# Introduction to the Graph-Oriented Programming Paradigm

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 on the backend side.
Graph-oriented programming is inspired by object-oriented programming, functional programming, design by contract, rule-based programming and semantic web. It integrates all those programming paradigm consistently. Graph-oriented programming enables to build enterprise software that does not generate technical debt. Its use is particularly adapted for enterprise software that must manage high complexity of 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 is the phenomenon of implementing software code and data structures that generate artificial dependencies compared to the underlying semantic concepts. Temporal couplings are artificial dependencies generated by ~~the habit of~~ maintaining the business rules and their evolutions in the same programs, while rules and data structures may change over time. Original rules may not be applicable to changed data structures anymore.

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 will 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 tend towards zero. In order to address this core issue of enterprise software, a lot of engineering-oriented work-arounds 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 ~~that enable~~ 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 proposing an alternative programming paradigm not to collect 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 (no foreign keys); (2) Designing programs so that the knowledge about relationships between entities (business nodes) is restricted to functional code located "outside" of the nodes, encapsulated in graph transformations; (3) Using graph transformation design best practices ~~in order~~ to guarantee a minimal or even zero 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 addition of new software rather than by existing software modification. At last, graph-oriented programming enables to think differently about the user interface and propose a new user experience that is closer to our mental way of representing things.

### The approach taken in GraphApps

At GraphApps, we developed a graph-oriented designer in Eclipse whose purpose is to model node and relationship types and to group them in semantic domains. Code generators, coupled to the designer, generate parameterized web pages proposing a default view of a root node type. Each semantic domain generates an independent `jar` file. We developed ~~also~~ a graph-oriented web framework, which loads the `jar` files and enables to integrate them in the graphical web framework. All domains can be integrated without the code of the domains to depend on each other. Each domain may include custom code, in order to implement graph transformations, web page modifications or new pages. The framework proposes ~~also~~ generic components that can offer generic reusable mechanisms such as business node classification, history, user links or alternate navigation.

Those tools enable a quick prototyping of large and complex applications, the implementation of time-based business rules and the cooperative works of several teams collaborating to the same core model. The way the code is organized enables to modify the behaviors of the core system, without having to modify existing code, ~~or~~ migrating data, or performing non-regressing testing.

We 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 graph-oriented programming paradigm 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 life cycle, and 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