Automatización del Software

Bheudek.com

1 Prefacio

Este documento es una compilación de los diferentes artículos publicados en el blog de la página web. http://bheudek.com/blog/

A través de estos artículos hemos intentado explicar la tecnología desde un punto de vista académico y el modo en que la hemos implementado en Bheudek.

Para cualquier pregunta o información puedes contactarnos en info@bheudek.com.

2. Introducción - Automatización del Software

La automatización del desarrollo de software, como todo proceso de automatización, consiste en **delegar a la máquina algunas de las tareas que implica este proceso productivo**. El objetivo es conseguir una **metodología de desarrollo más rápida**, y por tanto con **menos costes**, así como una **mayor calidad** en el producto final.

La automatización en el SW ha sido una meta perseguida con anterioridad aunque no con mucho éxito. **Estándares de diseño tipo UML o herramientas de tipo CASE** (Computer Aided Software Engineering) **no han conseguido los frutos esperados debido a que intentan dar una solución genérica** (de "propósito general") al conjunto de problemas a los que el SW intenta dar respuesta.

Por el contrario, **los nuevos métodos de automatización ofrecen la posibilidad de diseñar**, de forma simple y rápida, **lenguajes que se adapten y puedan describir cada problema particular**. Es este enfoque específico, y no de propósito general, el que permite lograr casos de éxito en la automatización del desarrollo de SW.

La automatización se asocia principalmente a la generación de código porque al fin y al cabo, el código fuente, es el resultado del proceso productivo, pero no conviene olvidar que **una vez que se utilizan estas técnicas, la programación pasa de ser una mera declaración funcional a una verdadera representación del conocimiento**, a partir de lo cual se pueden aplicar diversas disciplinas como la **semántica** o el **razonamiento automatizado**.

Podemos concluir esta introducción diciendo que **a lo largo de la historia el SW a optimizado muchos procesos productivos, es el momento de que optimice el suyo propio**.

3. ¿Por Qué Generación de Código?

La generación de código no es un nuevo estilo o técnica, **es el camino seguido por los lenguajes de programación para hacer frente a la complejidad**, desde la codificación en binario hasta la primera, segunda y demás generaciones de lenguajes. Es lo que los compiladores han estado haciendo desde el inicio.

El tema clave aquí es “hacer frente a la complejidad”. Mientras más complejo sea el problema más abstracto tiene que ser la forma de pensar para resolverlo. En otras palabras, es necesario elevar el nivel de abstracción. Y esta regla se aplica igualmente a las herramientas que se utilizan para resolver el problema: los lenguajes de programación.

Por lo tanto, podemos afirmar que **“elevar el nivel de abstracción es el objetivo perseguido en la evolución de los lenguajes de programación”**.

Los lenguajes comunes que se utilizan hoy en día para resolver los problemas (Java, C #, C + + , Delphi …), se conocen como “lenguajes de propósito general” (GPL en inglés), y aquí está el problema: “propósito general”, que significa que pueden resolver “todos” los problemas, pero desde una perspectiva global. Pueden resolver desde un nivel de abstracción lo suficientemente amplio como para llegar a la solución, pero no tan alto como lo que necesitaríamos en cada problema particular.**Existe una brecha entre el nivel de abstracción que utilizamos para lidiar con el problema y el nivel de abstracción que utilizamos para resolverlo** a través de GPLs.

¿Cómo podemos cubrir esa brecha? Obviamente, con la **generación de código**.

Como conclusión, para abordar adecuadamente un problema tenemos que encontrar un lenguaje particular para definir la solución en el nivel de abstracción que cada problema requiere. Con el fin de hacer que la solución sea computable, tenemos que generar código, por lo general en el nivel inferior más cercano: el de GPL.

Esos lenguajes particulares se conocen como “lenguajes específicos de dominio” (DSL en inglés), pero este tema se abordará en otro post.

Como conclusión, decir que este enfoque **no es solo aplicable a problemas de negocio sino también a la resolución de problemas técnicos**. Por ejemplo, los nuevos retos que ofrece la programación de aplicativos web de escritorio requieren un enfoque más abstracto que integre todas las tecnologías: HTML, CSS, JavaScrpit, AJAX y otros.

4. Buenas Prácticas en la Generación de Código

El error más común cuando generamos código es verlo como una caja negra, pensando que lo importante es “lo que hace” y no “cómo lo hace”. Esto es un error. **Como siempre, la calidad importa**.

Estas son algunas de las características que un buen código generado debería tener:

- **Independiente:** el código manual y el generado deben estar en archivos diferentes, de lo contrario se corre el riesgo de perder el primero en el caso de que tengamos que volver a generar el código (y sucederá).
- **Inmutable:** no se debe cambiar, por dos razones: es peligroso, por ser desconocido, y por la misma razón que en el caso anterior.
- **Legible:** eso significa: nombres de variables y funciones significativos, comentarios, sangría, organizados en carpetas, archivos, etc. El código generado debe estar presentable para recibir la visita de los desarrolladores: para saber cómo funciona y ¿por qué no?, para aprender de él. **Debemos generar un código del que sentirnos orgullosos**.
- **Extensible:** por diferentes razones es posible que tenga que implementar manualmente algunas funcionalidades, por lo que el código generado debe dejar algunas puertas abiertas. **La mejor manera es diseñar el código generado como un Framework**, donde el código manual puede extender sólo algunas funcionalidades permitidas y en un entorno seguro.
- **Estructurada:** elevar el nivel de abstracción requiere un buen conocimiento del campo que se está tratando. Un código mal estructurado puede ser un síntoma de que ese campo no está completamente bajo control. **Una buena generación de código requiere un buen arquitecto**.
- **Robusto:** el código generado puede fallar, por supuesto. El control de errores, la gestión de excepciones, la validación de las entradas, validaciones internas, etc deben ir siempre incluidas en el código. Este tipo de políticas de seguridad se pueden implementar fácilmente en la generación de código y debe ser una de las razones de su calidad.
- **Potente:** una vez dicho lo anterior, deberíamos ver la generación de código como una forma de escribir un código más potente, eso significa pensar en estrategias, en el código generado, que nunca usaríamos si lo hiciéramos a mano (por lo general por razones de mantenimiento).

En resumen, **las buenas prácticas en la generación de código son una mezcla de las buenas prácticas tradicionales y una forma más amplia de pensar**.

5. 10 Ventajas de la Generación de Código

Vayamos al grano. He aquí la lista:

- **Calidad SW:** En todos los aspectos: **rendimiento**, **fiabilidad**, **seguridad**, etc.
- **Estandarización:** no sólo en el código fuente: en la interfaz de usuario, en las estructuras de base de datos, etc.
- **Centralización:** políticas globales tales como el manejo de errores, la gestión de excepciones, el formato de visualización de datos, las validaciones de datos, comprobar los permisos, etc. están centralizados en el generador. Este tipo de políticas son también conocidos como *funcionalidades transversales* y es un tema abordado por la Programación Orientada a Aspectos (AOP en inglés) en la programación tradicional. La centralización evita este problema.
- **Refactorización:** relacionado con el beneficio anterior, la refactorización de código es **fácil y segura**.
- **Productividad: Menor coste** y **menor tiempo de lanzamiento al mercado** (entre versiones).
- **Habilidades Analíticas:** la generación de código requiere un análisis más profundo del dominio antes de implementar la solución a través del generador.
- **Habilidades de Diseño:** requiere un buen arquitecto, con una visión más amplia.
- **Crecimiento Sano:** previene la degradación de la arquitectura.
- **Integración de Nuevos Miembros:** la cultura o las normas de desarrollo son menores cuando se trabaja con generación de código.
- **Nivel de abstracción:** la programación a un nivel más abstracto, además de fácil de entender (es más intencional), **abre la puerta a nuevas posibilidades**, tales como: generación de pruebas unitarias, auto-documentación, carga automática de datos, semántica, racionamiento automático, etc.

La generación de código no es fácil, la implementación de un generador requiere de tiempo y esfuerzo, y más aún si se trata de un *Language Workbench*, pero, sin duda, **los beneficios son enormes**.

6. DSLs

Los lenguajes específicos de dominio (Domain-Specific Languages – DSLs) son lenguajes de programación diseñados para definir, de una manera más **precisa y expresiva**, áreas particulares, bien sean técnicas o de negocio.

Se denominan así en contraposición a los lenguajes de propósito general (General Purpose Languages – GPLs – Java, C#, C++, etc), ofreciendo un enfoque menos amplio pero más preciso, es decir, su objetivo es **cubrir únicamente el área o dominio para el que se diseñan pero hacerlo con las estructuras gramaticales y/o abstracciones gráficas que mejor le definen**.

Para entender estos lenguajes vamos a verlos desde dos puntos de vista: como una evolución a partir de la generación de código y como una evolución desde los GPLs.

DSLs desde la generación de código

Existen diferentes formas, más o menos sofisticadas, para generar código: macros, datos estructurados en tablas, generación dinámica, parseo de estructuras simples, modelados tipo CASE, etc, pero siempre que hablemos de un nivel elevado hablaremos de lenguajes (de tipo texto o gráfico), donde se define de manera formal las **estructuras lingüísticas, su representación y su interpretación**.

De este modo, entendemos que **los DSLs son la vía más sofisticada en la generación de código**.

DSLs desde los GPLs

Los GPLs son potentes porque permiten definir todos los problemas (Turing completo) pero en muchos casos son expresivamente pobres debido al **salto entre la definición del problema (mundo real) y su solución (código fuente)**. Esto hace muy complicada la programación y el mantenimiento porque se hace difícil entender lo que se pretende solucionar. Pongamos por ejemplo la definición de una interfaz de usuario web y su representación en HTML: el salto expresivo es enorme.

En base a esta necesidad surgen los DSLs.

Características y ventajas de los DSLs

- **Mayor nivel de abstracción**. Definen conceptos más complejos, más abstractos y por tanto más expresivos.

- Tienen **menos grados de libertad**. Normalmente no son Turing completos. Permiten definir el dominio, nada más que el dominio y con las reglas que rigen el dominio, lo cual les dota de una enorme potencia (en ese dominio, claro).
- **Aumentan la productividad** ya que permiten programar de una manera más rápida y eficiente.
- **Mejoran la calidad** del software. Abstraen de la complejidad técnica, generalmente resuelta por el generador de código, evitando errores.
- **Soporte IDE** (entorno de desarrollo integrado). Validaciones, comprobación de tipos, autocompletar, etc. Esto es una gran diferencia respecto a la definición del dominio mediante APIs o Frameworks.
- **Independientes de la plataforma**.
- En general todos las **ventajas de la generación de código**.

Los DSLs son comunes en el mundo real, a lo largo de la historia han sido creados en matemáticas, ciencia, medicina… es el momento de usarlos en el desarrollo de software.

7. Language Workbech

En anteriores publicaciones vimos lo que son los DSLs y por qué son necesarios y útiles en el desarrollo de software. Una vez que decidimos apoyarnos en ellos, nos encontramos ante la **necesidad de una herramienta que nos permitan diseñarlos y utilizarlos**. Esta herramienta se denomina técnicamente Language Workbench (LW).

Un LW está formado por dos partes fundamentales:

- Diseño del lenguaje.
- Uso del lenguaje. Programación.

Es posible que en el futuro la herramienta se divida en dos, de tal manera que, dentro o fuera de una organización, existirán dos roles perfectamente diferenciados: quienes diseñen el lenguaje y quienes se encarguen de utilizarlo, de programar en él.

Diseño del lenguaje

Un LW debe ser capaz de proveer las utilidades para definir las diferentes partes que forman el lenguaje:

- **Sintaxis abstracta**. La estructura gramatical/conceptual que define el lenguaje. Puede ser entendido también como el meta-modelo.
- **Sintaxis concreta**. La representación o representaciones visuales de dichos conceptos. Pueden ser representaciones en formato texto y/o gráfico. Para entendernos, es la definición de la interfaz visual con la que trabajará el programador.
- **Semántica estática**. Define aquellas restricciones o reglas que el lenguaje debe cumplir (aparte de ser sintácticamente correcto).
- **Semántica dinámica**. Sería sobre todo la traducción a lenguajes tradicionales aunque, como mencionaremos luego, aquí se encuentra el mayor potencial de esta metodología de desarrollo.

Uso del lenguaje

Una vez definidos los puntos anteriores, la herramienta es capaz de interpretarlos y **proveernos de un entorno de desarrollo (IDE)**. Según sea más o menos sofisticado, a parte de la edición, nos podrá proveer de utilidades como: autocompletar, validaciones estáticas, resaltar elementos sintácticos, mostrar diferentes vistas e incluso debug.

A parte de las características anteriores, este entorno nos permitirá generar código e incluso podrá dotarnos de un proceso de building para obtener el aplicativo final.

Potencial futuro

Todo lo comentado hasta ahora nos permite tener un proceso de desarrollo análogo al tradicional pero con las ventajas que ofrecen los DSLs y la generación de código, lo cual es un enorme avance en el que se apoyan los defensores e investigadores de esta metodología.

Estando de acuerdo en lo anterior, para nosotros el verdadero potencial, aun por descubrir, es el hecho de que **la programación deja de ser una mera declaración funcional y pasa a ser una representación del conocimiento**. Una vez que definimos los conceptos y sus reglas, la semántica puede ser capaz de ofrecernos muchos más servicios que la simple generación de código.

8. Representación del Conocimiento y Automatización de Software

La representación del conocimiento es una disciplina que persigue la **representación de la información del mundo real de una manera que pueda ser interpretada por las máquinas** para resolver, mediante inferencia, problemas complejos.

Tradicionalmente ha sido una disciplina de la inteligencia artificial y últimamente **ha adquirido gran relevancia por su utilización en el ámbito de la semántica**. El proyecto de la Web Semántica, liderado por la W3C, es un claro ejemplo de ello.

Aunque existen muchos enfoques para la representación del conocimiento, comúnmente todos persiguen: **definir los conceptos, las relaciones y las reglas que definen la información**. Mediante los diferentes conceptos podemos **clasificar la información** y mediante las relaciones y reglas podemos **inferir** (razonar) sobre ella. Por lo tanto, en vez de tener información “plana” tendremos además una meta-información que nos permitiría procesarla.

Desde la perspectiva de los lenguajes

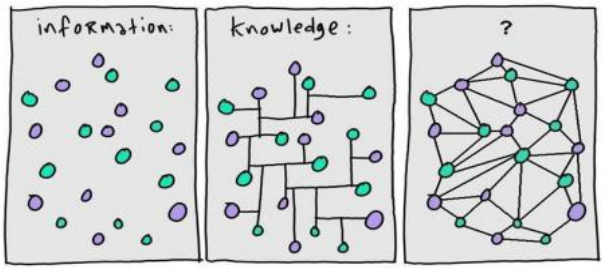
En anteriores publicaciones vimos como **los Language Workbenches definen los lenguajes mediante las sintaxis abstracta y concreta y las semánticas estáticas y dinámicas**. Si nos damos cuenta esa definición sería la **meta-información que hace que un programa sea una representación del conocimiento** y que, por tanto, podemos utilizar todo el potencial que esta disciplina nos aporta.

Se suele pensar en los Language Workbenches como generadores de código pero, enfocados desde el punto de vista de la representación del conocimiento y de la semántica, **pueden ofrecernos muchos más servicios**. Enumeramos algunos:

- Generar **baterías de pruebas**.
- Generar **cargas iniciales de datos**. Relleno de estructuras de bbdd para pruebas de rendimiento.
- **Auto-documentación**.
- **Auto-validación**.
- **Análisis estadísticos** de los datos y de los programas.
- **Inferir comportamiento** de los usuarios, clientes, etc.
- Facilitar la **importación y exportación** de datos. Por ejemplo XBRL: lenguaje de presentación de informes de negocio extensible.

- Enlazar con **ontologías estándar**. Por ejemplo FIBO: ontología del negocio financiero.
- Facilitar la **integración con otros sistemas**.
- **Razonamiento automático** (machine reasoning).

Conclusión



La representación del conocimiento esta siendo un área de investigación enfocada sobre todo en el tratamiento de los datos: estructurar la información de los buscadores, análisis semánticos aplicados al Big Data, definiciones de ontologías asociadas a diferentes negocios, etc.

Construir **metodologías de desarrollo que doten a los programas de esas capacidades es una puerta hacia el futuro con potenciales aun por descubrir**.

9. Diseñadores de Lenguajes

Una vez que disponemos de una herramienta para diseñar lenguajes de forma ágil, viene la tarea más difícil: diseñarlos.

El diseño de un buen lenguaje es la parte principal del proceso ya que **será la herramienta de los desarrolladores y lo que va a albergar la inteligencia semántica del sistema**.

Asimismo, dado que los conceptos que estructuran el lenguaje forman en sí mismo el modelo, **antes de diseñarlo es necesario conocer bien el dominio que se quiere modelar**.

Características de un buen lenguaje

- **Alto nivel de abstracción**. Mientras mayor sea el nivel, más potente será el lenguaje y mayor carga semántica tendrán sus conceptos. Asimismo, **un alto nivel de abstracción denota un alto conocimiento del dominio** que se modela.
- **Simple**. Debe ser fácil de utilizar y de leer. Un lenguaje **simple suele ser sinónimo de un alto nivel de abstracción**.
- **Diferentes niveles de complejidad**. A la vez que debe ser simple, también debe permitir vías para profundizar en el detalle por parte de aquellos que lo necesiten.
- **Estética agradable**.
- **Semánticamente potente**. Para que un lenguaje sea productivo simplemente es necesario dotar a los conceptos de su representación gráfica y su traducción a lenguajes tradicionales, pero si queremos que realmente sea completo, debemos dotar a los conceptos de más interpretaciones semánticas: auto documentación, auto validación, reglas de inferencia, etc.

Requisitos de un buen diseñador

A partir de las características de un buen lenguaje podemos inducir los requisitos:

- **Orientación a negocio**. Conocer bien el dominio para diseñar el lenguaje requiere un alto interés por conocer todos los procesos que lo rigen.
- **Capacidad de abstracción**. Conocido el dominio, se requiere una capacidad analítica que permita identificar, con el mayor nivel de abstracción posible, su más pura esencia.

- **Enfoque hacia la semántica**. El lenguaje ha de diseñarse con el objetivo de dotarle de una alta capacidad de representar el conocimiento.
- Cualidades enfocadas a la simpleza y la estética.

Conclusión

Este nuevo paradigma de desarrollo requiere un perfil particular para diseñar los lenguajes, donde no solo son importantes las antiguas **cualidades analíticas** sino que se han de añadir **cualidades de usabilidad y de representación del conocimiento**.

Inicialmente puede parecer complejo pero al fin y al cabo **forma parte de la evolución de la tecnología**, donde los perfiles que más aportan son aquellos que tienen mayor capacidad de abstracción.

10. Frameworks: la Media Naranja de la Generación de Código

Podemos definir un Framework (FW) como una **estructura software con funcionalidades genéricas las cuales pueden ser adaptadas o enriquecidas para obtener un aplicativo final**.

Comúnmente pueden ser confundidos con las librerías, pero se trata de un enfoque totalmente diferente. He aquí las principales características que definen esta diferencia:

- **La inversión de control**. El flujo de control lo define el FW y no el aplicativo que usa sus servicios. Esto también es conocido como el **principio de Hollywood**: *"no nos llames, nosotros te llamaremos"*.
- **Extensibilidad**. Algunas funcionalidades del FW no están "cerradas" como sucede en las librerías, al contrario están diseñadas para ser particularizadas según el problema particular que deban resolver.
- **Comportamiento predeterminado**. los FWs poseen un comportamiento global predeterminado, comportamiento que define el flujo de control. Como se ha visto antes, la extensión es lo que permite adaptar el comportamiento global al problema particular que cada aplicativo requiera.

Los frameworks son la **base de las arquitecturas plug-in** y de los sistemas enfocados a **ecosistemas de desarrollo** (facebook, twitter, amazon…), pero sobretodo son la arquitectura perfecta para hacer de base al código generado.

Técnicamente, dependiendo del lenguaje de programación que usemos, habrá varias formas de implementarlo. En lenguajes que lo permitan lo mejor es apoyarse en **clases abstractas, clases parciales, delegados y tipos genéricos**. En caso contrario se pueden usar otras estrategias como: intérpretes, funciones intermedias, tipo patrón visitor, que resuelvan la llamada y otros.

Ventajas en la generación de código

Estas son las características que aconsejan su uso:

- **Extensible**. Permiten realizar el trabajo en dos partes, lo cual facilita la definición del generador de código.
- **Estructurado**. Diseñar por un lado la parte genérica y en el generador la particular, permite tener una arquitectura mejor estructurada.

- **Seguridad**. Permite tener físicamente separados el código manual y el código generado, lo cual evita perdidas de código.
- **Reutilizable**. Una vez definido un FW podemos utilizarle en diferentes proyectos, facilitando nuevamente el desarrollo del generador.
- **Legibilidad**. La separación de lo genérico y lo particular facilita el entendimiento del código.

Conclusión

Cuando se aborda la generación de código se suele pensar que se va a generar el sistema al 100%. Una vez que se profundiza en ello se ve que es complejo además de ser un error de diseño y una distribución de trabajo desequilibrada.

Para conseguir un buen diseño arquitectónico y una buena distribución del trabajo **los frameworks se muestran como la media naranja ideal para nuestro sistemas generados**.

Por último, **es aconsejable que el código generado tenga también una estructura de framework**, esto nos permite poder implementar aquellas particularidades que no convenga automatizar y a su vez permite que nuestros sistemas puedan ser extendidos por un ecosistema de desarrolladores

11. Automatización: Proceso Completo

A lo largo de diferentes artículos hemos ido viendo el conjunto de elementos que forman la metodología, compilamos en éste todo el proceso.

Generación de código

A través del Language Workbench, una vez definidos los lenguajes (DSLs), realizamos el proceso de programación. Y a medida que vayamos avanzando iremos **generando código para compilarlo y ver el resultado en el aplicativo final**.

Siempre que utilicemos lenguajes que soportan clases parciales, clases abstractas, tipos genéricos, delegados, etc., podemos generar el código sin peligro ya que lo haremos en ficheros separados, sin riesgo de pisar código manual o de FWs. En caso contrario deberemos utilizar otro tipo de estrategias para evitar que esto ocurra.

Fusión con Frameworks

Como en el caso anterior, dependiendo de los lenguajes finales que utilicemos, la fusión será más o menos simple. **En el caso de lenguajes que soporten FWs** (clases abstractas, parciales, etc) no se requiere tal fusión porque **lo soluciona la propia sintaxis** del lenguaje.

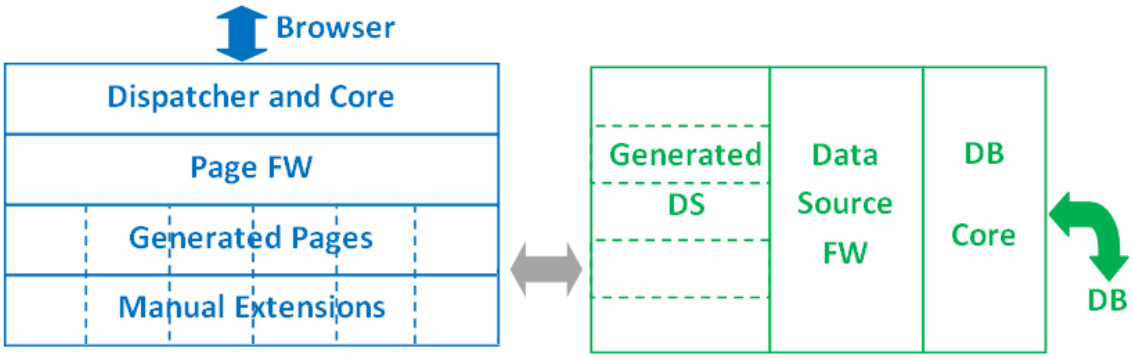
En otro tipo de lenguajes habrá que ajustar alguna funcionalidad o algún fichero de configuración, lo cual podrá ser también automatizado por el propio LW.

Extensiones manuales

En la mayoría de los casos **existirán ciertas funcionalidades que no conviene automatizar**, de lo contrario complicaríamos nuestros DSLs perdiendo, por tanto, parte de su potencia.

Lo ideal sería definir, en la propia herramienta, esos lenguajes de extensión teniendo integrado todo el desarrollo en la herramienta. En caso contrario podremos programar estas extensiones en el propio lenguaje final en el que generamos el código.

Si añadimos las extensiones en el lenguaje final, nuevamente será interesante buscar un enfoque de FWs, esto es, hacer que el propio código generado tenga una estructura extensible y así podamos tener las partes manuales en ficheros diferentes, evitando el riesgo de pérdida o substitución.



Componentes del servidor web

Building

Una vez que tenemos todo el código: FWs, generado, extensiones manuales, ficheros de configuración y otros, **un proceso automático se encargará de su fusión**, si fuera necesario, **de su compilación y de su publicación**.

Dependiendo del sistema operativo tendremos diferentes herramientas para hacerlo, aunque lo más común en todos ellos es usar **ficheros de instrucciones batch** que realiza todos los pasos del proceso.

Para concluir indicar que hemos detallado el proceso tal y como se encuentra en este momento, pero cabe resaltar que es una **tecnología en constante evolución y que persigue integrar todo el proceso en el propio LW**: lenguajes, debug, fusión, building, etc.

APPENDIX I – GLOSARIO

AOP

Aspect-oriented programming. La programación orientada a aspectos es un paradigma de programación que permite la separación de aspectos trasversales. Aspectos diferentes, tales como seguridad, log contable, avisos, etc. pueden ser definidos en áreas diferentes, obteniéndose el comportamiento final del sistema mediante a través de un proceso de *tejido*.

Dominio

Un dominio es un área, de negocio en nuestro caso, en el cual reside un conocimiento particular y una terminología específica para definirlo.

DSL

Domain Specific Language. Un lenguaje específico de dominio es un lenguaje de programación diseñado para representar los conceptos de un dominio dado. Puede ser tanto gráfico como texto.

FW

Framework. Un framework es una plataforma software reutilizable para desarrollar aplicaciones. Se trata de una abstracción donde sus funcionalidades genéricas pueden ser extendidas de manera selectiva para conseguir un aplicación software específica.

FM

Feature Model. Modelo para representar las partes comunes y opcionales que puede haber en una línea de producción. Es utilizado en PLE.

LW

Language Workbench. Herramienta que permite la definición y uso de lenguajes formales: sintaxis abstracta y concreta y semántica estática y dinámica. Se podría considerar como una herramienta para definir el lenguaje, el compilador y el IDE de un determinado dominio.

MDA

Model Driven Architecture. Estándar OMG para el desarrollo de software basado en modelos. Se puede considerar como un caso particular de MDSD.

MDSD

Model-driven Software Development. Se trata de un paaradigma de programación que basa el desarrollo de software a nivel de modelos.

Meta-modelo

Reglas, construcciones, relaciones, etc. que definen los modelos que pueden ser construidos en un dominio determinado.

PLE

Product Line Engineering. Disciplina para la creación de software a través de líneas de producción de software (SPL).

SPL

Software Product Lines. Metodología de desarrollo enfocada en la producción de software tal y como se hace en la producción de otros bienes: compartiendo los procesos comunes y aislando los particulares.

APPENDIX II – APLICACIONES WEB DE ESCRITORIO

Más allá de la publicidad o la identidad digital, las empresas cada vez más ven en la web la vía para mejorar su productividad y servicios: objetivos como **descentralizar la gestión (back offices) o dotar a los clientes de una actividad directa online** son ejemplos de ello.

Aunque la web no fue diseñada en origen para una interacción dinámica y bidireccional, **la aparición de nuevas las tecnologías y la mejora en los navegadores**, nos permiten diseñar aplicaciones cada vez más cercanas a las tradicionales aplicaciones de escritorio.

Estas aplicaciones web de escritorio **son diferentes a las páginas web y a los aplicaciones web** (e-commerce, por ejemplo), porque se enfocan a **usuarios que necesitan una interacción continua, prolongada y ágil**. Pongamos como ejemplo cualquier usuario de call center: atención al cliente, servicio técnico, gestión de recobros, etc.

Requisitos de una aplicación web de escritorio

- **Teclado**. Es indispensable que la aplicación pueda ser gobernada por el teclado y no solamente por el ratón. Tabulador, intro, flechas, etc.
- **Velocidad**. No solo por motivos de comodidad sino de productividad.
- **Seguridad**. Los datos con los que tratan estas aplicaciones son muy sensibles.
- **Sin efecto parpadeo**. La carga de contenido ha de ser parcial y dinámica, el refresco continuo de toda la página produce un efecto incómodo e insano.
- **Estética agradable y estándar**. Debe ser limpia y nunca recargada.

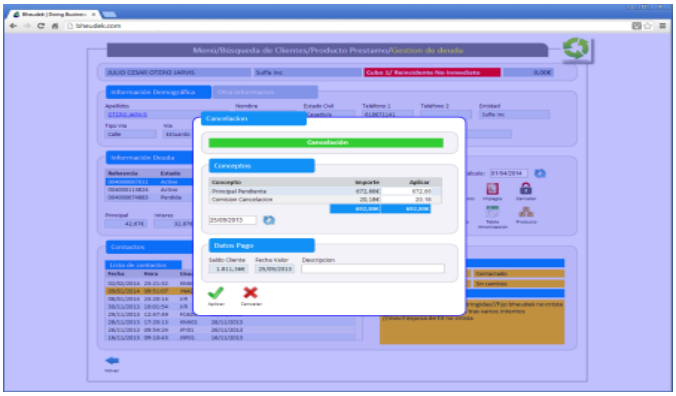
Consejos para conseguirlo

- **Patrón SPA** (Single Page Application). Se trata de un patrón de página web única donde los contenidos se van cargando dinámicamente. Permite políticas como carga inicial de ficheros (js, css, imágenes, etc), control de sesión compartido, mayor procesamiento en el cliente, etc.
- **Interfaz estándar** en todo el aplicativo. Las pantallas, los botones, las pestañas, los grids de datos y demás controles han de tener un aspecto y operativa común a lo largo del aplicativo.

- **Framework JavaScrtipt**. Necesario para estandarizar la interfaz y dotarla de los servicios para gestionar los controles y ventanas. Por motivos de seguridad y adaptabilidad se aconseja que sea desarrollo propio y encapsulado en un único objeto.
- Control de **eventos de teclado**. El framework debe implementarlos y asociarlos a los controles.
- **AJAX**. La vía para el dinamismo.
- **Minimizar uso de la red**. El trasiego de datos debe ser el mínimo, aunque eso suponga mayor procesamiento en el lado del cliente. Estándares de intercambio como JSON son aconsejables.
- **Pantallas modales**. Mejoran la experiencia de usuario y el dinamismo del aplicativo.
- Uso de **imágenes sprites**. Permiten cargar todas las imágenes de la aplicación al principio.

Un **desarrollo automatizado de la interfaz** de usuario apoyada en un **framework** js potente es el mejor camino para cumplir los objetivos anteriores.

Conclusión



Cuando hablamos de apps de escritorio hablamos de aplicaciones con unos requisitos muy exigentes ya que son la herramienta de trabajo en la que los usuarios pasarán gran parte de su tiempo.

Podrán seguir apareciendo nuevos dispositivos y nuevas maneras de interactuar con los sistemas, pero **cuando hablamos de interacción, requisitos tradicionales como el teclado, la velocidad, la seguridad y la ergonomía se hacen imprescindibles**.

APPENDIX III – PRIMER PRODUCTO: BHEUDEK FINANCE

Con el fin de acudir al mercado y poner a prueba nuestras herramientas, hemos desarrollado nuestro primer producto: un servicer financiero.

Basándonos en nuestra experiencia en proyectos similares podemos afirmar que, con nuestras herramientas, **¡Hemos reducido el tiempo total del proyecto en un 75%!**, mejorando no solo el tiempo de desarrollo sino también el de test y refactorización.

Menú principal.

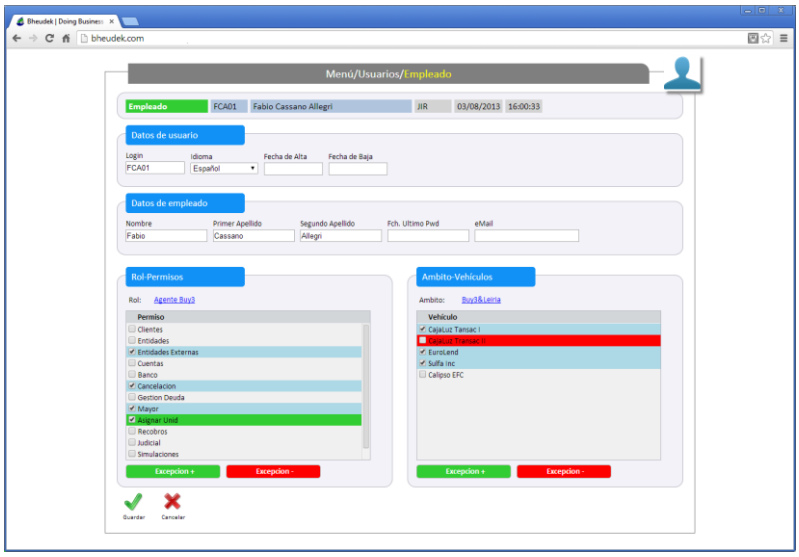
Características Funcionales

Como servicer financiero, estas son las funcionalidades principales de la aplicación:

- Migración de carteras.
- Gestión de clientes.
- Gestión de entidades, familias de productos y acuerdos.
- Ciclo de vida complete del préstamo: remesas, impagos, pagos parciales, recobros, fallido, etc.
- Gestation de recobros: amistoso, judicial y agencias externas.
- Mayor.
- Contabilidad.
- Gestión de usuarios, roles, permisos y ámbitos.
- Configuración: bancos, juzgados, simulación de préstamos, etc.

De manera más general queremos destacar estas características:

Multi-Ambito:similar a multi-compañia pero en diferentes niveles. La parametrización y la visión del usuario se pueden encuadrar a grupos de carteras, de compañías, de inversores, etc.

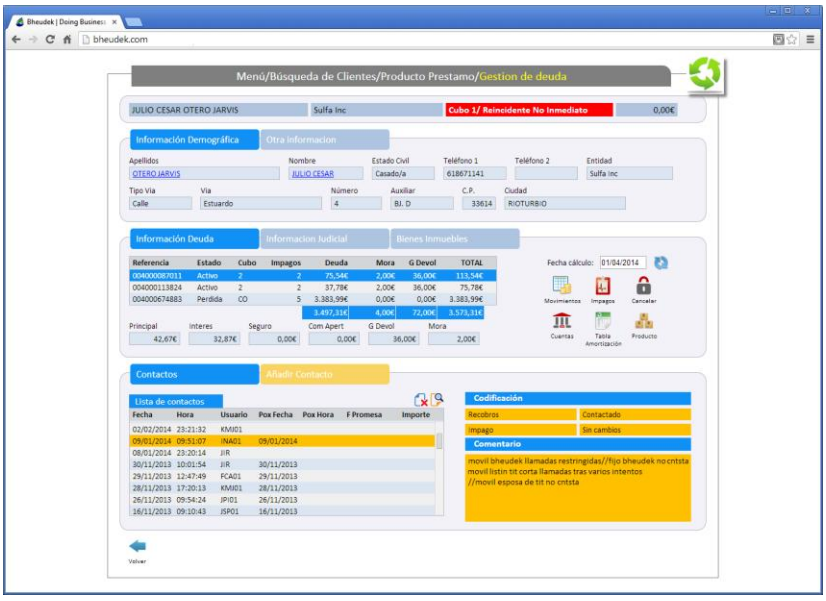


Página de gestión de usuario: selección de ámbito.

Estanqueidad: cada cartera se comporta a lo largo de la aplicación como un elemento aislado, sin existir filtraciones de datos o de procesos entre ellas.

Rastreable: se puede seguir el curso de cada entrada de dinero en el sistema.

Integrado: toda la gestión se integra en una única aplicación y a su vez, procesos que tradicionalmente eran independientes, han sido integrados en uno solo.

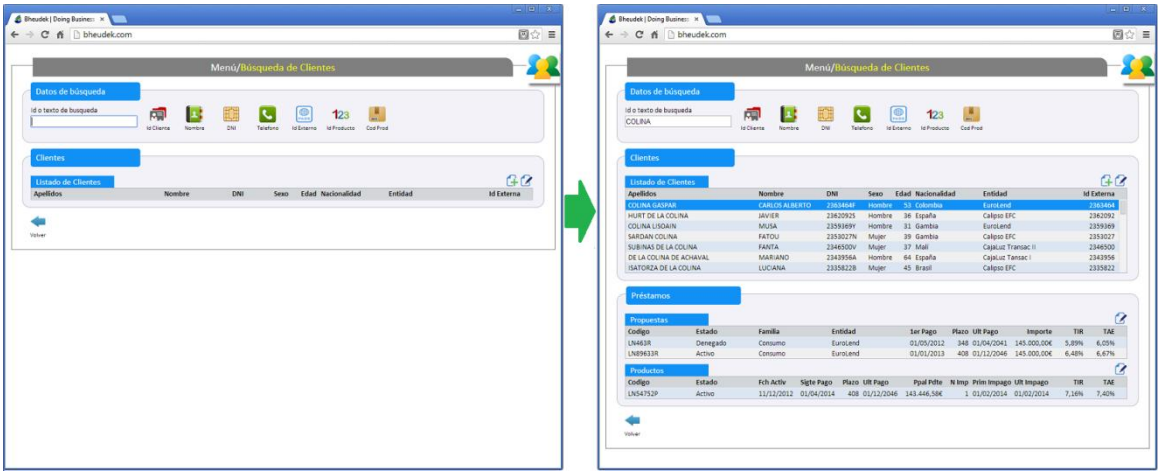


Proceso de recobros unificado: amistoso y judicial.

Caracteristicas Técnicas

Se trata de **aplicación web HTML5 basada en el patrón SPA (Aplicación de página única)**, lo que significa: carga dinámica de páginas sin el efecto blinking. Esta potente característica se consigue gracias a la velocidad de procesamiento de peticiones que ofrece el código generado.

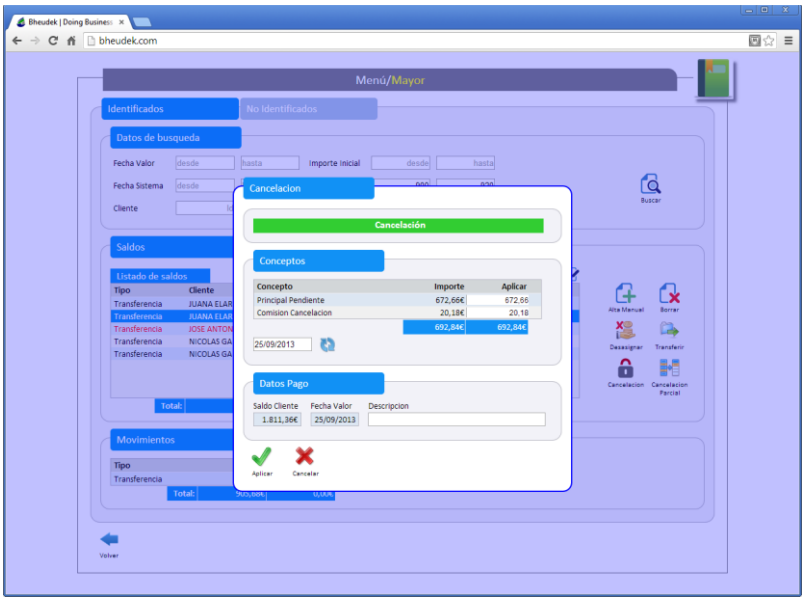
Una de sus características más importantes es que puede ser considerada como una **Aplicación Web de Escritorio**. Este tipo de aplicaciones web son aplicaciones tradicionales de escritorio pero con una interface web. Estas aplicaciones son diferentes a las páginas web y a los aplicaciones web (e-commerce, por ejemplo), porque se enfocan a usuarios que necesitan una interacción continua, prolongada y ágil. Pongamos como ejemplo cualquier usuario de call center: atención al cliente, servicio técnico, gestión de recobros, etc.



Ejemplo 1 Carga dinámica de la página

Estas son las características principales de una Aplicación Web de Escritorio:

- **Teclado**. Es indispensable que la aplicación pueda ser gobernada por el teclado y no solamente por el ratón. Tabulador, intro, flechas, etc.
- **Velocidad**. No solo por motivos de comodidad sino de productividad.
- **Seguridad**. Los datos con los que tratan estas aplicaciones son muy sensibles.
- **Sin efecto parpadeo**. La carga de contenido ha de ser parcial y dinámica, el refresco continuo de toda la página produce un efecto incómodo e insano.
- **Pantallas modales**. Mejoran la experiencia de usuario y el dinamismo del aplicativo.
- **Estética agradable y estándar**. Debe ser limpia y nunca recargada.



Ejemplo 2 – Ventana modal

Otra característica fundamental de la aplicación es la **seguridad**, la cual viene soportada por tener un único Framework JS y estar encapsulado en un único objeto y por las políticas de seguridad del servidor: sesión de testigo dinámico, validación de peticiones, validación de datos, permisos, etc. Este tipo de políticas globales que se implementan a lo largo de toda la aplicación se denominan cross-cutting concerns y son resueltas y garantizadas por el código generado.

APPENDIX IV – Beneficios de la Automatización del Software en el FinTech

El sector financiero se encuentra ante su mayor reto tecnológico: el fenómeno FinTech. Afrontar esta cambio, que implica a muchas áreas y tecnologías, requiere un enfoque global para evitar el caos operativo y tecnológico.
Ante este escenario, la automatización se presenta como la solución unificada que requiere el problema. Características como la semántica y la automatizacion de procesos ofrecen ventajas a los cambios que se abordarán en las diferentes áreas: servicios de negocio, servicios online y gestión de datos.

El sector financiero se halla sumergido en un proceso de renovación tecnológica. **Las oportunidades que ofrecen las nuevas tecnologías y la irrupción de nuevos actores**, muchos de ellos provenientes de otros sectores (Google, Amazon, Apple..), han obligado al sector a mover ficha.

El fenómeno FinTech está planteando **actuaciones en áreas muy diversas y con tecnologías muy diversas**, lo cual nos lleva a un escenario complejo donde se hace necesario que **el problema sea abordado desde una perspectiva global**, de lo contrario se corre el riesgo de entrar en un caos operativo y tecnológico.

Ante esta situación, las nuevas técnicas en la automatización del software se plantean como la mejor vía para afrontar este cambio.

> ***La automatización del software eleva la solución técnica al nivel conceptual en el que opera el negocio. Este punto de vista más elevado es el que permite dar un enfoque más global y unificado al reto que plantea el FinTech.***

Desde una visión más técnica, la automatización del software se apoya en la creación de lenguajes de programación más abstractos que permiten automatizar los detalles técnicos y dotar al sistema de una mayor riqueza semántica. En ámbitos técnicos, estos lenguajes se denominan DSLs y las herramientas para gestionarlos Language Workbench.

Ventajas

1.Automatización de procesos.

Trabajar en un nivel más abstracto permite automatizar procesos técnicos pero también a procesos tradicionales de negocio. Motores de decisión, ofertas basadas en riesgo, predicción de fallidos y detección del fraude son ejemplos de procesos que pueden verse mejorados bajo la perspectiva de la automatización.

2.Semántica.

La nueva forma de definir los lenguajes de programación, a partir de los conceptos que los estructuran, abre las puertas a laRepresentación del Conocimiento. Esta disciplina, tradicionalmente de la inteligencia artificial, se aplica cada vez más al tratamiento de datos (visualización, búsqueda, análisis, toma de decisiones …) y en el futuro será fundamental en cualquier proceso de análisis y de interacción, bien con clientes, bien con otros sistemas. XBRL (lenguaje de presentación de informes de negocio extensible) y FIBO (ontología del negocio financiero) son ejemplos de aplicación de la semántica en el sector financiaro.

3.Menor complejidad técnica.

Nuevamente, el mayor nivel de abstracción de los lenguajes nos permite resolver, de manera automática, gran parte de los detalles técnicos que requieren las diferentes soluciones. Conseguir automatizar esta complejidad redunda en la calidad, seguridad, estandarización y otras características del software.

4.Multi-plataforma.

Una vez que tenemos el dominio de los lenguajes de programación, un mismo desarrollo puede traducirse a múltiples plataformas, sin necesidad de tener que llevar a cabo un nuevo proyecto para cada nuevo dispositivo.

5.Orientación a negocio.

Los lenguajes de programación se hacen más cercanos al negocio, lo cual permite a los desarrolladores enfocarse en dar una solución desde el punto de vista propiamente del negocio y no desde una perspectiva técnica. A su vez esto derriba la barrera que ha existido siempre entre las áreas de desarrollo y el resto de áreas de la empresa.

Áreas beneficiadas

El FinTech actúa en múltiples áreas en las que la automatización puede aportar sus ventajas. Las agrupamos en tres bloque principales: servicios de negocio, servicios online y gestión de datos.

1.Servicios de negocio.

La demanda de nuevos servicios por parte del cliente y la búsqueda de mejoras en la productividad, está llevando al sector a actuar sobre sus estructuras más tradicionales. Los puntos en los que la automatización aporta ventajas son:

Descentralización. Para poder llevar a cabo la descentralización de servicios (call centers, back office, etc), es necesario migrar hacia el entorno web, lo cual resulta un reto importante porque los sistemas deben mantener las características de seguridad, velocidad y manejabilidad de las aplicaciones de escritorio tradicionales. La automatización nos permite crear, de manera eficiente, Aplicativos Web de Escritorio.

Gestión del portfolio. La automatización de procesos tradicionales nos permite una gestión más optimizada. Por otro lado, la semántica nos permite la gestión unificada de portfolios heterogéneos y la integración de portfolios externos.

Compliance. La semántica, junto con el uso de lenguajes orientados a negocio, facilitan los procesos de análisis y documentación de auditoría. A su vez, la automatización de procesos es la mejor vía para solucionar problemas de compliance.

Sistemas heredados (legacy). El mayor nivel de abstracción de los lenguajes nos permite diseñar, de manera simple, interfaces con los sistemas ya existentes.

2.Servicios online.

La aparición de nuevos dispositivos, la mejora en las comunicaciones y la nueva cultura digital han hecho que las entidades abran sus puertas para permitir una gestión más directa por parte de sus clientes. Nuevamente la automatización ofrece beneficios en éste área:

Accesibilidad. La multi-plataforma permite multiplicar las vías de acceso de los clientes a sus datos.

Promover la educación financiera. La semántica dota a los sistemas de unas capacidades de visualización y documentación que facilitan su manejo y entendimiento.

Gestión personal. Como resultado de las puntos anteriores, el cliente pasa a tener una gestión más personal de sus datos y sus activos. Como hemos dicho, la semántica juega un papel fundamental en la interacción con el cliente.

New Banking. Multi-plataforma, semántica y automatización de procesos permiten nuevas líneas de negocio como: ofertas personalizadas, plataformas de pago, cripto-monedas, análisis proactivos de riesgo y otros.

3.Gestión de datos.

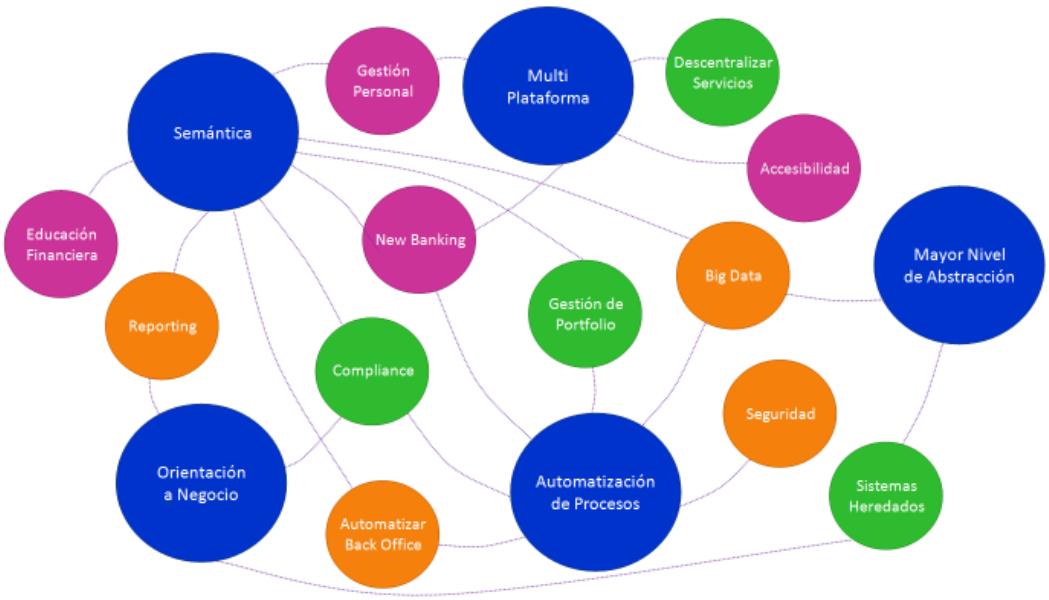
La interacción más directa con el cliente y los nuevos metodologías para procesar esa informacion han revolucionado el tratamiento de los datos y las tradicionales técnicas de CRM. En este último apartado la automatización también ofrece ventajas:

Automatización del back office. La automatización del back office pasa por la automatización de su workflow. Las posibilidades que ofrece la semántica en el diseño de motores de decisión más “inteligentes”, junto con la automatización de procesos, son líneas fundamentales para conseguir este objetivo.

Reporting. En general el reporting cada vez más tiende a la semántica, un ejemplo de ello es XBRL.

Seguridad. La automatización de procesos permite un tratamiento más seguro de los datos.

Big Data. Podemos resumir, ya que este punto requeriría un artículo entero, que la semántica, la automatización de procesos y un mayor nivel de abstracción nos permiten, por un lado, manejar la variabilidad de los datos y por otro, mitigar los riesgos que supone la desestructuración asociada al Big Data .



Conclusión

El sector financiero se encuentra ante uno de los retos tecnológicos más grandes de toda su historia, reto en el que se ven afectadas múltiples áreas y tecnologías.

La automatización del software aporta el enfoque general y unificado que requiere este escenario, de lo contrario, las actuaciones individualizadas pueden llevar a las compañías a un panorama tecnológico difícilmente gobernable.