# Migración de sistemas heredados hacia microservicios con el soporte de especificaciones mediante ejemplos

Carlos Fontela[1]

[1] Laboratorio de Métodos de Desarrollo y Mantenimiento de Software (LIMET), Facultad de Ingeniería Universidad de Buenos Aires, Argentina
cfontela@fi.uba.ar

**Resumen.** El presente trabajo plantea una propuesta metodológica para recuperar sistemas monolíticos heredados, migrando hacia una arquitectura de microservicios. La propuesta se basa en especificar con ejemplos, una práctica que ha se ha utilizado con éxito en el desarrollo de software nuevo, pero que también resulta ventajosa para la recuperación de software. La evolución hacia microservicios trae aparejada también una mayor facilidad de mantenimiento futuro. Varias de estas ideas se encuentran en la literatura, pero no hemos encontrado evidencia de la validación empírica del planteo, por lo que estamos validando la propuesta en trabajos finales de carrera de alumnos de grado.

**Palabras clave:** software heredado, recuperación, especificaciones mediante ejemplos, microservicios.

## 1 Introducción

En esta sección vamos a establecer a qué llamamos software heredado, explicaremos en qué consiste la práctica metodológica denominada Specification By Example y presentaremos la arquitectura de microservicios. La sección 2 explora trabajos relacionados. La sección 3 es el planteo de la propuesta metodológica en sí, cuya validación empírica, que estamos abordando, se explica en la sección 4. Las conclusiones se presentan en la sección 5.

### 1.1 Software heredado

Llamamos *software heredado* (o *legacy*) a aquél que ha alcanzado un grado de éxito relativo, pero que resulta difícil de mantener porque se ha perdido el conocimiento que permitiría hacerlo [1].

El haber caído en esta categoría se puede deber a una o más de las siguientes razones:

- Han sido desarrollados por equipos de trabajo que ya no trabajan en él.
- No hay pruebas de aceptación ni técnicas que permitan probar el funcionamiento del código o entender cómo funciona.
- Tienen escasa documentación funcional, técnica o para usuarios.
- Muchas funcionalidades son desconocidas por los desarrolladores y usuarios.
- Mantienen un gran volumen de deuda técnica que impide probarlos y comprenderlos.

### 1.2 Especificaciones mediante ejemplos como práctica de desarrollo

Cuando Kent Beck presentó Extreme Programming hizo especial hincapié en especificar mediante historias de usuario acompañadas por pruebas de cliente [2]. Luego de ello, fueron surgiendo algunas prácticas, relacionadas entre sí, que se basan en utilizar pruebas de aceptación para derivar el comportamiento del sistema [3, 4, 5, 6, 7].

De todas ellas, la que nos parece más adecuada para nuestro análisis es la denominada SBE (Specification By Example), que fue presentada originalmente como una práctica para mejorar la comunicación entre los distintos roles de un proyecto de desarrollo, y ha ido evolucionando hasta transformarse en una práctica colaborativa de construcción basada en especificaciones mediante ejemplos que sirven como pruebas de aceptación [6].

El procedimiento propuesto por SBE consiste en trabajar de manera incremental, tomando de a una historia de usuario a la vez. Para cada historia de usuario se determinan los criterios y las pruebas de aceptación. De allí en más, se trabaja mediante un enfoque de afuera hacia adentro, partiendo de funcionalidades y sus casos de aceptación y refinando el diseño en varios ciclos de desarrollo basados en pruebas unitarias y de componentes, de las cuales se deriva el código. El éxito de los criterios de aceptación es el que permite conocer cuándo se ha satisfecho un requerimiento: hay que ver si todas las pruebas de aceptación de la historia de usuario están funcionando.

Por lo tanto, la idea subyacente en SBE es usar ejemplos como parte de las especificaciones, y que los mismos sirvan para probar el sistema, haciendo que todos los roles (usuarios, analistas, desarrolladores, testers) se manejen con los mismos ejemplos. De esta manera, los ejemplos sirven como herramienta de comunicación, son más concretos para acordar en las conversaciones con clientes, no hay divergencia entre lo que comprende cada rol y sirven como especificaciones ejecutables (potencialmente automatizables) que hacen de pruebas de aceptación.

Una de las recomendaciones básicas de SBE es que las especificaciones se construyan en talleres multidisciplinarios, de los cuales participan todos los interesados. 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.

Un subproducto interesante de los talleres de especificaciones es la emergencia de un lenguaje común, en el sentido del lenguaje ubicuo que plantea Eric Evans en Domain Driven Design (DDD) [8]. 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, de modo tal que se convierte en un proceso de construcción de conocimiento [9].

### 1.3 Microservicios como patrón arquitectónico

La arquitectura de microservicios plantea una aplicación como un conjunto de pequeños componentes, cada uno corriendo en su propio proceso y comunicándose mediante mecanismos livianos [10]. Los componentes pueden desplegarse independientemente mediante procesos totalmente automatizados. A la vez, cada uno puede ser escrito en un lenguaje de programación y usar un mecanismo de persistencia distintos. Cada microservicio puede verse como un componente cohesivo y autónomo trabajando en conjunto con otros microservicios para entregar valor de negocio [11].

Se trata de una evolución de las arquitecturas SOA que emergieron en la década de 1990 [12]. En efecto, ya SOA preconizaba separar la implementación de componentes con interfaces bien definidas, alentando el bajo acoplamiento, la reusabilidad y la autonomía, pudiendo desarrollar distintos componentes con distintas tecnologías. La arquitectura de microservicios busca corregir los desvíos que se produjeron en la industria respecto de los objetivos originales de SOA [13].

Según una visión, la adopción de esta arquitectura ocurre por la necesidad de las organizaciones centradas en el software de aumentar la velocidad con la que liberan sus productos [14]. De esta manera, la necesidad de aumentar la velocidad de las liberaciones es la que provoca la organización de equipos polifuncionales (DevOps [15]) y cambios en los procesos, que a su vez lleva a la necesidad de entregar el producto en forma de microservicios independientes.

Las características generales de esta arquitectura son:

- Los servicios se organizan alrededor de capacidades individuales de negocio.
- Los microservicios incluyen todas las capas del sistema, incluyendo acceso a datos persistentes, modelo de negocio e interfaces externas.
- Los servicios deben ser desplegables en forma independiente, y posiblemente en plataformas diferentes.
- Los límites entre microservicios deben estar bien definidos.
- No hay un control centralizado de los servicios, sino que son los propios microservicios los que poseen la lógica de control.

En general, y si bien no es estrictamente necesario por definición, se requiere automatizar todo lo posible la infraestructura y los despliegues, como lo propone la práctica de Continuous Delivery [16]. Por otro lado, la comunicación suele basarse en eventos asincrónicos mediante eventos, más que en llamadas sincrónicas.

Como contrapartida, los sistemas de software cuyos módulos no se puedan desplegar y ejecutar independientemente, los llamaremos *sistemas monolíticos*.

Hay algunas ventajas que se han ido poniendo en evidencia de las aplicaciones basadas en microservicios, que han aumentado mucho su popularidad. Entre ellas se destacan:

- Cada cambio en un microservicio se puede desarrollar y desplegar independientemente del resto del sistema.
- Reduce el tiempo de liberación para cada funcionalidad.
- Cada microservicio puede ser desarrollado usando las tecnologías más adecuadas en cada caso. Esto mismo permite adoptar nuevas tecnologías de a partes.
- El sistema resultante será más escalable y tolerante a fallas.
- Si se encuentra un problema en producción debido a un despliegue reciente, se puede aislar el microservicio y corregir sin afectar al resto del sistema.
- Permite reemplazar partes de un sistema monolítico sin tener que atarse a tecnologías del monolito previo y sin desecharlo.

No obstante, hay varios desafíos que se plantean también al usar microservicios. La mayor parte de los mismos surgen de la mayor complejidad de los sistemas distribuidos, que impactan en las pruebas, las depuraciones, la seguridad, el desempeño y la dificultad de las refactorizaciones, entre otros. Tanto es así que Fowler propone siempre comenzar por un sistema monolítico y llevarlo a una arquitectura de microservicios más adelante, incluso comenzando por servicios de grano grueso [17].

## 2 Trabajos relacionados

Existen numerosos trabajos que destacan la importancia de la recuperación del software heredado con vistas a su evolución, así como también comienzan a aparecer artículos que tratan la migración de sistemas monolíticos hacia microservicios. En esta sección haremos un breve repaso de los más relevantes para nuestra propuesta.

### 2.1 Sobre la importancia de la recuperación del software heredado

Hace décadas que se reconoce la importancia del mantenimiento y la evolución del software [18]. Hay un antiguo estudio del Software Engineering Institute (SEI) 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 [19].

Más allá de todos estos antecedentes y del tiempo transcurrido, que parecen darle al mantenimiento cierta relevancia, el material disponible contrasta con 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. Incluso hay bastante literatura que plantea técnicas y procesos para el reemplazo liso y llano del software heredado, en vez de aplicarse a su recuperación [20, 21].

Sin embargo, algunos trabajos iniciales brindan algunas ideas para la recuperación. Peter Naur sostiene que programar es construir conocimiento, y que si un producto ya no cuenta con el equipo que tenía el conocimiento que aquél representa, el programa debe considerarse muerto, de modo tal que lo debemos encarar es un proceso de "resurrección" [22]. Si bien la analogía nos parece exagerada, toda vez que, en realidad, el software en cuestión no lo podemos considerar muerto si está siendo utilizado, adherimos a las ideas del autor, cambiando el término por "recuperación".

Respecto de cómo recuperar ese conocimiento, el SEI viene estudiando a fondo el tema desde 1997, cuando por primera vez 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 [23]. Unos años más tarde, el mismo SEI publicó un estudio de los distintos enfoques para modernizar sistemas heredados [19]. Aquí se aboga por lo que llama modernización de caja negra, basada en envolturas (*wrapping*) que deben rodear al sistema heredado con una capa de software que oculte la complejidad no deseada del antiguo sistema y exponga una nueva interfaz. A su vez, en 2001 el SEI comienza a tratar la modernización incremental de sistemas heredados, sobre todo debido a la necesidad de mantener operativos en todo momento a sistemas grandes y complejos [24].

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 para 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.

El auge del movimiento ágil llevó a buscar planteos alternativos. El más conocido es el de Michael Feathers, que parte de la premisa de que el código heredado es aquél que carece de pruebas, refiriéndose mayormente a pruebas unitarias automatizadas [25]. 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.

Entendemos que ninguna de las propuestas anteriores es del todo satisfactoria:

- El enfoque de tirar y reescribir nos permitiría reconstruir el sistema completo desde cero siguiendo buenas prácticas de ingeniería, de lo cual se esperaría que surja un producto de mejor calidad. Sin embargo, esta solución es poco factible por el costo y el tiempo de desarrollo. Además, 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 producto en un blanco móvil.
- El planteo de algunos autores, que sostienen que, dado que lo que hay que comprender es el comportamiento observable, habría que recurrir a los manuales de usuario y a la documentación de las interfaces programáticas (API), lo vemos riesgoso [23]. Uno de los problemas habituales de la documentación que ha sobrevivido separada del código mucho tiempo es que está desactualizada o fuera de sincronía con el código. Resulta por lo tanto peligroso dar por buena a la documentación, sabiendo que el sistema funciona bien, y que lo que funciona está representado en el código, no necesariamente en documentos.
- El enfoque basado en pruebas unitarias, o a lo sumo de componentes, tampoco lo vemos del todo satisfactorio. Necesitamos saber qué es lo que el usuario espera del sistema, y para ello las pruebas más adecuadas son las de aceptación. Por otro lado, en el caso del 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 de componentes. Por eso necesitamos pruebas que evidencien el comportamiento desde la perspectiva de interacciones reales de usuarios.

Por supuesto, existen planteos muy interesantes desde la academia. Uno de ellos aborda un enfoque metodológico más centrado en la gente que en los documentos [26]. 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 de Bari, en el que se pone a prueba un enfoque basado en pruebas de aceptación de historias de usuario [27]. El enfoque, planteado con el nombre de Storytest-Driven Migration (STDM), trabaja con las 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, lo cual está en línea con nuestra propuesta.

### 2.2 Enfoques para la migración a microservicios

Hay bastante literatura que refiere a migrar sistemas heredados a arquitecturas SOA, que es aplicable también a microservicios [28]. La mayoría de las propuestas sugieren que hay que empezar por separar contextos de negocio (similares a los "*bounded contexts*" de DDD [8]) que luego devendrán servicios. En cuanto a la migración incremental, una posibilidad es generar una capa de servicios, con su interfaz, alrededor de los componentes heredados.

Asimismo, hay una publicación que plantea varios patrones de migración hacia microservicios [29]. La propuesta incluye el uso de DDD para identificar subdominios del negocio y convertir cada subdominio en una unidad de despliegue, separar por frecuencia de cambios y descomponer el monolito por el acceso a datos. 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 [30].

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 [31]. La idea básica es ir rodeando al antiguo sistema con partes nuevas e ir estrangulándolo hasta que desaparezca. En los últimos años han ido surgiendo algunos casos de estudio interesantes que plantean el uso de este patrón para la migración de un monolito heredado hacia una arquitectura más moderna [32, 33]. La idea que ha ido ganando aceptación es la de 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. Este planteo se puede ver en detalle en una tesina que propone interceptar las llamadas al sistema viejo y reenviarlas al nuevo a través de su interfaz programática (API) [34].

Una idea parecida surge en el marco de DDD, cuando Evans introduce la noción de una capa anti corrupción [8]. El autor plantea hacer una API del sistema monolítico, que así se convierte en un servicio gigante, que de a poco vamos estrangulando.

## 3 Migrando hacia microservicios con la ayuda de ejemplos

El presente trabajo encara una práctica metodológica para migrar de manera segura sistemas monolíticos heredados a una arquitectura de microservicios.

Dado que en la mayor parte de las situaciones en que se debe trabajar con sistemas heredados el principal problema es la reconstrucción del conocimiento, la práctica de especificar con ejemplos, con su carga de actividad colaborativa y de descubrimiento de requerimientos se nos presenta como el enfoque ideal para ir recuperando el conocimiento que los mismos representan.

Uno de los autores que sugiere usar especificaciones con ejemplos para la evolución de software heredado es Ken Pough, que basa su propuesta en la afirmación de

que las pruebas de aceptación documentan cómo funciona realmente el sistema [35]. Sin embargo, como pasa en tantos otros casos, queda como una enunciación sin ejemplos concretos ni evidencia empírica de haberlo llevado adelante. Precisamente por la falta de trabajos que traten este tema a fondo y lo validen en casos concretos, es que lo hemos abordado como un tema de investigación.

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

Para evitar regresiones, frecuentes en sistemas monolíticos heredados, conviene realizar pruebas de regresión a menudo, y por lo mismo es importante automatizar las pruebas de aceptación, o al menos una proporción importante de las mismas.

Proponemos también aplicar el método solamente por demanda, es decir, reuniendo a los talleres a medida que surgen pedidos de cambios sobre el sistema. 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, sea porque funcionan bien como están o porque no se usan demasiado. Una manera consistente de hacerlo es definir una API para esos fragmentos y convertirlos en servicios que queden implementados como estaban en el sistema monolítico.

Esta forma de trabajo implica reemplazar el sistema de manera incremental, extrayendo módulos que pasan a interactuar con el sistema antiguo a través de alguna API. Estos módulos son los que se transforman en microservicios. La ventaja de este enfoque es que los módulos del sistema que más se usan y más cambios requieren irán mejorando su calidad poco a poco, por lo que cada vez resulta más sencillo mantener las partes más críticas del sistema. A la vez, el sistema heredado sigue vivo y la evolución no afecta a la base de usuarios.

Podemos ver el procedimiento de manera gráfica en la Fig. 1.

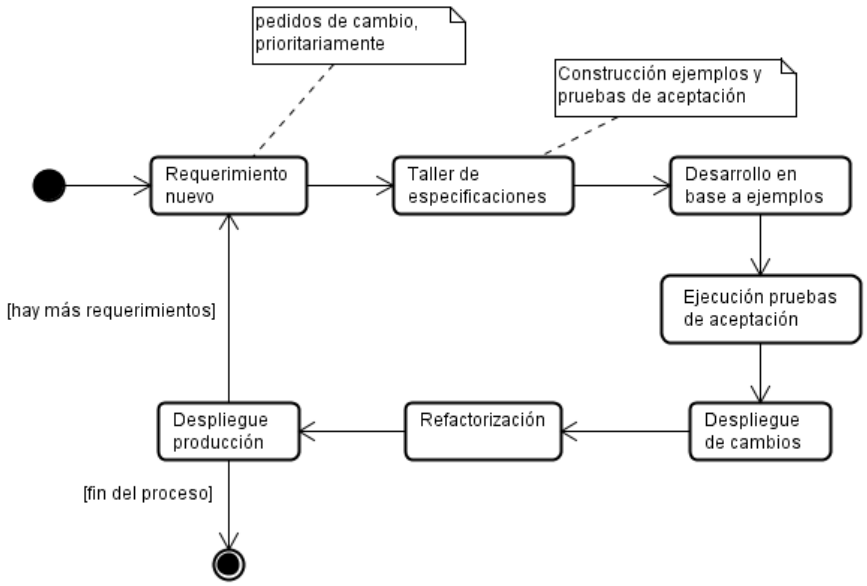


**Fig. 1.** Procedimiento propuesto.

## 4 Sondeos experimentales

El presente trabajo presenta una idea que ha sido validada sólo parcialmente, en el marco de algunas tesinas de grado dirigidas por el autor. Por ejemplo, la tesina de Nicolás Dascanio ya planteaba utilizar pruebas de aceptación para facilitar el mantenimiento de software heredado [36]. También se abordó un enfoque parecido en el trabajo final de carrera de Lionel Raymundi, en este caso previo a una reingeniería de un sistema bancario [37]. Pero no se ha seguido el procedimiento completo ni se ha intentado una migración hacia microservicios.

Como creemos que las prácticas de la ingeniería de software deben descansar fuertemente en evidencia empírica, lo anterior no alcanza. Por lo tanto, para una validación más sólida de la metodología propuesta, estamos comenzando a trabajar con alumnos de grado que están desarrollando trabajos finales de la carrera de Ingeniería Informática de la Universidad de Buenos Aires. A continuación se enumeran los trabajos que hemos comenzado:

- Un Trabajo Profesional consistente en migrar un sistema sin documentación a una arquitectura parcialmente basada en microservicios, guiados por especificaciones mediante ejemplos. En el mismo trabaja un equipo de tres alumnos que irán simulando los distintos roles del proceso.

- Otro Trabajo Profesional para modernizar un sistema de avisos clasificados en línea guiados por especificaciones mediante ejemplos. En el mismo trabaja un equipo de dos alumnos más uno o dos usuarios reales de la aplicación para validar los ejemplos y dar sustento real a la práctica.
- Un tercer Trabajo Profesional que pretende la recuperación del conocimiento de un sistema del gobierno federal argentino, trabajando con especificaciones mediante ejemplos. En el mismo trabaja un alumno con varios usuarios reales de la aplicación para validar los ejemplos.
- Una tesis de grado en la que un alumno estudiará patrones de migración a microservicios y su adecuación a un sistema existente, en parte implementado con esta arquitectura y monolítico por el resto.

En los cuatro casos los alumnos trabajarán guiados por uno o más docentes del LIMET, que oficiarán de tutores o co-tutores de los mismos.

Esperamos que estos experimentos nos sirvan para validar en mayor profundidad algunas prácticas que hemos planteado. También esperamos contar pronto con más trabajos que nos permitan seguir explorando nuestro planteo y encontrando limitaciones al mismo.

## 5 Conclusiones

La arquitectura de microservicios se plantea como una respuesta a la necesidad de liberar más frecuentemente incrementos de producto en las organizaciones basadas en software. Por otro lado, en general, no se llega a la arquitectura de microservicios desde el primer momento del desarrollo, sino luego de haber desarrollado un sistema monolítico exitoso. Sin embargo, muchos sistemas monolíticos caen en la categoría de software heredado, en el sentido de que se ha perdido todo o parte del conocimiento que había sobre su funcionamiento.

Por lo tanto, hemos elaborado una propuesta metodológica de recuperación de software heredado y monolítico hacia una arquitectura de microservicios. La propuesta plantea los siguientes requisitos:

- La migración debe ser incremental.
- Los incrementos deben encararse prioritariamente por demanda de los clientes.
- Toda la funcionalidad no migrada seguirá como estaba en la versión previa a la migración hasta tanto se decida migrarla.
- En todo momento se deberá trabajar con los usuarios mediante el procedimiento propuesto por SBE.
- Las pruebas se deberán automatizar para permitir pruebas de regresión, a la vez que quedan especificaciones ejecutables para el futuro.

- La mejora en el diseño a medida que se avanza en la migración sirve para ir realimentando la sencillez y velocidad de las mejoras futuras.
- La mejora en las especificaciones a medida que se avanza en la migración sirve para ir realimentando la calidad de las mismas y facilitar las mejoras futuras.

Como también dijimos, la propuesta está comenzando a ser validada en trabajos finales de carrera en la UBA, todos ellos con un fuerte anclaje empírico y sobre sistemas existentes reales.

## Referencias


1. Bernstein, D. S.: Beyond legacy code: nine practices to extend the life (and value) of your software. Pragmatic Bookshelf (2015)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional; US Ed edition (1999)
3. Reppert, T.: Don't just break software: make software. Better Software, (July/August), (2004) 18-23
4. Mugridge, R.: Managing agile project requirements with storytest-driven development. IEEE software, 25(1) (2008)
5. Adzic, G.: Bridging the communication gap: specification by example and agile acceptance testing. Neuri Limited (2009)
6. Adzic, G.: Specification By Example. How successful teams deliver the right software. Manning Publications (2011)
7. Gärtner, M.: ATDD by example: a practical guide to acceptance test-driven development. Addison-Wesley (2012)
8. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
9. Park, S. S.: Communicating domain knowledge through example-driven story testing. University of Calgary, Department of Computer Science (2011)
10. Fowler, M., & Lewis, J.: Microservices. ThoughtWorks. (2014) http://martinfowler.com/articles/microservices.html (visitado el 4/4/ 2018)
11. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly Media, Inc. (2015)
12. Perrey, R., & Lycett, M.: Service-oriented architecture. In Applications and the Internet Workshops. Proceedings 2003 Symposium. IEEE (2003) 116-119
13. Richards, M.: Microservices vs. service-oriented architecture. O'Reilly (2015)
14. Betts, T: Patterns for Microservice Developer Workflows and Deployment. Q&A with Rafael Schloming. The InfoQ eMag: Microservices - Patterns and Practices (2018). https://www.infoq.com/minibooks/microservices-patterns-practices (visitado el 4/4/ 2018)
15. Walls, M.: Building a DevOps culture. O'Reilly Media, Inc. (2013)
16. Humble, J. & Farley D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley (2010)
17. Fowler, M.: Monolith First. (2015) https://martinfowler.com/bliki/MonolithFirst.html (visitado el 4/4/ 2018)
18. Era, I. T.: Software Engineering-Introduction. Wall Street Journal (1964)

19. Comella-Dorda, S., Wallnau, K., Seacord, R. C., & Robert, J.: A survey of legacy system modernization approaches (No. CMU/SEI-2000-TN-003). Carnegie-Mellon University. Software Engineering Institute (2000)
20. Brodie, M. & Stonebraker, M.: Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan Kaufmann Publishers (1995)
21. Seng, J. & Tsai, W.: A Structure Transformation Approach for Legacy Information Systems- A Cash Receipts/Reimbursement Example, Proceedings on the 32nd Hawaii International Conference on System Sciences (1999)
22. Naur, P.: Programming as theory building. Microprocessing and microprogramming, 15(5), (1985) 253-261
23. 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 University. Software Engineering Institute (1997)
24. 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 University. Software Engineering Institute (2001)
25. Feathers, M.: Working Effectively With Legacy Code. Prentice Hall (2004)
26. Stevenson, C., & Pols, A.: An agile approach to a legacy system. Lecture notes in computer science, 3092, (2004) 123-129
27. Abbattista, F., Bianchi, A., & Lanubile, F.: A storytest-driven approach to the migration of legacy systems. Agile Processes in Software Engineering and Extreme Programming, (2009) 149-154
28. Almonaies, A. A., Cordy, J. R., & Dean, T. R.: Legacy system evolution towards service-oriented architecture. In International Workshop on SOA Migration and Evolution (2010) 53-62
29. Balalaie, A., Heydarnoori, A., & Jamshidi, P.: Microservices migration patterns. Tech. Rep. TR-SUTCE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran (2015)
30. Levcovitz, A., Terra, R., & Valente, M. T.: Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. arXiv preprint arXiv:1605.03175 (2016)
31. Fowler, M.: StranglerApplication (2004). https://www.martinfowler.com/bliki/StranglerApplication.html (visitado el 4/4/ 2018)
32. Hammant, P., Strangler Applications (2013). https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/ (visitado el 4/4/ 2018)
33. Rook, M.: The Strangler Pattern in Practice (2016). https://www.michielrook.nl/2016/11/strangler-pattern-practice/ (visitado el 4/4/ 2018)
34. Zaymus, M.: Decomposition of monolithic web application to microservices (2017). https://www.theseus.fi/bitstream/handle/10024/131110/Zaymus_thesis.pdf (visitado el 4/4/ 2018)
35. Pugh, K.: Lean-Agile acceptance test-driven-development. Pearson Education (2010)
36. 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 (visitado el 6/4/ 2018)
37. Raymundi, L.: Reingeniería de un sistema bancario legacy desarrollado en Java. Trabajo profesional de grado de Ingeniería Informática en la Universidad de Buenos Aires, Argentina (2015). Inédito.