Mantra Development Paradigm Principles White Paper

Principles to be followed to develop
lean and highly evolvable projects.

www.mantrajs.com

REASONS WHY SOFTWARE TENDS TO BECOME CORRUPTED

Software development in a growing project leads, sooner or later, to a corrupted project if most of the best practices are not consistently and faithfully followed.

A software project is considered to be corrupt when:

- It becomes increasingly difficult to maintain.
- The cost (measured in time) of correcting errors is increasing.
- The cost and difficulty of adding new features or changing existing ones are also increasing.

This effect, if not avoided, has enormous economic and productivity repercussions, often causing a product or project to become unviable.

In this regard, this characteristic of corruptibility is consubstantial to software development and, fundamentally, at its base, is because in most projects:

- It is not possible to dedicate enough time and effort to refactor continuously.
- No effort is dedicated to redesigning the existing when new requirements are implemented.
- There is no correct testing policy.
- There is not a sufficiently mature culture among developers to understand how technical debt accumulates and how to avoid it.
- Not thinking from a framework perspective.

On the other hand, it is common in startups, prototypes, entrepreneurial projects, and others poorly specified by the customer or with a deficient functional analysis, that there is a lack of knowledge of future requirements, which, when implemented without any improvement in the design, architecture and code quality of the existing ones, makes the inclusion of these requirements increasingly difficult and costly, causing more significant technical debt and a higher degree of corruptibility, hence the impossibility of starting with static and too rigid designs and architectures that, later, perhaps months or years later, have to be equally compatible with the new requirements to be incorporated in the project.

In general terms, this is how a big ball of mud is produced, which gradually grows until at some point it is so unsustainable that the project has to be redone in whole or in large part (sometimes fictitiously labeling the new project as a superior version of the previous one...) if the corporate entity that supports it can assume the costs when the project is not considered unfeasible and is abandoned.

All this gets even worse if technologies are chosen that are too changeable or not mature enough to require resources (and their impact on costs) to be devoted to their migration, when the software is not developed according to a specific deployment platform, thus limiting some design and architecture decisions that fit better with the nature of the project.

In this respect and challenging to maintain, the consequences of a corrupt software project go far beyond just technical and stress consequences for developers. They can jeopardize the very viability of a business, which happens very often in the software industry.

From all of the above, which is just a very abbreviated summary of why software tends to become complicated and obfuscated, we can see the need to develop software in a certain way, following certain principles, to avoid or minimize the above effects, especially in lean projects without a clear scope or that are going to change or pivot a lot during their lifetime.

This inherent characteristic of software can be avoided or minimized by following some principles, such as those presented in this document.

This whitepaper outlines these paradigmatic principles that should be followed by a software project that:

- It will change and evolve considerably.
- It is mainly unknown a priori in which direction it will change and evolve in the future.
- Maintainability, readability, and code reusability are intended to be the cornerstone of the project without extra cost.
- Development speed should be as fast as possible.
- The accumulation of relevant technical debt is avoided as much as possible.
- Creating homogeneous, small, and well-organized code blocks that can be reused in other projects is encouraged.

The following development principles enable all of the above and complement others such as S.O.L.I.D., dependency injection, etc. Some of these principles are considered best practices in software development or follow well-known design practices.

All these principles should be used together, allowing the implementation of complex and large applications or systems and platforms (with a set of related applications), reducing technical debt, and facilitating all of the above.

Building maintainable and evolvable software involves the design, architecture, testability, and methodology concepts.

These principles are listed below to define them as simply as possible.

PRINCIPLES

RADICAL COMPONENTIZATION

The software project must be organized into components. These are the minimum units of code. Everything in the project must be wrapped under the structure of a component.

A project is composed of a set of components that are related.

Usually, a project is composed of tens or hundreds of components well organized by their responsibility.

A component is the minimum unit of code and must comply with:

- It has a single purpose.
- It is as small as possible in terms of lines of code (at most, a few hundred lines).
- The functionality that the component implements is simple, concrete, and belongs to the same level of abstraction or the same level of the domain.
- The component exposes its functionality or the assets it defines and implements to the rest of the project.

- Components work decoupled from each other.
- There are no hard dependencies between components, so replacing one with an improved or different version will not affect the project's performance. For this purpose, the components expose their functionality through integration APIs.

COMPONENTS COMMUNICATE THROUGH APIS

Using the functionality of one component by another is done through the definition of APIs. The invocation of these APIs is not direct, but a framework must serve as an intermediary.

COMPONENTS INDICATE RELEVANT FACTS THROUGH EVENTS.

Components indicate situations or facts in the system by emitting events so that other components can subscribe to them to perform decoupled third actions without the need to bind the code with hard dependencies.

VERY HIGH-LEVEL FUNCTIONALITY IS IMPLEMENTED AS A COMPONENT ARRANGER

As mentioned above, most components must implement basic or low-level functionality. The implementation of higher-level functionality, business logic, or more complex business processes is also implemented in components that utilize the former.

MINIMALISTIC AND INDEPENDENT DATA MODELS

A component may indicate that it requires to have information persisted utilizing a data model. Since the component's functionality must be concrete and straightforward, its data model to be maintained will also be simple and concrete: few data entities with few properties each.

There are no hard dependencies between the data model from one component and another. These dependencies are semantic consensuses or assumptions between components.

REPOSITORIES AND DATA PERSISTENCE MUST BE TRANSPARENT TO THE COMPONENTS.

Data models are exposed by the components or indicated by definition and never refer to any specific persistent technology.

DIFFERENT TYPES OF PERSISTENCE IN THE SAME PROJECT

Although the data models of the components are simple and are defined by a statement, and the kind of persistence is transparent to the component, different types of persistence should be able to coexist in the same project or system, depending on the nature of the data models to be implemented.

MINIMUM AND INCREMENTAL UPGRADES

Components may be upgraded and indicate this circumstance. In the context of the project and its production run, these updates are small, frequent, and incremental.

ISOLATION OF NON-STANDARD THIRD-PARTY TECHNOLOGIES

The use of very specific third-party technologies is done utilizing wrappers that serve as a bridge between them and the

rest of the project to be replaceable at any given time.

A project or system is comprised of a set of applications that share components.

In the same project, different applications related to different purposes coexist so that they are also small and reuse a set of system components for their deployment.

PERFORMANCE BY STATEMENT

Some repetitive aspects of the system, within its particular domain, should be abstracted in statements, not in ad hoc programming.

PRACTICAL CONSEQUENCES

The practical consequences of following the above principles are described below through explanation and justification.

Radical componentization

The application or system is implemented in its entirety by components that should be as small as possible, which allows a better organization of the system.

Being small components, they will be more reusable in other projects.

Components can be organized according to their purpose, the area of the system they implement, or according to their hierarchy in a layered distribution.

A system (set of applications), or an application, can be composed of tens or hundreds of components that operate in a coordinated manner through the context (framework).

The component's functionality must “look” as little as possible to the nature of the application to be implemented. This will increase the degree of its reusability in other projects.

The components act wholly decoupled from each other

A component may depend on others and, at the same time, expose functionality that is consumed by as many others.

However, this dependency is not direct but is managed by the system context that acts as an intermediary between the components.

Since the components are small and highly decoupled from each other, tests for each component are more straightforward and more independent to implement.

Components Communicate via APIs

A component exposes its functionality by defining APIs and, in turn, consumes the functionality of others through their APIs. The context takes care of the intermediate work of communicating one component with another.

The components indicate relevant facts through events

Since the system requirements will change significantly over the software project's life, issuing events to manage the very high-level logic makes the coupling of components even smaller, allowing even more flexibility to add or modify functionality in the future.

High-level functionality is implemented as a component arranger

The high-level functionality or workflows are implemented in components whose mission will be to implement what to do when system events are emitted.

Thus, it is possible to have in the project a section with the arranger components to distinguish them from the rest, thus improving the organization and localization of the code.

Minimalistic and independent data models

Since the components are small and implement, by definition, a very particular functionality, it follows that the data entities they need will be small and easy to define, following a simple table model.

In addition, persistence in any data repository, whatever its type, will be simplified and complex; large and inefficient databases that are difficult to evolve will be avoided.

This will also simplify updates and migrations since by managing a few data entities with a reduced number of properties in a single component, and this routine work will be easier to perform and less prone to errors.

Each component manages its data model through the context independently so that a single application can have dozens of

components, each with its own different data model.

Repositories and data persistence must be transparent to the components

The context handles all the data model persistence work.

Implementing this context should provide the component with an easy and transparent way to manage its data model by isolating it from anything to do with its actual persistence of any kind (in the form of databases, in-memory cache, files, etc.).

Similarly, it is the context responsible for the creation or instantiation of the data models and the way they are consumed.

Different types of persistence in the same project

It is natural that in an application with hundreds of components, different types of persistence and even different instances of them may be necessary.

By setup, the project tells the context which type of persistence to use for each component (for example, Redis, MySql, Sql Server, Postgresql, etc.).

This way, the component only indicates its data model and

uses the context to manage it without worrying about all the plumbing necessary for its persistence.

As a natural consequence, the same application or system can use several different databases with optimizations adapted to each data model.

Minimal and incremental updates

Since the data models are simple and only consist of several entities with few properties, their evolution throughout the project's life will also be simple: adding a new property, modifying the type of an existing property, adding a new entity, etc.

Thus, migrating a data model to its next version will be easier than using an extensive database with hundreds of tables and hundreds of relationships and constraints.

Updates are intended to be small but frequent, which fits better in an environment with a DevOps and CD/CI (continuous development / continuous integration) culture, favoring a greater culture of testing and constant system updates with less risk.

The context provides the component with everything necessary to perform this type of migration in its data models.

Isolation of non-standard third-party technologies

In this paradigm of complex application development, we try not to depend "heavily" on third-party technologies that will undoubtedly change throughout the project's life, including a significant effort in migrations.

If one of these technologies is to be used, it must be adequately insulated in components that serve as wrappers and can be replaced at any time.

A system is composed of a set of applications that share components

In the usual ecosystem of a growing system, the need for different applications with different purposes will naturally arise.

The context must allow different applications to be implemented using the same set of components that make up the system.

For example, there is nothing worse than trying to implement in the same user interface, the end-user or customer functionality, plus the administration functionality, plus the ticket or incident management functionality, plus the analytics control panel, etc.

By creating independent applications with different purposes

and using the same components, the size of these applications will be smaller, allowing a much simpler and easier maintainability and evolution.

Similarly, this will also make it possible to deploy each of the applications that make up the system in different environments, with different levels of performance or security.

Performance by Statement

Some specific activities of the domain to be implemented in the application are usually repetitive or very functionally related.

When this is detected, it is required to implement a more abstract functionality so that the concrete one is defined by statement (xml, json, etc.).

For example, the generation and maintenance of forms are costly in any project: a better approach would be to define them by a statement in json objects indicating with properties the content of each form. At the same time, implementing a specific component (abstraction) would take care of generating it (rendering it) and managing its performance.

The above foundational principles are implemented in Mantra to allow the construction of complex applications (and systems composed of several applications) that will evolve and change significantly during their development, with the most reusable composition possible.

Rafael Gómez Blanes, 2022

www.mantrajs.com