Enhancement of Renew to Version 4.0 using JPMS

Laif-Oke Clasen[1], Daniel Moldt[1], Marcel Hansson[1], Sven Willrodt[1] and Lukas Voß[1]

[1]*University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics*
https://www.informatik.uni-hamburg.de/TGI/renew/

Abstract

Tools must be continuously developed further. For the Java reference nets tool Renew, the latest results of an extensive transformation process are presented. Starting from our good plugin architecture, we were able to convert the Java plugins into Java modules. Most of the architectural hindrances arose from the programming language itself. Java 9 introduced the Java Platform Module System (JPMS), and the Long-Term Support versions of Java 11 and 17 now provide modules and layers as central concepts to support better architectures. Their application is discussed using the transformation from Renew 2.5 to Renew 4.0, which applies the module concept to a medium-sized software toolset. Experiences from this extensive process are highlighted.

Keywords

Tool, Petri Nets, Release, Java, Module, JPMS, Migration, Renew

1. Introduction

Renew is a continuously developed extensible modeling and execution environment for Petri nets with various formalisms and other modeling techniques (see http://www.renew.de and [1, 2, 3]). The most prominent Petri net formalism of Renew is the Reference net formalism [4], which combines the concept of nets-within-nets [5] with reference semantics and the expressive power of object-oriented programming in the form of Java as the inscription language.

Reference nets in Renew can handle Java objects as tokens and Java expressions in transition inscriptions to execute Java code during the simulation. With the nets-within-nets concept, it is possible to build dynamic hierarchies of arbitrary height and nestedness. Multiple nets can communicate using synchronous channels, which enable the bidirectional exchange of information via parameters. Combining several channels on transitions allows for the synchronous execution of an arbitrary number of transitions (bindings) across arbitrary many net instances. The formalism is based on true concurrency semantics and, therefore, is well-suited for the modeling and implementation of concurrent software systems.

Renew is written in Java and is available for multiple platforms (including Windows, Linux, and Mac). This contribution covers the newly released new modularized version Renew 4.0, available free of charge, including the source code. [1]

PNSE'22: International Workshop on Petri Nets and Software Engineering, 2022, Bergen, Norway
7clasen@informatik.uni-hamburg.de (L. Clasen); moldt@informatik.uni-hamburg.de (D. Moldt); 5hansson@informatik.uni-hamburg.de (M. Hansson); 5willrod@informatik.uni-hamburg.de (S. Willrodt); 8voss@informatik.uni-hamburg.de (L. Voß)
© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

[1]Download for Renew 2.6 and Renew 4.0 can be found at: http://www.renew.de.
In addition to version 4.0, another new version based on the non-modularized code basis, Renew 2.6, realizes bug

CEUR-WS.org/Vol-3170/paper9.pdf

The development of Petri net tools is a challenge as any other tool support. Software architecture *ages* due to changing environments or technical debts arise[6]. Adaptation of tools to this dynamic environment is therefore necessary. Renew's development spans now more than two decades [2]. Consequently, Renew was adapted to the new Java concepts of modules and layers. Such a change in software architecture means a significant workload for software systems engineers. For example, the adaptation of the Java Language Compiler to the new features took quite a some time: see the Jigsaw project. As a consequence, these language extensions provided by the Java Platform Module System (JPMS) caused considerable efforts for the Renew team since 2017 [3]. Besides this process, new requirements popped up continuously and led to the continuous growth of the software system. Maintaining a running software system and changing the software architecture in parallel put some extra burden on the whole Paose (Petrinet-, Agent- und Organisation-oriented Software Engineering; see [10, 11, 12]) team that managed and performed the development process.

This contribution is a summary of the lessons learned and is intended to provide guidance to other tool developers in the Java community about migrating their software systems to the JPMS.

Section 2 addresses the foundations. Section 3 explains the goals of the development of Renew and emphasizes our development process. Section 4 sketches the way Renew is modularized. Examples of the modularization are covered in Section 5. In Section 6 the results are discussed before we conclude with Section 7.

2. Foundations

As explained above, Renew is a widely used tool for modeling and simulating Petri net formalisms, especially reference nets (see [4]). Important to know is that Renew has been built with a special architecture. First, structured object-oriented components were the building blocks. These were replaced by the plugin-based architecture as introduced in [13, 14]. Agent-oriented concepts influence the architectural style (see [15]). Service orientation and the agent ideas inspired us to strive for a more flexible architecture with dynamic structures at runtime. Java has relevant limitations with respect to dynamic (un)loading of software components, which was covered partly with our plugin architecture by special loading and unloading of classes and nets. For our multi-agent systems dynamic change of system structure is an important concept. While the modeling of reference nets allow this in principle, Java did not forget classes easily.

All this motivated us to start with the migration of our software system to the JPMS. To the best of our knowledge, our architecture is one of the first few examples of a nontrivial software system that migrated to the new concept of modules. Furthermore we seem to be the only Java based software system that uses several layers intensively.

fixes and improvements like a new GUI, which e.g. includes the ability to zoom. Renew 3.0 is an earlier internal prototype based on OSGi. Due to our own plugin architecture, we skipped that version for official releases.

[2]The first official release was published in [1]. After that major topics and enhancements were: architecture guide of Renew 1.5 [7], plug-in architecture at Petri Nets 2004 [8], development environment [9, 3]

[3]Java (Oracle): https://www.oracle.com/java; OpenJDK: https://openjdk.java.net/; Java Community Process: https://jcp.org; background information on Jigsaw the predecessor of the JPMS, starting in 2008 / 2014: https://openjdk.java.net/projects/jigsaw/

The JPMS provided since Java version 9 attempts to improve two important aspects of the Java software system: First of all the modularization/encapsulation of software components is addressed, which is mainly covered by the concept of modules, module path, etc. (see [16]). The concept of a module is based on the concept of information hiding (see [17, 18]) The layer concept covers some improved dynamic structure, which allows for loading and unloading modules at runtime (dynamic class loading). While the module system has already found its way into several software systems (including the Java compiler), the layer concept is rarely used to our knowledge.

The four types of Java modules (basic, automatic, unnamed, and open modules) (see [19, 20]) allow a smooth transition from former Java architectures to modularized architectures . For Renew, we strive for the simple use of basic modules, as the others weaken the encapsulation requirements. However, some exceptions are still required for third-party libraries, as they are not modularized.

This contribution presents the use of basic modules for our own components of Renew to illustrate the software technical improvements. Special keywords describe dependencies between modules. In particular this ensures the existence of *required* modules and covers explicit *export* of accessible parts of the modules. Dependencies that only exist at runtime are covered by *uses* and *provides*. With the keyword *opens*, it is possible to use Reflection arbitrarily, unlike with the keyword *export* where it is only possible on accessible parts. Dynamic aspects of layers allow to group modules and to load/unload them at runtime. We refer the interested reader to the Java ecosystem mentioned above for the technical details.

3. Objectives and Development Process

As mentioned above, our software Renew and its surrounding ecosystem grew over the years due to increased demands with respect to the software features. Maintenance of the software code became more and more difficult. Even our separation of the software components into plugins did not allow for a complete separation of development processes. Even more severe was the fact that the Java code was not compliant with newer Java standards, as there has been a new Java version every half a year since 2017. All this required some decisions about the general setup of the development process and the software architecture. The decision was to migrate to a new third major software architecture line.

To illustrate the different stages of our decision process we explain our basic goal set. The first goals of the project were to evaluate if a migration is possible and worthwhile. A first teaching project and a thesis ([21]) showed that is is possible and promising as the software architecture should improve even further. After that several hinderances conceptually and technically had to be addressed and the general goal was to complete the migration for the whole set of Renew plugins. Especially the change of build environment had to be added to the goal set and addressed (see [22, 23]).

Additional goals were added when it became clear that the migration will be successful: Our Mulan framework (see [24]) and our largest application, the Settler of Catan Game (see [25]) were addressed to run as non-modularized Java plugins first and then become migrated to. To test the applicability several concurrently running thesis developed new plugins directly for the

new architecture (see e.g. [26, 27, 28]). An overall goal was to test and adapt the Paose-approach for such kind of projects and to strive for an improvement of the software quality and the software development process in general.

Renew as a mid-sized software requires non-trivial efforts to introduce the JPMS of Java 9. There are only very few systems of the same or larger size that have successfully migrated. To illustrate the complexity to introduce the modularity concept to Java systems, we hint at the time scale which was necessary to develop the Java Platform Modul System (JPMS), which is modularized now. The transition took from August 2008 until the release of Java 9 about 9 years (see [29, 30]). For Renew the transition has taken four years until the official Release 4.0. Although not all Paose members thought the migration made sense at the beginning of the migration, they were convinced by the resulting architecture. [4]. As a good starting point for the modularization our plugins and plugin architecture proved as advantageous.

Accompanied was the technological push by some application requirements resulting from our Paose approach. Some new major features have been developed, and several (primarily internal) prototypes have been built during the last ten years to provide tool support for Paose. The need to maintain the existing version of Renew 2.x for several ongoing theses was obvious.

Therefore we had to cover also a parallel development of the two major versions. The plugin architecture, our development environment and the Paose-approach as our project basics allowed to maintain both software lines concurrently. Nevertheless discipline was necessary to properly cherry pick improvements and bug fixes from one version to the other. In the beginning this flow was more from Renew 2.5 towards the ongoing development of release 4.0. In the end this flow is now from 4.0 to 2.6. Development of Renew and all other software more or less followed our internal Paose development process ([31, 15, 13, 32]). Everyone agreed to follow this line for the next version of Renew, which meant to apply- our agile, agent-oriented development process. The software architectural style should be service- and agent-oriented, following the newest Java standards.

4. Modularization of Renew

In [13] we used reference nets to describe our plugin system's general software design, including the management within a plugin system. This concept was implemented in Renew and proved to be very useful over the years. Based on our plugin and nets-within-nets concepts, we can exchange net instances (i.e., Java objects) at runtime without significant issues. In contrast to an exchange, the complete removal of net templates (i.e., Java classes) with our former architecture was impossible without serious efforts due to the missing modularity in Java 8. Frameworks like OSGi were not considered suitable for our purposes as it required a different architecture and had the burden of a larger external framework.

The introduction of JPMS into the core of Java underlines the general problem with external and hence more abstract architecture support, in this case, the new layer of abstraction on top of the package structure in the form of modules. Restrictions of Java 8 and all former versions prevented removing classes at runtime based on the standard Java classloader. Therefore, we

[4] Disadvantages have to be noted in the integration of non-modularized external programs and libraries

introduced our own loading and unloading net templates (as classes). To opt for a standard mechanism was one major motivation for the initial investigation of the new concept, which proved to be very helpful.

We introduced the architectural guideline for the modularized Renew that each plugin corresponds to a module. With this, the plugins are better encapsulated, and the plugin interfaces now become visible as module interfaces and are therefore more explicit. Introducing a module layer for each module also enables the complete unloading (removal from runtime) of plugins. Due to the separation of plugins into Java module layers, each plugin now has its own Java(!) based classloader. Plugins can now be removed at runtime directly using Java language features.

The module layers are organized in a hierarchy formed by the parent relationship between the module layers. Only leaf nodes can be removed from this hierarchy, as Java's garbage collector only removes objects to which references no longer exist. A non-leaf layer is still a parent to another layer and is thus referenced by it. It can first be removed if all its child layers are removed as well, in which case it becomes a leaf node. This ensures for Renew that Plugins can only be removed as long as the architecture guideline (one plugin $\widehat{=}$ one module $\widehat{=}$ one layer) is preserved. So again, language and architecture harmonize well.

5. Modularization Examples

In this section of the paper, we present a brief graphical overview of our module or plugin architecture (see Figure 1). Then some explanations about the architecture are provided.

In Figure 1 three different plugins are shown as boxes. Each box is a plugin of Renew. The boxes with a red border are the publicly available Renew components / plugins. All green boxes are modularized. The white boxes have not been modularized, as they are currently only used in conjunction with Renew 2.6. But due to the architecture a modularization would be possible at any time. A similar diagram exists for our Mulan/Capa framework, which has also been modularized to the same extent in the meantime.

In our modularization process, it became clear early on that we needed to migrate our build environment from Ant to Gradle. Reasons for this were easier dependency management, support for multi-project builds and the builds should become slimmer, simpler and faster. One problem we ran into is that only weak tool support exists to convert Ant to Gradle. Therefore everything had to be developed from scratch, based on the knowledge and information contained in the Ant files. Adversely this allowed new bugs to be made that were not in the Ant files before.

Modularization clarified the dependencies of the plugins and shifted some control towards the Java compiler. Main reason for this is, that for modularized Java modules must be available at compile time. In Java 8 code cannot be really encapsulated. Reflection allowed to spy into every software and reveal internal code. Basic modules protect the modules from been inspection via reflection. For smooth transition of the whole Java eco system the can be weakened in a controlled and explizit manner. While we do not use this option for our Renew code, this might be an interesting option for our multi-agent framework Mulan, where dynamic changes of system structures are more common on the level of code.

One of the main reasons for our adaptation of the architecture is that we wanted to dynamically load or unload plugins even simpler at runtime. For this the module layers could be used well,

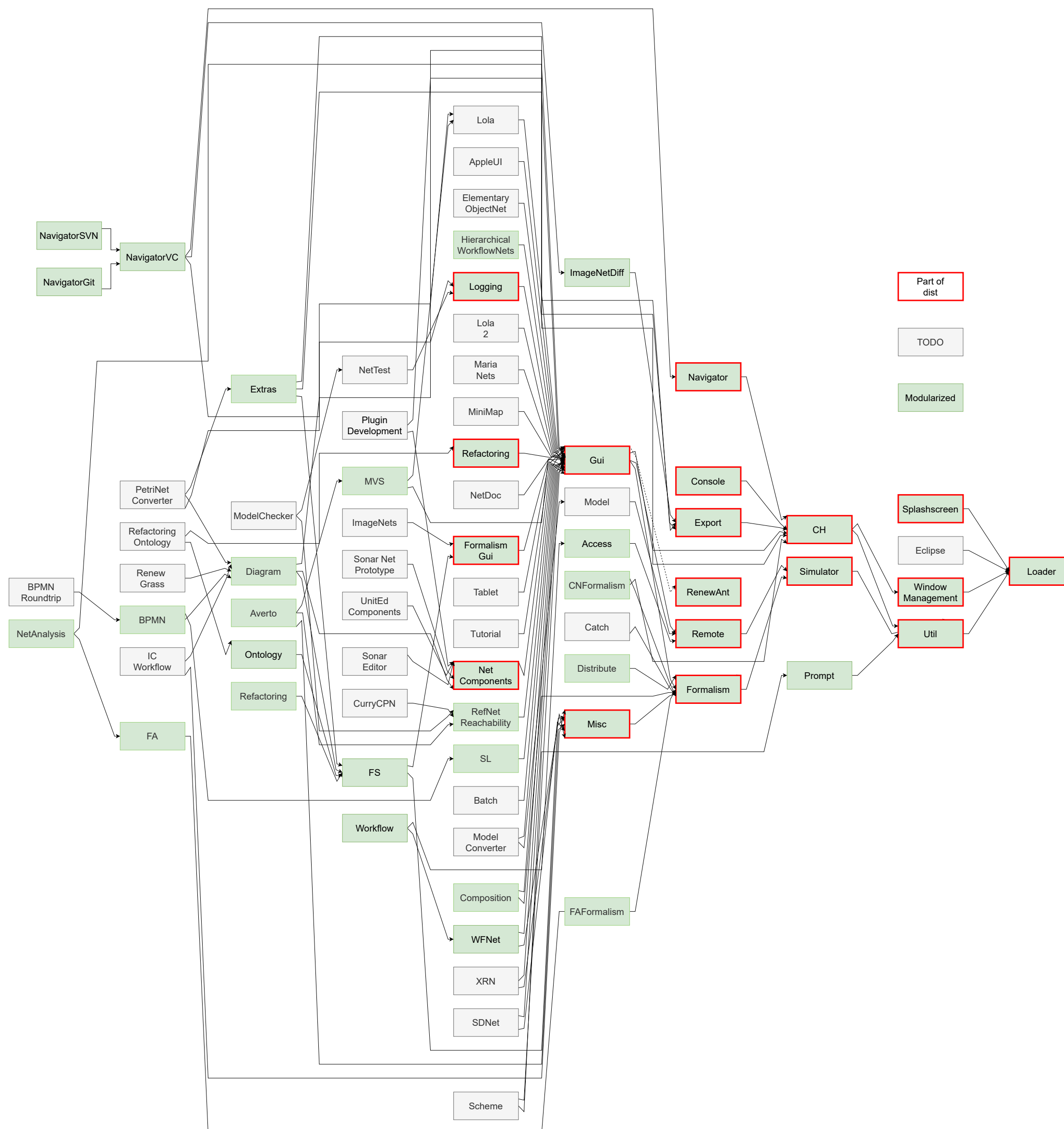

Figure 1: Renew dependencies with modularization progress [22, p.18]

since removing a layer removes the complete code, even from the JVM. Which opens up the possibility that classes can be loaded multiple times at different times.

A major drawback of modularization within our setting is that the module layers together with the module system cause problems if non-modularized parts of the software exist. The class and module paths (see [33]) must be handled separately. Renew uses external libraries that are not modularized. This weakens the approach because it does not guarantee the same

level of encapsulation as the basic modules.

6. Discussion

This contribution's primary goal was to present the results of our transformation process toward a non-trivial, modularized Java system. Experiences like this are hardly documented up to now, and therefore we discuss the pros and cons in the following.

First of all, we reached our goals of the modularization of Renew. Modules improve the encapsulation of our plugins. Their explicit representation of dependencies implies a better architectural style. Tools from the Java ecosystem and development environments support dependency management. All this leads to higher software quality.

In general, our experience is that the higher quality provides a more precise separation of the development of functional features. As we have already used this architecture for about two years, several (minor) software deficiencies that were not seen before became evident and could be handled. Again this improves the quality of our software.

As positive results of the transformation to the JPMS (for Java systems in general and Renew especially) can be named: a) Explicit declaration of the interfaces, b) support of decomposition of the overall system (what was already well prepared by the plugin system), c) reduction of the memory usage by loading only the necessary code (in the form of modules), d) reduction of complexity, since the restricted interfaces between modules, compared to the former package structure, enforce a stronger encapsulation and e) support for the idea of plugins by the language: Our plugin architecture is enhanced by the restrictions of modules (and layers).

Negative aspects that need to be mentioned are: First of all, our restricted usage of modules (module $\widehat{=}$ layer) for the current Renew architecture as it limits the Java language concepts. However, we have already experimented with some modifications. This can be done within one layer when a plugin is split up into several modules, e.g., since we have realized too much functionality inside a single module. Up to now, we have mainly concentrated on the transformation of the plugins into modules. As this was successful, we can now again address more of the inner plugin features and improve the software in that direction. Doing so we will have (one plugin $\widehat{=}$ several modules $\widehat{=}$ one layer) or (several plugins $\widehat{=}$ several modules $\widehat{=}$ one layer).

A second negative aspect of transforming existing software systems is that the build environment must be adapted. For Renew, this holds especially true since the former `Ant` tasks consisted of several files larger than 50KB. Renew's build environment and the accompanying Paose environment had to undergo a severe change, e.g., with the transition from `Ant` to `Gradle`. Nearly 50% of the workload can be attributed to that transition, due to the aforementioned large size of the former `Ant` tasks. Therefore, we recommend moving to a modern build system as early as possible, as overlapping transitions further complicate the process. Build environments, which are mostly software systems, continuously require some maintenance. For Renew, we have reached a relatively stable state by now, so most of the work goes into the Java software system from now on.

A third negative aspect is that related frameworks must be considered. Some frameworks rely on Renew but are parts of our other projects within the Paose context. These related projects

also had to be migrated. Most notably, our Mulan/Capa framework was addressed. In the first version Mulan was treated like any other external plugin system. Integration was realized via the Classloader. It successfully had the Mulan functionality available via the traditional Java path management as it is the default for legacy software. With our largest implementation example, Settler, validation could be provided. [5] As the Mulan framework also uses a plugin architecture, we migrated most of its plugins in the context of two BSc theses and as one out of five sub-projects of our last teaching project in 2021/2022 [25, 24]. [6] Settler and Mulan are running with Renew 2.6 and Renew 4.0, illustrating that we managed the parallel development due to the intended Java option of embedding legacy code. The quality of the former plugin architecture and the provisioning of a smooth transition option by Java as a language can be seen. Due to its size, Mulan has a similar complexity as Renew. We use it as the test environment for the smooth transition of plugin-based systems toward JPMS-based systems. Teaching students this kind of software evolution on practical examples is worth the effort. Most of the Java systems today still need to be migrated.

Overall, one can conclude that migration implies a high workload. However, it is worth the effort when a sufficient starting point is available. The plugin architecture already provides a good system architecture which now gets supported well by the JPMS. With respect to pure new functionality, we have modularized MoMoC and built SignatureCheck from the start in a modularized version. [7] Both versions are larger software systems that illustrate the potential of modularized architectures. Both main developers started with no experience in JPMS. Nevertheless they had no problems building their plugins as modules. As a result, we now have two signature-related plugins (from the user AND software perspective) that have been developed as modules directly.

These examples illustrate that concurrent and distributed teams can contribute to a single system (see release 2.6 and 4.0). Process-oriented and structural teams can be established following our matrix-organization of agent applications to drive Renew even further into this direction, based on the Paose approach. A service-oriented implementation ([34, 35]) is one of our technical solutions for the specific architectural directions while using the agent-oriented perspective as the conceptual background. It is important to note that a smooth transition is possible when the software is organized in packages/plugins. The architecture of the software allows this part to be encapsulated by modules in a straight forward fashion.

A different viewpoint can be taken when teaching software development is taken into account. Students should learn the newest state-of-the-art in their discipline. As Java undergoes a major technological change, it is essential to provide an environment that reflects this. Adding functionality in a structured way can now be done on a higher level of abstraction since the architectural guidelines are now more obvious by the language itself. Dependencies need to be handled more explicitly than before. All this allows us to use our system to teach students and provide them with a commercial-like development environment. The basis for a stricter

[5] We have a Renew binary (Mulan Technology Preview) at https://paose.informatik.uni-hamburg.de/paose/wiki/Downloads available since 2010. There some of Mulan's features can be seen. *The Settler of Catan* game can be played, e.g., against a computer player.

[6] See https://paose.informatik.uni-hamburg.de/paose/wiki/Settler for the current state of the transformation.

[7] See MoMoC: https://paose.informatik.uni-hamburg.de/paose/wiki/MoMoC
SignatureCheck: https://paose.informatik.uni-hamburg.de/paose/wiki/SignatureCheck

separation of plugin specification and implementation and the decomposition of larger modules into several smaller modules has been created.

7. Conclusion

The migration of Renew, its plugins, and its frameworks, e.g. Mulan, is based on the work of many people (special mention should be made of [21, 22, 34, 24, 23] and the work of about 100 participants in our teaching projects and thesis projects the last four years). Besides the Java code, the build environment needed to be modernized, which led to a transition from `Ant` to `Gradle`.

This contribution's primary goal is to present the results of our transformation of a mid-sized software project from Java 8 to the JPMS. Tools tend to become larger and larger due to continuously increasing demands and requirements. Our results are especially relevant for other (Petri net) tools developers who use Java as the main development language. Results presented here follows on from our long-term experiences. They show: **(a)** Architecture is essential for long-term or complex tools, especially when developing with a larger team. **(b)** Migration from Java 8 to the JPMS of newer Java version is possible for larger software tools. **(c)** Migration is also possible within a university context with BSc and MSc students as developers. **(d)** The workload is high, but the quality of the software basis improves. **(e)** The build environment is an important component for the long term maintenance process to ensure that software quality requirements are met. **(f)** Our Paose approach with its special modeling perspective and project management proved to be suited as a the basis for the migration process and for the general team communication, coordination and collaboration. **(g)** Development of larger software is also possible in a university context based on the contributions of BSc and MSc students who can learn working in cooperative software development contexts of non trivial software systems.

The current functional part of Renew 2.6 has been kept and developed concurrently beside our Renew 4.0 release. Presenting the 4.0 release here shows that the software's architectural division into plugins and now modules are viable. From our experience an important aspect of the successful migration was to already start from a plugin based architecture. Renew 4.0 now encompasses more explicit interfaces for the modules due to Java's technical requirements, respectively, the JPMS. Stronger encapsulation is automatically enforced when enriching/restricting the plugins to be a module. Beside being able to apply on the newest Java versions for the maintenance of Renew now, the software quality improved due to clearer encapsulation.

Overall our experiences during this transformation are auspicious in nearly any direction of future teaching, research, and commercial projects based on Renew. We can recommend undertaking the effort either from the beginning of new projects or transforming an already existing project.

For our own Renew and Paose context, we will be able to extend our various prototypes individually without too much interference: We will be able to introduce new features to Renew and our other software components. First areas include distributed simulation in clusters / the cloud and verification of reference nets. Further examples are net-specific tools like

MoMoC or specific Renew and Mulan/Capa based applications, like the Settler game. General feature improvements are e.g. the SignatureCheck approach [26], which introduces trust via certificates on nets and our software modules / plugins. For all these types of software entities, good encapsulation is an important feature.

One major current enhancement efforts will affect the build environment. In a first thesis (see [28]) we experimented with the transition towards Maven repositories. This means that the current central repository is split up into about 80 Maven repositories. Each plugin is embedded within a single Maven repository. By introducing versioning via Maven repositories, we will be able to provide stable and experimental versions of Renew side by side. A further topic is to change our current architecture guideline from (one plugin $\widehat{=}$ one module $\widehat{=}$ one layer) towards (one plugin $\widehat{=}$ one or more modules $\widehat{=}$ one (or eventually more) layer(s)).

References

[1] O. Kummer, F. Wienberg, Renew – the Reference Net Workshop, Available at: http://www.renew.de/, 1999. URL: http://www.renew.de/, release 1.0.

[2] O. Kummer, F. Wienberg, M. Duvigneau, M. Köhler, D. Moldt, H. Rölke, Renew – the Reference Net Workshop, in: E. Veerbeek (Ed.), Tool Demonstrations. Petri Nets (ATPN 2003)., TU Eindhoven, 2003, pp. 99–102.

[3] L. Cabac, M. Haustermann, D. Mosteller, Software development with Petri nets and agents: Approach, frameworks and tool set, Sci. Comput. Program. 157 (2018) 56–70.

[4] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002.

[5] R. Valk, Petri nets as token objects - an introduction to elementary object nets, in: J. Desel, M. Silva (Eds.), Petri nets ATPN 1998, number 1420 in LNCS, Springer, Berlin, 1998, pp. 1–25.

[6] C. Lilienthal, Langlebige Software-Architekturen, volume 3, dpunkt.Verlag, Heidelberg, 2020.

[7] O. Kummer, F. Wienberg, M. Duvigneau, Renew – Architecture Guide, release 1.5 ed., University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, 2001. URL: http://www.renew.de/, available at: http://www.renew.de/.

[8] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, R. Valk, An extensible editor and simulation engine for Petri nets: Renew, in: J. Cortadella, W. Reisig (Eds.), Petri Nets 2004, ICATPN 2004, volume 3099 of *LNCS*, Springer, Berlin, 2004, pp. 484–493.

[9] L. Cabac, M. Haustermann, D. Mosteller, Renew – the reference net workshop, in: D. Moldt, H. Rölke, H. Störrle (Eds.), PNSE'15, Brussels, Belgium, June 22-23, volume 1372 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015, pp. 313–314. URL: http://CEUR-WS.org/Vol-1372.

[10] D. Moldt, Petrinetze als Denkzeug, in: B. Farwer, D. Moldt (Eds.), Object Petri Nets, Processes, and Object Calculi, FBI-HH-B-265/05, Bericht Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2005, pp. 51–70.

[11] D. Moldt, PAOSE: A way to develop distributed software systems based on Petri nets and

agents, in: J. Barjis, U. Ultes-Nitsche, J. C. Augusto (Eds.), MSVVEIS Proceedings, 2006, pp. 1–2.
[12] L. Cabac, Modeling Petri Net-Based Multi-Agent Applications, Dissertation, Universität Hamburg, Department Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010.
[13] L. Cabac, M. Duvigneau, D. Moldt, H. Rölke, Modeling dynamic architectures using nets-within-nets, in: G. Ciardo, P. Darondeau (Eds.), Petri Nets, ICATPN 2005, volume 3536 of *LNCS*, Springer, 2005, pp. 148–167.
[14] M. Duvigneau, Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen, volume 4 of *Agent Technology – Theory and Applications*, Logos Verlag, Berlin, 2010.
[15] H. Rölke, Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen, vol. 2, Agent Technology – Theory and Applications, Logos Verlag, Berlin, 2004.
[16] Oracle, Understanding Java 9 Modules, 2017. URL: https://www.oracle.com/corporate/features/understanding-java-9-modules.html, [Online; accessed 19.05.2022].
[17] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (1972) 1053–1058. URL: https://doi.org/10.1145/361598.361623. doi:10.1145/361598.361623.
[18] D. L. Parnas, A technique for software module specification with examples, Commun. ACM 15 (1972) 330–336. URL: https://doi.org/10.1145/355602.361309. doi:10.1145/355602.361309.
[19] A. Jecan, Java 9 Modularity Revealed, Springer, 2017.
[20] K. Sharan, Erratum to: Java 9 revealed: For early adoption and migration, in: Java 9 Revealed, Springer, 2018, pp. E1–E1.
[21] A. Daschkewitsch, Modularisierung des Renew-Plugin Systems, Masterarbeit, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2019.
[22] J. R. Janneck, Modularizing a Plugin System Using Java Modules: Application to a Medium-Sized Open-Source Project, Masterarbeit, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[23] J. Johnsen, Sicherung der Lauffähigkeit von Renew 4.0 bei einer Umstellung der Java LTS Version, Bachelor thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2022.
[24] J. Kronziel, Modularisieren eines Plugin Systems am Beispiel von Mulan, Bachelor thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[25] F. Beese, Untersuchung von Optionen bei der Erweiterung einer Multiagentenanwendung im Kontext von PAOSE am Beispiel von Settler, Bachelorarbeit, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[26] R. Jürgensen, Untersuchung des Zertifikateeinsatzes im Java-Umfeld – Beispielhafte Diskussion von Zertifikaten für die Open-Source Software Renew, Master thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[27] C. Künemund, Entwicklung eines Plugins zur Berechnung und Visualisierung von Invarianten in P/T-Netzen mit synchronen Kanälen in Renew, Bachelor thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.
[28] A. Heinze, Umstellung des Quellcodes der Software Renew auf eine Multi-Repository-Struktur unter Verwendung der Abhängigkeitsmanagement-Funktionalität von Gradle,

Bachelor thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2022.

[29] Oracle, Project Jigsaw, 2017. URL: https://openjdk.java.net/projects/jigsaw/, [Online; accessed 19.05.2022].

[30] Oracle, Project Jigsaw: JDK modularization, 2020. URL: http://openjdk.java.net/projects/jigsaw/doc/jdk-modularization.html, [Online; accessed 19.05.2022].

[31] D. Moldt, F. Wienberg, Multi-agent-systems based on coloured Petri nets, in: P. Azéma, G. Balbo (Eds.), Petri Nets, ATPN 1997, number 1248 in LNCS, Springer, Berlin, 1997, pp. 82–101.

[32] L. Cabac, M. Duvigneau, D. Moldt, H. Rölke, Applying multi-agent concepts to dynamic plug-in architectures, in: J. Mueller, F. Zambonelli (Eds.), AOSE VI. Revised Selected Papers, volume 3950 of *LNCS*, Springer, Berlin, 2006, pp. 190–204.

[33] Oracle, Setting the classpath, 2018. URL: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html, [Online; accessed 18.05.2022].

[34] L.-O. Clasen, Untersuchung von Java Service Mustern für die Verwendung im modularen Renew, Bachelorarbeit, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2021.

[35] J. H. Röwekamp, submitted PhD thesis: Skalierung von nebenläufigen und verteilten Simulationssystemen für interagierende Agenten, Ph.D. thesis, Universität Hamburg, FB Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2022.