project. Therefore team meetings should have a casual ambience and encourage open discussions.

6.5. Best Practices in Context

Obviously some of the presented best practices have different relations to each other. Some are prerequisites for others or the implementation of one practice positively influences others. Figure 4 shows a sketch of those dependencies.

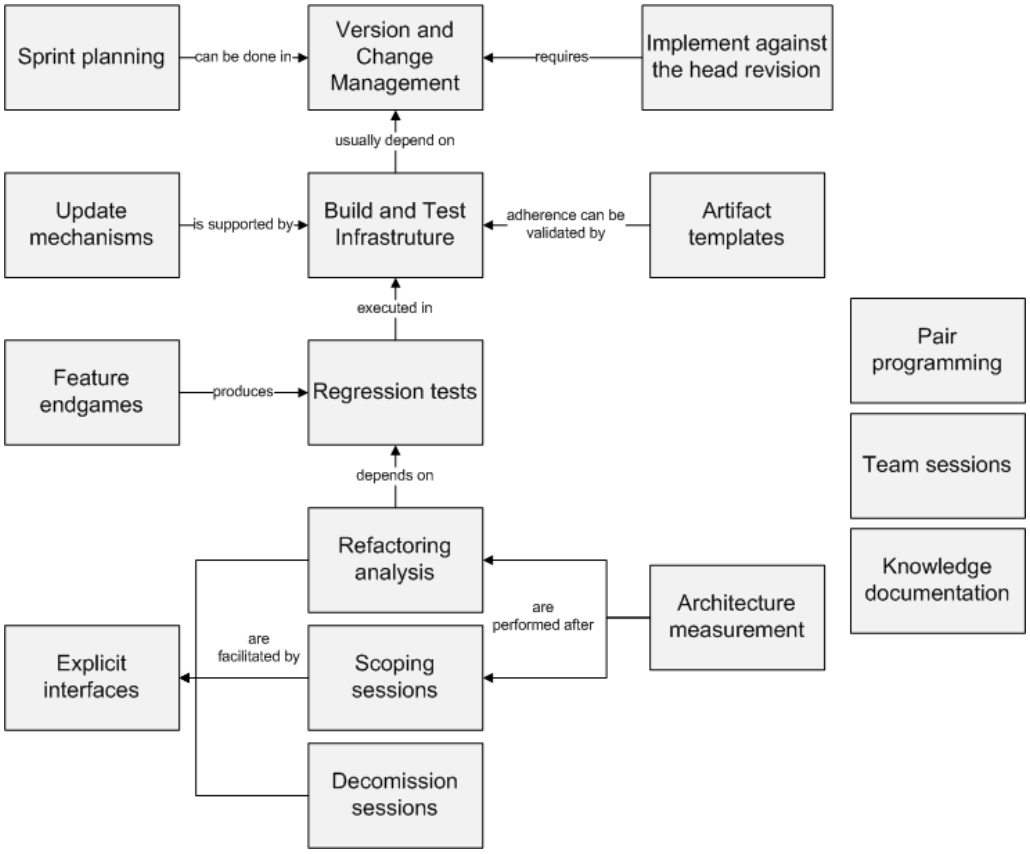

Figure 4: Best practices in context

These dependencies obviously influence the sequence to establish practices in a project, typically practices should be introduced in related groups.

7. Related Work

Feature oriented development processes have been discussed in several publications. According to the feature driven development approach proposed by [13] all features are defined upfront. Afterwards all features are developed in an iterative fashion, where in each iteration one feature is realized. Several agile development processes like Scrum [18] or XP [2] use features for the planning of iterations but during devlopment all features of the iteration are developed together by one development team.

The product-line engineering community [14] introduced the idea of a common platform and scoping techniques for its maintenance. However, those approaches mainly focus the management of variability of multiple products. Thus they are not specific enough for scientific software development projects with only one single product, which consists of a platform and several features.

Best practices for development processes have be discussed by several authors. [3] e.g., focusses mainly human aspects in development projects whereas [1] describes a quite elaborated set of process patterns for large-scale object-oriented systems. In resent years several open-source projects like Gnome [8] or Eclipse [6] have evolved to software ecosystems where one project builds the platform for the development of several others. Most of those define a development process with a milestone oriented release planning and best practices. [16] discusses different management aspects of those projects. However, those projects don't aim for one integrated solution. Thus no exlipicit scoping and revalidation of the platform is planned.

Today none of the current development approaches explicitely concerns the specific characteristics of scientific development projects. Although many of the discussed approaches and techniques can be usable in scientific development projects most of them need to be adopted to those projects specific needs.

8. Conclusion and Outlook

In this paper we discussed the importance of a feasible development process as basis for the creation of high quality software in long living scientific development projects and we depicted the specific aspects of scientific software projects that impact a feasible development process. We outlined a generic process frame, that addresses the specifics of scientific software projects and sketched a set of best practices for the presented process frame. Furthermore we presented the ViPER project as one case study for a successful scientific development project, that was managed according to the presented best pratices.

We have sucessfully performed the QMetrics project [17, 15] with the described best practices and we are currently running several other scientific projects we perform at the faculty and with different industrial cooperation partners, with the presented process frame and the described best practices. Our experience in all these projects are also very positive. Additionally we plan to do a systematic analysis of the process and its respective best practices. Therefore we intend to perform a series of questionnaire oriented qualitative analyses in several scientific projects that may or may not implement our process frame.

Until now the described process frame only considers generic basic best practices. In the future we plan to enrich the best practice section with a set of optional measures for specific project settings. Moreover we plan to provide strategies for the introduction of a best practice based process frame in a new project or the migration of a running project to such a process.

References

[1] Ambler, S. W., 1998. Process patterns: building large-scale systems using object technology. Cambridge University Press, New York, NY, USA.

[2] Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.
[3] Brooks, F. P., August 1995. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition), 2nd Edition. Addison-Wesley Professional.
[4] Bugzilla Project, Jun. 2010. Bugzilla - project website.
URL http://www.bugzilla.org/
[5] CVS Project, Jun. 2010. Concurrent version system - project website.
URL http://www.cvshome.org/
[6] Eclipse Project, Jun. 2010. Eclipse - project website.
URL http://www.eclipse.org/
[7] Fujaba Project, Jun. 2010. Fujaba - project website.
URL http://www.fujaba.de/
[8] Gnome Project, Jun. 2010. Gnome - project website.
URL http://www.gnome.org/
[9] Hello2Morrow, Jun. 2010. Sotoarc - project website.
URL http://www.hello2morrow.com/products/sotoarc
[10] Metrics Project, Jun. 2010. Metrics - project website.
URL http://metrics.sourceforge.net/
[11] MontiCore Project, Jun. 2010. Monticore - project website.
URL http://www.monticore.de/
[12] Nyßen, A., Lichter, H., Streitferdt, D., Nenninger, P., 2008. Medusa - a model-based construction method for embedded and real-time software. In: COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference. IEEE Computer Society, Washington, DC, USA, pp. 1376–1382.
[13] Palmer, S. R., Felsing, J. M., February 2002. A Practical Guide to Feature-Driven Development (The Coad Series). Prentice Hall PTR.
[14] Pohl, K., Böckle, G., Linden, F. J. v. d., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
[15] QMetric Project, Jun. 2010. Qmetric - project website.
URL http://www.qmetric.org/
[16] Sandred, J., 2001. Managing Open Source Projects: A Wiley Tech Brief. John Wiley & Sons, New York, NY, USA.
[17] Schackmann, H., Jansen, M., Lischkowitz, C., Lichter, H., 2009. Qmetric - a metric tool suite for the evaluation of software process data. In: ICSE Companion. IEEE, pp. 415–416.
[18] Schwaber, K., Beedle, M., February 2002. Agile Software Development with SCRUM, illustrated edition Edition. Prentice Hall.
[19] Scmbug Project, Jun. 2010. Scmbug - project website.
URL http://freshmeat.net/projects/scmbug/
[20] SESAM Project, Jun. 2010. Sesam - project website.
URL http://www.iste.uni-stuttgart.de/se/research/sesam/
[21] ViPER Project, Jun. 2010. Viper - project website.
URL http://www.viper.sc/