Universidad de Buenos Aires

Facultad de Ingeniería

Laboratorio de Métodos de Desarrollo y Mantenimiento de Software (LIMET)

Proyecto: un planteo metodológico para la recuperación del software heredado basado en especificaciones mediante ejemplos y microservicios

Whitepaper Marzo 2018

Autor: Carlos Fontela

Índice de temas

1 Introducción

1.1 Resumen

El software heredado es un activo fundamental de las organizaciones de todo tipo. No obstante, debido que se desconocen sus funcionalidades, sus atributos de calidad o su arquitectura y diseño, el mantenimiento del mismo es muy costoso y riesgoso. A su vez, se suele tratar de sistemas que requieren cambios constantes, lo cual genera un círculo vicioso difícil de romper.

El presente trabajo plantea un enfoque basado en especificaciones con ejemplos y microservicios para ir descubriendo la funcionalidad y la arquitectura de sistemas heredados, y a la vez documentarlas con ejemplos de uso que sirvan de pruebas de regresión ante cualquier cambio evolutivo, sea una refactorización que no afecte el comportamiento o un cambio funcional. El uso de este enfoque permitiría ir refactorizando y haciendo evolucionar el sistema de manera más segura y económica.

1.2 Preguntas[1] que busca responder este proyecto

- ¿En qué situaciones es más económico mantener el software heredado que simplemente reescribirlo desde cero?
- ¿Existen enfoques probados que permitan recuperar el software heredado?
- ¿Puede el enfoque de Specification By Example servir para redescubir conocimiento alojado en sistemas heredados?
- ¿Es la arquitectura de microservicios una opción válida para reemplazar de manera incremental partes de un sistema heredado?

[1] Research questions.

2 El problema del mantenimiento y evolución del software heredado

2.1 Planteo del problema

2.1.1 A qué llamamos software heredado

Habitualmente, en la jerga informática, software heredado (o legado, o *legacy*, como se le dice en inglés) significa tanto antiguo como difícil de mantener. Siguiendo la primera acepción, hay quienes usan el término *legacy* para referirse a software desarrollado con tecnologías anticuadas [Sneed 2006].
En este trabajo nos referimos a la segunda acepción de la expresión, la que sostiene que el código heredado es aquél que es particularmente difícil de arreglar, mejorar y trabajar con él, ya que es frágil y casi imposible de extender[2] [Bernstein 2015].

Lo que ocurre es que existen productos de software que funcionan bien, en el sentido de que hacen lo que sus usuarios desean y sin errores. Pero en muchos casos se trata de productos que caen en alguna o varias de las siguientes categorías:

- Han sido desarrollados por equipos de trabajo que ya no están en el proyecto (productos huérfanos).
- No hay pruebas de aceptación ni técnicas que permitan probar el funcionamiento del código o entender cómo funciona (código inasequible).
- Tienen escasa documentación funcional, técnica y para usuarios (productos "flojos de papeles").
- Muchas funcionalidades son desconocidas por los desarrolladores y usuarios (productos tipo "caja de sorpresas").
- Mantienen un gran volumen de deuda técnica que impide probarlos y comprenderlos (Big Ball of Mud [Foote 1997]).

A su vez, como se trata de productos exitosos, suelen evolucionar, siguiendo la antigua máxima de Fred Brooks Jr. que dice que al software exitoso siempre se lo cambiará [Brooks 1975]. Con todo, esa evolución tiende a ser muy costosa y riesgosa, ya que se suele carecer de elementos que permitan conocer con suficiente precisión qué hace el sistema y de qué manera lo hace.
A esos productos los llamaremos software heredado.

[2] "…is particularly difficult to fix, enhance, and work with", "brittle", "nearly impossible to extend".

Definición: **software heredado**
Se trata de sistemas de software que han sido desarrollados y han alcanzado un grado de éxito relativo, pero resultan difíciles de mantener porque se ha perdido el conocimiento que permitiría hacerlo.

2.1.2 La necesidad de mantenimiento y evolución del software heredado

Hace décadas que se reconoce la importancia del mantenimiento y la evolución del software. Desde la década de 1960 se sostiene que el mantenimiento ocupa más de la mitad del esfuerzo económico del desarrollo [Era 1964]. Ya en 1969 Meir Lehman comenzó una serie de investigaciones formales que condujeron a la formulación de las leyes de Lehman [Lehman 1980, Lehman 1997].

Hay también un estudio del SEI[3] que pone el énfasis en los sistemas heredados como activos de las organizaciones, que han incorporado conocimiento a lo largo de la vida de las mismas [Comella-Dorda 2000].

Curiosamente, el SWEBOK[4] no aborda el problema [Bourque 2014]. Si bien define al mantenimiento como parte integral del ciclo de vida del software, y dice que su objetivo es la modificación mientras se mantiene la integridad del sistema, no dedica su atención al software heredado más que lateralmente.

Más allá de todos estos antecedentes y del tiempo transcurrido, que parecen darle al mantenimiento cierta relevancia, ni siquiera se acerca a la enorme cantidad de publicaciones dedicadas al desarrollo de nuevos sistemas. Tampoco en la industria se ha logrado una adopción de procesos o enfoques metodológicos para el mantenimiento acorde al volumen económico del mismo.

2.1.3 La complejidad del mantenimiento y la evolución

El software es una construcción compleja, por muchas y variadas razones. Sin embargo, hay una percepción generalizada de que modificar un sistema de software es algo sencillo. Esto no es así.

Tan complejo es mantener software que la literatura más antigua referida a los sistemas heredados plantea técnicas y procesos para el reemplazo liso y llano del software heredado [Brodie 1995, Seng 1999].

[3] Software Engineering Institute.

[4] Software Engineering Body Of Knowledge.

Uno de los mejores análisis de la complejidad del mantenimiento y la evolución del software se lo debemos a Peter Naur [Naur 1985]. Naur sostiene que programar es construir conocimiento[5], y como la construcción de conocimiento supera en complejidad a la simple manipulación de textos, las modificaciones a un sistema de software son mucho más difíciles que el simple cambio de un programa fuente escrito en texto.

La visión de Naur se complementa con la noción de que el principal problema para modificar software surge cuando el equipo que lo concibió ya no trabaja en el desarrollo: en este sentido, podríamos especular que sugiere que un programa se convierte en software heredado cuando cae en lo que hemos denominado productos huérfanos. Explica también que no alcanza con que un nuevo programador pueda comprender el código y otros tipos de documentación, ya que es imposible restablecer el conocimiento reunido durante el desarrollo mediante la simple lectura de textos, sino que debe trabajar con otras personas que estén en posesión del conocimiento[6], llegando al punto de sugerir que si no queda nadie del equipo original es probable que convenga reescribir el sistema desde cero.

Como veremos luego, Feathers es menos drástico: sugiere que lo que debemos reconstruir son las pruebas que nos dan pistas sobre cómo es el comportamiento del programa que tenemos entre manos [Feathers 2004].

2.1.4 A qué llamamos recuperación del software

Volviendo a Naur, él sostiene que si un producto ya no cuenta con el equipo que tenía el conocimiento que aquél representa, el programa debe considerarse muerto [Naur 1985]. Por lo tanto, lo único que queda por hacer es intentar revivirlo o recuperarlo con otro equipo de trabajo[7].

---

[5] En realidad, Naur dice "theory buiding", definiendo que "theory is understood as the knowledge a person must have in order not only to do certain things intelligently but also to explain them, to answer queries about them, to argue about them, and so forth".

[6] "who already possess the theory"

[7] Textualmente: "During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. The actual state of death becomes visible when demands for modifications of the program cannot be intelligently answered. Revival of a program is the rebuilding of its theory by a new programmer team.".

En nuestro contexto, los programas heredados caen mayormente en esta situación: están muertos[8], en el sentido que Naur le da a la palabra, pues si bien se pueden seguir usando, no se le pueden hacer modificaciones de manera segura.
Lo único que nos queda por hacer, entonces, es revivirlos[9] o recuperarlos para poder trabajar en su evolución.

Definición: **recuperación del software**
Es la acción o conjunto de tareas que permiten llevar un producto de software que se ha vuelto imposible de mantener a uno que vuelve a ser posible de modificar, principalmente mediante la reconstrucción del conocimiento contenido en él.

2.2 Soluciones propuestas

2.2.1 Primeras ideas

A lo largo del tiempo se han propuesto muchas soluciones.
El SEI viene estudiando a fondo el tema desde 1997 [Weiderman 1997]. En este primer trabajo aparecen algunas distinciones entre mantenimiento y evolución, diferenciándolos por la granularidad de los cambios y por sus beneficios económicos y estratégicos. También se propone que para entender el funcionamiento de los programas debería prestarse más atención al comportamiento externo de los módulos que al funcionamiento interno de los mismos. Las propuestas rondan en torno a proponer el uso del paradigma de objetos, con su carga de encapsulamiento que favorece el análisis del comportamiento observable y los sistemas distribuidos que fuerzan la necesidad de clarificar interfaces. Ese mismo año se propuso un framework que provee una guía para la evolución de sistemas heredados [Bergey 1997]. Algunas de estas recomendaciones van en línea con nuestra propuesta.
Unos años más tarde, el mismo SEI publica un estudio[10] de los distintos enfoques para modernizar sistemas heredados [Comella-Dorda 2000]. Aquí se estudian distintas opciones: la del mantenimiento, la modernización o la del reemplazo total. Dentro de la modernización, aboga por lo que llama modernización de caja negra[11], basada en envolturas[12], lo cual consiste

[8] El término "muerto" no nos parece adecuado para un producto en uso, pero es el que le da el autor.

[9] Así como no estamos de acuerdo con el término "muerto", tampoco hablaremos de "revivir" en este trabajo.

[10] "survey"

[11] "black-box modernization"

[12] "wrapping"

en rodear al sistema heredado con una capa de software que oculta la complejidad no deseada del antiguo sistema y exporta una interfaz moderna[13].
Recién en 2001 el SEI comienza a tratar la modernización incremental de sistemas heredados [Seacord 2001]. El planteo parte de la base de que el tamaño y la complejidad, sumado a la necesidad de mantener el sistema operativo en todo momento, hacen que la opción más sensata sea la incremental.
Incluso hay un informe de un proyecto que trata de la reingeniería de un sistema en el campo de las telecomunicaciones [Mosler 2006].

Hay muchos elementos de estos primeros trabajos que son valiosos para nuestra forma de ver el mantenimiento del software heredado:

- El foco en el comportamiento observable de los módulos por sobre el funcionamiento a menor granularidad.
- El planteo arquitectónico de envolver comportamiento con capas que expongan interfaces accesibles a las nuevas partes del sistema.
- La necesidad de ir mejorando la mantenibilidad del sistema a la vez que se mantiene la funcionalidad heredada.
- La necesidad de proceder con un enfoque incremental de reemplazo del software heredado por el nuevo.

2.2.2 Algunas propuestas de la comunidad ágil

El auge del movimiento ágil llevó a buscar planteos alternativos.
Feathers planteó un método para trabajar con código heredado que parte de la premisa de que el código heredado es aquél que carece de pruebas, refiriéndose mayormente a pruebas unitarias automatizadas [Feathers 2004]. Siguiendo el formato de los catálogos de refactorizaciones, plantea escenarios típicos y elabora técnicas para ir rompiendo dependencias en el código mediante la introducción de pruebas automatizadas que cubran dicho código. Algunas de las pruebas son de mayor alcance, pero el foco de Feathers está puesto en las pruebas unitarias.
Feathers, con su caracterización del código heredado como aquél que carece de pruebas, apunta en gran medida a la solución. Está en línea también con lo que planteaba Kent Beck de que uno de los objetivos de las pruebas unitarias en código es brindar información sobre el comportamiento de los objetos [Beck 1999].

[13] "surrounding the legacy system with a software layer that hides the unwanted complexity of the old system and exports a modern interface".

Feathers es el caso más conocido en la literatura moderna vinculada al movimiento ágil, al menos en ámbitos organizacionales. Pero hay otros dos planteos muy interesantes desde la academia.

Uno de ellos aborda un enfoque metodológico más centrado en la gente que en los documentos [Stevenson 2004]. La idea de trabajar en conjunto con usuarios, el planteo de tener herramientas que comparen resultados entre el sistema anterior y el migrado, y la noción de que las pruebas de aceptación automatizadas introducidas desde el comienzo del proceso podrían ser de gran ayuda – más allá de que ellos mismos sólo han probado trabajar con pruebas unitarias – va en línea con nuestro planteo.
Mucho más cerca de nuestra idea está el trabajo desarrollado en la Universidad Aldo Moro, de Bari, en el que tres investigadores en conjunto con dos alumnos de esa casa de estudios ponen a prueba un enfoque basado en pruebas de aceptación de historias de usuario [Abbattista 2009]. El enfoque, planteado con el nombre de Storytest-Driven Migration (STDM) plantea un enfoque ágil y el trabajo con pruebas de aceptación, a la manera de especificaciones ejecutables, que deben funcionar tanto en el sistema heredado como en su reemplazo. Hay un fuerte énfasis en la revisión de las especificaciones por todos los interesados.
También hubo otros intentos de abordar el tema, como el de Bernstein [Bernstein 2015], pero en vez de decirnos cómo trabajar con el código heredado, se limita a presentar nueve prácticas basadas en Extreme Programming, Scrum y Lean para desarrollar software que sea fácil de mantener y evite convertirse en código heredado.

2.3 Fragilidad de las soluciones propuestas

2.3.1 Fragilidad del enfoque de tirar y reescribir

Una opción que se plantea a menudo cuando hay que realizar grandes cambios en sistemas en producción es la de tirar y reescribir. Como decíamos, hasta hace un par de décadas era el enfoque más habitual, y aún hoy en día sigue siendo una opción que se analiza como válida.
Hay quienes defienden este enfoque diciendo que reimplementar las funcionalidades tal como están en el sistema heredado es generar nuevo código heredado, sin valor para el negocio [Stevenson 2004]. Así planteado, parece una ligereza: si el sistema se está usando es porque la mayor parte es correcta, al menos desde el punto de vista funcional. Aun cuando reconocemos que en el software heredado puede haber características que ya no tengan sentido: por ejemplo, porque se hicieron para un cliente que ya no las usa, o simplemente parecieron buenas ideas que luego se dejaron de lado y quedaron en el producto.

La ventaja real que acarrea esta práctica es que podremos construir el sistema completo desde cero siguiendo buenas prácticas de especificación, diseño y construcción, de lo cual se espera que surja un producto de mejor calidad. Sin embargo, esta suele ser una solución poco factible. Salvo que se requiera un sistema totalmente diferente o que las tecnologías que se usaron para desarrollarlo estén muy obsoletas, el costo y el tiempo de desarrollo hacen impracticable esta solución. A eso hay que sumarle que, en general, los clientes no están dispuestos a esperar el desarrollo de un nuevo sistema sin seguir pidiendo cambios al anterior, lo cual convierte al nuevo producto en un blanco móvil. Finalmente, estos enormes emprendimientos son difíciles de estimar, con lo cual las fechas de puestas en producción resultan poco fiables.

2.3.2 Fragilidad del enfoque documental

Algunos de los primeros planteos de mantenimiento de software heredado, al sostener que lo que hay que comprender es el comportamiento observable, dicen que habría que recurrir a los manuales de usuario y a la documentación de las APIs[14] [Weiderman 1997].

Creemos que es un planteo muy riesgoso. Uno de los problemas habituales de la documentación que ha sobrevivido separada del código mucho tiempo es que no está actualizada o en sincronía con el código. Pasa con los modelos de arquitectura, con la documentación de interfaces, y hasta con los manuales de usuario poco frecuentados.

Resulta por lo tanto peligroso dar por buena a la documentación, siendo que sabemos que el sistema funciona bien, y que lo que funciona bien está representado en el código, no necesariamente en documentos.

2.3.3 Fragilidad del enfoque basado exclusivamente en pruebas técnicas

El enfoque de Feathers ya visto es suficiente cuando el problema es la documentación del uso y funcionamiento de pequeñas porciones de código. Sin embargo, lleva a micro mejoras[15], sin que los problemas de diseño arquitectónico terminen de ser abordados del todo.

Nuestro proyecto sostiene que se necesita mucho más que pruebas unitarias para definir el comportamiento de un sistema mediano o grande. Necesitamos contar con especificaciones de características, por ejemplo en la forma de historias de usuario acompañadas de casos de aceptación. Asimismo, precisamos saber cómo probar el sistema para saber qué es lo que el usuario espera de él, y para eso podríamos volver a usar los casos de aceptación. También

[14] API significa Application Programming Interface o “interfaz de programación de aplicaciones”

[15] enhancements in the small

sería deseable tener alguna forma de descripción de la arquitectura, lo cual podría venir de la mano de pruebas de integración.

En definitiva, creemos con Feathers que las pruebas son necesarias, y nos parece que su trabajo es pionero en cuanto a que señala una dirección correcta en general. Pero también creemos que con pruebas unitarias solamente no nos alcanza.

Otra razón para desechar el uso de pruebas unitarias es que, precisamente en el software heredado, si bien sabemos que el sistema como un todo cumple su función satisfactoriamente, no podemos decir lo mismo de cada porción de código y de las implementaciones particulares de las funcionalidades, que es lo que prueban las pruebas unitarias y técnicas de integración. Por eso necesitamos pruebas que evidencien el comportamiento desde la perspectiva de interacciones reales de usuarios.

3 Especificaciones mediante ejemplos al rescate

3.1 Especificaciones mediante ejemplos en la historia del desarrollo de software

En general, cuando especificamos el comportamiento esperado de un producto de software solemos usar un enfoque descriptivo, como casos de uso, historias de usuario o especificaciones complementarias. Pero también hace ya mucho que se utilizan ejemplos en las especificaciones, en general acompañando especificaciones descriptivas.

Cuando Kent Beck presentó Extreme Programming hizo especial hincapié en especificar mediante historias de usuario acompañadas por pruebas de cliente[16] [Beck 1999]. Está bastante claro en sus primeras publicaciones que Beck se refiere a pruebas de aceptación. Pero con el tiempo, la práctica fue derivando en lo que se conoció como TDD[17], basado en pruebas unitarias que sirven como especificación del comportamiento de pequeñas porciones de código [Beck 2003].

A nuestro juicio, esta transición ha sido desafortunada, ya que – si bien es una buena práctica guiar el diseño mediante pruebas unitarias – ha minimizado la importancia de las pruebas de aceptación para derivar el comportamiento del sistema.

Con el tiempo, fueron surgiendo algunas prácticas que buscaron subsanar este problema. Bastante tempranamente, Ward Cunningham y Ken Auer comenzaron a usar la expresión *Agile Acceptance Testing* para implementar las pruebas de cliente de las que hablaba Beck [Reppert 2004]. He analizado en detalle estas prácticas, que al día de hoy se presentan bajo diferentes nombres, en un trabajo de 2012 [Fontela 2012]. Lo que sigue es una simple enumeración:

- BDD (Behavior Driven Development) [North 2006]: inicialmente surgió como una práctica para hacer bien TDD y fue convergiendo a una manera de especificar el comportamiento esperado mediante escenarios concretos que se puedan automatizar como pruebas de aceptación.
- STDD (Storytest Driven Development) [Mugridge 2008]: es una práctica que pretende construir el software basándose en ir haciendo pasar las pruebas de aceptación – que se pretenden automatizadas – que acompañan las historias de usuario.

[16] "customer tests"

[17] Test-Driven Development.

- ATDD (Acceptance Test Driven Development) [Koskela 2007, Gärtner 2012]: similar a la anterior, construye el producto en base a pruebas de aceptación de usuarios, con menos énfasis en la automatización de las pruebas y más en el proceso en sí.
- SBE (Specification By Example) [Adzic 2009, 2011]: presentada originalmente como una práctica para mejorar la comunicación entre los distintos roles de un proyecto de desarrollo, en especial entre los desarrolladores y los clientes y usuarios, ha ido convirtiéndose en una práctica colaborativa de construcción basada en especificaciones mediante ejemplos que sirven como pruebas de aceptación.

Desde el punto de vista del uso, y de este informe, vamos a considerar a las cuatro prácticas como sinónimos, sobre todo porque en todos los casos se parte de ejemplos que sirven tanto como especificaciones del usuario, guías para el desarrollo y casos de aceptación a probar, que es lo que nos importa aquí.

3.2 SBE en la práctica

3.2.1 Las historias de usuario conducen el desarrollo mediante sus criterios de aceptación

Las historias de usuario[18] son un artefacto introducido por Extreme Programming para representar los requerimientos [Beck 1999].

La idea de SBE es tomar una historia de usuario, determinar los criterios de aceptación para, a partir de ellos, derivar las pruebas de aceptación, y luego escribir el código. De esta manera, se va construyendo el sistema de manera incremental en base a historias de usuario que tienen valor para el cliente.

El éxito de los criterios de aceptación es el que nos permite conocer mejor cuándo se ha satisfecho un requerimiento, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de la historia de usuario están funcionando. Adicionalmente, da una visibilidad mayor de los avances parciales para ambos.

Lo importante del uso de historias de usuario es que éstas son requerimientos simples y no representan el cómo, sino solamente quién necesita qué y para o por qué. Al fin y al cabo, los interesados no técnicos se sienten más cómodos priorizando características u objetivos de negocio que tareas específicas de bajo nivel o cuestiones de diseño de software.

Sin embargo, las historias de usuario representan el requerimiento, no lo especifican [Cohn 2004]. Las pruebas de aceptación de una historia de usuario funcionan como complemento de

[18] User story es el término en inglés, que a menudo también se usa en castellano.

la misma, al convertirse en sus condiciones de satisfacción. Por eso es que aparecen estas últimas guiando el desarrollo.

3.2.2 Enfoque de afuera hacia adentro

SBE trabaja mediante un enfoque de afuera hacia adentro[19], partiendo de funcionalidades y sus casos de aceptación para luego ir refinando el diseño. En definitiva, por cada prueba de aceptación que debemos encarar, hay que hacer varios ciclos de desarrollo basados en pruebas unitarias y de integración técnicas.

Por supuesto, al ir avanzando el desarrollo, estas pruebas técnicas van a terminar cubriendo el mismo código que las pruebas de aceptación que las motivaron. De todas maneras, esto dista de ser un problema, pues la coexistencia de pruebas a distintos niveles va a facilitar el mantenimiento del sistema, sea cuando hay que realizar cambios de comportamiento o cuando haya que hacer grandes refactorizaciones [Dascanio 2014, Fontela 2013-2]. Por eso, es una buena idea mantener todas las pruebas, aun cuando haya redundancia en la cobertura: a lo sumo, si muchas pruebas implican mucho tiempo de ejecución de las mismas, se puede no ejecutar todas las pruebas en todas las integraciones, siguiendo las ideas de Test Impact Analysis [Hammat 2017].

3.2.3 Ejemplos como casos de aceptación

La idea subyacente en SBE es usar ejemplos como parte de las especificaciones, y que los mismos sirvan para probar la aplicación, haciendo que todos los roles se manejen con ejemplos idénticos.

Al fin y al cabo, habitualmente un analista de negocio escribe especificaciones, que le sirven a los desarrolladores y a los testers. Los desarrolladores construyen su código, en principio basándose en las especificaciones recibidas, y también definen pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas. Los testers, por su lado, desarrollan casos de prueba, que contienen a su vez ejemplos. En algunas ocasiones, para entender mejor las especificaciones y evitar el síndrome del teléfono descompuesto[20], los desarrolladores y los testers le solicitan a los analistas que les den ejemplos concretos para aclarar ideas. E incluso hay ocasiones en que los propios analistas proveen escenarios, que no son otra cosa que requerimientos instanciados con ejemplos concretos.

En definitiva, hay varias ocasiones en que se plantean ejemplos que, de alguna manera, podrían servir como complementos de requerimientos. De allí que se haya pensado en

[19] "outside-in"

[20] "Telephone game" o "Chinese whispers" en inglés.

especificar directamente con ejemplos, o al menos acompañando los requerimientos con ejemplos.

Estos requerimientos con ejemplos tienen ciertas ventajas, de las cuales las más importantes son:

- Sirven como herramienta de comunicación.
- Se expresan por extensión, en vez de con largas descripciones y reglas en prosa, propensas a interpretaciones diversas.
- Son más sencillos de acordar con los clientes, al ser más concretos.
- Evitan que se escriban ejemplos distintos, una y otra vez por roles distintos, con su potencial divergencia.
- Sirven como pruebas de aceptación: más aún, las pruebas de aceptación son la especificación.

3.2.4 Roles y talleres de especificaciones

En el enfoque tradicional, los distintos roles están establecidos de la siguiente manera:

- El analista de negocio hace las veces de "traductor" entre los clientes, por un lado, y los desarrolladores y testers, por el otro. Escucha expectativas, deseos y necesidades, y con ellos elabora especificaciones de requerimientos y los prioriza.
- El tester es quien valida el producto contra las especificaciones. Recibe especificaciones, elabora casos de prueba en base a escenarios y ejecuta casos de prueba. En general no habla directamente con el cliente, sino que usa como interlocutor al analista.
- El desarrollador es el diseñador y constructor del producto. Recibe especificaciones, elabora un diseño y lo plasma en código, para luego ejecutar pruebas unitarias y de integración técnicas. Como ocurre con el tester, tampoco habla directamente con el cliente, sino a través del analista.
- El cliente o usuario, por su lado, sólo habla con el analista.

En cambio, una de las recomendaciones básicas de SBE es que las especificaciones se construyan en talleres multidisciplinarios, de los cuales participan todos los interesados, con especial énfasis en los roles de usuario, analista de negocio, desarrollador y tester. La participación de distintos roles en una discusión abierta hace que los ejemplos que surjan del taller sean más ricos y cubran una mayor cantidad de casos particulares. Por eso, en el enfoque de SBE:

- El analista de negocio debe ser un facilitador de intercambio de conocimiento. Hay ocasiones en que se puede prescindir de este rol, si tenemos algún usuario que pueda ejercerlo.

- El desarrollador, además del experto técnico, es uno más de los participantes en la elaboración de los escenarios y ejemplos, que luego le sirven para desarrollar el producto y las pruebas técnicas.
- El tester es otro de los participantes en la elaboración de los escenarios y ejemplos, que luego le sirven para probar la aplicación.
- El usuario tiene un rol más activo, como autor principal y dueño de las pruebas de aceptación con ejemplos, asistido por el resto de los interesados.

En algunos enfoques, se reducen los roles a tres, pudiendo el usuario y el analista de negocio la misma persona [Pugh 2010].

En cuanto a lo metodológico, SBE propone realizar un taller por iteración. Estos sirven para intercambiar ideas entre los participantes, además de establecer un lenguaje común más cercano al del negocio. También en estas sesiones surgen requerimientos implícitos y aquellos que nadie hubiese contemplado o que se dan por sobreentendidos, aparte de que se reduce la brecha entre lo que entiende uno y otro participante. El propio formato de taller hace que haya una revisión implícita y que se tenga más presentes a los requerimientos que cuando hay que leerlos de un documento escrito.
Siguiendo la premisa principal de TDD, no deberían desarrollarse funcionalidades que no estén acompañadas por una prueba de aceptación. Y si durante una prueba manual, o en producción, surgiera un problema, habría que analizar qué pasó con la prueba de aceptación correspondiente antes de resolverlo.
En definitiva, SBE es un conjunto de prácticas[21] que buscan construir el producto que el cliente espera basándose en realizar especificaciones precisas con ejemplos concretos que sirvan a todo el equipo de trabajo y otros interesados. De este modo, todos entienden de la misma manera qué hay que construir, sin ambigüedades y sabiendo cuándo una pieza de trabajo está terminada.

3.2.5 Lenguaje ubicuo

Dijimos que uno de los objetivos de los talleres de requerimientos que plantea SBE es establecer un lenguaje común más cercano al del negocio. Ese lenguaje es el que usan los participantes del taller, luego los testers al escribir los casos de aceptación y los programadores en su código. En ese sentido, se parece al lenguaje ubicuo[22] que plantea

[21] [Adzic 2011] dice que es un "set of key process patterns".

[22] "ubiquitous language"

DDD[23] [Evans 2004]. Evans sugiere que el éxito de un proyecto está fuertemente relacionado con la calidad del modelo de dominio, que a su vez está soportado por un lenguaje ubicuo que se usa en ese dominio. De esta manera, el lenguaje de los interesados es el que se utiliza durante todo el desarrollo, desde los requerimientos hasta las pruebas, pasando por los nombres de clases, métodos, funciones, etc., de modo tal que se convierte en un proceso de construcción de conocimiento [Park 2011].

Ahora bien, ¿qué es este lenguaje ubicuo?

Se trata de una serie de conceptos clave que definen el dominio y el diseño. El dominio, para evitar las confusiones en el uso de términos similares que significan cosas distintas. Respecto del diseño, estos conceptos, por un lado definen una jerga del proceso de desarrollo, y por el otro van a servir para dar nombres a las clases, métodos, módulos, o cualquier artefacto del diseño.

En realidad el lenguaje ubicuo lo define el propio equipo del proyecto: no es necesariamente el lenguaje del negocio, ni un estándar de la industria, ni la jerga que usan los expertos, sino un conjunto de términos, con sus propiedades y significados, que utiliza el equipo de desarrollo. Ello se debe a que es el lenguaje colaborativo que usa el equipo para capturar los conceptos y términos de un dominio y lograr una comprensión compartida[24]. Por eso no es ubicuo para todo un proyecto ni una organización, sino solamente dentro del equipo del proyecto.

Tampoco se trata de un lenguaje estático: el lenguaje ubicuo puede ir cambiando con el tiempo en la medida que el equipo así lo considere. Lo importante es que, ante cambios en el lenguaje, estos se deberán ver reflejados en los nombres que usamos para los distintos artefactos del proyecto, incluyendo el código.

3.2.6 Automatización

Notemos que hasta ahora no hemos afirmado nada de la automatización de los ejemplos. Si bien esto suele hacerse, y en este trabajo vamos a apuntar en ese sentido, la técnica tiene su fundamento principal en la mejora de comunicación usando el lenguaje del negocio, y que ello llegue al diseño y al código.

No obstante, si bien pusimos el énfasis en que no necesariamente las especificaciones por extensión deben automatizarse, es interesante poder hacerlo, ya que eso las convierte en especificaciones ejecutables, mucho más económicas de ejecutar una y otra vez como pruebas

[23] Domain Driven Design.

[24] “shared understanding”

de aceptación, con la ventaja de que en este caso estamos partiendo de genuinos criterios de aceptación.

3.2.7 Formatos y herramientas

Hay diversos formatos que se usan para realizar las especificaciones en el marco de SBE: tablas, texto libre, guiones[25] y código. Cada uno de ellos tiene ventajas e inconvenientes, defensores y detractores, y hay herramientas para automatizar todas ellas.

Las tablas resultan una notación usual para representar entradas y salidas calculables, aunque son definitivamente inadecuadas para representar flujos de tareas, y requieren bastante trabajo de programación para generar pruebas automáticas. Entre las herramientas más populares que usan este formato están Fit y FitNesse.

El texto libre tiene el mismo problema de la dificultad de generación de código de pruebas, a lo que se suma una menor riqueza en la comunicación de entradas y salidas. Por otro lado, tiene la ventaja de que todos los perfiles pueden entenderlo. Existen algunos lenguajes de especificación, entre los que destaca Gherkin, que facilitan la generación del código a costa de establecer una estructura un poco menos libre. Algunas de las herramientas más utilizadas son Cucumber y RSpec.

Existe también la posibilidad de expresar los criterios de aceptación a través de guiones que simulen el comportamiento de un humano utilizando el sistema. Si bien este formato permite la prueba del sistema de punta a punta[26] y utilizando el sistema real, tiende a generar pruebas frágiles que se rompen al cambiar la interfaz de usuario. Además, poner el foco en el comportamiento y en pruebas de comportamiento permite aislar cuestiones de interfaz que a veces se corresponden con una sola plataforma[27]. La herramienta más utilizada, en el mundo de las aplicaciones web, es Selenium. También existe Selenium Webdriver, una API que permite escribir código en varios lenguajes de programación y que genera pruebas basadas en guiones[28] como Selenium. Su uso o no hay que determinarlo con cuidado: es cierto que son más frágiles y que pueden estar probando comportamiento más de una vez por cada plataforma de interfaz de usuario, pero también lo es que es el tipo de pruebas que mejor entienden los usuarios.

---

[25] "scripts"

[26] "end to end"

[27] Por ejemplo, para el caso de un sistema que debe tener interfaz web y móvil a la vez: las pruebas orientadas al comportamiento van a ser las mismas; las orientadas a la interfaz, no.

[28] "scripted tests"

En cuanto a las especificaciones en código, su obvia ventaja radica en la facilidad de generación de pruebas automáticas, a veces incluso a un costo nulo. Y si bien son las menos adecuadas como herramientas de comunicación y provocan una resistencia casi instintiva de los roles no técnicos, existen algunas extensiones a las últimas versiones de los framework xUnit que permiten simular el formato tabular más amigable (ver anexos).

3.3 El germen de una idea

Dado que en la mayor parte de las situaciones en que se deben mantener sistemas heredados el principal problema es la reconstrucción del conocimiento, se nos ocurrió que la práctica de especificar con ejemplos, con su carga de actividad colaborativa y de descubrimiento de requerimientos, era una posibilidad cierta para el abordaje de este problema.

Markus Gärtner nos dio otra pista, al sostener que las pruebas son el activo más precioso en cualquier proyecto de desarrollo de software: no sólo muestran que el sistema sigue siendo útil de algún modo, sino que también documentan lo que el sistema hace[29] [Gärtner 2012].

En la tesis de maestría de Carlos Fontela se aborda tangencialmente el tema, al plantear que la única manera segura de hacer grandes refactorizaciones es tener pruebas de aceptación que sirvan como última red de contención que asegure la preservación del comportamiento [Fontela 2013]. Pero lo que no aborda dicha tesis es lo que ocurre si esas pruebas de aceptación no existen.

Hay dos trabajos finales de carrera dirigidos por Fontela que también exploran este escenario. En especial, la tesina de Nicolás Dascanio muestra que el impacto de cambios realizados sobre sistemas que sólo tienen pruebas unitarias es mayor que cuando se realizan sobre sistemas con pruebas de aceptación [Dascanio 2014]. Pero – nuevamente – no explora cómo proceder en el caso de que no haya pruebas de aceptación.

El que más explícito hace el planteo de usar especificaciones con ejemplos para la evolución de software heredado es Ken Pough, al decir que, antes de hacer un cambio sobre el sistema, hay que crear pruebas de aceptación alrededor de la porción del sistema involucrada con el cambio[30] [Pugh 2010]. Incluso nos recuerda que las pruebas de aceptación documentan cómo funciona realmente el sistema[31]. Sin embargo, como pasa en tantos otros casos, queda como una enunciación sin ejemplos concretos ni evidencia empírica de haberlo llevado adelante.

---

[29] Textualmente: “working tests are the most precious asset in any software development project. They not only show that your system is still useful in some way, but they also document what the system does”.

[30] “before making a change, create acceptance tests around the portion of the system that is involved with the change”

[31] “acceptance tests document how the system currently works”

3.4 Ejemplos como especificaciones para recuperar software heredado

Nuestra propuesta consiste en ir acotando el sistema existente por funcionalidades, las cuales se deberán especificar con ejemplos que sirvan como casos de aceptación, y luego proceder a hacer los cambios.

Siguiendo el procedimiento que indica SBE, estos ejemplos deberían surgir de talleres en los cuales participen usuarios, analistas de negocio, programadores y testers.

Es decir, cada funcionalidad o grupo de funcionalidades separables del sistema deberán ser analizadas en talleres y construir ejemplos que sirvan de documentación para luego refactorizar o cambiar el sistema. Esos mismos ejemplos servirán luego para probar la aplicación.

3.5 Un problema que subsiste

Ahora bien, si el sistema con el que estamos trabajando es mediano o grande, parece evidente que no va a ser posible encarar una documentación exhaustiva de la totalidad del mismo antes de atender los nuevos pedidos de los usuarios. En efecto, se supone que el sistema está vivo[32] y que requiere constantemente cambios y nuevas características. Por lo tanto, resulta ingenuo pretender que el cliente esté dispuesto a asumir los costos de un trabajo largo de documentación a la vez que se mantiene congelado el producto.

Por eso, siempre debemos tener en cuenta es que todo cambio tiene que tener valor para el negocio: si no, no se va a hacer. Más aún si pretendemos que haya analistas de negocio y usuarios que participen en talleres de requerimientos.

Un enfoque pragmático sería aplicar el método solamente por demanda, entregando valor desde el principio[33]. Es decir, a medida que surgen cambios en algunas funcionalidades del sistema, reunir a los distintos interesados en el taller, construir ejemplos para esas funcionalidades, e ir implementando los cambios. Este enfoque tiene varias ventajas: vamos a lograr mayor compromiso de los usuarios, estaremos trabajando e invirtiendo dinero solamente en lo que es prioritario y no invertiremos esfuerzo en partes del sistema que no haya que cambiar. En el límite, puede que haya partes del sistema que queden sin tratar, pero sería porque no fue necesario cambiarlas tampoco, sea porque funcionan bien como están o porque no se usan demasiado.

Esto implicaría que el reemplazo del sistema se debería hacer por partes, de manera incremental. De alguna manera, el sistema como un todo queda igual y se les van extrayendo

[32] Por esto es que no coincidimos con [Naur 1985] y su caracterización del sistema como “muerto”.

[33] “delivering value from day one”

partes que interactúan con el sistema viejo a través de alguna API. La ventaja de este enfoque es que las partes del sistema que más se usen y más cambios requieran van a ir mejorando su calidad poco a poco, con lo que cada vez va a ser más sencillo mantener las partes más críticas del sistema.

Esto de recortar el sistema por funcionalidades y de hacer interactuar las partes nuevas con el viejo sistema a través de APIs nos hace planearnos una arquitectura que debería favorecer nuestro enfoque, y así llegamos a la propuesta de usar microservicios, que veremos a continuación.

4 Microservicios como solución técnica

4.1 Qué es la arquitectura de microservicios

4.1.1 Microservicios y sus antecedentes

La arquitectura de microservicios es un enfoque para desarrollar una aplicación como un conjunto de pequeños[34] componentes, cada uno corriendo en su propio proceso y comunicándose mediante mecanismos livianos [Fowler 2014]. Estos servicios son construidos en correspondencia con capacidades de negocio[35] y pueden desplegarse independientemente mediante procesos totalmente automatizados[36]. A la vez, pueden ser escritos en diferentes lenguajes de programación y usar distintas tecnologías de almacenamiento de datos[37] (esto último a veces es referido como "persistencia políglota").

Cada microservicio puede verse como un componente cohesivo y autónomo trabajando en conjunto con otros microservicios para entregar valor. Estos microservicios son reutilizables por varios sistemas: por ejemplo, en una organización, se puede construir un microservicio que maneje la seguridad y los permisos y que sea utilizado por el resto de los sistemas de la organización.

La idea de los microservicios se ha visto popularizada por el auge de las aplicaciones en la nube[38] y sus necesidades (escalar fácilmente, desplegar a una alta frecuencia, etc.), habiendo todavía poco estudio académico e investigación empírica del tema [Dragoni 2017].

Las características generales de esta arquitectura son:

- Los servicios se organizan alrededor de capacidades individuales de negocio.

---

[34] Lo de pequeños, o "micro" no tiene que ver tanto con el tamaño del servicio sino con una interfaz pequeña, con un propósito simple y bien definido.

[35] "business capability"

[36] Textualmente: "is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweigxht mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery".

[37] Textualmente: "written in different programming languages and use different data storage technologies".

[38] Las aplicaciones en la nube (en inglés "cloud computing") cobran auge a partir de 2006, cuando Amazon introduce Elastic Compute Cloud, y en los años siguientes ingresan al rubro compañías tales como Microsoft y Google. De todas las implementaciones posibles de computación en la nube, la más difunida es PaaS (Platform as a Service), en la cual se provee una máquina virtual lista para el despliegue de aplicaciones.

- Los microservicios incluyen todas las capas del sistema[39], incluyendo acceso a datos persistentes[40], modelo de negocio e interfaces externas.
- Los servicios deben ser deplegables en forma independiente, y posiblemente en plataformas diferentes.
- Se requiere automatizar todo lo posible la infraestructura y los despliegues[41]. Por eso, se propone CD[42] como mejor opción para el desarrollo [Humble 2010] y se usan contenedores para el despliegue [Dragoni 2017].
- Los límites entre microservicios deben ser bien definidos y cohesivos.
- No hay un control centralizado de los servicios, sino que son los propios microservicios los que poseen la lógica de control.
- Los servicios suelen comunicarse vía APIs RESTFull[43] livianas sobre HTTP[44].
- La comunicación suele basarse en un estilo de coreografía[45] más que orquestación[46]: la inteligencia y la coordinación está más bien puesta en las terminales[47] que en un mecanismo de manejo centralizado.
- La comunicación suele basarse en eventos asincrónicos más que en llamadas sincrónicas: no pedimos a los servicios que hagan algo, sino que les comunicamos un evento y esperamos que reaccionen.
- Los equipos se especializan, no por especialidades, sino por áreas de negocio (cada una implicando un producto distinto) que cubren todas las capas y especializaciones. Esto incluye programadores, testers, analistas de negocio, gestión, especialistas en

[39] Es decir, dentro de un microservicio puede haber capas, pero cada microservicio debe incluirlas todas y no definir microservicios por capas.

[40] Por lo tanto, trabajan cada uno con su almacenamiento persistente, y no puede haber otras partes del sistema que les estén cambiando datos sin pasar por ellos.

[41] Cada microservicio debería tener su batería de pruebas de todos los tipos.

[42] Continuous Delivery.

[43] REST (REpresentational State Transfer) es un estilo arquitectónico inspirado en la web y HTTP.

[44] Esto es sólo la tendencia dominante: hay soluciones basadas en microservicios que se comunican por RPC u otros mecanismos.

[45] "choreography"

[46] "orchestration"

[47] "endpoints"

experiencia de usuario y hasta operaciones. Por eso, se propone DevOps[48] como modelo de equipo [Walls 2013].

- Los equipos se asignan a productos, no a proyectos, durante toda la vida del producto.

La idea de usar microservicios en vez de sistemas monolíticos ya había surgido con las arquitecturas SOA[49] en la década de 1990 [Perrey 2003]. En efecto, ya SOA preconizaba separar la implementación de componentes detrás de interfaces flexibles y bien definidas, alentando el bajo acoplamiento, la reusabilidad y la autonomía, pudiendo desarrollar distintos componentes con distintas tecnologías.
De hecho, microservicios es una evolución de SOA, que busca corregir los desvíos que se produjeron en el mundo real respecto de los objetivos originales de SOA, muchas veces debido a los intereses de las empresas comercializadoras de middleware y a cierta preferencia por la comunicación sincrónica y orquestada de las aplicaciones basadas en SOA [Richards 2015].

4.1.2 Algunas definiciones

Antes de seguir, introduzcamos un par de definiciones de dos conceptos que vamos a usar frecuentemente, contrastándolos. Las definiciones son intencionalmente minimalistas y fueron tomadas de [Dragoni 2017].
Monolito: es una aplicación de software cuyos módulos no se pueden ejecutar independientemente.
Microservicio: es un proceso cohesivo e independiente que interactúa mediante mensajes.

4.1.3 Ventajas de los microservicios frente a las aplicaciones monolíticas

Las ventajas frente a las aplicaciones monolíticas son varias:

- Cada cambio en un microservicio se puede desarrollar y desplegar independientemente del resto del sistema.
- Reduce el tiempo de salida al mercado[50] para cada funcionalidad.

[48] DevOps es un modelo de organización de equipos que implica la descentralización. Los equipos incluyen todas las especialidades (textualmente: "decentralization of skill sets into cross-functional teams") y son responsables del ciclo de vida completo del producto, incluyendo el desarrollo y el soporte.

[49] Service Oriented Architectures o arquitecturas orientadas a servicios.

[50] "time to market"

- Cada microservicio puede ser desarrollado en el mejor lenguaje que se aplique para ese problema, con la base de datos más acorde y la interfaz más adecuada[51]. Esto mismo permite adoptar nuevas tecnologías de a partes.
- Permite ir cambiando de a poco una aplicación monolítica, incorporando los microservicios de manera incremental (ver más adelante).
- Es más sencillo escalar hacia arriba o abajo cada servicio en forma autónoma de los demás (cuando escalamos un monolito debemos escalar todo junto, pero no todas las partes de un monolito tienen las mismas necesidades).
- Hay mejor comprensión del código, al estar separado por contextos encerrados[52] con su vocabulario específico (ver más adelante).
- Cuando se encuentra un problema en algo que recién se desplegó, se puede aislar fácilmente el microservicio, se vuelve atrás y se corrige sin afectar al resto del sistema.
- Permite reemplazar partes de un sistema monolítico sin tener que atarse a tecnologías del monolito previo y sin tirarlo sin más.

4.1.4 Desafíos que plantea la arquitectura de microservicios

Sin embargo, hay algunas cuestiones a tener en cuenta y algunas desventajas también. Lo que pasa es que, al trabajar con microservicios, estamos desarrollando sistemas distribuidos. Y no debemos olvidar que los sistemas distribuidos son más complejos de desarrollar y probar en general que los sistemas monolíticos.

Una cuestión poco explorada es el de las pruebas de microservicios, que son considerablemente más complejas que en el caso de los sistemas monolíticos [Dragoni 2017]. En este tema hay opiniones diversas, pero se plantean al menos los siguientes niveles de pruebas:

- Pruebas de pequeñas porciones de código dentro de los microservicios. Casi todos los autores coinciden en llamar pruebas unitarias a las mismas [Sundar 2017, Ghahrai 2017]. Algunos autores les asignan escaso valor a estas pruebas [Sridharan 2017].

[51] Hay quienes opinan que esto no es una buena idea, pues de esta manera se llega a situaciones en que cada equipo trabaja en su tecnología ("technology stack") preferida, generándose una fragmentación cultural que hace muy difícil la transferencia de personas entre equipos, así como compartir código y herramientas de monitoreo [Ranney 2016]. Esto hace que la idea de usar distintas tecnologías en distintos servicios deba verse como una posibilidad a usar con cuidado.

[52] "bounded contexts"

- Pruebas de cada microservicio, aislándolo del resto del sistema mediante dobles que simulen a los demás microservicios. Hay quien las denomina pruebas de contratos[53] [Sundar 2017], pruebas de componentes [Ghahrai 2017] e incluso quien las considera pruebas unitarias[54] [Sridharan 2017]. Son interesantes como documentación de la API que exponen los microservicios[55].
- Pruebas que verifiquen las comunicaciones e interacciones entre servicios, para detectar defectos en las interfaces y tratando a los microservicios como cajas negras. Estas pruebas a veces son llamadas pruebas de integración [Ghahrai 2017, Sundar 2017]. El problema de estas pruebas es que requieren de entornos en los que se ejecuten varios microservicios a la vez, y eso suele ser muy exigente para un entorno de desarrollo.
- Pruebas del sistema completo, de punta a punta[56], en el sentido de las pruebas de aceptación. Se hacen muy difíciles por la complejidad de reproducir tal cual el ambiente productivo, por lo que no se suelen hacer sobre todos los posibles escenarios, sino sólo los más críticos o los que son fuentes frecuentes de problemas. El resto de los escenarios se prueban directamente en producción.
- Pruebas en producción, esenciales en arquitecturas impredecibles con inumerables interacciones posibles, como las de microservicios. Son las únicas que realmente prueban el uso real del sistema, con las interacciones que realmente se realizan, aun las inesperadas durante el desarrollo. En esta familia podemos incluir el monitoreo, las pruebas exploratorias, las pruebas A/B[57], los patrones Blue/Green Deployment[58]

[53] "Integration contract testing"

[54] En realidad, no son estrictamente pruebas unitarias, porque si bien prueban al microservicio como una unidad, acceden a bases de datos y otras dependencias del microservicio.

[55] Es lo mismo que ocurre en TDD, que las pruebas quedan como documentación del uso de métodos y clases.

[56] "end to end"

[57] Las pruebas A/B son un tipo de pruebas beta que se usan habitualmente para ver cómo responden a una variable binaria, a modo de experimento sobre un conjunto de usuarios, antes de habilitar a todos ellos para que usen la nueva implementación. Si bien es un mecanismo típico de las pruebas de usabilidad, se pueden usar para pruebas de comportamiento también, para detectar problemas en producción.

[58] Blue/green deployment consiste en desplegar dos versiones del sistema y utilizar un proxy para acceder a una u otra conforme se la va actualizando. Al principio, todos los usuarios acceden a la versión antigua, pero progresivamente se va haciendo que el proxy vaya derivando cantidades mayores de solicitudes a la versión nueva.

[Fowler 2010] o Canary Release[59] [Sato 2014], etc. [Sridharan 2017]. Sobre estas pruebas hay opiniones encontradas: hay quienes plantean que el costo de una corrección de un defecto en producción es muy alto [Sundar 2017] y quienes plantea que se debería llegar a producción sin verificar tanto y dejar que allí se pruebe una vez desplegado y en uso [Sridharan 2017].

La depuración (*debugging*) de una aplicación con microservicios, al no poder tener todo junto el ambiente, el estado del sistema y el flujo de ejecución, es mucho más compleja que la que se hace sobre un sistema monolítico.

Un desafío adicional lo presenta la necesidad de que cada microservicio maneje su propio almacenamiento persistente. Esto dificulta, entre otras cosas, la posibilidad de realizar transacciones en el sentido tradicional. En una publicación reciente se ha propuesto cambiar la idea de transacciones por la de "sagas" [Betts 2018].

Además, la seguridad es un aspecto de mucha mayor relevancia en microservicios que en los sistemas monolíticos, lo cual se ve afectado por la heterogeneidad de plataformas y los múltiples puntos de ataque.

El desempeño de un sistema basado en microservicios haciendo llamadas remotas constantemente también suele ser menor que el de un sistema monolítico. Esto empeora si un servicio debe invocar en forma reiterada a otros servicios para efectuar una operación. Por eso, se requiere que las APIs sean de granularidad mayor a la que se espera de las interfaces de objetos[60].

[59] "Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody. Similar to a BlueGreenDeployment, you start by deploying the new version of your software to a subset of your infrastructure, to which no users are routed.".

[60] Esta aclaración viene bien porque, topológicamente hablando, la arquitectura de microservicios se asemeja mucho a la que surge de un buen diseño orientado a objetos, y en ambos casos se considera un buen diseño a aquél que garantiza el encapsulamiento. Por otro lado, el problema del límte entre microservicios sigue siendo el mismo de siempre: el de establecer el límite entre módulos, y en algún sentido, algunos principios de diseño de la orientación a objetos son muy útiles, como aquél que recomienda poner juntas las cosas que cambian por las mismas razones.

Por otro lado, construir y mantener una cadena[61] de despliegue para cada microservicio tiene costos importantes, más aún cuando pueden ser de ambientes distintos.

Otro tema es el costo de las refactorizaciones que implique cruzar los límites entre servicios, que resultan también mucho más costosas que cuando se refactoriza un sistema monolítico, principalmente por las llamadas remotas y por las tecnologías potencialmente distintas.

4.1.5 Digresión[62]: microservicios reactivos

Hay un tipo especial de microservicios que se denomina *reactivo*. Decimos que una arquitectura de microservicios es reactiva cuando tiene las siguientes propiedades:

- Autonomía[63]: cada servicio trabaja independientemente del resto de los servicios.
- Elasticidad[64]: implica la posibilidad de escalamiento horizontal, permitiendo variar automáticamente la cantidad de instancias en función de la carga que recibe. En general implica que los servicios sean lo menos dependientes posibles de su estado[65][66].
- Resiliencia[67]: implica la habilidad para manejar fallas y recuperaciones sin bloquear la aplicación, probablemente mediante replicación.
- Pasaje de mensajes asincrónico[68]: esto evita demoras por caídas de servicios.
- Adaptabilidad de los propios servicios en función de los estímulos que reciben.
- Los servicios son descubiertos programáticamente usando técnicas de descubrimiento de servicios[69] automáticas.

Hemos planteado este apartado sólo como una digresión, ya que desde el punto de vista del proyecto no nos interesa la escalabilidad o la reactividad de los microservicios, sino más bien

---

[61] "pipeline"

[62] Este apartado excede las pretensiones de este proyecto, pero puede tener interés para otros proyectos del LIMET.

[63] "Autonomy"

[64] "Elasticity"

[65] "stateless"

[66] Desde nuestro punto de vista, esto es un tanto utópico, toda vez que los datos persistentes son fundamentales en los sistemas de negocio, y de alguna manera, los servicios van a tener que acceder a este estado y actuar en consecuencia. Mover los repositorios persistentes a servicios externos sólo traslada el problema.

[67] "Resilience"

[68] "asynchronisity message passing"

[69] "service Discovery"

su adecuación para permitir la evolución de sistemas heredados en forma incremental, dejando un sistema de más fácil mantenimiento como resultado.

4.2 Migrando a microservicios

4.2.1 Por qué migrar

Algunos planteos sugieren que no se buscar construir con una arquitectura de microservicios por los microservicios mismos, sino que la necesidad suele surgir en las organizaciones centradas en el software[70] cuando desean aumentar la velocidad con la que liberan sus productos [Betts 2018]. Según esta visión, la necesidad de aumentar la velocidad de las liberaciones es la que provoca la organización de equipos polifuncionales (DevOps) y cambios en los procesos, que a su vez lleva a la necesidad de liberar el producto en la forma de microservicios.

4.2.2 Aspectos metodológicos

Cuando necesitamos reemplazar partes de un sistema, sea porque una funcionalidad debe ser reemplazada por otra o porque hay que realizarle algunos cambos, usar microservicios en el código nuevo es una buena opción, que tiene la ventaja de ir incorporando una arquitectura más moderna, escalable, con menores tiempos para llegar al mercado y más sencilla de mantener y desplegar [Newman 2015].

De alguna manera, cuando reemplazamos partes de un sistema por uno o más microservicios sin cambiar su funcionalidad estamos realizando una refactorización de arquitectura (una reingeniería, dirían algunos). Como todo cambio arquitectónico, debe ser incremental y de a pasos pequeños, de modo de no afectar a los usuarios actuales del sistema.

4.2.3 Microservicios y su relación con DDD: contextos encerrados

Hay una fuerte relación de los microservicios con una técnica de diseño que lleva más de una década de existencia: DDD[71] [Evans 2004]. Esto resulta interesante en nuestro contexto, ya que DDD también está relacionado con SBE por el uso del lenguaje ubicuo.

La relación con microservicios viene porque DDD propugna que el diseño arquitectónico se base en contextos encerrados[72]: subconjuntos del modelo de dominio consistentes en sí mismos. Y se ha planteado en numerosos estudios que es una buena idea limitar los

[70] "software-driven organizations"

[71] Domain Driven Design.

[72] "bounded contexts"

microservicios dentro de los contextos, a la manera de DDD [Balalaie 2015, Fowler 2014, Nadareishvili 2016, Newman 2015].

Pero, ¿qué es un contexto encerrado?

Se trata de una solución arquitectónica que fracciona el espacio de la solución, de modo tal que cada subdominio de la solución debería mapear uno a uno con un contexto. Como pasa con los microservicios, los contextos encerrados cruzan varias capas de la aplicación.

Incluso se ha propuesto el uso de contextos encerrados para mejorar sistemas de arquitectura inconsistente, entre los cuales casi siempre caen los sistemas heredados [Vernon 2013].

Hay una fuerte relación entre contextos encerrados y lenguaje ubicuo. Esto es así porque cada contexto tiene su lenguaje ubicuo, más allá de que haya términos compartidos con otros, que en ese caso evolucionarían cada uno por su lado. En cada contexto encerrado, los conceptos tienen su significado particular. Por eso decimos que un contexto encerrado es un límite lingüístico y que hay un contexto por lenguaje ubicuo.

Volviendo a lo que ya vimos de los lenguajes ubicuos, de que son ubicuos para cada equipo, ahora podemos agregar que se trata de lenguajes ubicuos dentro de un contexto encerrado.

4.2.4 Pasando a microservicios

Hay bastante literatura que refiere a migrar sistemas heredados a arquitecturas SOA [Almonaies 2010]. Y si bien no hay tanta referida a la migración hacia microservicios, muchos de los conceptos que surgen de la migración a SOA son aplicables también a microservicios.

La mayoría de las propuestas sugieren que hay que empezar por separar capacidades y convertir esas capacidades en contextos encerrados: aquí el problema suele presentase porque las funciones de negocio, que uno enmarcaría en contextos, probablemente no estén en un único módulo sino desperdigadas en varios. También existe la sugerencia de separar las partes que cambian poco en servicios distintos de las que cambian mucho.

En cuanto a la migración incremental, una posibilidad es generar una capa de servicios, con su interfaz, alrededor de algún componente heredado[73].

De todas maneras, en los últimos años ha ido surgiendo literatura específica para la migración hacia microservicios.

Hay una publicación reciente que plantea unos cuantos patrones de migración hacia microservicios [Balalaie 2015]. Sintetizando algunas que nos parecen más relevantes:

- Usar DDD para identificar subdominios del negocio y convertir cada subdominio en un contexto encerrado que sea una unidad de despliegue[74].

[73] "wrapping"

- Separar por la frecuencia de cambio[75].
- Separar por requerimientos no funcionales[76].
- Descomponer el monolito por el acceso a datos[77].

Una tesina de la Facultad politécnica en Jyväskylä (JAMK University of Applied Sciences), Finlandia, se enfoca en un caso práctico de migración de un monolito a una arquitectura de microservicios [Zaymus 2017]. Volvemos sobre esto al analizar en detalle nuestra propuesta.

Es interesante el planteo de Fowler, que en una publicación reciente se plantea si no será mejor siempre comenzar por una aplicación monolítica y llevarla a una arquitectura de microservicios más adelante, incluso comenzando por servicios no tan pequeños y de grano grueso [Fowler 2015]. Si bien Fowler sostiene que no hay suficiente evidencia empírica que indique que éste sea el mejor camino, lo cual se debe a que la arquitectura de microservicios recién está mostrando sus bondades[78], sostiene que la mayor parte de los éxitos de una arquitectura de microservicios se han evidenciado al migrar desde un monolito, a la vez que los mayores problemas han surgido de un planteo temprano de esta arquitectura. Esta idea descansa fuertemente en el antiguo principio de diferir la introducción de mejoras en el diseño hasta tanto las necesitemos realmente [Beck 1999]. Por otro lado, encontrar contextos encerrados y lenguajes ubicuos para cada uno son tareas que requieren experiencia en el dominio, y eso sólo se logra luego de un tiempo de trabajar con el sistema. Al fin y al cabo – y seguimos con el razonamiento de Fowler – un monolito permite explorar mejor la complejidad y encontrar límites entre contextos, permite refactorizar de manera más sencilla, y siempre hay tiempo para migrar hacia los microservicios cuando la complejidad del sistema aumente. Incluso se podría ir migrando hacia servicios de grano grueso hasta tanto estemos seguros de la pertinencia del modelo de microservicios y la arquitectura esté más estable. Lo que resulta muy interesante de este planteo – si bien aún habría que validarlo empíricamente – es que está muy en línea con nuestra propuesta de recuperación de sistemas heredados usando microservicios por demanda.

Otras publicaciones sugieren otros modos de derivación de microservicios desde un monolito, como el planteo de una técnica basada en casos de uso que representen un conjunto de operaciones de diferentes subsistemas sincronizados por la acción de un actor [Levcovitz 2016].

---

[74] Textualmente: “DDD should be used to identify sub-domains of the business that the system is operating in. Then, each sub-domain can constitute a Bounded Context that is a deployable unit”

[75] Textualmente: “different change frequency rate”

[76] Textualmente: “Different parts of the system have different non-functional requirements, e.g. scalability”

[77] Textualmente: “Decompose the monolith based on data ownership”

[78] “These are early days in microservices”

Todas las recomendaciones sugieren no pensar en los microservicios como objetos tradicionales. Si bien el modelo de microservicios descansa, como el modelo de objetos, en el encapsulamiento y la abstracción, el hecho de que los microservicios deban correr en ambientes distribuidos en los que el desempeño y la disponibilidad de la red no sean óptimas, hace que las interfaces de alta granularidad que se recomiendan en el diseño orientado a objetos no sean recomendables en este caso.

4.2.5 Migración de las bases de datos a microservicios

Uno de los problemas más serios de la migración a microservicios tiene que ver con la migración de las bases de datos, que así como el código, deben fraccionarse para que cada microservicio tenga la suya. Una recomendación básica es hacer los cambios de a poco. Por ejemplo, si bien la arquitectura de microservicios permite usar distintos paradigmas de bases de datos en cada servicio, no es una buena idea hacerlo desde el principio. En el caso más típico estaremos cambiando la arquitectura de un sistema con una base de datos relacional: por lo tanto, siguiendo esta recomendación, terminaremos la primera migración con varias bases de datos relacionales [Yanaga 2017].

El principal desafío de la evolución del esquema de la base de datos es hacerlo sin detener el sistema[79] y ofreciendo al mismo tiempo consistencia. Si deseamos una consistencia fuerte, todos los nodos mostrarán la misma información a expensas de la velocidad de respuesta, mientras que una consistencia más flexible, temporariamente inconsistente, es lo más recomendable cuando la velocidad de respuesta sea lo más importante. Si bien hay muy poca literatura y publicaciones académicas, hay un libro que aborda una serie de modelos de refactorización para bases de datos relacionales hacia el modelo de microservicios [Yanaga 2017].

La solución más habitual es la de compartir tablas de una base de datos entre varios microservicios [Yanaga 2017]. Claramente esto atenta contra la propia definición de la arquitectura hacia la que estamos yendo, además de presentar un diseño muy acoplado. Por lo tanto, si bien tiene como ventaja la rapidez y la sencillez en implementar la solución, sólo debería usarse temporariamente antes de elegir otro mecanismo de integración.

Hay muchos otros mecanismos de integración, con sus pros y contras. Por ejemplo, las vistas de bases de datos tienen la ventaja de cierta sencillez pero problemas de desempeño, aunque si DBMS que utilizamos permite actualizar vistas puede ser muy práctico. Las vistas materializadas que implementan algunos DBMS, trabajando sobre tablas físicas funcionan mejor si el DBMS permite escrituras consistentes. Otra variante es la virtualización de datos,

[79] "zero downtime deployments"

que provee una capa de abstracción sobre varias fuentes de datos, probablemente heterogéneas, y accede directamente a los datos reales, por lo que casi siempre es muy flexible y consistente. El libro que citamos anteriormente cubre estos mecanismos y varios más [Yanaga 2017].

4.3 Incorporación de microservicios en nuestra propuesta

Cuando planteamos nuestro enfoque metodológico para la recuperación de software heredado, SBE nos dio el marco metodológico. Sin embargo, lo metodológico no puede quedar aislado de una propuesta más técnica. Por eso nos pareció que el enfoque debería incluir cuestiones de arquitectura.

Por otro lado, debido a que nuestra propuesta de recuperación de sistemas heredados debe ser incremental, y dado que SBE – que es el enfoque metodológico que planteamos – pone especial énfasis en el lenguaje, que es una de las premisas de DDD, nos pareció adecuado proponer a los microservicios – que también tienen raíces en DDD – como propuesta arquitectónica. Esto es algo que empiezan a destacar varios autores en los últimos años [Rahman 2015].

Entre las ventajas que le vemos al uso de microservicios en nuestra propuesta están:

- Se lleva muy bien con la idea de ir reemplazando el sistema de manera incremental.
- El sistema resultante será más escalable y tolerante a fallas.
- Habremos incorporado la posibilidad de liberar el producto en pequeñas iteraciones independientes y de manera continua.
- Al final del proceso, vamos a quedar con una arquitectura más moderna, modularizada en función de capacidades de negocio (o contextos encerrados) y más sencilla de evolucionar hacia el futuro.

Como aclaración, nuestra propuesta de usar arquitecturas de microservicios se basa en sus bondades para la separación de incumbencias, no por la escalabilidad, la posibilidad de usar diferentes tecnologías o la facilidad de ser desplegados de manera continua y en la nube.

De todas maneras, no es obligatorio usar microservicios en todos los casos en que se siga nuestra propuesta de recuperación de sistemas heredados con las prácticas de SBE. Creemos que en la mayor parte de los casos se puede aplicar y que se realimenta positivamente con el resto del enfoque, pero puede haber ocasiones en que convenga otro enfoque arquitectónico. En este caso, el profesional deberá aplicar su criterio.

Uno de los aspectos a analizar es que, como dijimos antes, la refactorización es más difícil en sistemas distribuidos, y esto podría ser un elemento disuasorio para el mantenimiento futuro.

5 La propuesta en lo metodológico

5.1 Características deseables

Luego de todo lo explicado hasta aquí, queda claro que el nuestra propuesta deberá tener al menos las siguientes características:

- La migración debe ser incremental.
- Toda la funcionalidad no migrada seguirá como estaba en la versión previa a la migración hasta tanto se decida migrarla.
- Los incrementos deben encararse prioritariamente por demanda de los clientes.
- Solamente se encararán incrementos no solicitados por los clientes cuando quede tiempo ocioso en el equipo.
- En todo momento se deberá trabajar con los usuarios.
- Las pruebas se deberán automatizar para permitir pruebas de regresión, a la vez que dejamos especificaciones ejecutables para el futuro.
- El sistema deberá quedar modularizado en base a contextos encerrados que se implementarán como microservicios.
- La mejora en el diseño a medida que se avance en la migración servirá para ir realimentando la sencillez y velocidad de las mejoras futuras.
- La mejora en las especificaciones a medida que se avance en la migración servirá para ir realimentando la calidad de las mismas y facilitar cada vez más las mejoras futuras.

5.2 Propuesta básica

El procedimiento que proponemos consiste en los siguientes pasos:

- Ante un nuevo requerimiento (pedido de cambio del cliente, error reportado del sistema en producción o simplemente inactividad del equipo de desarrollo), reunir al taller de especificaciones.
- En el marco del taller de especificaciones se detallan los ejemplos de uso del sistema que van a guiar el desarrollo y servir como pruebas de aceptación.
- Desarrollo de la migración necesaria usando microservicios.
- Probar el sistema tomando los ejemplos como casos de aceptación.
- Se despliegan los cambios.
- Se analiza la necesidad o no de hacer una refactorización mayor, y se la hace de ser necesaria, usando los mismos ejemplos como red de seguridad ante cambios de comportamiento.
- Se despliegan los cambios provenientes de la refactorización.

La Fig. 1 muestra este procedimiento.

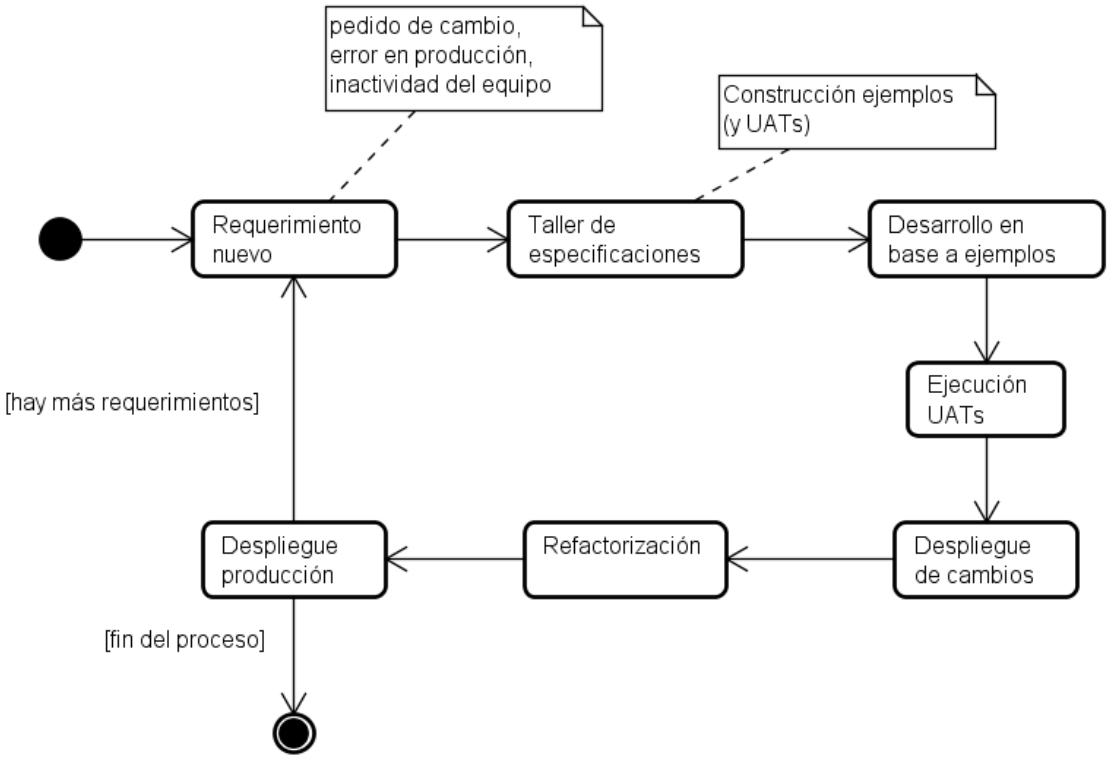

Fig. 1. Procedimiento básico propuesto, que muestra cómo comienza el trabajo prioritariamente ante un pedido de cambios del cliente.

5.3 Consideraciones especiales

5.3.1 La importancia de los talleres multidisciplinarios

SBE plantea claramente que las especificaciones ilustradas con ejemplos deben surgir de talleres en los que participen roles diversos en conjunto. Este es un aspecto muy importante de SBE y también de nuestro enfoque, ya que los distintos roles aportan visiones diferentes sobre los ejemplos, y conversando en un mismo espacio pueden proporcionar ejemplos desde sus particulares modos de ver el sistema.

Los testers, por ejemplo, son buenos tratando de encontrar casos excepcionales o extraños que no están cubiertos por los ejemplos[80]. Los programadores son los especialistas técnicos y

[80] Por ejemplo, qué pasa si acceden concurrentemente 1000 usuarios.

suelen ser detallistas para encontrar situaciones de borde que deban ser consideradas, y que les interesan para saber cómo las van a implementar luego. Los usuarios son quienes mejor conocen el sistema, sus vericuetos, cuestiones a las que acceden mediante atajos o forzando comportamientos inesperados, aunque tal vez se centren en el camino feliz si no son bien orientados.

Por eso es fundamental trabajar en estos talleres, y que todos puedan aportar sus visiones con libertad y confianza.

Es en estos talleres también en los que surge el lenguaje ubicuo, que debemos usar consistentementea lo largo del proceso, y que nos va a permitir definir contextos que sirvan para delimitar microservicios.

5.3.2 La necesidad de los usuarios

Una posible debilidad de nuestro planteo sería que depende de que los usuarios colaboren en los talleres de especificaciones. Debemos entender que el rol del usuario es central, pues en sistemas heredados suelen ser quienes mejor conocen qué hace realmente el sistema y qué debería seguir haciendo, esté documentado o no.

Por lo tanto, hay que hacer todo lo posible por lograr que el usuario se sienta parte del equipo, acomodarse a su agenda y facilitarle siempre su participación. También es una buena idea hacer demos para los usuarios de manera frecuente, una por cada entrega como mínimo para que validen lo que se hizo, ya que el feedback en este momento es fundamental. Si se está siguiendo alguna forma de CD que implique entregas automáticas, al menos una vez a la semana se deberían hacer demos al usuario.

Ahora bien, ¿qué ocurriría si los usuarios no pudieran o no quisieran colaborar?

Como siempre pasa en estos casos, no hay una respuesta fácil. Dada la importancia del usuario en estos talleres, lo primero es entender qué lo impulsa a no participar: si es falta de seguridad, si percibe que el sistema es mucho mayor de lo que él usa, si entiende que se trata de una tarea muy técnica, y por lo tanto reservada a los especialistas en software.

Una vez entendido el problema, debemos afrontarlo. Si el problema es una percepción de tarea demasiado técnica, tendremos que tratar de enfocar los talleres en las necesidades de los usuarios por sobre cualquier otra consideración y usar el vocabulario del negocio. Si la percepción del usuario es que él o ella no conoce la totalidad del sistema, tendremos que buscar más usuarios que nos permitan cubrir más. Si el usuario simplemente no desea colaborar, tal vez haya que cambiar de usuario por uno más comprometido con la evolución del producto.

Una pregunta que cabría hacerse es por el tipo de usuario que necesitamos. En realidad, la literatura sobre SBE nombra en ocasiones al analista de negocio [Adzic 2009, Gärtner 2012],

en otras al cliente [Koskela 2007, Pugh 2010] y en otras al usuario [Cohn 2004, Mugridge 2005]. En realidad no importan tanto los nombres, sino que sea alguien que conoce el sistema (aquí los usuarios concretos parecen más adecuados, sobre todo tratándose de software heredado), que tenga capacidad de decisión para priorizar y tomar decisiones (tal vez un representante del cliente sea más adecuado para esto) y que pueda traducir las necesidades de clientes y usuarios cuando no sea posible contar con ellos (lo que habitualmente hace un analista).
Lo que sí es seguro es que de los usuarios – o como los llamemos – no podemos prescindir.

5.3.3 Sobre la necesidad o no de refactorizar

Una cuestión que surje es si debemos o no refactorizar antes de hacer los cambios. Las buenas prácticas indicarían que sí. Incluso la tesis de Alan Jonsson en la Universidad de Uppsala estudia la conveniencia de refactorizar para mejorar las métricas de código en sistemas heredados [Jonsson 2017].
Sin embargo, lo esencial no pasa por refactorizar o no refactorizar. Está claro que refactorizar va a mejorar la calidad del código y hacerlo más mantenible. El problema está en que estamos a punto de modificar algo: escribir pruebas de aceptación que evidencien el comportamiento previo al cambio, para luego refactorizar, modificar esas pruebas recién escritas y luego modificar el código para volver a probar parece un trabajo redundante. Incluso hay quienes afirman que ni siquiera hay que documentar código heredado porque no agrega valor al negocio [Stevenson 2004].
Ahora bien, por cada uno de los cambios que mostramos en el diagrama de la Fig. 1 podríamos ir desarrollando el código siguiendo el planteo de Feathers, que implica pequeñas refactorizaciones previas y luego el uso de Test-Driven Development (TDD) [Feathers 2004]. Además, el protocolo TDD exige que se haga otra refactorización dentro de cada ciclo.
Por lo tanto, el procedimiento para el desarrollo de lo que corresponde a cada ejemplo consiste en los siguientes pasos:

- Separar dependencias y agregar pruebas unitarias automatizadas, siguiendo las recomendaciones de Feathers.
- Refactorizar el código para facilitar los cambios.
- Desarrollar siguiendo el ciclo de TDD, que implica varios ciclos de escribir pruebas unitarias automatizadas que fallen, escribir código que haga pasar una a una las pruebas unitarias y refactorizar.

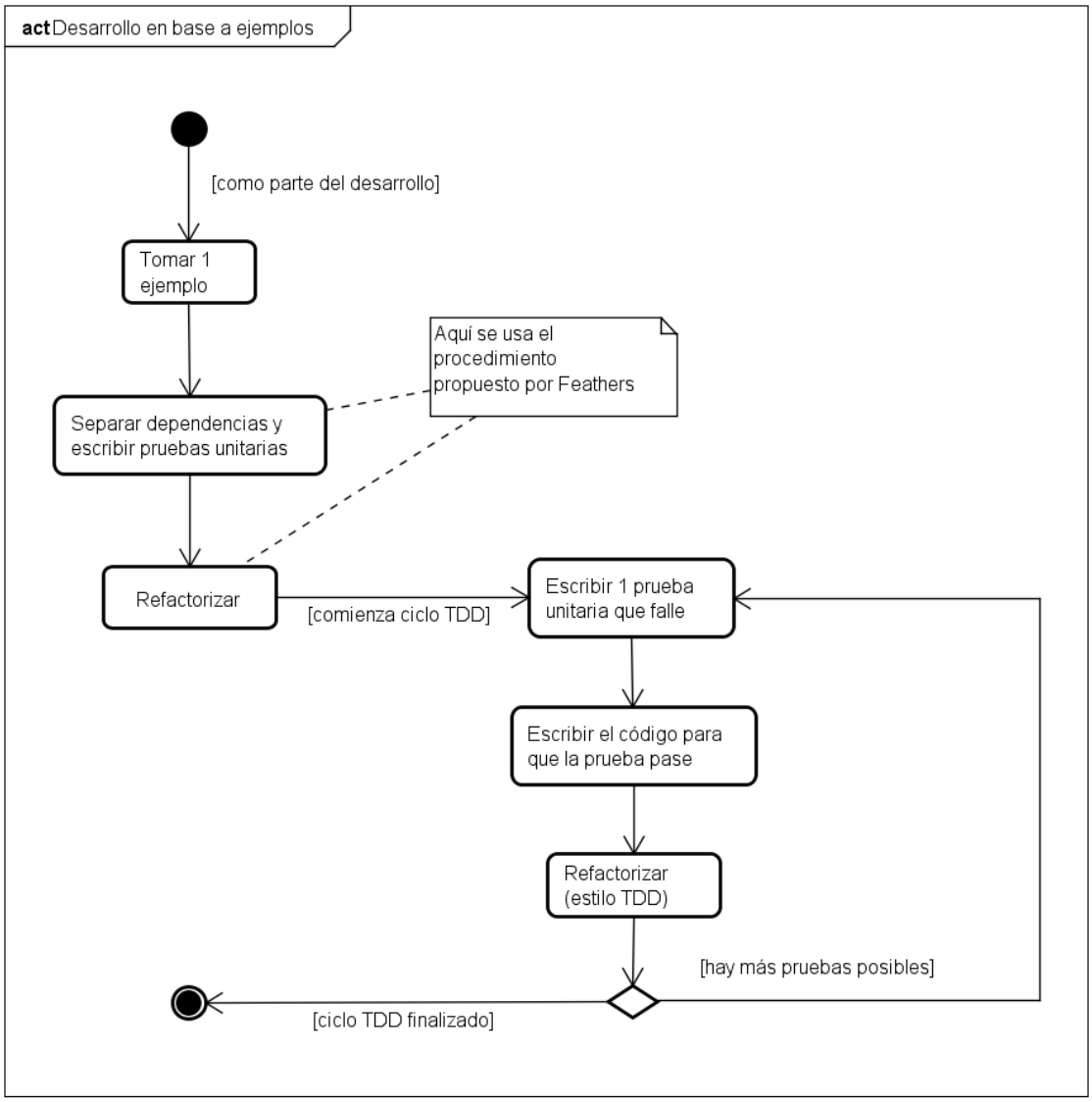

Fig. 2. Detalle de la actividad "Desarrollo en base a ejemplos" de la Fig. 1. Nótese el uso del protocolo de TDD (que incluye una refactorización). Recordemos (Fig. 1) que se deja lugar para una refactorización final antes del despliegue definitivo.

5.3.4 Regresiones en sistemas muy acoplados

Una debilidad del planteo surge ante la posibilidad de problemas de regresión en partes del sistema que aparentemente no estarían siendo afectadas por los cambios. Si el sistema no está bien modularizado – y es altamente probable que un sistema heredado no lo esté[81] – los

[81] Lo típico es que se haya ido convirtiendo en un Big Ball Of Mud [Foote 1997].

cambios que introduzcamos podrían provocar fallas en otras partes del sistema que no nos hayamos preocupado por especificar, y para las cuales no tendremos pruebas de aceptación. Esto implica que haya que hacer pruebas de regresión completas luego de cada cambio de funcionalidad grande.
Precisamente por esta necesidad de hacer pruebas de regresión a menudo, es que hemos planteado que el método debería apoyarse en la automatizar las pruebas de aceptación, o al menos una proporción importante de las mismas.

5.3.5 Sobre la necesidad o no de automatizar pruebas

El conjunto de prácticas que describe SBE incluye la generación de pruebas de aceptación que se deben usar para probar nuevas características y regresiones. Esas pruebas de aceptación surgen claramente de los ejemplos elaborados en los talleres de especificaciones.
Pero SBE no pone tanto énfasis en la automatización de las pruebas. Incluso Brian Marick se opone a la automatización de pruebas de aceptación, por su alto costo en tiempo y por su fragilidad [Marick 2008]. Él sugiere especificar con ejemplos, pero no automatizarlos, sino simplemente como base para derivar pruebas unitarias que luego se utilicen como guía del diseño[82].
Sin embargo, nada impide que automaticemos pruebas de aceptación en SBE ni en nuestro enfoque metodológico. De hecho, lo hemos recomendado explícitamente, no sólo para evitar regresiones, sino también para dejar especificaciones ejecutables y vivas en el nuevo sistema migrado, que permitan la evolución futura. Como consecuencia de lo anterior, podremos dejar de considerarlo software heredado.
Tal vez lo conveniente sea analizar cuánto automatizar de cada tipo de pruebas, siguiendo la pirámide de pruebas como guía [Eickelmann 1996]. Como las buenas prácticas indican, las pruebas unitarias deberían ser las más abundantes, seguidas de las de integración, luego las de comportamiento a través de una capa de servicios y sólo finalmente las pruebas de aceptación a través de la interfaz de usuario.

5.3.6 Sobre la necesidad o no de la entrega continua

Primero, algunas definiciones.
Integración continua (CI[83]): se refiere a compilar, construir y probar, todo en el ambiente de desarrollo[84]. Como mínimo se ejecutan las pruebas unitarias y de integración técnicas.

---

[82] Citamos esto con cierto desacuerdo, pero teniendo en cuenta que es la opinión de un referente fundamental de testing en la comunidad ágil.

[83] Continuous Integration.

Entrega continua (CD[85]): implica CI[86] más asegurar que el código siempre está en condiciones de ser desplegado[87]. Por lo tanto, se deben ejecutar todas las pruebas de aceptación automatizadas que tengamos, como mínimo una vez al día, mediante el uso de una cadena de despliegue[88] que automatice todo el proceso hasta el ambiente productivo. El patrón por excelencia del CD es la cadena de despliegue, que debe comenzar sencilla y básica, incluyendo algunas pruebas unitarias y de aceptación.
Hay una técnica más avanzada, llamada habitualmente despliegue continuo[89] que implica, no sólo la potencialidad de desplegar, sino el despliegue efectivo en forma continua por cada pequeño cambio[90].

Respecto de si es o no necesario usar CD, la respuesta corta es no. Por supuesto, si estamos modernizando un producto de software, puede ser conveniente introducir buenas prácticas, que a su vez se realimentan con la arquitectura que estamos utilizando, pero es algo a analizar en cada caso, y excede nuestro proyecto.

5.3.7 Cómo ir introduciendo la nueva arquitectura: Strangler Application y Test Toggles

Como vimos, ha habido unos cuantos planteos de procedimientos para llevar una arquitectura monolítica a otra de microservicios. Hay también algunos patrones que conviene explorar.
Tal vez el planteo más interesante es el basado en el patrón Strangler Application, presentado someramente hace más de una década, y con pocas aplicaciones hasta hace pocos años [Fowler 2004]. La idea básica es simple: ir rodeando al antiguo sistema con partes nuevas e ir estrangulándolo hasta que desaparezca. Se plantea que, de esa manera, el riesgo es menor que pretendiendo hacer un cambio grande todo a la vez[91].
En los últimos años han ido surgiendo algunos casos de estudio interesantes que plantean el uso de este patrón [Hammant 2013, Rook 2016] para la migración de un monolito heredado hacia una arquitectura más moderna. La idea que ha ido ganando aceptación es la de

---

[84] "integrating, building, and testing code within the development environment"

[85] Continuous Delivery.

[86] "Continuous Delivery builds on this, dealing with the final stages required for production deployment"

[87] "ensuring our code is always in a deployable state"

[88] "deployment pipeline"

[89] Continuous Deployment.

[90] "every push goes to production"

[91] "big bang rewrite"

establecer un proxy que derive funcionalidad hacia el sistema viejo y también hacia las nuevas implementaciones, de modo tal de ir cambiando de a poco, hasta que el monolito quede sin uso y se pueda desactivar[92].

Este planteo de un proxy se puede ver en detalle en una tesina reciente de la Facultad politécnica en Jyväskylä (JAMK University of Applied Sciences), Finlandia, que propone interceptar las llamadas al sistema viejo y reenviarlas al nuevo a través de su API [Zaymus 2017]. Esto se basa en la noción de Request Router, que es una implementación particular de Strangler Application. Las Fig. 3, 4 y 5 muestran los pasos que se podrían seguir aplicando este método.

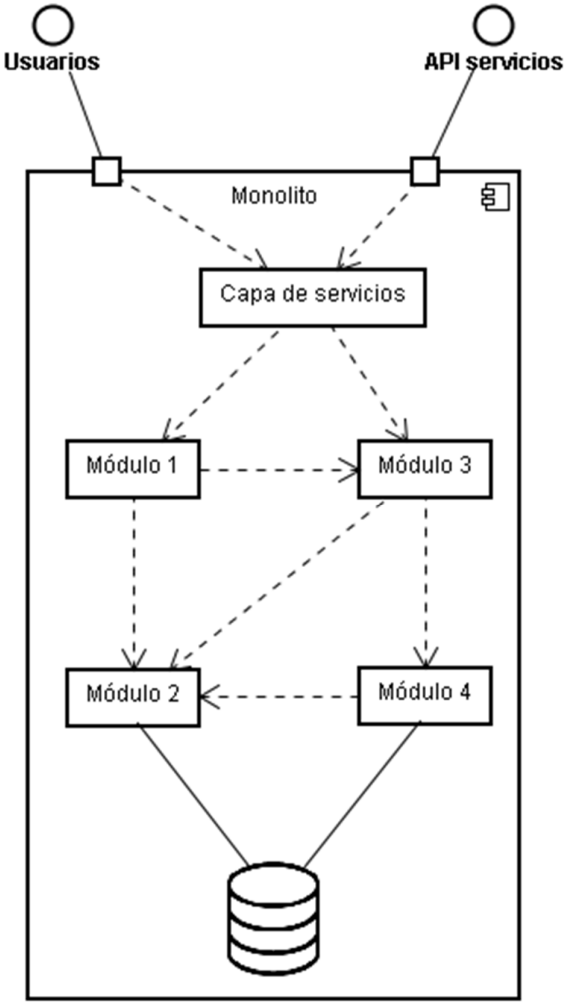

Fig. 3. Diagrama de arquitectura inicial de la aplicación monolítica

[92] Este enfoque permitiría también volver al sistema viejo ante un problema que se detecte en el nuevo.

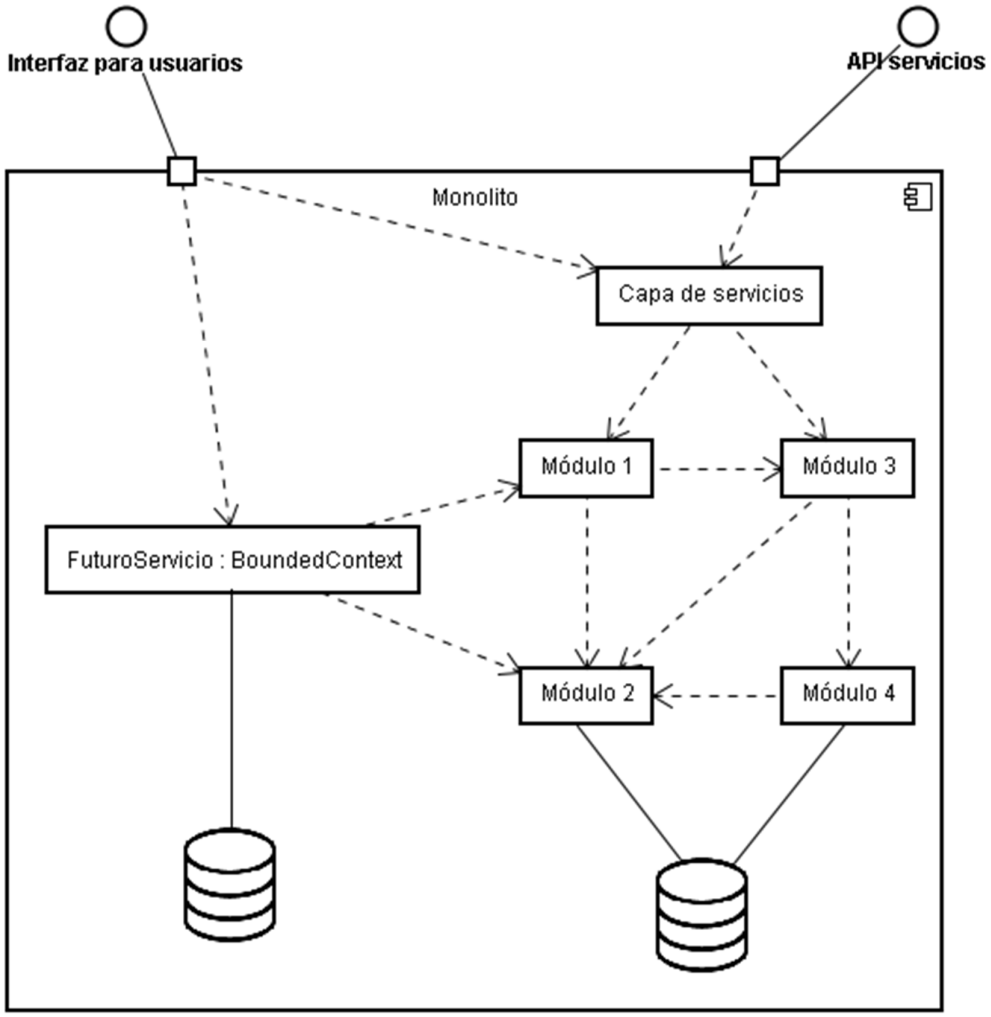

Fig. 4 Estado intermedio con contextos identificados

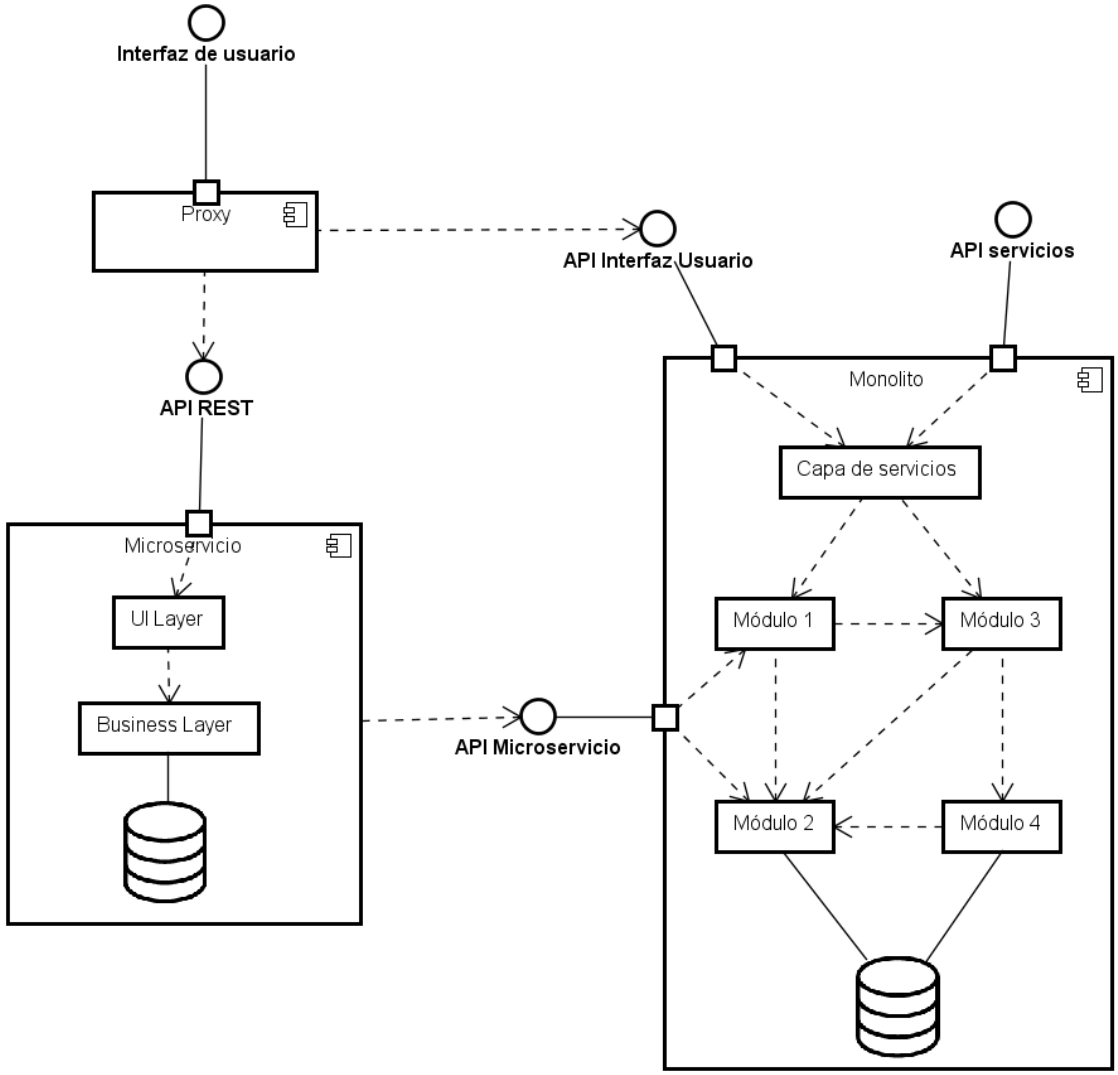

Fig. 5 Diagrama de arquitectura final con microservicio separado

Las propuestas alrededor del patrón Strangler Application también sugieren ir estrangulando y a la vez introduciendo nueva funcionalidad, lo cual está muy en línea con nuestro planteo metodológico.

Una posibilidad que se nos ocurre que se podría implementar son las pruebas A/B[93] o los patrones Blue/Green Deployment[94] [Fowler 2010] o Canary Release[95] [Sato 2014] para analizar la aceptación en producción de la nueva implementación.

[93] Las pruebas A/B son un tipo de pruebas beta que se usan habitualmente para ver cómo responden a una variable binaria, a modo de experimento sobre un conjunto de usuarios, antes de habilitar a todos ellos para que

Una idea parecida surge en el marco de DDD, cuando Evans introduce la noción de una capa anti corrupción[96] [Evans 2004], que plantea hacer una API del sistema monolítico, que así se convierte en un microservicio gigante, que de a poco vamos estrangulando.

Test Toggles[97] es un patrón que utiliza Martin Fowler sin caracterizarlo [Fowler 2013] y al que hemos bautizado en este proyecto mezclando los nombres de los Feature Toggles, un conjunto de patrones que permiten desplegar de una sola vez más de una versión del sistema [Rahman 2016], y el hecho de que lo que estamos introduciendo aquí son pruebas beta, similares a las A/B.
Según Fowler para obtener feedback del usuario necesitamos hacer despliegues continuamente. Pero si no estamos dispuestos a que toda nuestra base de usuarios sea afectada, podemos desplegar a un subconjunto de nuestros usuarios y probar la aceptación en producción antes de desplegarlo a todos ellos[98].

Habría que probar estos enfoques en pruebas de concepto.

---

usen la nueva implementación. Si bien es un mecanismo típico de las pruebas de usabilidad, se podría ensayar utilizarlo para pruebas de comportamiento también, para analizar cuál versión brinda mayor valor al negocio.

[94] Blue/green deployment consiste en desplegar dos versiones del sistema y utilizar un proxy para acceder a una u otra conforme se la va actualizando. Al principio, todos los usuarios acceden a la versión antigua, pero progresivamente se va haciendo que el proxy vaya derivando cantidades mayores de solicitudes a la versión nueva.

[95] "Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody. Similar to a BlueGreenDeployment, you start by deploying the new version of your software to a subset of your infrastructure, to which no users are routed.".

[96] "anti corruption layer"

[97] En castellano sería "interruptor de pruebas" o, siendo menos literal pero más explícito, "conmutador de pruebas".

[98] "User feedback does require you to be doing continuous deployment. If you want that, but don't fancy getting new software to your entire user base, you can deploy to a subset of users. In a recent project of ours, a retailer deployed its new online system first to its employees, then to an invited set of premium customers, and finally to all customers."

5.3.8 Cómo migrar la base de datos

#TODO

5.4 Cómo mantener vivo al sistema durante el proceso

La idea, como ya dijimos, no es descartar el sistema heredado de una sola vez, cosa que no vamos a lograr que nuestros usuarios acepten de buena gana. Por eso, hemos hablado de ir introduciendo las partes migradas incrementalente, usando SBE y microservicios. De esta manera, el sistema heredado sigue vivo y la evolución es gradual, de modo tal que la migración no afecte a la base de usuarios.
El uso de Strangler Application, de los Test Toggles, etc., apunta en esta dirección.

5.5 Cuándo terminar

Un producto de software exitoso siempre evoluciona [Brooks 1975]. Sin embargo, puede que haya partes del mismo que pasen años sin cambiar. Dado que nuestro procedimiento lo estamos basando en cambiar lo que necesitemos por las demandas de cambios de los clientes, puede ocurrir que, en algún momento, vayan a quedar partes del sistema monolítico que funcionen bien y no hayan sido migradas. Nuestra recomendación en este caso es no invertir esfuerzo en ellas: una manera elegante de hacerlo es definir una API para esos sectores y convertirlos en servicios que queden implementados como estaban.

5.6 El procedimiento completo

El diagrama de la Fig. 4 muestra el procedimiento completo que proponemos.
#TODO

6 Pruebas de concepto

Todo lo anterior resulta meramente especulativo si no se valida empíricamente en sistemas reales. Por ello, hemos encarado algunos trabajos académicos como pruebas de concepto.

6.1 Tesina sobre el uso de microservicios para reemplazar partes de un sistema heredado

6.1.1 Resumen del trabajo

#TODO

6.1.2 Hallazgos que aportan evidencia a este proyecto

#TODO

6.1.3 Amenazas a la validez de los resultados y trabajos futuros

#TODO

6.2 Tesina sobre el uso de SBE para recuperar el conocimiento de un sistema heredado

6.2.1 Resumen del trabajo

#TODO

6.2.2 Hallazgos que aportan evidencia a este proyecto

#TODO

6.2.3 Amenazas a la validez de los resultados y trabajos futuros

#TODO

6.3 Tesina sobre el uso combinado de SBE y microservicios

6.3.1 Resumen del trabajo

#TODO

6.3.2 Hallazgos que aportan evidencia a este proyecto

#TODO

6.3.3 Amenazas a la validez de los resultados y trabajos futuros

#TODO

6.4 Tesina que compara el costo de tirar y reescribir versus recuperar software heredado

6.4.1 Resumen del trabajo

#TODO

6.4.2 Hallazgos que aportan evidencia a este proyecto

#TODO

6.4.3 Amenazas a la validez de los resultados y trabajos futuros

#TODO

6.5 Tesina que analiza SBE como práctica metodológica para migrar a microservicios

6.5.1 Resumen del trabajo

#TODO

6.5.2 Hallazgos que aportan evidencia a este proyecto

#TODO

6.5.3 Amenazas a la validez de los resultados y trabajos futuros

#TODO

7 Conclusiones

7.1 Limitaciones

7.1.1 Limitaciones de la práctica de especificar con ejemplos

SBE plantea especificar por extensión. Sin embargo, no hay que olvidar que los requerimientos por intensión tienen un nivel de abstracción mayor y expresan cuestiones que no siempre podemos llevar a los ejemplos de una manera razonable: la generación de números al azar, las cuestiones de experiencia de usuario (en las que la interacción es un requerimiento más importante que el comportamiento) y tantas otras.

Ya Gojko Adzic llamaba la atención sobre este punto en sus primeras publicaciones [Adzic 2009]. Por lo tanto, así como toda historia de usuario debe ir acompañada de pruebas de aceptación, en SBE debemos poner el foco en que los escenarios o pruebas de aceptación deben estar agrupados por historias de usuario.

Estas mismas recomendaciones debemos trasladarlas a nuestro enfoque. Las especificaciones con ejemplos son la herramienta principal del mismo, pero hay lugar para otros tipos de especificaciones que en ocasiones deben acompañar a los ejemplos.

La crítica principal del enfoque de especificar con ejemplos vino de Bertrand Meyer: según él, son los contratos los que deben dirigir el diseño y las pruebas [Meyer 2011]. Su postura es que, si bien se pueden derivar pruebas individuales de los contratos, es imposible el camino inverso, ya que miles de pruebas individuales no pueden reemplazar la abstracción de una especificación contractual.

La respuesta habitual es que si bien las especificaciones abstractas son más generales que las pruebas concretas, estas últimas, precisamente por ser concretas, son más fáciles de comprender y acordar con los analistas de negocio y otros interesados [Mugridge 2005]. Lasse Koskela amplía esto diciendo que los ejemplos concretos son fáciles de leer, fáciles de entender y fáciles de validar [Koskela 2007].

7.1.2 Otras críticas

#TODO

7.2 Aportes del proyecto

#TODO

7.3 Trabajos relacionados

#TODO

7.4 Trabajos futuros

#TODO: Pero aun cuando rehacer el sistema fuera la mejor opción, necesitamos que se mantenga el comportamiento previo en la mayor parte del mismo, por lo que el método a seguir podría ser una variante menor del planteado en este proyecto.
#TODO: hay que probar lo que se dice en las limitaciones

8 Bibliografía y referencias

[Abbattista 2009] Abbattista, F., Bianchi, A., & Lanubile, F. (2009). A storytest-driven approach to the migration of legacy systems. Agile Processes in Software Engineering and Extreme Programming, 149-154.

[Adzic 2009] Adzic, G. (2009). Bridging the communication gap: specification by example and agile acceptance testing. Neuri Limited.

[Adzic 2011] Adzic, G.: Specification By Example. How successful teams deliver the right software. Manning Publications (2011)

[Almonaies 2010] Almonaies, A. A., Cordy, J. R., & Dean, T. R. (2010, March). Legacy system evolution towards service-oriented architecture. In International Workshop on SOA Migration and Evolution (pp. 53-62).

[Balalaie 2015] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2015). Microservices migration patterns. Tech. Rep. TR-SUTCE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran, Iran.

[Beck 1999] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional; US Ed edition (1999)

[Beck 2003] Beck, K. (2003). Test-driven development: by example. Addison-Wesley Professional.

[Bergey 1997] Bergey, John; Northrop, Linda; & Smith, Dennis. Enterprise Framework for the Disciplined Evolution of Legacy Systems (CMU/SEI-97-TR-007, ADA 330880). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Bernstein 2015] Bernstein, D. S. (2015). Beyond legacy code: nine practices to extend the life (and value) of your software. Pragmatic Bookshelf.

[Betts 2018] Thomas Betts. Patterns for Microservice Developer Workflows and Deployment. Q&A with Rafael Schloming. The InfoQ eMag: Microservices - Patterns and Practices. https://www.infoq.com/minibooks/microservices-patterns-practices (visitado el 29 de marzo de 2018).

[Bourque 2014] Bourque, P., & Fairley, R. E. (2014). Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press.

[Brodie 1995] Brodie, M. & Stonebraker, M. Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan Kaufmann Publishers, 1995.

[Brooks 1975] Brooks, F.: The Mythical Man-Month: Essays on Software Engineering, Addison Wesley (1975)-

[Cohn 2004] Cohn, M. (2004). User Stories Applied: For agile software development. Addison-Wesley Professional.

[Comella-Dorda 2000] Comella-Dorda, S., Wallnau, K., Seacord, R. C., & Robert, J. (2000). A survey of legacy system modernization approaches (No. CMU/SEI-2000-TN-003). Carnegie-Mellon univ pittsburgh pa Software engineering inst.

[Dascanio 2014] Dascanio, N.: Análisis de impacto de cambios realizados sobre sistemas con pruebas automatizadas con distinta granularidad. Tesina de grado de Ingeniería Informática en la Universidad de Buenos Aires, Argentina (2014). http://materias.fi.uba.ar/7500/Dascanio.pdf

[Dragoni 2017] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. In Present and Ulterior Software Engineering (pp. 195-216). Springer, Cham.

[Eickelmann 1996] Eickelmann, N. S., & Richardson, D. J. (1996, March). An evaluation of software test environment architectures. In Software Engineering, 1996., Proceedings of the 18th International Conference on (pp. 353-364). IEEE.

[Era 1964] Era, I. T. (1964). Software Engineering-Introduction. Wall Street Journal.

[Evans 2004] Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional. (2004)

[Feathers 2004] Feathers, M.: Working Effectively With Legacy Code. Prentice Hall (2004)

[Fontela 2012] Fontela, M. C. (2012). Estado del arte y tendencias en Test-Driven Development. Trabajo final de especialización en la Universidad Nacional de La Plata (2012). http://163.10.34.134/bitstream/handle/10915/4216/Documento_completo.pdf?sequence=1 como estaba el 17 de marzo de 2018.

[Fontela 2013] Fontela, C.: Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras. Tesis de maestría en la Universidad Nacional de La Plata, Argentina (2013). http://sedici.unlp.edu.ar/handle/10915/29096

[Fontela 2013-2] Fontela, C., Garrido, A., & Lange, A. (2013). Hacia un enfoque metodológico de cobertura múltiple para refactorizaciones más seguras. In Proceedings of the 14th Argentine Symposium on Software Engineering (ASSE 2013), Córdoba, Argentina.

[Foote 1997] Foote, B., & Yoder, J. (1997). Big ball of mud. Pattern languages of program design, 4, 654-692.

[Fowler 2004] Fowler, M. (2004). StranglerApplication, https://www.martinfowler.com/bliki/StranglerApplication.html

[Fowler 2010] Fowler, M. BlueGreenDeployment. https://martinfowler.com/bliki/BlueGreenDeployment.html [last accessed on November 8, 2017]

[Fowler 2013] Fowler, M. Continuous delivery. https://martinfowler.com/bliki/ContinuousDelivery.html [last accessed on November 8, 2017]

[Fowler 2014] Fowler, M., & Lewis, J. (2014). Microservices. ThoughtWorks. http://martinfowler. com/articles/microservices.html [last accessed on September 26, 2017].

[Sato 2014] Sato, Camilo. CanaryRelease. https://martinfowler.com/bliki/CanaryRelease.html [last accessed on November 8, 2017]

[Fowler 2015] Fowler, M. 2015. Monolith First. Accessed on 1 April 2017. Retrieved from https://martinfowler.com/bliki/MonolithFirst.html

[Ghahrai 2017] "Testing Microservices – A Beginner's Guide", Amir Ghahrai, https://www.testingexcellence.com/testing-microservices-beginners-guide/

[Gärtner 2012] Gärtner, M. (2012). ATDD by example: a practical guide to acceptance test-driven development. Addison-Wesley.

[Hammant 2013] Hammant, P., Strangler Applications, https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

[Hammant 2017] Hammant, P., The Rise of Test Impact Analysis. ThoughtWorks, https://martinfowler.com/articles/rise-test-impact-analysis.html

[Humble 2010] "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Jez Humble, David Farley, Addison-Wesley, 2010.

[Jonsson 2017] Jonsson, A. (2017). The Impact of Refactoring Legacy Systems on Code Quality Metrics.

[Koskela 2007] Koskela, L.: Test Driven: TDD and Acceptance TDD for Java Developers. Manning Publications (2007).

[Lehman 1980] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060-1076.

[Lehman 1997] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997, November). Metrics and laws of software evolution-the nineties view. In Software Metrics Symposium, 1997. Proceedings., Fourth International (pp. 20-32). IEEE.

[Levcovitz 2016] Levcovitz, A., Terra, R., & Valente, M. T. (2016). Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. arXiv preprint arXiv:1605.03175.

[Meyer 2011] Bertrand Meyer, "Test or spec? Test and spec? Test from spec!", http://www.eiffel.com/general/column/2004/september.html, como estaba en junio de 2011.

[Mosler 2006] Mosler, C.: E-CARES Project: Reengineering of PLEX Systems. Softwaretechnik-Trends, 26(2), 59-60. (2006)

[Marick 2008] Brian Marick, "Exploration Through Example", http://www.exampler.com/blog/2008/03/23/an-alternative-to-business-facing-tdd/, como estaba en noviembre de 2017.

[Mugridge 2005] Rick Mugridge y Ward Cunningham, "Fit for Developing Software: Framework for Integrated Tests", Prentice Hall, 2005.

[Mugridge 2008] Mugridge, R. (2008). Managing agile project requirements with storytest-driven development. IEEE software, 25(1).

[Nadareishvili 2016] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice Architecture: Aligning Principles, Practices, and Culture. " O'Reilly Media, Inc.".

[Naur 1985] Naur, P. (1985). Programming as theory building. Microprocessing and microprogramming, 15(5), 253-261.

[Newman 2015] Newman, S. (2015). Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.".

[North 2006] North, D.: Behavior Modification. The evolution of behavior-driven development. Better Software Magazine. Volume-Issue: 2006-03. (2006)

[Park 2011] Park, S. S. (2011). Communicating domain knowledge through example-driven story testing. University of Calgary, Department of Computer Science.

[Perrey 2003] Perrey, R., & Lycett, M. (2003, January). Service-oriented architecture. In Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on (pp. 116-119). IEEE.

[Pugh 2010] Pugh, K. (2010). Lean-Agile acceptance test-driven-development. Pearson Education.

[Rahman 2015] Rahman, M., & Gao, J. (2015, March). A reusable automated acceptance testing architecture for microservices in behavior-driven development. In Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on (pp. 321-325). IEEE.

[Rahman 2016] Rahman, M. T., Querel, L. P., Rigby, P. C., & Adams, B. (2016, May). Feature toggles: practitioner practices and a case study. In Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on (pp. 201-211). IEEE.

[Ranney 2016] Ranney, M. 2016. What I wish I had known before scaling Uber to 1000 services. Video recording from GOTO Conference in Chicago 2016. Accessed on 28 March 2017. Retrieved from https://www.youtube.com/watch?v=kb-m2fasdDY

[Reppert 2004] Reppert, T. (2004). Don\'t just break software: make software. Better Software, (July/August), 18-23.

[Richards 2015] Richards, M. (2015). Microservices vs. service-oriented architecture. O'Reilly.

[Rook 2016] Rook, Michiel, The Strangler Pattern in Practice, https://www.michielrook.nl/2016/11/strangler-pattern-practice/

[Seacord 2001] Seacord, R. C., Comella-Dorda, S., Lewis, G., Place, P., & Plakosh, D. (2001). Legacy System Modernization Strategies (No. CMU/SEI-2001-TR-025). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[Seng 1999] Seng, Jia-Lang & Tsai, Wayne. "A Structure Transformation Approach for Legacy Information Systems- A Cash Receipts/Reimbursement Example", Proceedings on the 32nd Hawaii International Conference on System Sciences, 1999.

[Sneed 2006] Sneed, H. M. (2006, February). Wrapping legacy software for reuse in a SOA. In Multikonferenz Wirtschaftsinformatik (Vol. 2, pp. 345-360).

[Sridharan 2017] "Testing Microservices, the sane way", Cindy Sridharan, https://medium.com/@copyconstruct/testing-microservices-the-sane-way-9bb31d158c16

[Stevenson 2004] Stevenson, C., & Pols, A. (2004). An agile approach to a legacy system. Lecture notes in computer science, 3092, 123-129.

[Sundar 2017] Arvind Sundar, White Peper "An Insight into Microservices Testing Strategies", Infosys, 2017, disponible en https://www.infosys.com/it-services/validation-solutions/white-papers/documents/microservices-testing-strategies.pdf

[Vernon 2013] Vernon, V. (2013). Implementing domain-driven design. Addison-Wesley.

[Walls 2013] Walls, M. (2013). Building a DevOps culture. " O'Reilly Media, Inc.".

[Weiderman 1997] Weiderman, N. H., Bergey, J. K., Smith, D. B., & Tilley, S. R.. Approaches to Legacy System Evolution (No. CMU/SEI-97-TR-014). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. (1997)

[Yanaga 2017] Yanaga, Edson, Migrating to Microservice Databases: From Relational Monolith to Distributed Data. O'Reilly Media, Inc.

[Zaymus 2017] Zaymus, M. (2017). Decomposition of monolithic web application to microservices. https://www.theseus.fi/bitstream/handle/10024/131110/Zaymus_thesis.pdf (octubre 2017)

9 Anexos

9.1 Anexo A: Sobre los términos utilizados en el proyecto

9.1.1 Ayuda de colegas

Partimos de: "enfoque metodológico para mantener sistemas X con escasa Y"
Recibí las siguientes sugerencias:
X = nativos, legacy, sistemas sin especificaciones formales, aplicaciones cuyos desarrolladores originales han dejado el proyecto, huérfanos (ya no están sus creadores disponibles)
Y = especificaciones formales, artefactos que posibilitan entender el modo de funcionamiento del sistema, conocimiento (DF: tal vez lo que se pierda no es la documentación sino el conocimiento)
EY plantea: sistemas sin especificaciones que posibiliten entender su funcionamiento

Una primera aproximación: Hacia un enfoque metodológico para el mantenimiento de sistemas de los que se ha ido perdiendo el conocimiento
En inglés: Towards an Evolution Approach for Legacy Software based in Specification By Example and Microservices

Naur habla de "revival" => reconstrucción, recuperación, restablecimiento, reanimación

9.1.2 Términos adoptados en castellano

Heredado como sinónimo de legacy, de que se ha perdido el conocimiento
En vez de hablar de mantenimiento ponemos el foco en la **recuperación**, que debería facilitar el mantenimiento y la evolución
En general, prefiero "**software heredado**" a "sistemas heredados", aunque depende del contexto

9.1.3 Términos adoptados en inglés

Legacy => es el término más difundido y el que usa Feathers, de mucha aceptación en la comunidad ágil; ojo que tal vez requiera aclaración en algunos ámbitos
Revival es el término que usa Naur, pero a mí me suena medio hippie
Sinónimos de revival que me suenan más: recovery, recuperation, restoration; creo que me quedo con **recovery** o revival

9.2 Anexo B: ejemplos de código de SBE

Cucumber y Java:

```
Feature: Frequent Flyer status is calculated based on points
As a Frequent Flyer member
I want my status to be upgraded as soon as I earn enough points
So that I can benefit from my higher status sooner

Scenario: New members should start out as BRONZE members
Given Jill Smith is not a Frequent Flyer member
When she registers on the Frequent Flyer program
Then she should have a status of BRONZE

Scenario Outline:
Given Joe Jones is a <initialStatus> Frequent Flyer member
And he has <initialStatusPoints> status points
When he earns <extraPoints> extra status points
Then he should have a status of <finalStatus>

Examples: Status points required for each level
| initialStatus | initialStatusPoints | extraPoints | finalStatus |
| Bronze | 0 | 300 | Silver |
| Bronze | 100 | 200 | Silver |
| Silver | 0 | 700 | Gold |
| Gold | 0 | 1500 | Platinum |
```

un test de BDD en JUnit

```
@RunWith(Parameterized.class)
public class WhenEarningStatusLevels {
@Parameters
      public static Collection pointsPerStatus() {
            return Arrays.asList(new Object[][]{
                  {Bronze, 0, 100, Bronze},
                  {Bronze, 0, 300, Silver},
                  {Bronze, 100, 200, Silver},
                  {Silver, 0, 700, Gold},
                  {Gold, 0, 1500, Platinum}
            });
      }
      Status initialStatus, finalStatus;
      int initialPoints, earnedPoints;
      public WhenEarningStatusLevels(Status initialStatus,
                  int initialPoints,
```

```
                  int earnedPoints,
                  Status finalStatus) {
            this.initialStatus = initialStatus;
            this.initialPoints = initialPoints;
            this.earnedPoints = earnedPoints;
            this.finalStatus = finalStatus;
      }
      @Test
      public void should_earn_new_status_based_on_point_thresholds() {
            FrequentFlyer member
                  = FrequentFlyer.withFrequentFlyerNumber("12345678")
                        .named("Joe", "Jones")
                        .withStatusPoints(initialPoints)
                        .withStatus(initialStatus);
            member.earns(earnedPoints).statusPoints();
            assertThat(member.getStatus()).isEqualTo(finalStatus);
      }
}
```