Benefits of Plugin-Based Heuristic Optimization Software Systems

Stefan Wagner[1], Stephan Winkler[2], Erik Pitzer[2], Gabriel Kronberger[2], Andreas Beham[2], Roland Braune[2], Michael Affenzeller[1]

Upper Austrian University of Applied Sciences
[1] Department of Software Engineering
[2] Research Center Hagenberg
Softwarepark 11, 4232 Hagenberg, Austria
{swagner,swinkler,epitzer,gkronber,abeham,
rbraune,maffenze}@heuristiclab.com

Abstract. Plugin-based software systems are the next step of evolution in application development. By supporting fine grained modularity not only on the source code but also on the post-compilation level, plugin frameworks help to handle complexity, simplify application configuration and deployment, and enable users or third parties to easily enhance existing applications with self-developed modules without having access to the whole source code.
In spite of these benefits, plugin-based software systems are seldom found in the area of heuristic optimization. Some reasons for this drawback are discussed, several benefits of a plugin-based heuristic optimization software system are highlighted and some ideas are shown, how a heuristic optimization meta-model as the basis of a thorough plugin infrastructure for heuristic optimization could be defined.

1 Introduction

In software engineering a high degree of modularity is generally considered to be an import software quality aspect ([6], [2]). Having to deal with changing requirements, new feature requests and short development times is the daily life of an application developer. So trying to build software systems that are divided into clearly separated modules communicating over well-defined interfaces is a critical factor of success. On the one hand, highly modular software systems encourage code reuse as several generic modules could be used in other projects. On the other hand, reacting on change and extension requests is easier as only the directly involved parts have to be replaced without needing to change (or even know) the application as a whole.

So when we take a look at the evolution of software development methodologies the trend towards more modularity is obvious. Starting with monolithic programs, going through first procedural and then object-oriented programming and ending with component-oriented or aspect-oriented development of complex software systems, one can see that in each step of development new concepts

were introduced to support clearer structuring and separation of software into modules (procedures, objects, components, aspects, etc.). However, all these decomposition approaches are mostly applied at the source code level. When it comes to compilation, the different modules of an application are often still linked to a single monolithic executable. When changing a module or providing an additional extension or new functionality is required, the changes in source code are clearly separated but the whole application has to be re-compiled or at least re-linked. This fact makes it hard to dynamically extend software systems on demand (especially with new modules that are for example developed by third parties).

As a consequence, plugin-based software systems became very popular, as they are able to overcome this problem. By not only splitting the source code into different modules but compiling these modules into enclosed ready to use software building blocks, the development of a whole application or complex software system is reduced to the task of selecting, combining and distributing the appropriate modules. Due to the support of dynamic location and loading techniques offered in modern application frameworks as for example Java or .NET, the modules do not need to be statically linked during compilation but can be dynamically loaded at runtime. Usually, a specific plugin management mechanism takes care of this task by finding all available parts, loading them on demand and providing mutual interaction by creating objects of specific types implementing required interfaces (contracts).

In the following a short overview of several existing plugin frameworks for different platforms is given (see Section 2). After pointing out general benefits of plugin-based application development in Section 3, the main focus lies on discussing plugin approaches in the light of heuristic optimization software systems (Section 4), as the authors think that especially in this area plugin-based approaches should be very effective but are not implemented in most of existing heuristic optimization frameworks (at least in those known to the authors). Finally a summary and concluding remarks are stated in Section 5.

2 Existing Plugin Frameworks

Especially in Java application development the idea of enhancing software with a plugin mechanism is more and more frequently realized. The OSGi Alliance offers a well developed and standardized architecture for plugin-based and service-oriented software systems called the OSGi Service Platform Specification [8]. The OSGi architecture is structured in several layers built on top of the Java Runtime Environment and providing mechanisms (enhanced class loading, module management, communication and cooperation between modules) required for application building. The application itself (also called bundle) can be situated on any layer depending on the required functionality. The layout of the OSGi architecture is shown graphically in Figure 1. Successful implementations of the OSGi Service Platform Specification are documented in several publications (for example [3]) as well as in open source and commercial software projects.

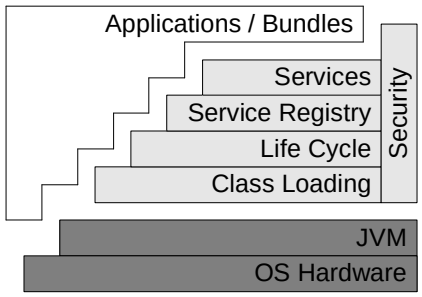

Fig. 1. OSGi Architecture (taken from [7])

Beside the OSGi Service Platform, also the well known Eclipse development platform and in particular the Eclipse Rich Client Platform (Eclipse RCP) intensively use the concept of plugins ([1], [12], [5]). Additionally, other popular applications can be found that follow similar ideas like the Mozilla web browser [11] or the NetBeans environment.

However, the idea of plugin oriented application development is not restricted to the Java platform at all. Realizing a powerful and flexible plugin infrastructure is possible with any kind of application development environment offering basic features such as dynamic loading and reflection. Several publications show how to implement plugin-based applications using the .NET Common Language Runtime ([9], [10], [4], [15]) and underpin the worth of this approach. Furthermore, the Razor project of Mark Belles also has to be mentioned, which goes beyond scientific analysis and prototypical implementation and focuses on developing a fully functional plugin framework based on the .NET framework.

3 General Benefits of Plugin-Based Application Development

As already stated in the introduction, a high degree of modularity is an important factor for building high quality software systems. Apart from the typical improvements of testability, reusability, readability, and maintainability that are already supported by classical modularization techniques such as object-oriented modeling, plugin-based software development further simplifies the following aspects discussed in the next subsections:

Complexity

Thinking of modularity not only at the source code level but also on the application level helps to handle the high degree of complexity large software systems have to deal with today. By separating an application into several plugins, where each plugin is responsible for a well-defined task, the complexity and size of a

single module is drastically reduced. Furthermore, when compiling a plugin into an executable unit, the plugin is effectively sealed and can be shared among different developers without having the need (or even the chance) to take a look at the original source code. As communication and interaction with the plugin is done using fixed interfaces, a developer does not have to be bothered with trying to understand other developers' source codes. Furthermore, the development of drivers and stubs to test each module independently is also simplified. However, to be honest complexity does not vanish. Instead it is transferred from the programming level up to the modeling level, as the final application has to be assembled by combining different small and simple plugins. Thereby the problem of glue code arises that is required to link and combine these modules. How to handle incompatibilities between different plugins and how to prevent the complexity from being shifted from the application modules into the glue code holding the whole application together is still an active field of research. For example, at this point generative software development comes into play as a concept for automatically generating glue code from system models instead of manually putting the required parts together.

Configuration, Customization, Deployment

In plugin-based software systems the main application's logic is reduced to the mere plugin hosting and management mechanism and does not offer any application specific functionality. The set of features of such an application is defined by different plugins loaded and managed by the plugin infrastructure. So it is much easier to customize a plugin-based software system to the specific requirements in a customer's scenario just by adding necessary and removing irrelevant parts. Besides, application deployment is simplified as functionality to deploy new or to update outdated plugins from update locations (i.e. servers) over a network can be realized easily.

Expendability

Finally, also users benefit from a plugin-based infrastructure, as they are equipped with a comprehensive and yet easy to use application programmer's interface (API). Extending an application with new individual features without understanding or even having access to the full source code of an application is plain sailing (just implement new plugins and register them in the application). This also offers the possibility for third parties to develop and sell extensions to an existing application independently.

4 Plugin-Based Heuristic Optimization Software Systems

Although the benefits of plugin-based software systems are well known in the software engineering community and several open source and commercial projects (especially related to development environments) demonstrate its use, in the area

of heuristic optimization software systems such approaches are not commonly used [13] yet. So what is the reason that developers of heuristic algorithms lag behind recent improvements of software engineering?

In the opinion of the authors there are several answers to this question. On the one hand it is a matter of fact that most heuristic optimization experts are originating from other research areas (mathematics, mechatronics, economy, logistics, etc.) and are therefore not so familiar with advanced application development techniques in contrast to software engineers. However, it should be quite easy to overcome this drawback by (prototypically) implementing a plugin-based heuristic optimization software system and showing its benefits. For example, the HeuristicLab optimization environment [14] tries to fulfill this demand. More details can be found on the HeuristicLab homepage[1].

On the other hand the more severe reason is the lack of a common model unifying heuristic optimization algorithms. Such a heuristic meta-model is required as the basis for developing a practicable plugin infrastructure for this domain. However, due to the vast amount of different algorithm variants as for example Evolutionary Algorithms (Genetic Algorithms, Genetic Programming, Evolution Strategies, or Evolutionary Programming), other population-based algorithms (Ant Colony Optimization, Particle Swarm Optimization, e.g.) or neighborhood-based optimization techniques (Simulated Annealing, Tabu Search, etc.) this is not a straightforward task.

So let us consider all those different kinds of optimization techniques. What do all of them have in common? In fact, heuristic optimization is always about manipulating promising solution candidates in order to reach high quality regions of the search space in reasonable time. This process can be split into a few main operations:

- Create one or more initial solution candidates (using a specific solution representation)
- Evaluate solutions using a given quality measure
- Manipulate a solution
- Create new solutions out of existing ones
- Regroup solutions

So we have three different aspects interacting with each other: the solution encoding, the evaluation function and a set of operators working on single solutions or sets of solutions. Standardizing this interaction can be done by defining a small set of fixed interfaces: a solution can be abstracted to an object exposing one (single objective optimization) ore more (multiobjective optimization) quality values and containing data in the form of an instance of a solution representation (for example a multidimensional vector of real values, a permutation, or a tree-like data structure). A solution processing operator is simply a method taking a single solution (for example evaluation) or a set of solutions (for example selection) and returning a single solution or multiple solutions again. One possible modeling of these interfaces is shown in Figure 2.

[1] `http://www.heuristiclab.com`

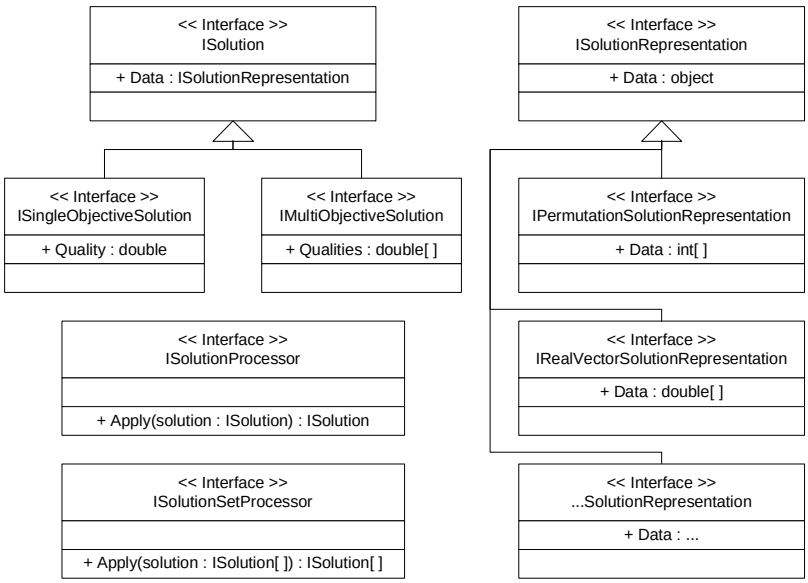

Fig. 2. Basic Heuristic Optimization Interfaces

Based upon this abstraction of heuristic optimization, a plugin-based approach can be realized. Interaction between the plugins is restricted to processing objects represented by the given interfaces leading to a high degree of exchangeability. A plugin itself is only responsible for a single and simple processing operation. When developing heuristic optimization software systems, additional benefits arise from applying this paradigm that go beyond the general aspects already discussed in Section 3.

Algorithm Development

By providing a set of plugins, each of which realizes a specific solution representation or operation, the process of developing new heuristic algorithms is revolutionized. Algorithms do not need to be programmed anymore but can be created by combining different plugins. This approach has a huge advantage: By providing a graphical user interface for combining plugins, no programming or software engineering skills are necessary for this process at all. As a consequence, algorithms can be modified, tuned or developed by experts of different fields (logistics, production, finance, or data analysis, e.g.) with less or no knowledge in the field of application development. This transfers development from software engineering to the concrete application domains profiting from the profound knowledge of the users and eliminating the communication overhead between optimization users and software engineers.

Experimental Evaluation

Secondly, the cumbersome and time-consuming task of evaluating different algorithmic approaches for a given optimization problem is simplified. New ideas can be prototyped rapidly (by simply adding new operators) and immediately evaluated within the application by comparing their results with existing classical algorithm configurations.

Combination of Different Optimization Paradigms

Finally, a plugin-based heuristic optimization software system helps to bridge the gap between different optimization paradigms like evolutionary algorithms, local neighborhood-based algorithms, swarm-based algorithms, or even exact mathematical optimization methods in general. Different concepts can be easily combined just by linking the relevant plugins. So for example realizing genetic algorithms equipped with a local optimization of solution candidates (memetic algorithms) is quite a walk-over. This will lead to algorithms that are individually tuned for specific applications bringing out the best of each paradigm.

5 Conclusion and Future Perspectives

In this paper the authors have highlighted the importance of modularity in complex software systems and discussed plugin-based application development as the next evolutionary step of programming paradigms. Several examples of plugin frameworks for modern development platforms such as Java or .NET show the benefits of this approach. These benefits are a better handling of complexity, a simplified configuration and deployment procedure and the ability to extend existing applications with new own or third-party modules without requiring access to the whole source code of an application.

However, when it comes to heuristic optimization software systems, modularity is mainly realized on the source code level alone by using classical decomposition techniques such as object-oriented modeling. Plugin-based approaches are seldom implemented. In the opinion of the authors this is a severe shortcoming of many systems in this domain. Having a plugin-based heuristic software system at hand would revolutionize the process of algorithm development as new heuristic algorithms do not have to be programmed anymore but can be modeled and assembled using a graphical user interface. As a consequence, no programming skills are required anymore, opening the field of algorithm engineering for many other users originating from different research areas such as logistics, production, finance, medicine, biology, or data analysis.

As one reason for the lack of plugin-based heuristic optimization software systems, the authors have identified the difficult task of developing a common meta-model for heuristic algorithms which is required as a basis for developing a practical plugin infrastructure. Some ideas how such a highly abstract meta-model might be defined are given in Section 4 and a prototypical implementation is available in form of the HeuristicLab optimization environment [14].

Future directions of research will focus on the one hand on further development of the plugin concept of the HeuristicLab environment. On the other hand a comprehensive set of heuristic optimization plugins would pave the way for realizing automation concepts for heuristic algorithm development. For Example, using Genetic Programming techniques to automatically evolve whole heuristic algorithms (i.e. different plugin combinations) in order to build new high-quality optimization algorithms specifically tuned to a given application is a very fascinating idea. Furthermore, fighting the problem of increased runtime (due to the overhead of complex plugin interactions) by using generative programming approaches is another research direction the authors will continue to work on.

References

1. Beck, K., Gamma, E: Contributing to Eclipse. Addison-Wesley (2003)
2. Cox, B.: Planning the software industrial revolution. IEEE Software 7(6) (1990)
3. Hall, R.S., Cervantes, H.: An OSGi Implementation and Experience Report. In: Consumer Communications and Networking Conference (2004)
4. Holm, C., Krüger, M., Spuida, B.: Dissecting a C# Application – Inside SharpDevelop. Apress (2003)
5. McAffer, J., Lemieux, J.-M.: Eclipse Rich Client Platform: Designing, Coding, and Packing Java Applications. Addison-Wesley (2005)
6. McIllroy, M.: Mass produced software components. Proceedings of the Nato Software Engineering Conference (1968) 138–155
7. OSGi Alliance: About the OSGi Service Platform. Technical Report, OSGi Alliance (http://www.osgi.org) (2005)
8. OSGi Alliance: OSGi Service Platform Specification (Release 4). Technical Report, OSGi Alliance (http://www.osgi.org) (2006)
9. Osherove, R.: Creating a Plug-in Framework. Technical Report, Microsoft Developer Network (2003)
10. Osherove, R.: Search Dynamically for Plug-ins. Technical Report, Microsoft Developer Network (2003)
11. Shaver, M., Ang, M.: Inside the Lizard: A look at the Mozilla Technology and Architecture. Technical Report, http://www.mozilla.org (2000)
12. Shaver, M., Ang, M.: Eclipse Platform Technical Overview. Technical Report, Object Technology International (http://www.eclipse.org) (2003)
13. Voss, S., Woodruff, D.: Optimization Software Class Libraries. Kluwer Academic Publishers (2002)
14. Wagner, S., Affenzeller, M.: HeuristicLab: A Generic and Extensible Optimization Environment. In Ribeiro, B., Albrecht, R. F., Dobnikar, A., Pearson, D. W., Steele, N. C. (eds.): Adaptive and Natural Computing Algorithms. Springer Computer Science, Springer Berlin Heidelberg New York (2005) 538–541
15. Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .net Platform. Proceedings of the Joint Modular Languages Conference 2006. Lecture Notes in Computer Science, Vol. 4228. Springer Berlin Heidelberg New York (2006) 287–305