Mantra Development Paradigm Principles White Paper

Principios a seguir para desarrollar
proyectos 'lean' y altamente evolucionables.

www.mantrajs.com

MOTIVOS POR LOS QUE EL SOFTWARE TIENDE A CORROMPERSE

El desarrollo de software en un proyecto en crecimiento conduce, tarde o temprano, a que éste se corrompa si no se siguen fielmente la mayoría de las buenas prácticas de forma continuada.

Se considera que un proyecto software está corrupto cuando:

1) Cada vez resulta más difícil de mantener.

2) El coste (medido en tiempo) de corregir errores es cada vez mayor.

3) El coste y dificultad de añadir nueva funcionalidad o cambiar la existente, también son mayores.

Este efecto, si no se evita, tiene enormes repercusiones económicas y de productividad, haciendo frecuente que un producto o proyecto se termine por convertir en inviable.

En este sentido, esta característica de corruptibilidad, es

consustancial al desarrollo de software y, en esencia, en su base, se debe a que en la mayoría de proyectos:

1) No se puede dedicar el tiempo y esfuerzos suficientes para refactorizar continuamente.
2) Tampoco se dedica esfuerzo a rediseñar lo existente cuando se implementan nuevos requisitos.
3) No existe una política correcta de testing.
4) No existe una cultura entre los desarrolladores suficientemente madura de forma que comprendan cómo se acumula deuda técnica y cómo evitarla.
5) No se piensa desde la perspectiva de un framework.

Por otra parte, es habitual en startups, prototipos, proyectos emprendedores y otros mal especificados por el cliente o con un análisis funcional deficiente, que haya un desconocimiento de futuros requisitos, que, al implementarlos sin una mejora del diseño, arquitectura y calidad de código de lo existente, hace que la inclusión de éstos sea cada vez más difícil y costosa, provocando mayor deuda técnica y un mayor grado de corruptibilidad, de ahí la imposibilidad de partir con diseños y arquitecturas estáticas y demasiado rígidas que, después, quizá meses o años más tarde, tengan que ser igualmente compatibles con las nuevas exigencias a incorporar en el proyecto.

A grandes rasgos, así es como se produce una gran bola de barro (*big ball of mud*) que crece paulatinamente hasta que en

algún momento es tan insostenible que hay que volver a rehacer el proyecto en su totalidad o en gran parte (a veces etiquetando ficticiamente el nuevo proyecto como una versión superior a la anterior…), si es que la entidad corporativa que lo sustenta lo puede asumir en costes, cuando no se considera el proyecto como inviable y es abandonado.

Todo esto empeora aún más si se eligen tecnologías demasiado cambiantes o poco maduras que obligarán a dedicar recursos (y su repercusión en costes) a sus migraciones, cuando no se desarrolla el software ceñido a una plataforma de despliegue concreta, limitando así algunas decisiones de diseño y arquitectura que encajan mejor con la naturaleza del proyecto.

Las consecuencias de un proyecto software corrupto en este sentido y difícil de mantener, van mucho más allá de las meramente técnicas y de estrés para los desarrolladores: pueden poner en dificultades la misma viabilidad de un negocio, como de hecho ocurre con mucha frecuencia en la industria del software.

De todo lo anterior, que no es más que un resumen muy abreviado de por qué el software tiende a complicarse y ofuscarse, se extrae la necesidad de desarrollar software de cierto modo, siguiendo ciertos principios, de forma que se eviten o minimicen los efectos anteriores, especialmente en proyectos lean sin un alcance del todo claro o que van a

cambiar o pivotar mucho a lo largo de su tiempo de vida.

Esta característica consustancial del software, se puede evitar o minimizar si se siguen algunos principios, como los expuestos en este documento.

En este *white paper* se presentan estos principios paradigmáticos que debe seguir un proyecto software que:

- Va a cambiar y evolucionar considerablemente.
- Se desconoce a priori en gran parte en qué dirección va a cambiar y evolucionar en el futuro.
- Se pretende que la mantenibilidad, legibilidad y reusabilidad de código sean la piedra angular del proyecto sin que esto suponga un coste extra.
- La velocidad de desarrollo sea la máxima posible.
- Se evite en la medida de lo posible la acumulación de deuda técnica relevante.
- Se fomente la creación de bloques de código homogéneos, pequeños y bien organizados que permitan ser reutilizados en otros proyectos.

Los siguientes principios de desarrollo permiten todo lo anterior y son complementarios de otros como S.O.L.I.D., inyección de dependencias, etc. Algunos de estos principios se consideran buenas prácticas en el desarrollo de software o siguen prácticas de diseño bien conocidas.

Todos estos principios deben ser utilizados juntos, lo que permite implementar aplicaciones complejas y grandes o sistemas y plataformas (con un conjunto de aplicaciones relacionadas) reduciendo la deuda técnica y facilitando todo lo anterior.

Construir software mantenible y evolucionable es una cuestión compleja en la que participan conceptos de diseño, arquitectura, testeabilidad y metodología.

Se indican a continuación estos principios, con la intención de definirlos de la forma más simple posibles.

PRINCIPIOS

COMPONETIZACIÓN RADICAL

El proyecto software debe estar organizado en componentes. Estos son las unidades mínimas de código. Todo en el proyecto debe estar envuelto bajo la estructura de un componente.

Un proyecto está compuesto por un conjunto de componentes que se relacionan.

Habitualmente, un proyecto está compuesto por decenas o cientos de componentes bien organizados por su responsabilidad.

Un componente es la unidad mínima de código y debe cumplir:

- Tiene un único propósito.
- Es lo más pequeño posible en términos de líneas de código (como mucho, unos cientos de líneas).
- La funcionalidad que implementa el componente es

sencilla, concreta y pertenece al mismo nivel de abstracción o el mismo nivel de dominio.

- El componente expone su funcionalidad o los activos que define e implementa, al resto del proyecto.

LOS COMPONENTES ACTÚAN DESACOPLADOS UNOS DE OTROS

No existen dependencias duras entre componentes, de modo que la sustitución de uno por una versión mejorada o diferente, no afectará al funcionamiento del proyecto. Para ello, los componentes exponen su funcionalidad mediante APIs de integración.

LOS COMPONENTES SE COMUNICAN MEDIANTE APIS

El uso de la funcionalidad de un componente por parte de otro, es realizado mediante la definición de APIs. La invocación de estas APIs no es directa, sino que debe actuar un framework como intermediario.

Los componentes indican hechos relevantes mediante eventos

Los componentes indican situaciones o hechos en el sistema

emitiendo eventos, de tal modo que otros componentes se pueden subscribir a ellos para realizar terceras acciones desacopladas sin necesidad de ligar el código con dependencias duras.

LA FUNCIONALIDAD DE MUY ALTO NIVEL ESTÁ IMPLEMENTADA COMO ORQUESTACIÓN DE COMPONENTES

Como se ha dicho, la mayoría de los componentes deben implementar funcionalidad básica o de bajo nivel. La implementación de funcionalidad de más alto nivel, lógica de negocio o procesos de negocio más complejos, se implementa también en componentes que utilizan los primeros.

MODELOS DE DATOS MINIMALISTAS E INDEPENDIENTES

Un componente puede indicar que tiene necesidad de persistir información mediante un modelo de datos. Puesto que la funcionalidad del componente debe ser simple y concreta, se deduce que su modelo de datos a persistir también lo será: pocas entidades de datos con pocas propiedades cada uno.

No existen dependencias duras entre el modelo de datos de

un componente y el de otro. Estas dependencias son consensos o asunciones semánticas entre componentes.

LOS REPOSITORIOS Y LA PERSISTENCIA DE DATOS DEBEN SER TRANSPARENTES PARA LOS COMPONENTES

Los modelos de datos son expuestos por los componentes o indicados por definición y nunca hacen referencia a ninguna tecnología de persistencia específica.

DISTINTOS TIPOS DE PERSISTENCIA EN EL MISMO PROYECTO

Si bien los modelos de datos de los componentes son simples y se definen por declaración, y el tipo de persistencia es transparente al componente, en un mismo proyecto o sistema, deben poder convivir distintos tipos de persistencia, según la naturaleza de los modelos de datos a implementar.

ACTUALIZACIONES MÍNIMAS E INCREMENTALES

Los componentes pueden ser actualizados e indican esta circunstancia. En el contexto del proyecto y su ejecución de producción, estas actualizaciones son pequeñas, frecuentes e incrementales.

AISLAMIENTO DE TECNOLOGÍAS DE TERCEROS NO ESTÁNDAR

La utilización de tecnologías muy concretas de terceros se realiza mediante componentes envoltorios (*wrappers*) que sirvan de puente entre éstas y el resto del proyecto, de forma que puedan ser reemplazable en un momento dado.

UN PROYECTO O SISTEMA ESTÁ COMPUESTO DE UN CONJUNTO DE APLICACIONES QUE COMPARTEN COMPONENTES

En el mismo proyecto, conviven diferentes aplicaciones relacionadas con propósitos también distintos, de modo que éstas sean también pequeñas y que reutilicen para su

implementación un conjunto de los componentes del sistema.

COMPORTAMIENTO POR DECLARACIÓN

Algunos aspectos repetitivos del sistema, dentro de su dominio particular, se deben abstraer en declaraciones, no en programación ad hoc.

CONSECUENCIAS PRÁCTICAS

Se describen a continuación las consecuencias prácticas de seguir los anteriores principios, a modo también de explicación y justificación de los mismos.

Componetización radical

La aplicación o sistema está implementada en su totalidad por componentes que deben ser lo más pequeños posible, lo que permite una organización del sistema mejor.

Al ser componentes pequeños, estos serán más reutilizables en otros proyectos.

Los componentes se pueden organizar según su propósito, el área del sistema que implementa o según su jerarquía en una distribución por capas.

Un sistema (conjunto de aplicaciones) o una aplicación,

puede estar compuesta de decenas o cientos de componentes que actúan coordinadamente a través del contexto (framework).

La funcionalidad de un componente tiene que "oler" lo menos posible a la naturaleza de la aplicación a implementar. De ese modo, aumentará el grado de su reutilización en otros proyectos.

Los componentes actúan totalmente desacoplados unos de otros

Un componente puede depender de otros, y, al mismo tiempo, exponer funcionalidad que es consumida por otros tantos.

No obstante, esta dependencia no es directa, sino que es gestionada por el contexto del sistema que hace de intermediario entre los componentes.

Puesto que los componentes son pequeños y están altamente desacoplados entre ellos, los tests para cada componente son más sencillos e independientes de implementar.

Los componentes se comunican mediante APIs

Un componente expone su funcionalidad mediante la definición de APIs y, a su vez, consume la funcionalidad de

otros mediantes sus APIs. El contexto se encarga del trabajo intermedio de comunicar un componente con otro.

Los componentes indican hechos relevantes mediante eventos

Puesto que los requisitos del sistema van a cambiar notablemente a lo largo de la vida del proyecto software, la emisión de eventos para gestionar la lógica de muy alto nivel, hace que el acoplamiento de los componentes sea aún menor, lo cual permite aún más flexibilidad para añadir o modificar funcionalidad en el futuro.

La funcionalidad de alto nivel está implementada como orquestación de componentes

La funcionalidad o flujos de trabajo de alto nivel, se implementa en componentes cuya misión será implementar qué hacer cuando los eventos del sistema sean emitidos.

De esta forma, se puede tener en el proyecto una sección con los componentes orquestadores, para distinguirlos del resto, mejorando así la organización y la localización del código.

Modelos de datos minimalistas e independientes

Puesto que los componentes son pequeños e implementan, por definición, una funcionalidad muy concreta, de aquí se

extrae que las entidades de datos que necesiten, serán pequeñas y fáciles de definir, siguiendo un modelo tipo *simple table model*.

Además, la persistencia en cualquier repositorio de datos, del tipo que sea, se simplificará y se evitará tener bases de datos complejas, grandes e ineficientes difíciles de evolucionar.

De este modo, se simplifican también las actualizaciones y las migraciones, puesto que al gestionar un componente pocas entidades de datos con un número también reducido de propiedades, este trabajo habitual será más sencillo de realizar y menos dado a errores.

Cada componente gestiona su propio modelo de datos a través del contexto de forma independiente, de modo que una misma aplicación puede contar con decenas de componentes y cada uno de ellos con su propio modelo de datos diferente.

Los repositorios y la persistencia de datos deben ser transparentes para los componentes

El contexto se encarga de todo el trabajo de persistencia de los modelos de datos.

La implementación de este contexto, debe proporcionarle al componente una forma fácil y transparente de gestionar su propio modelo de datos aislándole de todo lo que tenga que ver

con su persistencia real en cualquier tipo (en la forma de bases de datos, caché en memoria, archivos, etc.).

Del mismo modo, es el contexto el que se encarga de la creación o instanciación de los modelos de datos así como de la forma de consumirlos.

Distintos tipos de persistencia en el mismo proyecto

Es natural que en una aplicación con cientos de componentes, puedan ser necesarios distintos tipos de persistencia e incluso diferentes instancias de las mismas.

Por configuración, en el proyecto se le indica al contexto qué tipo de persistencia usar para cada componente (por poner unos ejemplos, Redis, MySql, Sql Server, Postgresql, etc.).

De este modo, el componente tan solo indica su modelo de datos y utiliza el contexto para gestionarlos, sin preocuparse de toda la fontanería necesaria para su persistencia.

Como consecuencia natural, una misma aplicación o sistema, puede utilizar varias bases de datos diferentes y de distintos tipos con optimizaciones también adaptadas para cada modelo de datos en particular.

Actualizaciones mínimas e incrementales

Puesto que los modelos de datos son simples y apenas

compuestos por varias entidades con pocas propiedades, su evolución a lo largo de la vida del proyecto, será también sencilla: añadir una nueva propiedad, modificación del tipo de una propiedad existencia, adición de una nueva entidad, etc.

De este modo, la migración de un modelo de datos a su siguiente versión, será más sencilla que si se utilizase una gran base de datos con cientos de tablas y cientos de relaciones y restricciones.

Se pretende que las actualizaciones sean pequeñas pero frecuentes, lo cual encaja mejor en un entorno con cultura DevOps y CD/CI (*continuous development / continuous integration*), favoreciendo una mayor cultura de testing y actualizaciones del sistema continuas con menos riesgos.

El contexto le ofrece al componente todo lo necesario para realizar este tipo de migraciones en sus modelos de datos.

Aislamiento de tecnologías de terceros no estándar

En este paradigma de desarrollo de aplicaciones complejas, se intenta no depender de forma "fuerte" de terceras tecnologías que van a cambiar sin duda a lo largo de la vida del proyecto, incluyendo en éste un esfuerzo significativo en migraciones.

En caso de tener que usar una de estas tecnologías, se debe

aislar correctamente en componentes que actuarán a modo de "envoltorios" (*wrappers*) y que podrán ser sustituidos en cualquier momento.

Un sistema está compuesto de un conjunto de aplicaciones que comparten componentes

En el ecosistema habitual de un sistema que crece, surgirá de forma natural la necesidad de que existan distintas aplicaciones con diferentes propósitos.

El contexto debe permitir que se puedan implementar diferentes aplicaciones consumiendo el mismo conjunto de componentes que constituye en sistema.

Por poner un ejemplo, nada peor que intentar implementar en la misma interfaz de usuario, la funcionalidad de usuarios finales o clientes, más la funcionalidad de administración, más la de gestión de tickets o incidencias, más el panel de control de analíticas, etc.

Al poder crear aplicaciones independientes con propósitos diferentes y que consumen los mismos componentes, el tamaño de estás será más pequeño, permitiendo una mantenibilidad y evolución mucho más sencilla y fácil.

Del mismo modo, esto permitirá también deplegar cada una de las aplicaciones que componen el sistema en distintos

entornos, con diferentes niveles de rendimiento o seguridad.

Comportamiento por declaración

Algunas actividades concretas del dominio a implementar en la aplicación, suelen ser repetitivas o estar muy relacionadas funcionalmente.

Cuando se detecta lo anterior, hay que implementar una funcionalidad más abstracta de modo que la concreta sea definida por declaración (xml, json, etc.).

Por poner un ejemplo, la generación y mantenimiento de formularios es muy costosa en cualquier proyecto: un mejor enfoque sería definirlos por declaración en objetos json indicando con propiedades el contenido de cada formulario, mientras que la implementación de un componente específico (abstracción) se encargaría de generarlo (renderizarlo) y gestionar su comportamiento.

MANTRA FRAMEWORK

Todos los principios fundacionales indicados anteriormente están implementados en el framework Mantra para permitir construir aplicaciones complejas (y sistemas formados por varias aplicaciones) que van a evolucionar y cambiar notablemente durante su desarrollo, con una componetización lo más reutilizable posible.

Rafael Gómez Blanes, 2022

www.mantrajs.com