Introduction to graph-oriented programming

The role of graph transformations in solving the technical debt problem

Olivier Rey

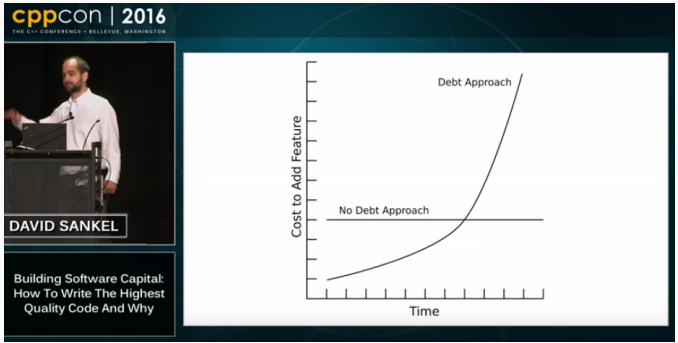
June 26 2018

GraphApps for STAF/ICGT 2018

Table of contents

Technical debt in enterprise software

The famous technical debt



Since decades, the software engineering world knows this exponential function: The cost of the function point, in a software, tends quickly to infinity with time

Proposed definition of the technical debt

The difference between the cost of implementation of a software module M, developed alone, and the costs of development of the same module M developed in the context of an existing software E, all costs included (testing and data migration for instance)

Couplings in enterprise software

The way the software industry currently builds the enterprise software generates a lot of "couplings"

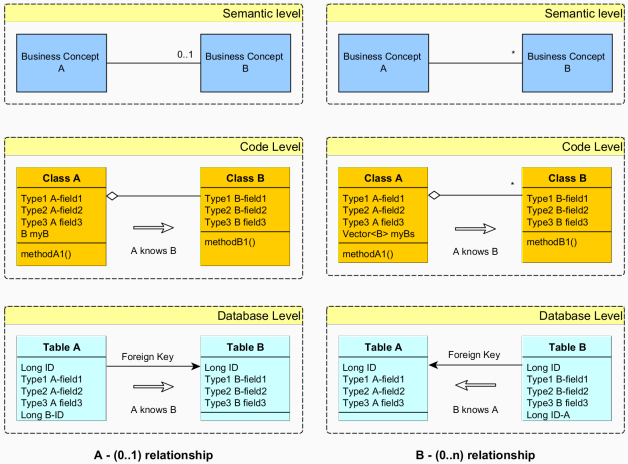
- Inside the code
- Inside the data
- Between the code and the data

We divided those couplings into two categories:

- Structural couplings
- Temporal couplings

We will explore briefly both of them in the context of object-oriented programming and RDBMS

First category: Structural couplings



A - (0..1) relationship **B - (0..n) relationship**

The "encoding" of the semantic knowledge is largely sub-optimal

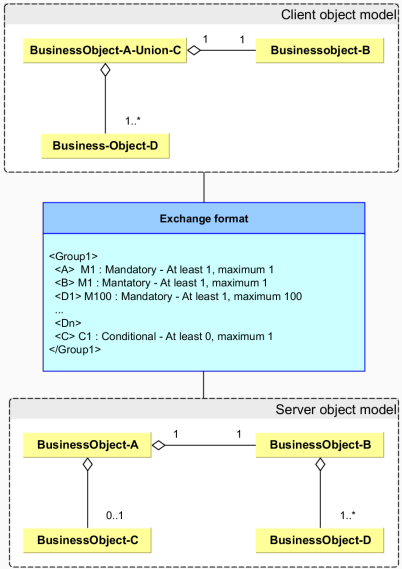
- In the code
- In the data

The result is a double coupling for the very common (0..n) relationship

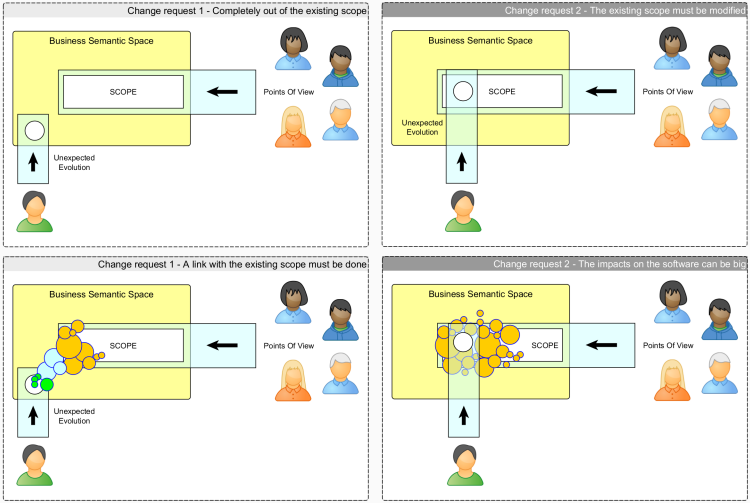
No unique way of representing things

Common software practices generate structural couplings whereas there is no unique way of representing things, and so, our representation may have to change with time

On the right: 3 ways of representing the same reality in a classic client/server exchange



There are always many scope changes



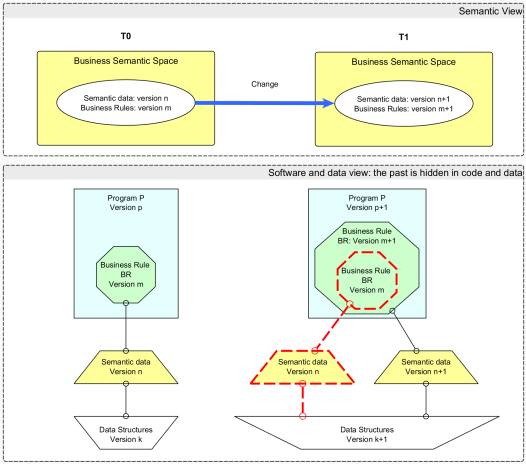
Scope changes

- Cannot really be anticipated
- Are more or less difficult to accommodate in an existing software

Second category: Temporal couplings

Temporal couplings result from the very common way of doing software evolution

- Upgrade an existing code (BR version `m+1` "contains" BR version `m`)
- Upgrade the database (version `k+1`) to fit the new version of BR (`m+1`)
- Migrate old data to the new format (semantic data version `n`)



The current code embeds hidden past business rules that apply to past data attached to hidden data structured encoded in the last version of data structures

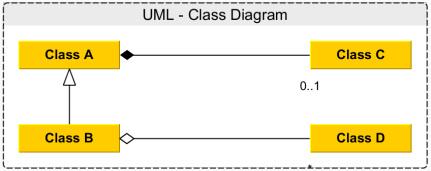
Addressing the technical debt

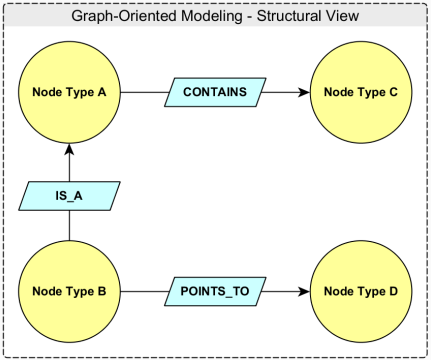
For decades, the software industry found "work-arounds" to address the technical debt problem, without really questioning the true nature of the technical debt

- The object-oriented design patterns
  - Promise: being able to anticipate extensibility and reusability
  - Reality: it is not really possible to anticipate extensibility
- Software architecture
  - Promise: software component reuse, separation of concerns, dependency minimization, etc.
  - Reality: good for technical bricks but does not solve business code technical debt
- Refactoring methods
  - Promise: being able to make the software evolve longer
  - Reality: costly, risky, *a posteriori* approach

Graph-oriented programming: Structural aspects

Node types and relationships types





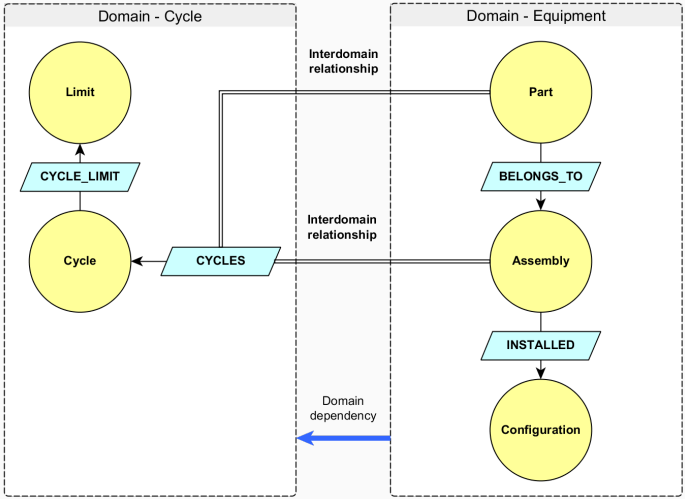
If we suppose we have two kinds of artifacts, node types and relationship types (that become a first class citizen), the object-oriented approach can be "extended"

Important points:

- Node and relationship types have attributes
- Relationships are named
- Node types do not "embed" the relationship knowledge, i.e. they do not know the graph topology

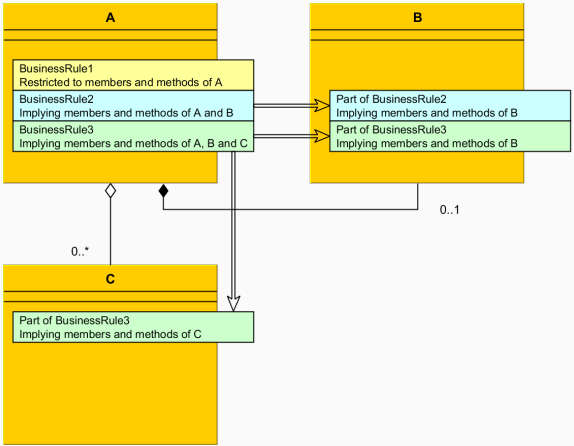
Grouping artifacts per semantic domain

We can group artifacts per semantic domains. This practice will be very useful in the software, as we will see it.
This exhibits the role of certain inter-domain relationship types that can appear as creating domain dependencies



Graph-oriented programming: Dynamic aspects

Business rules implementation in object-oriented programming



In object-oriented programming, business rules often originate in a class and are spread among several other classes

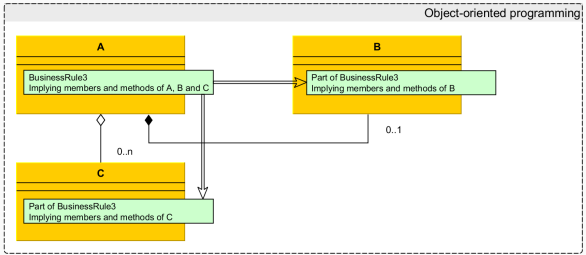
In this example, `BusinessRule3` is spread among 3 classes and the part located in class `A` knows the graph topology and `B` and `C` methods

Removing topology dependency with a functional approach on the subgraph

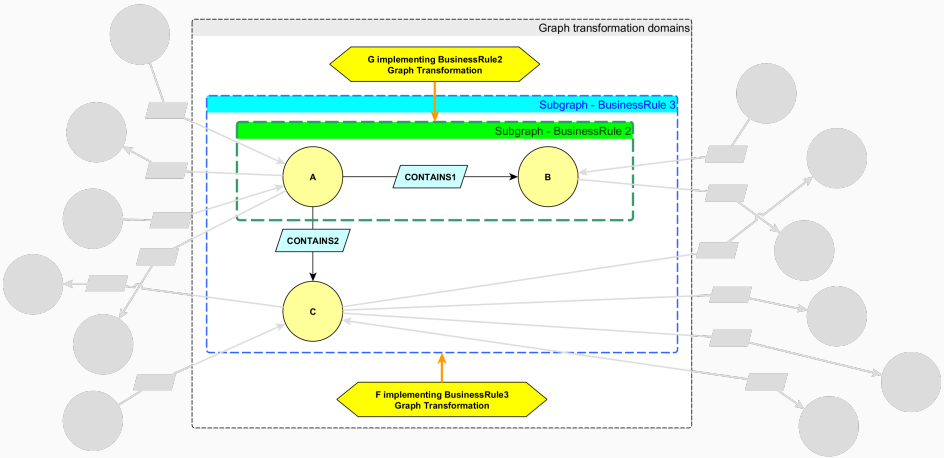
In the bottom figure, we implemented `BusinessRule3` differently

- We created an external function `F` that only calls node types methods which knowledge is limited to their very node
- `F` knows how to navigate the graph through `CONTAINS1` and `CONTAINS2` relationships
- In a way, `F` orchestrates the graph to implement `BusinessRule3`

Considering F can alter the graph, `F` is a graph transformation

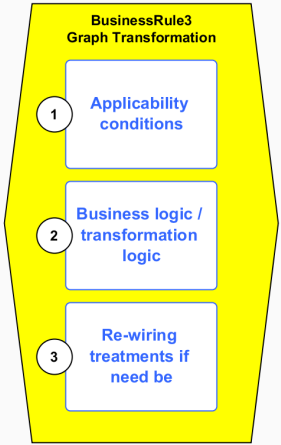


Nodes, graph and sub-graphs



The domain of the graph transformation is the origin subgraph

Structure of a graph transformation



A graph transformation should be defined in 3 steps

1. The first step is to check applicability conditions, in particular, the topology of the provided sub-graph
   - This is crucial because the graph transformation should only the strict topology required for its treatment
   - In case the applicability conditions are not met, the graph transformation should declare itself `NOT-APPLICABLE`
2. The business logic will manipulate the graph with or without side-effects
3. The last part of the treatment should be to rewire the nodes of the subgraph if need be

About graph transformations

Graph transformations are defined for all subgraphs of the full graph and can have a unified API

- For $g \in sub(G), F : g \to F(g)$ is a subgraph of the transformed graph G or $F(g) = \texttt{NOT-APPLICABLE}$
- Graph transformations are always invokable
- This property is very important in a maintenance and evolution perspective

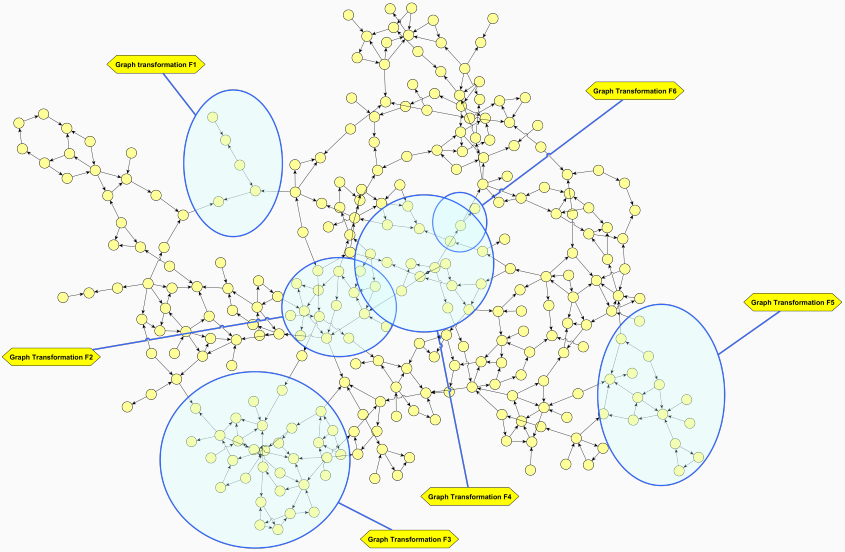
Graph transformations can be composed, which enables a certain level of reusability

- Let F_1 and F_2 be graph transformations, $F_1 \circ F_2$ and $F_1 \circ F_2$ are valid graph transformations

A (just a bit) new programming paradigm

If we consider the graph as being a new kind of “object”, graph transformations appear as the “methods of the graph”

In a functional programming perspective, graph transformations are a special functions applicable to graphs



Managing evolutions

Maintenance and evolutions in enterprise software

Despite the fact that most of the R&D efforts are focused on new projects and new technology, most of the IT budgets worldwide are spent in the maintenance and evolution phase

- Average figures that we can find in the industry states that the maintenance and evolution phase costs around 75% of the overall project for around 25% for the project phase

So software evolution costs are not "just a pain point" in software engineering, they indeed are *the major problem of the industry*

We will analyze how the use of graph-oriented programming can change the perspective

- For the structural couplings
- For the temporal couplings

Structural evolutions

Modifications in graph-oriented programming impact only *what should be impacted*

Programming model	OOP	Graph-oriented programming
Evolutions of semantic model (concepts and links) inside the code	The impacts are generally important if the structure of the model evolves. It requires code refactoring, and quite often full non regression testing on top of the evolution testing	The modified node and relationship type have to be retested. However, if new classes and new relationships are added, there is no need to retest the unmodified artifacts (here comes the importance of the domains)
Evolutions of semantic model (concepts and links) inside the database	Database refactoring, data migration and data retesting must generally be done	Only the touched entities must be modified. In some database, when there is no schema, there is nothing to do at all

Dynamic evolution implying a topological change

We take the case of the evolution of a business rule encoded as a graph transformation

Rule 1: If the topological applicability conditions change, then the graph transformation should be forked

- In other terms, an evolution of topological conditions creates a new rule that cannot be considered as an "evolution" of the previous rule
- Once the modification is done, the system contains two active graph transformation with two different applicability conditions (probably on separate sets of data)
- Note: changing attributes in node or relationship types is a topological change

Dynamic evolution not implying a topological change

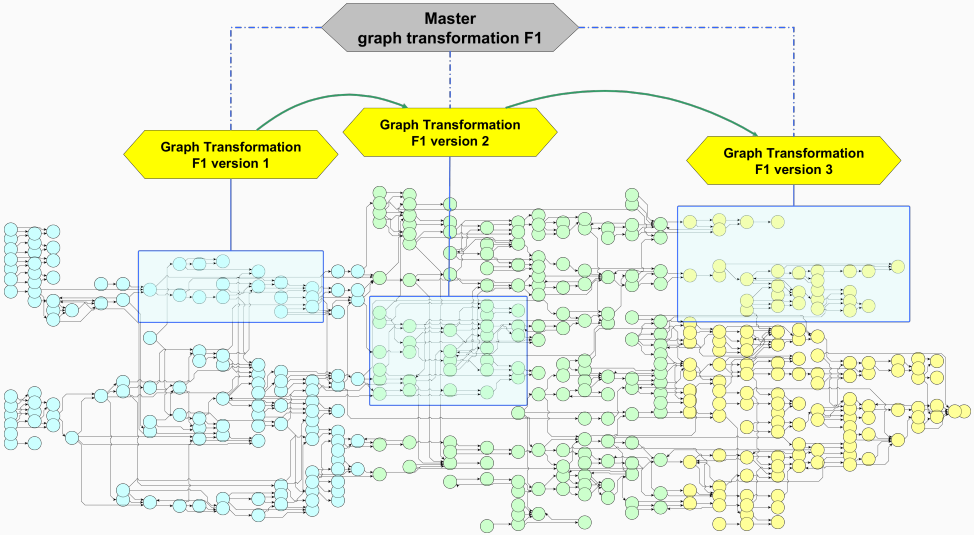
In that case, no structure change in the database but the business rule evolves

Rule 2: If all data in the database are submitted to the new version of the business rule, the graph transformation should be modified
If not, it depends

- The graph transformation can manage several cases
- It can be decided to fork the graph transformation if, for instance, the nature of the rule really changed and/or the new rule clearly applies from a certain moment on newly created nodes and relationships

In that case, there are still design choices to make!

Dynamic evolution rules forecast a new kind of software and patterns



Graph-oriented programmings solves superfluous structural couplings

Graph transformations appear as the smallest code entity that encapsulate (at least) the topological knowledge associated to a business rule

Programming model	**OOP**	**Graph-oriented programming**
Code level superfluous structural couplings	–Inside aggregations –Inside methods	–None at the node and relationship type levels (independence of artifacts the ones from the others) –Strictly necessary at the graph transformation level (minimal topology knowledge restricted to the business rule itself)
Database level superfluous structural couplings	Inside the RDBMS	None in an attributed directed graph database

Graph-oriented programmings solves superfluous temporal couplings

Graph transformations "fork evolution approach" appears to enable "timelining" the application:

- Timelining data but also timelining data structures and enable to keep data in the original structure (if relevant)
- Timelining business rules and enable graph transformation time sensitivity, protected by the "always invokable" graph transformation principle
- Restricting dependencies to their necessary semantic core and not allow implementation of data storage concerns generate superfluous couplings that will result into a divergent generation of technical debt
- Enable new patterns to emerge and new design best practices in a world where, most of the time, refactoring, non regression testing and data migration is no more required

Implementation aspects

Minimum set of requirements for a graph-oriented implementation

Requirement	Sample in OO language	Sample in functional language
Node type programming representation	Class	List/struct with typed members
Relationship type programming representation	Class with a source node ID attribute and a target node ID attribute	List/struct with typed members with two members for the source and target node IDs
Graph programming representation	There must be a class `Graph` enabling graph manipulation.	There must be a structure representing the graph and enabling graph manipulation.
Graph manipulation API	Methods on the `Graph` class	Functions acting on the graph structure

Graph-oriented programming can be implemented in existing programming languages

Minimum set of requirements for a graph manipulation API

The graph transformation must use a graph manipulation library to work on the graph itself

#	Category	Requirement Description
01	Basic	Create graph from select query
02	Basic	Get graph root node (when applicable)
03	Basic	Add nodes and relationships inside the graph
04	Basic	Delete a node or a relationship inside the graph
05	Basic	Modify a node or a relationship inside the graph
06	Basic	Get nodes and relationships from the graph to access them in a object-oriented or functional way
07	Advanced	Assert a topology condition on the graph (returning `true` or `false`)
08	Advanced	Search for nodes and relationships with some criteria (such as per attribute value)
09	Advanced	Merge two different graphs
10	Advanced	Persist the graph
11	Advanced	Match a pattern in the graph
12	Advanced	Perform some other complex operations on graphs (for instance, for two graphs G_1 and G_2, create the graph $G_3 = G_1 \cap G_2$)

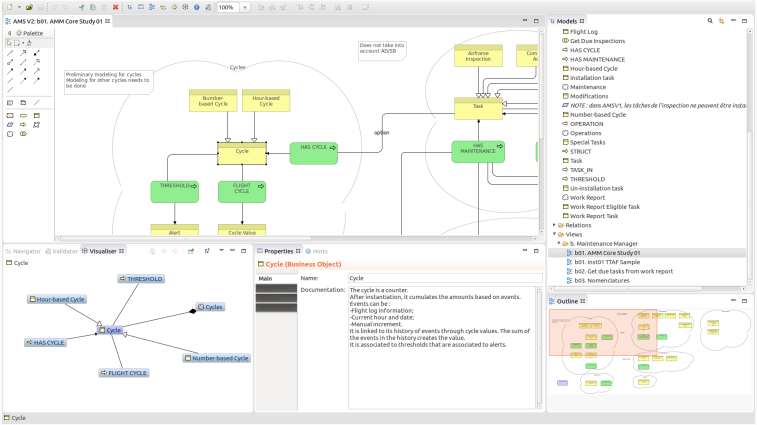
More

High level overview of GraphApps tools

GraphApps designer, code generators and web framework

1. GraphApps designer
   - Based on Eclipse
   - Define a meta model that enables to work on a certain semantic domain
   - Artifacts are tagged in order to enrich the code generation
   - Conceptually, GraphApps designer enables to model complex businesses in a multi-view environment at a structural level (not instance-based programming)
2. Code generators
   - Take in input the designer model plus other configuration parameters and generate a web framework "plugin" per semantic domain
   - The code generator enable to use the glue-ing mechanism of the web framework to limit the plugin dependencies
3. GraphApps web framework
   - Complete multi-layer web framework proposing a graph-oriented framework approach and using completely the "non-adhesive" properties of "plugins" and "modules" More

GraphApps Designer

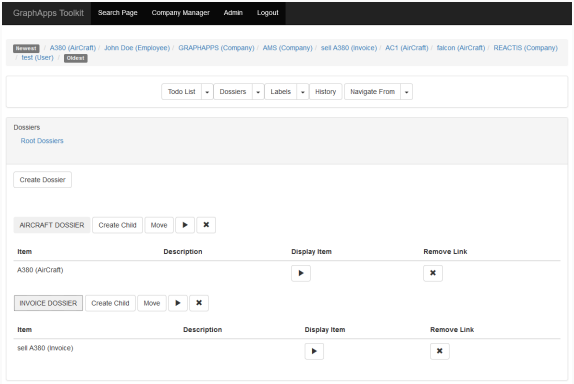


- The Designer proposes a very simple metamodel
- The Designer frame is composed of various sections
- The modeling is performed at a structural level (similar to UML class diagrams) and at an instance level (similar to UML collaboration/sequence diagrams)
- Many views can reference the same artifacts (enabling cross-check)
- The “graph view” enables to work with the full graph model (union of all views)

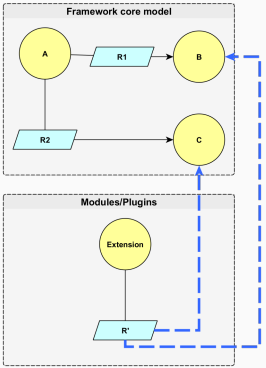
GraphApps Framework Screenshot

The GraphApps Framework (showed here with no CSS) proposes many features including:

- Alternate navigations
- Reusable components for user defined management (Dossiers, Labels and To-do lists)
- Attachments (that can be multi-referenced)
- Geolocation
- History
- Custom navigation between business concepts
- User security
- Optional graph navigation
- Graph manipulation toolkit
- Etc.



Plugins, modules and semantic domains

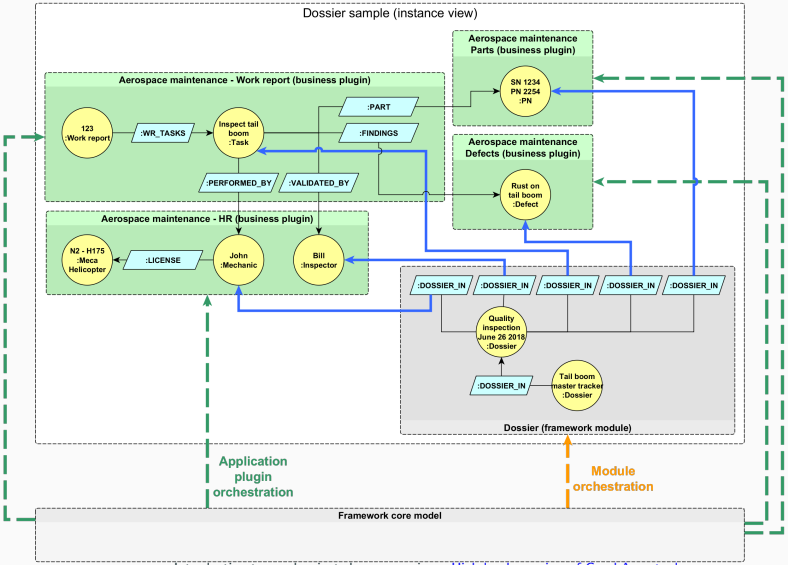


The framework manages navigation and knows the “root node” of every web page

The framework works with extensions of its core model

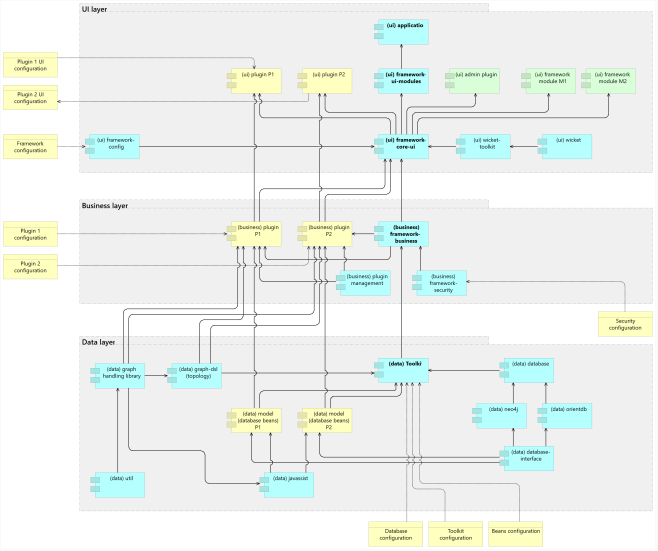
- Internal services can be added and available on every web page (for instance through the toolbar)
  - They can act on the root node and decorate it without interfering with the framework behavior or the application behavior
- Applications are loaded by the framework under the form of a plugin
  - Each plugin can customize its full UI environment
  - By respecting the contract of the framework, the plugin is glued in the framework, alongside with other plugins
  - In case of change, many elements won't have to be non regressed because they have no dependency between each other

Sample of framework module and plugins per semantic domain



Framework high level architecture

- In blue: the framework main components
- In green: framework modules and security plugin
  - Only top layer packages are shown
- In yellow: application code and configuration files



Conclusion

Application domain for GraphApps tools

GraphApps tools were designed with 3 requirements of unusual applications in mind

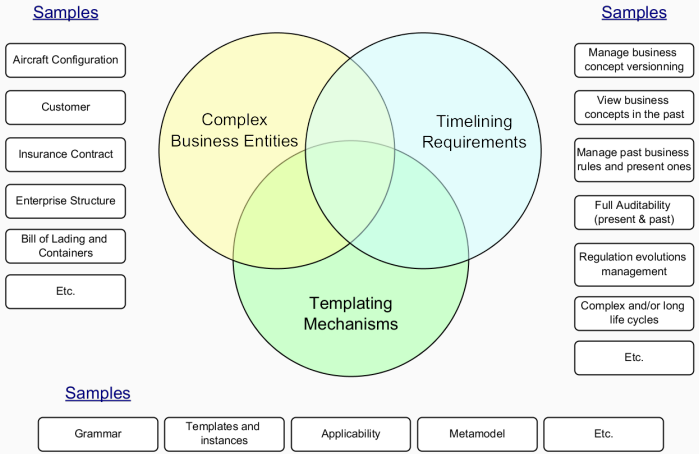
Application	Complexity
Complex container transport sales and logistic system	–Dossiers containing heterogeneous business objects –Advanced versioning of quotation system –Multiple points of view of the same business object
Complex public tax collection system on enterprises	–Enterprises are very complex business objects (graphs) –Corpus of thousands of business rules, some of them being "in competition" –Business rules apply or not on enterprises depending on patterns on enterprise structure Timelined business rules (regulatory)
Aerospace maintenance information system (MIS)	–Very complex "business object" (A/C configuration) –A/C templating mechanism –Timeline-oriented maintenance with versions of manuals and procedures

Applicability of graph-oriented programming

Some aerospace management applications domains were prototyped (MIS, SMS)

The kind of software that have between 1 and 3 of those features can be realized in OOP/RDBMS but with much pain and with design choices that will make important business evolutions very hard and costly

Indeed many systems could benefit from GOP (Financials, CRM, ERP, PLM, etc.)



Demo

Questions?

About

About the authors

Olivier Rey – rey.olivier@gmail.com – orey.github.io

- +20 years of experience in software companies and IT service companies involved in complex software projects
- +10 years as a senior enterprise architect using graph-oriented modeling (Archimate)
- +10 years as a program director in complex projects
- Creator of the graph-oriented programming approach
- Expertise in high end distributed transactional system and middleware

Alexandre Ricciardi – alexandre.ricciardi@gmail.com

- +15 years of experience in software companies and IT service companies
- Various experiences in enterprise creation, scientific calculation, 3D programming, and a lot in professional business applications in a recurring innovation context

ICGT 2018

11th International Conference on Graph Transformation

25-26 June 2018 - Toulouse, France - Affiliated with STAF

http://www.icgt-conferences.org

The conference takes place under the auspices of EATCS, EASST, and IFIP WG 1.3.
Proceedings will be published by Springer in the Lecture Notes in Computer Science series.

ICGT 2018 continues the series of conferences previously held in Barcelona, Rome, Natal, Leicester, Enschede, Bremen, York, L'Aquila, Vienna, and Marburg.

We are pleased to announce Olivier Rey (CEO, GraphApps, France) as invited speaker.

Important Dates:
Abstract submission: 23 February 2018
Paper submission: 2 March 2018
Notification: 9 April 2018
Conference: 25 - 26 June 2018

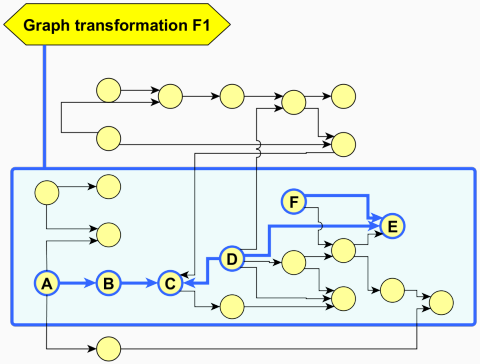
Steering Committee:
Paolo Bottoni, Andrea Corradini, Gregor Engels, Holger Giese, Reiko Heckel, Dirk Janssens, Barbara König, Hans-Jörg Kreowski, Ugo Montanari, Mohamed Mosbah, Manfred Nagl, Fernando Orejas, Francesco Parisi-Presicce, John Pfaltz, Detlef Plump, Arend Rensink, Leila Ribeiro, Grzegorz Rozenberg, Andy Schürr, Gabriele Taentzer, Bernhard Westfechtel

Program Chairs:
Leen Lambers, Jens Weber

Program Committee:
Anthony Anjorin, Paolo Baldan, Gábor Bergmann, Paolo Bottoni, Andrea Corradini, Juan de Lara, Jürgen Dingel, Rachid Echahed, Holger Giese, Annegret Habel, Reiko Heckel, Berthold Hoffmann, Dirk Janssens, Barbara König, Yngve Lamo, Mark Minas, Mohamed Mosbah, Fernando Orejas, Francesco Parisi-Presicce, Detlef Plump, Arend Rensink, Leila Ribeiro, Andy Schürr, Gabriele Taentzer, Bernhard Westfechtel, Albert Zündorf

Backup slides

Graph traversals: benefits and dangers



Object-oriented programming with a graph database can generate a massive amount of technical debt

If `A` is coded in OOP, to reach `F`, we could write:

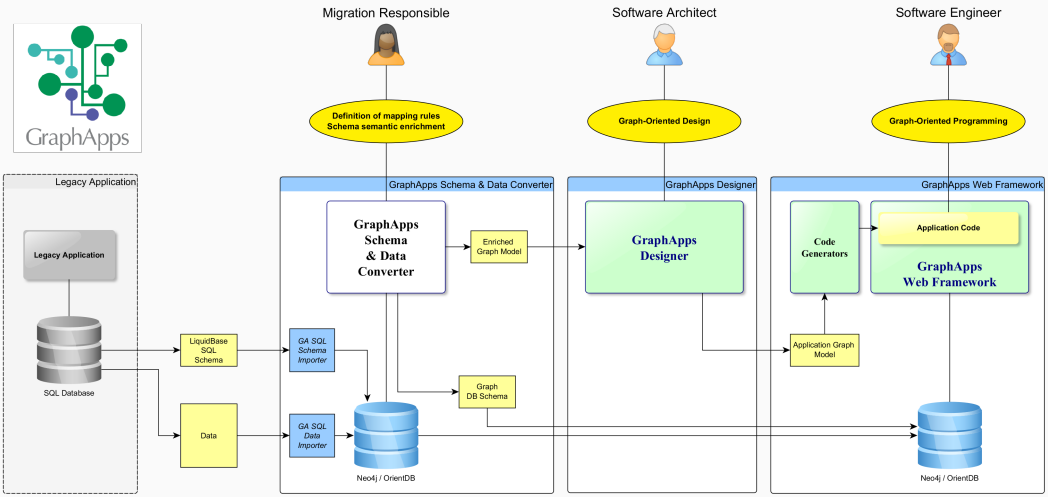
```
F = A.dest(B).dest(C).origin(D).
dest(E).origin(F)
```

generating a lot of technical debt

With a graph transformation using a graph library, this knowledge is in a graph transformation, any change to this graph will, in the worst case, make `F1 NOT-APPLICABLE`

Back

GraphApps full product suite



Back