QUALITY BY DESIGN

Prevent your Mendix application from becoming a software dragon

By:
Diederik Wentink, Mendix Solution Architect
Rolf Bolt, co-founder en Mendix Solution Architect EGALiT

MAINTAINABILITY AN ISSUE?

Organizations are forced to spend a large part of their IT budget on management and maintenance of applications. Unfortunately, the arrival of low-code solutions such as Mendix does not change this. In fact, if a low-code project is not handled properly, maintainability will be even worse than with traditional software development.

A focus of your organization on the maintainability of software will harvest many positive effects. But what are the driving factors that hampers and enhance maintainability? What effects does a shifting focus on maintainability have?

In this article we provide some answers to these questions. In addition to the answers, we will also introduce a concrete solution to make software developed in Mendix maintainable and flexible. This solution, a layered architecture, was devised by EGALiT in recent years and used in many projects.

SOFTWARE THE DRAGON IN THE BASEMENT

One day you go to the pet store and buy a nice little lizard for your terrarium. You feed this animal with mealworms and proudly show it to your children and friends. But the reptile is getting bigger and now needs crickets to stay alive.

Sometime later you have to start feeding the animal mice and it no longer fits in your terrarium. You decide to build a bigger cage in your basement and move the dragon there. It now eats rats and rabbits and you wonder whether you like the pet anymore.

The moment you have to feed the sheep, you prefer to get rid of your pet. But you are afraid that if you stop feeding, the dragon will come out of the cellar and eat you!

CAUSES OF LACK OF QUALITY

A 2018 Deloitte study found that more than half of organizations' IT budgets are spent on application maintenance[1]. Does this also apply to applications that have been realized in a low-code development environment?

SPEED SPEED SPEED…

The last 10 years, low-code development with environments like Mendix has taken off compared to traditional development in, for example, Java or .Net[2]. Because of the out-of-the-box functionality, the generation of software and easy deployment, this way of developing software is much faster. Mendix claims that software development is 6x faster than traditional development[3].

With this development speed, you quickly set up a first version of your application adding more functionality on the go. Low-code development therefore fits in seamlessly with the DevOps idea: how can I convert an idea of the business into a working “service” for the end user as quickly as possible? Unfortunately, the teams and management often forget to take measures in order to maintain the initial speed of the development process in subsequent phases of the project. Like traditional custom software, projects tend to become more difficult to maintain as its size increases. The question arise, what factors drive ‘unmaintainability’?

[1] IT Spending: From Value Preservation to Value Creation
[2] Enterprises are flocking to low-code tools, Mendix reports
[3] Rapid Application Development Tools & Software - Low-Code Platform

CITIZEN DEVELOPERS

One of the factors is a lack of experience with software development. Traditional software languages require quite some training and experience. Developing applications in Mendix on the contrary, is accessible, the toolset is easy to understand and use. This attracts people that have more affinity with the business than with IT (citizen developer[4]) and lowers the barriers of introducing bad designed software.

LACK OF DESIGN, ARCHITECTURE AND CODE STANDARDS

Another potential cause is the lack on focus on design, architecture and code standards. A Mendix team is small typically, and consists of a couple of T-shaped employees that are responsible for the complete delivery cycle. The design is done 'on the fly' and often one of the team members sets up an architecture at start and moves to another project afterwards. Combined with an 'Agile' approach which short iterations, the focus of the team is typically on delivering business functionality fast with insufficient attention to proper design, code standards and good architecture. Due to the high development speed this can lead to a quick decay of software quality resulting in sprawl, code duplication and, in the end, unmaintainable software.

SOFTWARE COMPLEXITY IS NOT SAME AS BUSINESS COMPLEXITY

The Mendix toolbox is taking a lot of code complexity under the hood. This gives an impression that software development as a whole is stripped from complexity. However, the business functionality itself can be quite complex and demands for highly skilled professionals. A good capacity for conceptual thinking and a drive to search for simple solutions is required in order to automate complex business solutions. It is therefore a misunderstanding that deploying Mendix in complex environments with only citizen developers and junior developers will yield good results.

‘STEP BY STEP’ DEVELOPMENT

Low-code tooling like Mendix invites the developer to think in processes instead of data structure, functions and properties. This is induced by the fact that microflows are shown as a ‘process’. The consequence is that a low-code developer often builds software by realizing (complex) business functionality 'step by step'. The necessary properties are added to the data- and process model on the fly. As a result, the overview of the correct operation and outcome of the business functionality disappears. This leads to a lot of (analysis) time and regression problems whenever the software needs to be adapted. Another consequence is that virtually no statements can be made about the correctness of the outcome, other than by programming a large number of test cases.

EFFECTS

These factors increase the risc that a substantial part of the IT budget is spent on maintenance. Like the dragon in the basement, the problem is not visible at the start of the project but is emerging in time. Due to the properties of a low-code platform, the chance is even greater than with traditional software development.

This article lists some principles for a solution in order to prevent your business software from becoming an uncontrollable and unstoppable sink. We then mention a number of important effects of these principles you can expect.

[4] Definition of Citizen Developer - IT Glossary

PRINCIPLES OF A SOLUTION

The most important principle to get and keep custom business software under control is by emphasizing the quality aspect 'maintainability'[5]. This focus calls for attention to the following principles:

- Separation of concern (separation and clear division of business logic and interaction / navigation) – Build business logic only once and ensure that it can be called up from different places, for example from screens.
- Code complexity reduction (KISS) – Break complex code into smaller, less complex units.
- Uniformity in coding – Make sure that all code from all modellers has the same structure.
- Ensure data quality – Separate business rules from application logic and use ACID (atomicity, consistency, isolation, durability) principles to enforce these business rules.
- Traceability of requirements – Establish a relationship to the requirement in the code.
- Low-threshold automatic testing – Perform automatic testing not only on the user interface, but also on the underlying logic.

EFFECTS OF MAINTAINABLE SOFTWARE

In the next paragraphs we will address some of the results that yield from implementing the outlined principles.

DEVELOPMENT COST AND PRODUCTIVITY

Working with a standardized architecture demands a consistent effort. At first glance, this seems to take more time than unstructured work. However, even when initially developing a system, time will be saved. This is caused by the fact that most of the development time will be spent on the most complex parts of the system.

Compare a 100 piece jigsaw puzzle with a 1000 piece jigsaw puzzle. You will be able to solve the smaller puzzle on your own in half an hour. The puzzle of 1000 pieces will probably take a few man-days. If you find a method that adds half an hour of prep time to solving a puzzle, but then halves the solving time (e.g. sorting puzzle pieces), you will save so much time solving it that the extra time will easily pay for itself

Naming standardization facilitates the discovery of realized code and thereby encourages code reuse. Standardization and proper division also facilitate the extension or modification of an existing requirement: a developer does not have to analyze all the code, but can rely on the correct functioning of the system after its modification.

An additional advantage is that it is much easier to perform automatic unit and API tests with standardized software than with unstructured software. These tests make up the largest (and cheapest) part of the test pyramid [6].

With continued development of software, the benefits in terms of costs and productivity become even greater:

- By separating logic and interaction / navigation, less regression will occur. The ability to automatically check for data consistency prevents errors in production data with costly remediation actions.
- Traceability of requirements ensures faster problem analysis with requirement changes.
- Automated unit and API tests are easier to maintain than automated UI tests.

TIME-TO-MARKET

The benefits mentioned in the cost reduction also contribute to a shorter time-to-market of changes to the system.

This starts with traceability of requirements: this enables a quick analysis of the impact of a change proposal. This leads to fewer errors during implementation. The errors are also faster to solve: because the teams build code with a lower complexity in a structured way, they can take advantage of the benefits described in the previous section.

Furthermore, correct code structures and correctly implemented data integrity checks are preventing

[5] ISO 25010 'Maintainability'

[6] Mike Cohn "Succeeding with Agile: Software Development Using Scrum"

side-effects of code changes. The software can be taken to production faster and release rollbacks due to data errors are less expected. In the unlikely event that there is an error in the release, the good analysability of the structured code ensures that errors can be quickly found and resolved.

EXPANDABILITY AND SCALABILITY

By properly dividing business logic, it is possible to decouple parts of the application from each other. This decoupling makes it easier to divide the application into multiple microservices, each of which has its own purpose. Dividing an application into microservices ensures better extensibility and scalability of the application, because, for example, the microservices can be distributed over different servers.

The uniform work by team members ensures that employees can work on all applications with a short training period. This allows a team to scale up quickly.

ERROR SENSITIVITY AND CORRECTNESS

In many applications, business logic is intertwined with user interaction. As a result, it can happen that the application gives different results if the same function is performed in different places. You prevent this by realizing business logic separately. The business logic can be (automatically) tested separately from the rest of the application to ensure continued operation.

Reducing complexity means fewer errors. Low complexity and separate realization of business logic makes it easier to detect and analyze errors. The uniform working method also contributes to this. Because all software has the same structure, an employee can easily review the code of his colleague and detect errors.

LEARNING CURVE NEW EMPLOYEES

For junior employees it is nice to work in a fixed structure. This provides guidance when realizing the software. An organization can choose to have employees with little experience only work on screens and navigation. These can be junior employees, but also people who cooperate from the business (citizen developers). Because there is no business logic in screens, the inexperienced employee will not get bogged down in complexity or introduce errors that lead to incorrect data.

The fixed structure allows a new employee to quickly add value to another team. And because the requirements are traceable, it is still possible for employees who are unfamiliar with the system to quickly understand how the logic of the system works. This provides flexibility in shifting team sizes or switching the applications a team is working on.

LAYERED ARCHITECTURE AS A SOLUTION

In recent years, EGALiT has come up with a solution to make software developed in Mendix maintainable and flexible. This solution is basically an architecture that consists of a layer model in which each layer in the software has its own specific responsibility (separation of concern).

MVC-LIKE PATTERN

The layer model is a more detailed elaboration of the Model-View-Controller (MVC) pattern that has long been used in traditional software development:

- All business logic is defined in the Model: validations, business rules and operations / calculations on the data.
- The View concerns all communication to the outside world. These can be screens, interfaces with other systems, but also events with a time trigger.
- The Controller takes care of executing the business rules in the correct order and writing them to the database. The Controller enforces the ACID principles.

The Model consists of a number of sublayers:

- Selections: Retrieve data from memory or database.
- Manipulating objects and data (operations, functions, derivations, processes and process steps).
- Validate data against validity of business rules.

View includes:

- The definition of the interfaces (web services, other links but also screens).
- The handling of screen interaction (navigation, screen events etc).

The Controller typically consists of only one layer that is called from the View and that combines the different layers from the Model.

CONDITIONS

The dividing of the Mendix code into layers is done on the basis of agreements. No additional software is required to apply the layer model. What is needed:

- Clear agreements and rules regarding building software.
- Good instruction and training for all modellers.
- A review process that monitors the application of the agreements and rules.
- A “layer model champion” in the team: the modeller/architect who can help to correctly apply the layer architecture in complex and deviant cases.

WHY IS NOT EVERYBODY DOING THIS?

As mentioned earlier, Mendix modellers often have no background in programming and architecture. The threshold for working in Mendix is low, so citizen developers can also build code. Due to the lack of knowledge and background, no thought is given to setting up a uniform structure.

To realize a good software architecture, it is necessary that there is sufficient IT and architecture knowledge in a team. But at the moment this knowledge is scarce, so teams start with too little seniority and knowledge.

Mendix modellers are often not architects. Due to a focus on the short term, architectural principles are not considered important enough by modellers. Experienced modellers feel limited in their creativity and working method by applying a uniform framework and will only recognize the importance after maintenance and extensibility problems.

Even in a still young project, a significant 'technical debt' can already exists. Experience shows that it is difficult to bring an existing low-code application under architecture afterwards. Not only because of complex code, but because implementing architecture requires an investment of money and time without immediate business value. As long as the ‘application works’, businesses are tending to spend their money on new functionality.

Occasionally it is possible to implement the architecture afterwards for only a part of the software. This can be a solution when complete overhaul is to expensive and most benefits can be reaped on complex parts of the software. Preconditions to this approach are a well-defined scope and a clear and well defined plan.

CONCLUSION AND CONTINUATION

We have written this article to make more people aware of the importance of working under a uniform architecture. The investment required to initially set up a uniform architecture is relatively low and the long-term benefits are significant. By choosing to work under architecture, you can include employees in this way of thinking from the start.

In a follow-up article we will go into more detail about the layer model and how to apply it in Mendix. We also explain a number of standard patterns, for example for status transitions.

Until then, you can contact us for more information.

Diederik Wentink, diederik@wentinksolutions.nl

Rolf Bolt, r.bolt@egalit.nl

For more information about the services of EGALiT, please contact Markus Travaille m.travaille@egalit.nl 06-19975135