Introduction to Graph-Oriented Programming

Olivier Rey[1 [0000−0003−4462−3712]]

GraphApps, France
rey.olivier@gmail.com – orey.github.io/papers

Abstract. Graph-oriented programming is a new programming paradigm that defines a graph-oriented way to build enterprise software, using directed attributed graph databases as backend. Graph-oriented programming is inspired by object-oriented programming, functional programming, design by contract, rule-based programming and the semantic web. It integrates all those programming paradigms consistently. Graph-oriented programming enables software developers to build enterprise software that does not generate technical debt. Its use is particularly adapted to enterprise software managing very complex data structures, evolving regulations and/or high numbers of business rules.

Couplings in enterprise software

The way the software industry currently builds enterprise software generates a lot of "structural and temporal couplings". Structural coupling occurs when software and, in particular, data structures, are implemented such that artificial dependencies are generated. A dependency is artificial if it occurs in the implementation but not in the underlying semantic concepts. Temporal couplings are artificial dependencies generated by holding several versions of business rules in the same program, those rules being applicable to data that are stored in the last version of the data structures.

Those couplings are at the very core of what is commonly called "technical debt". This debt generates over-costs each time a software evolves. Generally, the requirements change, the software is partially redesigned to accommodate the modification, the data structures evolve, the existing data must be migrated, and all programs must be non-regressed. In order to implement a small modification in an enterprise software, a change in regulation for instance, overcoming the technical debt may represent up to 90-95% of the total workload [5,6].

The software industry has, for a long time, identified the costs associated to technical debts, and in particular those costs seem to grow exponentially with time [5]. That means that the productivity of any maintenance team of fixed size will constantly decrease throughout the evolution process. In order to address this core issue of enterprise software, a lot of engineering-oriented workarounds can be found: design patterns that are supposed to enhance software extensibility [1], software architecture practices that define modules and layers inside an enterprise software [4,2], or best practices for software refactoring to reduce the costs of the refactoring phase itself [3]. However, every software vendor knows that the core problem of the technical debt has not been solved.

Graph-oriented programming

Graph-oriented programming is meant as an alternative programming paradigm not collecting technical debts. This paradigm is based on three concepts: (1) Using directed attributed graph databases to store the business entities without storing their relationships in the entities themselves, i.e. there are no foreign keys; (2) Designing programs so that the knowledge about relationships between entities (business nodes) is captured in functional code located "outside" of the nodes, encapsulated in graph transformation rules; (3) Using best practices in graph transformation design to guarantee a minimal or even no generation of technical debt. This programming paradigm can be applied using an object-oriented or functional programming language.

The expected advantages of using graph-oriented programming are multiple: reusability of software is increased due to less software dependencies; multiple views of the same data can be implemented in the same application; multiple versions of data structures and business rules can cohabit, meaning that the software and the data can be timelined; software maintenance can be done by adding new software rather than by modifying existing software.

At last, graph-oriented programming enables to build a different kind of enterprise software that proposes, through the use of a graph-oriented navigation, a new user experience, closer to our mental way of representing things.

The approach taken at GraphApps

At GraphApps, we developed a graph-oriented designer in Eclipse whose purpose is to model node and relationship types, as they occur in business applications, and to group them in semantic domains. Code generators, coupled to the designer, generate parameterized web pages proposing a default view of defined types of business entities. For each semantic domain, an independent `jar` file is generated. In addition, we developed a graph-oriented web framework, which loads the `jar` files and enables us to integrate them in the graphical web framework. All domains can be integrated without introducing any new code dependency. Each domain may include custom code, in order to implement graph transformations, web page modifications, or new pages. Moreover, the framework proposes reusable components that offer generic reusable mechanisms such as business node classification (every business node can be referenced in a tree of shared folders), business node timelines, navigation history, personalized links between business nodes, or alternate navigation.

Those tools support a quick prototyping of large and complex applications, the implementation of time-based business rules, and the cooperative work of several teams collaborating to the same core model. The way the code is organized enables us to modify the behavior of the core system, without having to modify existing code, migrating data, or performing non-regressing testing. We have used this set of tools for many business prototypes and we are using it currently to build a complete innovative aerospace maintenance information system (composed by many semantic domains) from scratch.

Conclusion

The paradigm of graph-oriented programming enables us to build a new generation of enterprise software that will be much easier to maintain and that can address the high complexity of business entity structures and their life cycles, as well as time-sensitive business rules. This paradigm may be used to rewrite a huge number of enterprise software in the coming decades in order to decrease drastically the maintenance costs, to enhance the capability of personalization of the software and to create new user experiences by proposing more intuitive ways to navigate within the software.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements Of Reusable Object Oriented Software*. Addison-Wesley, 1994.
2. F. Buschmann. *POSA Volume 1 - A System of Patterns*. Wiley, 1996.
3. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
4. D. Alur. *Core J2EE Patterns*. Prentice-Hall, 2nd edition, 2003.
5. A. Nugroho, V. Joost, and K. Tobias. *An empirical model of technical debt and interest*. Proceedings of the 2nd Workshop on Managing Technical Debt. ACM, 2011.
6. Z. Li,P. Avgeriou,and P. Liang. *A systematic mapping study on technical debt and its management*. Journal of Systems and Software, 101, 193-220. 2015