PhD THESIS

Promoting End-User Involvement in Web-based Tasks: a Model-Driven Engineering Approach to Form-Filling and User-Acceptance Testing

Itziar Otaduy Igartua

Supervisor: Óscar Díaz García

2020

Universidad del País Vasco Euskal Herriko Unibertsitatea

(cc) 2021 Itziar Otaduy Igartua (cc by-nc-sa 4.0)

ES

Resumen de la Tesis

Introducción

Con el uso extendido de dispositivos electrónicos tanto en el hogar como en el lugar de trabajo, la Web se ha convertido en una parte integral de la vida de muchas personas. Por lo tanto, muchas aplicaciones que fueron diseñadas para escritorio han hecho gradualmente la transición a la Web. Por ejemplo, Google y Microsoft ya ofrecen versiones en línea de editores de documentos, hojas de cálculo y presentaciones de diapositivas.

Esta difusión de las aplicaciones online ha dado lugar a un emergente interés por facultar a los usuarios para que comprueben, adapten y personalicen la forma en que navegan y hacen uso de esas aplicaciones. La Automatización Web (i.e., la automatización de la ejecución de flujos de navegación), la Aumentación Web (i.e., la personalización de la interfaz de la Web) o los "mashups" (i.e., la integración de información de la Web) son algunas de las técnicas que permiten que los usuarios saquen el máximo provecho de la Web. Los beneficios de estas propuestas son varios. Por su parte, la automatización web puede ahorrar tiempo y evitar cometer errores cuando los usuarios tienen que realizar tareas repetitivas o complejas. Dos ejemplos de ello son el rellenado de formularios y el testeo de aplicaciones web. En el primer caso, herramientas como iMacros son ampliamente utilizadas, y permiten a los usuarios crear sus propios flujos de trabajo automatizados utilizando una técnica de grabación y reproducción. El grabador de iMacros graba las acciones del usuario y permite luego reproducirlas para lograr, por ejemplo, el rellenado automático de formularios web. Esta automatización ahorra tiempo a la hora de realizar estas monótonas tareas. En lo que se refiere al testeo de aplicaciones Web, Selenium IDE es una herramienta muy conocida que, al igual que iMacros, proporciona un grabador para que el usuario pueda crear fácilmente sus propios scripts de test. Durante el desarrollo de una aplicación web, Selenium permite que cualquier persona pueda crear fácil y rápidamente conjuntos de tests que pueden ser más adelante

reproducidos por cualquier otra persona.

Dicho esto, las herramientas actuales podrían quedarse cortas en algunas situaciones. Por ejemplo, en el campo del rellenado de formularios podría ser necesario extender los scripts grabados si queremos utilizar para el rellenado información extraída de fuentes externas (archivos CSV, bases de datos). En el caso del testeo web los scripts podrían ampliarse, por ejemplo, con el objetivo de añadir bucles o condicionales a una simple secuencia de comandos para así testear un abanico más amplio de situaciones. Estos dos ejemplos ilustran la limitación de las herramientas actuales y muestran una necesidad de soluciones de programación más específicas. Aunque tanto iMacros como Selenium IDE ofrecen un lenguaje propio de scripting para que los script grabados puedan ser editados de forma manual, esto suele estar fuera del alcance de muchos usuarios debido a la falta de conocimientos de programación. Incluso, en muchas ocasiones, usuarios con ciertos conocimientos de programación prefieren trabajar en entornos familiares en lugar de aprender un lenguaje diferente para cada contexto. Además, las frecuentes actualizaciones de los sitios web hacen que estos scripts de automatización sean difíciles de mantener: si la estructura de la web cambia, entonces el script podría fallar en el intento de encontrar los elementos DOM de destino (por ejemplo, podrían fallar los localizadores XPath basados en IDs si estos IDs cambian). La conclusión es que no sólo el desarrollo de los scripts sino también su mantenimiento acarrean dificultades a la hora de tratar de democratizar el uso de los scripts de automatización.

Esta Tesis aborda dos escenarios donde se pueden aprovechar las ventajas de la automatización web: el *rellenado de formularios web* y las *pruebas de aceptación de aplicaciones web* por parte de los usuarios. En cuanto al primer escenario, nos centramos en el **rellenado de sitios web de formularios intensivos**, esto es, sitios que contienen numerosos formularios repartidos en varias páginas, lo cual hace que se requiera navegación para llevar a cabo el

rellenado. Por ejemplo, los programas de financiación europeos o los sitios de declaración de la renta. En estos escenarios, es habitual que los datos solicitados por los formularios estén disponibles en formato electrónico, como bases de datos u hojas de cálculo. Sin embargo, rellenar manualmente estos formularios no sólo es engorroso sino que también es muy propenso a errores tipográficos. Además, esta tarea suele ser realizada habitualmente por personas sin conocimientos de programación, generalmente de perfil administrativo. En esta Tesis tratamos de capacitar a los usuarios para crear sus propios scripts de autorellenado para formularios web.

El segundo trabajo se centra en las pruebas de aceptación de una aplicación web en un entorno ágil. Las metodologías ágiles consisten en un desarrollo iterativo de pequeños conjuntos de requisitos de la aplicación. En cada iteración se desarrolla y prueba un grupo de requisitos, obteniendo un producto potencialmente entregable. Debido al corto período de tiempo que duran estas iteraciones (de 1 a 4 semanas), es crucial que exista una buena y fluida comunicación entre los participantes. Esto es especialmente importante cuando se trata de realizar las **Pruebas de Aceptación de Usuario** (User Acceptance Testing, UAT), en las que los usuarios del sistema comprueban si el producto cumple con sus expectativas o no. En este escenario, nuestro segundo trabajo tiene como objetivo capacitar a los usuarios para dar un feedback apropiado, además de poder crear pruebas automatizables que puedan ser reproducidas posteriormente por el equipo de desarrollo.

Metodológicamente, en esta Tesis recurrimos a la Design Science Research (DSR, Investigación Científica Basada en el Diseño). En la DSR, los artefactos están diseñados para interactuar en el contexto de un problema con el fin de mejorar algo en ese contexto.

Para cada uno de los temas abordados, realizamos primero un análisis de

la causa raíz del problema. El resultado es un planteamiento descrito según la siguiente plantilla de Wieringa:

> *Mejorar <un problema en un contexto>*
> *al <(re)diseñar un artefacto>*
> *satisfaciendo <algunos requisitos>*
> *con el fin de <ayudar a los interesados a alcanzar algunos objetivos>*

Esta plantilla supone un contexto y un objetivo, y propone la creación de un artefacto que ayude a las partes interesadas a alcanzar su objetivo en ese contexto específico. Para cada uno de los dos problemas abordados en esta Tesis, proporcionamos: 1) el contexto y las definiciones clave, 2) el análisis de la causa fundamental del problema que se ha de resolver, 3) el planteamiento formulado según la plantilla de Wieringa, 4) el conjunto de requisitos que se han de abordar y 5) el artefacto creado con el fin de resolver el problema presentado.

A continuación se describen las dos principales cuestiones abordadas.

Automatización del rellenado de formularios web

En este trabajo proponemos un enfoque basado en modelos que abstrae los scripts, grabados mediante un grabador, en modelos XMI. Una vez obtenidos, estos modelos se utilizan para crear una aumentación de la interfaz de la web, permitiendo a los usuarios crear y ejecutar fácilmente sus scripts de rellenado sin salir de la ventana del navegador.

Contexto y definiciones

Rellenado automático de formularios Web. El rellenado automático se define como un programa que permite rellenar los formularios sin requerir

de la intervención del usuario. Esta automatización permite a los usuarios ahorrar tiempo y evitar errores especialmente a la hora de rellenar formularios web repetitivos. El principal objetivo de las herramientas de autorellenado ha sido habitualmente la reutilización de los datos introducidos previamente en otros formularios. En estos casos, los datos de rellenado pueden pedirse al usuario por adelantado (p.e., PIAFF) o ser recogidos automáticamente mientras se van rellenando campos en de diferentes formularios web (por ejemplo, nombre, dirección, código postal, etc.) (p.e., Firefox Autofill, Carbon). Una vez que la herramienta ha recogido los datos, el autorellenado puede ser activado cada vez que el usuario visite una página web que contenga un formulario con campos similares.

Grabación y reproducción Esta técnica consiste en registrar las interacciones realizadas por un usuario y generar un script que permita la posterior reproducción automatizada de dichas interacciones. Estos script son muy fáciles de obtener y normalmente no requieren de ninguna habilidad avanzada. Muchas de las herramientas existentes, como iMacros, Selenium o CoScripter, incluyen un grabador que permite a personas sin conocimientos técnicos crear sus propios script básicos sin escribir una sola línea de código. Sin embargo, para poder utilizar todas las características de estas herramientas normalmente se requiere de programación manual. Incluso si el lenguaje de programación suministrado es sencillo, tratar con código puede ser desalentador para muchos usuarios.

Aumento de la web La aumentación de sitios Web (Web Augmentation, WA) es una técnica que tiene por objeto mejorar las páginas web existentes a fin de ofrecer al usuario un conjunto más amplio de interacciones o información. La WA se basa en la interfaz de un sitio web existente, ampliando la experiencia del sitio web en cuestión. Estas aumentaciones pueden ser activadas por complementos del navegador, applets, código

javascript, etc. Como ejemplos encontramos el poder añadir enlaces adicionales a páginas web para facilitar la navegación entre sitios relacionados, la ampliación de la información de una página con datos extraídos de otros sitios o la presentación de sugerencias de navegación a través del sitio web.

Análisis de la causa del problema

Declaración del problema

La creación de scripts de autorellenado de formularios es costosa, especialmente cuando los datos que se van a introducir en el formulario tienen que ser extraídos de una fuente de datos externa, como una base de datos o una hoja de cálculo.

Causas

Hay dos causas principales que pueden provocar este problema:

- La creación de un script de rellenado de formularios requiere de una importante inversión inicial. Los sitios web con formularios intensivos requieren de scripts largos y complejos. Además, el desarrollo de scripts de autorellenado alimentados a partir de fuentes externas es complejo y requiere de un considerable esfuerzo de programación. El usuario tiene que programar tanto el acceso a las fuentes externas como el script, utilizando para ello lenguajes de programación. Además, el código resultante muestra habitualmente dependencias tanto con la estructura de las fuentes externas como con la estructura de las propias páginas HTML. Si se realizan cambios en alguna de estas estructuras (la estructura de la fuente de datos o de la página web), el código corre el riesgo de fallar.
- El rellenado de formularios suele ser realizado habitualmente por personas sin conocimientos de programación. Los perfiles administrativos

son quienes gestionan los documentos, las hojas de cálculo y las aplicaciones de base de datos que contienen la información que eventualmente alimentará el sitio web. Son estas personas las que mejor conocen el sitio web, los pasos a seguir para introducir la información, así como cualquier directiva relacionada con el proceso de rellenado. En otras palabras, son las expertas del dominio en lo que respecta al proceso de rellenado de formularios. Lamentablemente, no es habitual que los administrativos tengan conocimientos de programación, por lo que las herramientas actuales no son lo suficientemente asequibles para este tipo de usuarios.

Consecuencias

La dificultad de crear scripts de rellenado de formularios que utilicen fuentes externas podría dar lugar a dos problemas principales:

- El proceso se realiza de forma manual, lo cual hace que se invierta mucho tiempo en copiar y pegar manualmente la información de la fuente de datos en el formulario. Esto hace que la tarea de rellenar el formulario se convierta en una tarea que consume mucho tiempo.
- Se pueden cometer errores al rellenar el formulario, tanto humanos como tipográficos. La larga y tediosa tarea de copiar y pegar información de un soporte (una base de datos, una hoja de Excel) a otro (una página web) aumenta las posibilidades de cometer errores. Por ejemplo, se puede pegar un dato en el campo equivocado del formulario, o incluso se puede hacer clic en el botón o enlace equivocado, lo que obliga al usuario a retroceder y repetir las acciones.

Diseño del problema

Siguiendo el modelo de Wieringa, este trabajo tiene como objetivo:

> *Mejorar la creación de scripts de autorellenado alimentados a partir de fuentes externas*
> *al abstraer el código de los script como modelos*
> *satisfaciendo eficiencia y asequibilidad*
> *con el fin de que los usuarios finales puedan crear sus propios scripts de rellenado automático de formularios.*

Esta plantilla asume que, en el contexto de la creación de scripts de autorellenado alimentados a partir de fuentes externas, los usuarios finales deben estar facultados para crear sus propios scripts. El artefacto diseñado debe cumplir una serie de requisitos. En primer lugar debe ser eficiente, de modo que los usuarios finales estén motivados para utilizarlo. Es decir, el rendimiento y la utilidad del script deben compensar el tiempo empleado en la creación del mismo. Por lo tanto, este tiempo debe recuperarse después de un número reducido de ejecuciones del rellenado, en comparación con el método manual. En segundo lugar debe ser lo suficientemente asequible como para que los usuarios sin formación técnica puedan utilizarlo sin recurrir a la ayuda de programadores. Este segundo requisito supone un diseño cuidadoso de la interfaz de usuario y de las interacciones de usuario necesarias para lograr la creación del script.

En esta Tesis, proponemos abstraer el código de los script en modelos. Estos modelos se presentan luego de forma visual al usuario a través de una aumentación web sobre la interfaz del formulario, facilitando así el establecimiento de relaciones entre los campos del formulario y la fuente de datos externa sin salir de la ventana del navegador.

Automatización de los test de aceptación web

El segundo trabajo aborda la realización de pruebas de aceptación de usuario cuando se utilizan metodologías ágiles. Proponemos el uso de mapas mentales como una plataforma que nos permite ofrecer usabilidad (necesaria para los usuarios finales) y automatización (necesaria para los desarrolladores).

Contexto y definiciones

Prueba de aceptación del usuario Las pruebas de aceptación del usuario son una de las últimas etapas dentro del ciclo de desarrollo de un producto. Una vez que el software ha sido testeado a través de pruebas unitarias y de integración, los clientes deben intervenir para probar el producto desarrollado y comprobar su adecuación a sus necesidades reales. Durante la UAT (User Acceptance Testing, testeo de aceptación de usuario), el software es probado en un escenario real por el público al que está destinado. Algunos enfoques habituales para la realización de la UAT son la presentación de demostraciones de software a los clientes o la comprobación manual del nuevo producto por parte de sus usuarios potenciales.

Metodologías ágiles Las metodologías ágiles son procesos de gestión de proyectos en los que demandas y soluciones evolucionan gracias al esfuerzo colaborativo de equipos autoorganizados y multifuncionales y de sus clientes. Aplicadas al desarrollo de software, estas metodologías abogan por la entrega y la mejora continua a través del desarrollo iterativo de pequeños conjuntos de requisitos. Existen diferentes métodos ágiles, cada uno de los cuales aplica los mismos principios e ideas comunes del Manifiesto Ágil de diferente manera: XP (eXtreme Programming, programación extrema), Lean, Scrum, etc. De entre ellos, Scrum es el más conocido y extendido. En este método, el conjunto completo de requisitos del producto (también conocido como backlog de producto) se agrupa en colecciones más

pequeñas (también conocido como backlog de iteración) en cada iteración. A continuación, el desarrollo se lleva a cabo durante un conjunto de iteraciones o *sprints*. Cada sprint es un proceso de desarrollo completo: las fases de diseño, desarrollo y prueba se completan en cada una de las repeticiones. El resultado de cada sprint es un producto potencialmente entregable que incluye aquellas características seleccionadas en el backlog de la iteración. Los métodos ágiles requieren de altos niveles de colaboración entre el equipo y sus clientes para poder desarrollar de forma rápida características que aporten valor comercial en cada iteración.

Mapas mentales Un mapa mental es un diagrama que permite organizar visualmente la información. Estos mapas se van creando alrededor de un solo concepto, representado como un nodo en el centro de una página en blanco, al que se añaden ideas asociadas representadas como imágenes o palabras. Los mapas mentales son una notación común en las organizaciones para llevar a cabo lluvias de ideas.

Wikis Las wikis son herramientas colaborativas. Consisten en un conjunto estructurado de páginas enlazadas entre sí, que pueden ser editadas por cualquiera a través de un sistema de edición abierto. Para usar las wikis no se necesita conocimiento de HTML, ya que se basan en un conjunto simple de comandos que permiten dar formato a la información. Todas las ampliaciones y cambios de contenido se registran junto con el nombre de su autor. Las wikis pueden ser creadas para proyectos específicos, proporcionando una plataforma de colaboración que facilita el trabajo incluso en equipos con participantes que están distribuidos geográficamente. Como ejemplo, FitNesse es una wiki dirigida a crear y ejecutar el testeo de aplicaciones de forma colaborativa.

Análisis de la causa del problema

Declaración del problema

La UAT en los procesos ágiles requiere que los usuarios estén muy involucrados y en comunicación frecuente con el equipo de desarrollo. Esta implicación es difícil de conseguir.

Causas

Las causas que pueden llevar a este problema incluyen:

- **Falta de tiempo.** Las pruebas de UAT necesitan que los clientes dediquen un tiempo significativo al margen de sus tareas diarias para participar. Sin embargo, no es fácil conseguir que los interesados inviertan su tiempo en las pruebas. En muchos proyectos los clientes no tienen suficiente tiempo para poder participar como deberían. La distancia entre el equipo de desarrollo y sus clientes es otro factor importante que puede llevar a una falta de participación por parte de los clientes. Esta situación se agrava a medida que aumenta el número de personas involucradas, y especialmente en los entornos distribuidos a nivel mundial, donde la sincronización de agendas puede convertirse en una tarea difícil. Todo esto lleva muchas veces a los equipos a simplificar el proceso reduciendo el número de usuarios que participan en las pruebas de la aplicación. Además, es común que la UAT se realice de forma manual, lo que hace que pueda resultar tedioso y consuma mucho tiempo. Varias publicaciones indican que esto puede ser desalentador para muchos clientes .

- **Falta de motivación.** Los desarrolladores no deben subestimar las cantidad de tareas que tienen que realizar los clientes: establecer las prioridades adecuadas para el trabajo, identificar todos los detalles que los programadores necesitan, ajustarse al plazo para las revisiones y prue-

bas, etc. Es necesario que tanto la organización como los desarrolladores reconozcan este esfuerzo y sean conscientes de las horas que los clientes están invirtiendo en el proyecto. La falta de reconocimiento puede dar lugar a una falta de adhesión y menor responsabilidad por parte de los clientes respecto al éxito o fracaso del proyecto.

- **Falta de conocimiento.** El tiempo y el reconocimiento no garantizan buenos resultados. Los clientes pueden estar acostumbrados a metodologías tradicionales en las que la UAT se lleva a cabo únicamente al final del ciclo del desarrollo. Así pues, en lugar de la participación intensa que exigen las metodologías ágiles, algunos usuarios pueden preferir expresar sus necesidades en reuniones puntuales. Además, hacer que los clientes realicen las pruebas por su cuenta puede hacer que se sientan intimidados y desanimarlos a la hora de realizar sus tareas de testeo.

Consecuencias

La dificultad para conseguir que los usuarios participen en el testeo de aplicaciones podría tener las siguientes consecuencias:

- Que el feedback de las pruebas de usuario se dé de manera tradicional (en documentos de texto, llamadas telefónicas, etc.) o en reuniones puntuales. En los procesos ágiles, la necesidad de una comunicación frecuente entre usuarios y desarrolladores incrementa la importancia de este feedback y exige métodos más dinámicos y asíncronos. Además, el feedback debe ser lo suficientemente fácil para que los usuarios lo preparen y lo suficientemente específico para que los desarrolladores lo entiendan.

- Si los usuarios no tienen suficientes conocimientos técnicos, pueden terminar recurriendo a personal de control de calidad o informático para que les ayude a desarrollar sus pruebas de aceptación. Esto podría por

una parte ocupar demasiados recursos de la empresa (dos personas para realizar una sola prueba), y por otra reducir la libertad del usuario para revisar la aplicación por su cuenta y detectar errores.

- El coste en términos de tiempo de las pruebas y la falta de motivación de los usuarios podría dar lugar a una reducción del número de clientes que participan en el proyecto. En consecuencia, también disminuye el número de pruebas realizadas, lo que podría poner en peligro la corrección y la satisfacción del resultado final.

Diseño del problema

Teniendo en cuenta las causas mencionadas, este trabajo tiene como objetivo:

> *Mejorar las pruebas de aceptación de aplicaciones web*
> *al abstraer los script de test como mapas mentales*
> *facilitando el aprendizaje y la repetición de test*
> *con el fin de que los clientes puedan crear sus propias pruebas de aceptación*
> *y dar un feedback adecuado al equipo de desarrollo.*

Esta plantilla asume que, en el contexto de las pruebas de aceptación de aplicaciones web, los usuarios finales deben estar facultados para crear sus propias pruebas de aceptación y dar un feedback adecuado al equipo de desarrollo. El artefacto diseñado debe abordar un conjunto de requisitos. En primer lugar debe ser sencillo de aprender, de modo que los usuarios finales estén motivados para utilizarlo sin dificultad. En segundo lugar tiene que incluir capacidades de reproducción, de modo que las pruebas creadas puedan ser reproducidas por el equipo de desarrollo para saber exactamente qué acción del usuario o cadena de acciones desencadenó el error. Esta capacidad de repetición también permite que las pruebas puedan ser comprobadas tantas veces como sea necesario para asegurarse de que el error no vuelva a ocurrir, una vez que se haya resuelto. En

esta Tesis recurrimos a mapas mentales y wikis como plataformas asequibles para realizar la UAT y compartir sus resultados con todo el equipo.

Conclusiones

En esta tesis se abordan dos problemas: (1) cómo facultar a los usuarios finales para que generen sus propios scripts de autocompletado de formularios alimentados a partir de fuentes externas y (2) cómo pueden los clientes crear sus propias pruebas de aceptación y dar un feedback adecuado en un contexto de desarrollo ágil. Las causas y las posibles consecuencias de las dos contribuciones propuestas se han enumerado en este capítulo. Ambas contribuciones se han desarrollado siguiendo un enfoque DSR (Design Science research).

EN

Thesis

Promoting End-User Involvement in Web-based tasks: a Model-Driven Engineering approach to form-filling and user-acceptance testing

Dissertation
presented to
the Department of Computer Languages and Systems of
the University of the Basque Country (UPV/EHU)
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy
("*international*" mention)

Itziar Otaduy Igartua

Supervisor: *Prof. Dr. Oscar Díaz*
Donostia/San Sebastián, Spain, 2020

April 19, 2020

This work was hosted by the University of the Basque Country UPV/EHU (Faculty of Computer Sciences). The author enjoyed a doctoral grant from the Basque Government under the *"Researchers Training Program"*. This work was co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839, and the Ministerio de Industria, Turismo y Comercio under contract TSI-020500-2010-206.

Contents

Contents i

List of Figures v

List of Tables ix

Summary 1

Acknowledgements 3

1 Introduction 5

1.1 Overview . . . 7
1.2 Automation of web form filling . . . 10
Context and definitions . . . 10
Root-cause analysis . . . 12
Design problem . . . 14
1.3 Automation of web testing . . . 15
Context and definitions . . . 15
Root-cause analysis . . . 18
Design problem . . . 20
1.4 Research Methodology: Design Science Research . . . 21
1.5 Outline . . . 24

1.6 Conclusions 25

2 Web automation **27**

2.1 Introduction 29

2.2 WA goals: form autofilling & web testing 32

2.3 WA techniques: PBD & scripting 36

3 EUD for web form filling **41**

3.1 Overview 43

3.2 Problem Definition 44

3.3 Meta-requirements for a solution 48

3.4 Related Work 49

3.5 A brief on iMacros 52

3.6 From iMacros to WebFeeder: from coding to modeling 58

3.7 Abstracting the external sources 60

3.8 Abstracting the form filling process 62

3.9 Abstracting the mapping 66

3.10 Generating the *feeder* script 67

3.11 Facing upgrades 68

3.12 Evaluation 72

3.13 Future Work 80

3.14 Conclusions 80

4 User Acceptance Testing **83**

4.1 Overview 85

4.2 The Practice: UAT in Agile processes 86

4.3 Problem definition 90

The Problem: Causes and consequences 92

4.4 Meta-requirements for a solution 96

4.5 The process: iterative and collaborative UAT 102

4.6 The notation: test maps 107

The expressiveness of test maps 109
4.7 The tool: Assisting users to come up with test maps 111
Check-in 113
Roaming 115
Test Coverage 118
Enacting 120
Commenting 120
Check out 122
4.8 Evaluation 122
Methodology 123
Evaluation subjects 125
RQ1. Does TestMind alleviate the lack of time? 129
RQ2. Does TestMind alleviate the lack of motivation? 130
RQ3. Does TestMind alleviate the lack of knowledge? 132
Threats to validity 134
4.9 Architecture 136
4.10 Related Work 137
4.11 Conclusions 140

5 Conclusions **141**
5.1 Overview 143
5.2 What is being done 143
Automation of web form filling 143
Automation of web testing 144
5.3 Publications 147
5.4 Research stage 147
5.5 What is left 148
Automation of web form filling 148
Automation of web testing 150
5.6 Conclusions 151

Bibliography **155**

A Designing Test Maps **171**

B Testmind interview guide **177**

C Testmind installation guide **179**

List of Figures

1.1 Root-cause analysis for the problem of “The creation of externally-fed autofilling scripts for form intensive websites is costly” 12
1.2 Root-cause analysis for the problem of “Difficulty to involve customers in application testing” 17
1.3 DSR methodology process model 22
1.4 Chapter map. 24

2.1 Web automation tools according to their goal and used techniques 31

3.1 Root-cause analysis for form-intensive website filling 47
3.2 CDTI form example. Parameterized *iMacros* script (top) and its *VB* script configuration counterpart (bottom). 53
3.3 From script coding to script modeling. 54
3.4 *WebFeeder* extends *iMacros* to support the life-cycle of *feeders*: **record** (2.1), **weave** (2.2), **synthesize** (3) and **run** (4). 56
3.5 Use of *input masks* in Microsoft Excel 57
3.6 Pushing the buttons in iMacros: the **LoadDataSource** button (Figure 3.4(1)), the **Synthesis** button (Figure 3.4(3)), and the **Run** button (Figure 3.4(4)). 61
3.7 Harvesting the *Data* metamodel out of database catalogues. 63
3.8 The *FormFlow* metamodel. 64

3.9 Harvesting the *FormFlow* model out of iMacros scripts. 65

3.10 A *feeder* generated for the form in Figure 3.2 67

3.11 Updating the *FormFlow* model using the "Safe Mode" (see figure 3.6 for the normal mode) 70

3.12 Upgrading the sample form with a new field. 72

3.13 The SPRI and CDTI web forms used for testing Webfeeder 74

3.14 Use case 1: break-evens for a simple form filling 76

3.15 Use case 2: break evens for a complex form filling. 78

4.1 Process Outline. 87

4.2 Root-cause analysis for the lack of customer involvement. 93

4.3 Agile process with a focus on UAT activities. Customers and developers collaborate through the wiki pages. 103

4.4 Wiki structure and distinct scaffolding pages for the *Calendar* application: *Hint* page (1), *Kickoff* page (2), and *Test* page (3). Test pages are enacted through the *"Test"* button (A). 104

4.5 *Test maps*: mind maps to capture UAT concerns. Map nodes are overloaded with UAT significance. Their operational semantics is defined through a Domain Specific Language (see Appendix A). . 107

4.6 **Check-in.** Customer attention is drawn to two features: *Event-Modification* and *NewEventAddition* (2). For each feature, Kickoffs are displayed as sub-nodes (3). Select a *Kickoff* node for a description to pop up in the lower canvas (4). 114

4.7 **Roaming**. Obtaining *UATCases* through Selenium IDE. Enhancements to this IDE include: (1) comment bar; (2) hint bar; (3) expectation-placeholder bookmark; (4) stop recording. 116

4.8 **Test Coverage.** When a page node is selected on the test map, the lower canvas shows a table where new input data sets can be introduced. Here three scenarios are added (2). *TestMind* complements the empty cells as follows: blanks are replaced by default values (taken from the first row) while the *$empty* keyword is used to deliberately leave a blank value. The resulting table is depicted in (3). 119
4.9 **Commenting**. Customer adds two *ExpectationStatement* nodes for *Test Case row 190.* 121
4.10 Diverging Stacked Bar Chart for the Satisfaction Questionnaire using Likert scales. The "3" on the left means the three customers, i.e. C1, C2 and C3, *Strongly Disagree* while "3" on the right corresponds to all *Strongly Agree*. 124
4.11 *TestMind* architecture 137

A.1 UAT Feature Model. 172
A.2 TestMind Abstract Syntax. 173

List of Tables

2.1 Recorder and scripting language availability on web automation. . 32

3.1 Autofilling approaches. 49

3.2 A classification of upgrades for websites and database schema. Frequency is based on anecdotal evidences from the test case. 69

4.1 Issues and addressing meta-requirements. 96

4.2 Tasks involved during UAT. 112

4.3 Subject profiling along risen issues. 126

4.4 Validating meta-requirements as effective (✔), partially effective (‰) or unconclusive (??) as for the issue at hand. 128

Summary

Many applications which formerly were designed for the desktop have gradually made a transition to the Web. Accordingly, an increasing number of tasks can now be conducted through the Web. As a result, opportunities arise to achieve a higher level of automation that the one being previously possible with proprietary, OS-anchored desktop applications. This resulted in an emerging interest in empowering users to check, adapt and customize the way they navigate and make use of these applications. Web automation, Web augmentation, or Web mashups are performant approaches that pursuit this aim. This work explores the use of scripting for two tasks, namely, Web-form filling and Web-application User-Acceptance Testing (UAT). In both cases, the challenge rests on abstracting from scripting code to higher models that permit the notion of scripting to be hidden into more-affordable representations. Accordingly, this work abstracts scripts into platform-independent models. For Web-form filling, we tackle the problem of repetitive form-filling from external sources. The solution is realized through WebFeeder, a plugin for iMacros that introduces autofilling-script models as first-class artifacts in iMacros. The synthesis, enactment and maintenance of these script models are handled without leaving iMacros, minimizing users' cognitive load and involvement. As for UAT, we tackle the issue of the need for the regular physical-presence of stakeholders for

UAT in Agile methodologies. In this case, we resort to mind-maps as the model representation. These ideas are fleshed out in TestMind, an editor for FitNesse that permits to capture UAT sessions as test maps. Self-paced testing is then devised as an iterative and collaborative effort where developers set the testing scaffold through the wiki, while customers build the test map that ends up as FitNesse pages. TestMind is evaluated through a case study involving three real customers. Summing it up, the bottom line is that WebFeeder and TestMind showcase the benefits that Model-Driven Engineering can bring to Web Automation. By moving away from code to high-level models, Model-Driven Engineering reduces the entry barrier for the participation of end-users.

Acknowledgements

A Thesis is not only the result of the time spent on reading, thinking and writing. It is also the result of many encounters with colleagues, conversations with other people, moments shared with many persons that offered me their support and understanding.

First of all, I wish to express my sincere appreciation to my supervisor, Prof. Dr. Óscar Díaz, for all the lessons learned from his vast experience, for the support he offered me all these years. His will for learning new things and exploring new research paths made the development of this Thesis very interesting and instructive. I want also to thank his empathy and support when it came to difficult professional and personal moments.

Thanks to the *Onekin Research Group*, to all the colleagues that I have known during these years: Iñigo Aldalur, Cristóbal Arellano, Maider Azanza, Iker Azpeitia, Josune de Sosa, Jokin García, Felipe Ibáñez, Arantza Irastorza, Jon Iturrioz, Haritz Medina, Leticia Montalvillo, Juanan Pereira, Jeremías Pérez and Gorka Puente. They have been a big inspiration and they made me feel at home when we were working in the lab.

I wish to thank my colleagues at Hochschule Neu-Ulm, specially Erica Janke and Mitja Weilemann, for accompanying and supporting me during my stage. To Professor Philipp Brune, that welcomed and integrated me in his research group and offered me the unvaluable opportunity to visit real companies that helped us with our work.

Thanks to the Basque Government for the economical support I have received during the years 2011 to 2015, which made possible the development of this Thesis.

To the *Pako's* staff, specially to Amparo and Josune, for providing me with the required amounts of coffee and for the good vibes they transmit at any moment of the day.

I would like to show my gratitude to Labox, for their deep understanding, their support, and all the facilities they gave me to be able to write this Thesis.

I wish to acknowledge the support and great love of my family and specially to my parents, Conchi and Carlos. This work would not have been possible without their encouragement and patience from the very first moment of this journey.

To each and every one of my friends, that unconditionally understood my occasional absences during these years. For the time shared with them, that made the difficult moments to become lighter. Also to my university colleagues, with whom I shared joy, concerns, frustrations, and many good moments in *sagardotegis*.

Last but not least, I want to give my best thanks to Unai; for his unconditional love, support, and understanding. Thanks for always cheering me up and staying with me both in the good and bad times.

1

Introduction

1.1 Overview

With the extensive use of electronic devices at both home and workplaces, the Web has become an integral part of people's lives [CDLN10]. Thus, many applications which were formerly designed for the desktop have gradually made the transition to the Web [AWDP17]. As an example, Google and Microsoft offer online versions of document[1], spreadsheet[2] and slideshow editors[3]. This spread of Web applications resulted in an emerging interest in empowering users to check, adapt and customize the way they navigate and make use of these applications. Web automation (i.e., automating the execution of navigation flows), Web augmentation (i.e., customizing the web interface) or Web mashups (i.e., joining web information) are performant approaches for users to get the best out of the Web. [CDLN10]. Benefits are manyfold. Specifically, Web automation can save time and prevent errors when employees have to perform repetitive or complex tasks [LHML08]. Web form filling and Web testing are two cases in point. On the first case, tools like iMacros [IMa] are widely used, and permit regular users to create their own automated workflows by using a record-and-replay technique. The iMacros recorder saves the user actions and allows then to replay them in order to achieve the filling of usual web forms (i.e., login into sites, filling in repetitive data such as address or phone number), thus saving time when performing these monotonous tasks. As for

[1]Google Docs (https://www.google.es/intl/es/docs/about/) and Microsoft Word online (https://office.live.com/start/Word.aspx)

[2]Google Spreadsheets (https://www.google.es/intl/es/sheets/about/) and Microsoft Excel online (https://office.live.com/start/Excel.aspx)

[3]Google Slides (https://www.google.es/intl/es/slides/about/) and Microsoft PowerPoint online (https://office.live.com/start/PowerPoint.aspx)

Web testing, Selenium IDE [Selb] is a well-known tool that also provides with a recorder for the user to create testing scripts. During the development of a web application, Selenium permits to easily and rapidly create test suites to be replayed by any user.

That said, current tools might fall short in some scenarios. For instance, form filling scripts might need to be extended to get the filling information from external sources (CSV files, databases). On the testing side, scripts might be enriched, for example by adding loops to a simple sequence of commands. These two examples illustrate the limitation of the "recorder approach" and the need for more specific coding solutions. Unfortunately, coding is very often out of reach for many end-users [CDLN10, BCBG16], which might require of appropriate tools that permit them to easily create their own scripts [LPKW06]. Even skilled users might prefer familiar environments instead of learning a different language for each different context [Pay00]. Besides, the frequent upgrades on web sites makes automation scripts difficult to maintain: if the web structure changes, then the script might fail in the attempt to find the target DOM elements (e.g., XPath locators based on IDs might fail if these IDs change [AWDP17]). The bottom line is that not only development but also maintenance impose stringent demands on the effort to democratize the power of automation scripts.

In this setting, this work tackles two scenarios to leverage users to take advantage of web automation: the filling of web forms and web user-acceptance testing. As for the former, we address the filling of form-intensive websites, that is, sites that account for numerous web forms spread across several web pages. For example, European funding programs or income tax return sites. In these scenarios, the data requested by the forms is usually available in electronic format, such as databases or spreadsheets. However, manually filling in these forms is not only cumbersome but also prone to typos. Moreover, this task is a

clerical work, that is, it is usually performed by people with no programming skills. Our first effort then (Chapter 3) strives to empower users to create their own autofilling scripts.

The second endeavor focuses on the testing phase of a web application on an Agile setting. Agile methodologies consist on an iterative development of small sets of application requirements [Mey14]. On each iteration a set of requirements is developed and tested, obtaining a potentially shippable product at the end. Due to the short period that these iterations last (from 1 to 4 weeks), a good and fluid communication between every participant is crucial. Specially, when it comes to User Acceptance Testing (UAT), where system users check whether the product is compliant with their expectations or not. In this scenario, our second approach (Chapter 4) aims at empowering users to give appropriate feedback and to create automated tests that can be later replayed by the development team.

Methodologically, we resort to Design Science Research (DSR). In DSR, artifacts are designed to interact with a problem context in order to improve something in that context [Wie14]. Section 1.4 delves deeper into this research methodology.

For each of the tackled problems we first conduct a root-cause analysis of the problem. The outcome is a design question described along Wieringa's template:

> *Improve <a problem context>*
> *by <(re)designing an artifact>*
> *that <satisfies some requirements>*
> *in order to <help stakeholders achieve some goals>*

This template assumes a context and a stakeholder goal and calls for an artifact that helps stakeholders achieve their goal on that specific context. For each of the two problems addressed in this Thesis, we provide: (1) the context and key definitions, (2) the root-cause analysis of the problem to be solved,, (3) the design problem formulated along Wieringa's template, (4) the set of meta-requirements to be addressed and (5) the artifact created with the aim of solving the presented problem.

This chapter provides an overview of the Thesis. Sections 1.2 and 1.3 describe the two issues tackled in this dissertation. The research methodology is presented in Section 1.4. An outline of the chapters is depicted in 1.5. Finally, section 1.6 concludes this chapter.

1.2 Automation of web form filling

Chapter 3 tackles how non-programmers can be empowered to create their own form autofilling scripts. Our solution uses a model-based approach that abstracts recorded scripts into models. These models are next used to deliver a web augmentation interface. This interface permits users to create and execute their scripts without leaving the browser window.

Context and definitions

Web form autofilling Autofilling is defined as a feature of a computer program that allows filling in forms without requiring user intervention [AGLH10]. This automation permits users to save time and avoid errors when filling in repetitive web forms. The main focus of autofill-

ing tools has usually been the reuse of data from previously filled out forms. In these cases, filling data is asked to the user in advance (PI-AFF [WGV+11]) or it is collected while he navigates the web and fills in web forms containing some fields (e.g., name, address, zip code, etc.) (Firefox Autofill [fir], Carbon [AGLH10]). Once the tool has gathered the data, the autofilling can be triggered whenever the user visits a web page containing a form.

Record and replay (R&R) This technique (a.k.a. Capture & Replay) consists in recording the interactions performed by a user and generating a script that provides such actions for automated, unattended re-execution [LCRT13]. R&R scripts are very easy to obtain and actually do not require any advanced skills. Many existing tools, such as iMacros [IMa], Selenium [Selb] or CoScripter [LHML08] include a recorder that permits non-technical people to create their own basic scripts without writing a single line of code. However, in order to be able to use all the features of these tools, manual scripting is usually required. Even if the provided scripting language is simple, dealing with code might be discouraging for many users [GHS12].

Web augmentation Web augmentation (WA) is a technique that aims at improving existing web pages in order to offer the user a larger set of interactions or information. WA builds on top of the rendering of an existing website, framing the new development within the Web experience of the website at hand [Día12]. Augmentations can be triggered by browser add-ons, applets, javascript code, etc. WA examples include adding extra links to web pages to ease the navigation between related sites, extend-

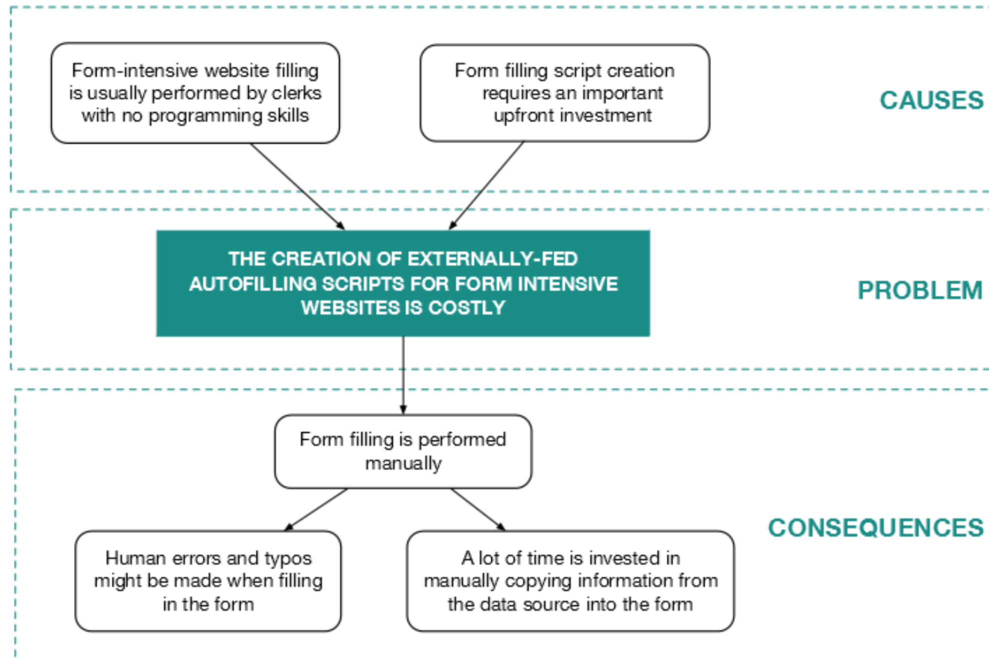


Figure 1.1: Root-cause analysis for the problem of "The creation of externally-fed autofilling scripts for form intensive websites is costly"

ing information with data extracted from other sites or showing hints for users to be guided through the use of a website [Bou99].

Root-cause analysis

Problem statement

The creation of form-filling scripts is costly, specially when the data to be introduced into the form is to be extracted from an external data source, such

as a database or a spreadsheet. Figure 1.1 outlines the causes and consequences of this problem.

Causes

There are two main causes that lead to this problem:

- **Form filling script creation requires an important upfront investment**. Form-intensive websites necessarily lead to large scripts. Also, externally-fed autofilling script development is programming intensive. The user has to code both the access to the external sources and the script using general-programming languages. In addition, this code shows external dependencies with the structure of both the external sources and the HTML pages. If upgrades are made on the structure of either the data source or the website, the code risks to fall apart [LHML08]. This involves a lot of time and resources to be invested in script writing.

- **Form filling is usually performed by clerks with no programming skills**. Form filling is a clerical work. Clerks manage the documents, spreadsheets and database applications that contain the data that will eventually feed the website. They know the site map, the possible flows for introducing the information as well as any directive related to the feeding process. They are the domain experts as far as the feeding process is concerned. Unfortunately, clerks do not usually have programming skills, so current scripting tools are not affordable enough for this type of users [KAB+11].

Consequences

The difficulty to create externally-fed form-filling scripts for form-intensive websites might lead to two main problems:

- **A lot of time is invested in manually copying information from the data source into the form**. Thus, the task of filling in the form becomes a long and time-consuming task [Cyp12].
- **Human errors and typos might be made when filling in the form**. The long and tedious task of copying and pasting information from one support (e.g. Excel) to another (e.g. Web page) increases the chance of making human errors. For example, one piece of information might be pasted into the wrong form field, or even the wrong button or link can be clicked, forcing the user to go back and repeat the actions [AC11].

Design problem

Following Wieringa's template, this work aims at:

> *improving the creation of externally-fed autofilling scripts*
> *by abstracting the scripts from code to models*
> *satisfying efficiency and affordability*
> *so as end-users are empowered to create their own form-filling scripts.*

This template assumes that, in the context of externally-fed autofilling script creation, end-users should be empowered to create their own scripts.

The designed artifact must address a set of requirements. First, it needs to be efficient, so end-users are motivated to use it. That is, the script's performance and usefulness must compensate the time spent on script creation. Thus, this time should be recovered after a reduced number of form filling executions, compared to the manual method. Second, it has to be affordable enough so users with no technical training are able to use it without resorting to programmers. This second requirement involves a careful design of the user interface and user interactions required to achieve the script creation.

In this Thesis, we propose to abstract scripts from code to models. These models are then presented to the user by augmenting the web form interface, and thus facilitating users to set relationships between the form fields and the data source without ever leaving the browser window.

1.3 Automation of web testing

Chapter 4 tackles User Acceptance Testing when Agile methodologies are used. We propose the use of mind maps as a reasonable trade-off between affordability (required by end-users) and automation (required by developers).

Context and definitions

User Acceptance Testing (UAT) User acceptance testing comes as one of the latest stages of product development. After the software has been unit and integration tested, customers are required to intervene in order to test the developed product and check its adequacy to their actual needs [Mey14]. During UAT, the software is tested in a real setting by

the intended audience. Common UAT approaches are the presentation of software demos to customers [Mey14] or the manual checking of the new product by its potential users [PT15].

Agile methodologies Agile methodologies stand for project management processes where demands and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customers [Mey14]. When applied to software development, these methodologies advocate for early delivery and continuous improvement through the iterative development of small sets of requirements [HD09]. Different Agile methods emerged (eXtreme programming (XP), Lean, Scrum, etc.), each one applying the same common principles and ideas of the Agile Manifesto in different ways [Mey14]. Among them, Scrum has come to dominate the agile scene [Mey14]. In this process, the complete set of product requirements (a.k.a. product backlog) is grouped into smaller collections (a.k.a. iteration backlog) to be developed on the following iteration. Then, development is carried out during a set of *iterations* or *sprints*. On each sprint a complete development process is performed: design, development and testing phases are completed into each one of the repetitions [SW07, HD09]. The output of each sprint is a potentially shippable product that accounts for the features in the iteration backlog. Agile methods require high levels of collaboration between the team and their customers in order to frequently release features that deliver business value in each iteration [HNM11].

Mind maps A mind map is a diagram to visually organize information. These maps are often created around a single concept, represented as a node in the center of a blank page, to which associated representations of ideas such as images or words are added [BB06]. Mind maps are a common notation among organizations for brainstorming and idea forming [Buz14].

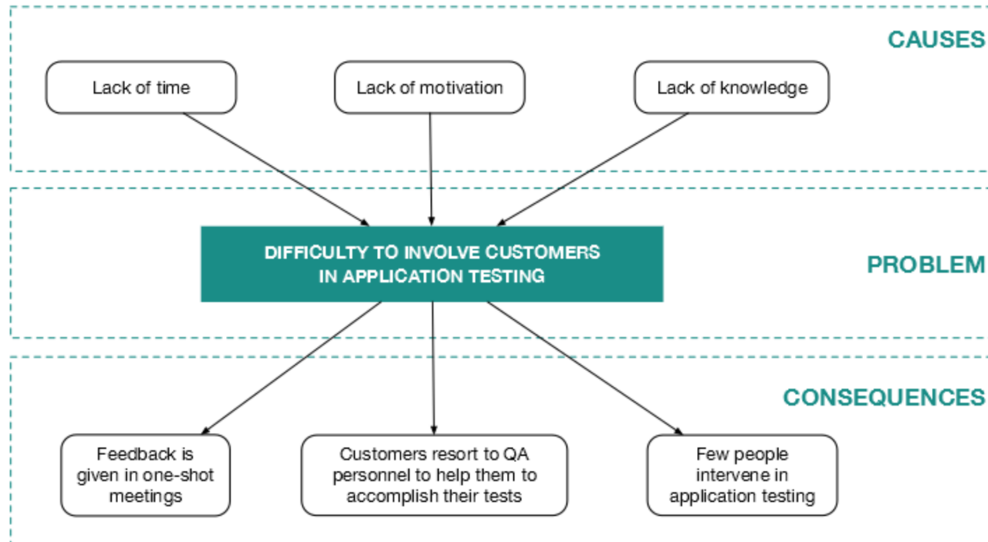


Figure 1.2: Root-cause analysis for the problem of "Difficulty to involve customers in application testing"

Wikis Wikis are an intensely collaborative tool. They consist of a structured set of pages linked to each other, that can be edited by anyone through an open-editing system. To use wikis no knowledge of HTML is needed, as they use a simple set of formatting commands to lay out the information [GJ03]. All the addition and changes of content are logged along with the name of their author. Wikis can be created for specific projects, providing a collaborative environment, even for teams with participants who are geographically distributed [LDP+12]. As an example, *FitNesse* [Fit] is a wiki aimed at collaboratively create and execute application tests.

Root-cause analysis

Problem statement

UAT in Agile processes requires users to be highly involved and in frequent communication with the development team. This involvement is hard to achieve. Figure 1.2 outlines the causes and consequences of the problem at hand.

Causes

Causes that might lead to this problem include:

- **Lack of time**. UAT testing implies customers to dedicate significant time away from their daily roles to participate [Sha08]. However, getting stakeholders to invest their time on testing is not easy. In many projects customers do not have enough time to be able to participate how they should [HNM11]. The distance between the development team and their customers is another important factor leading to this lack of customer involvement [HNM11]. This situation aggravates as the number of people involved increases, and specially in globally distributed environments, where agenda synchronization might become a challenging task. This inconvenience might lead teams to simplify the situation by reducing the number of users involved in application testing. Besides, it is common for UAT to be conducted manually, which makes the approach tedious and time consuming. This is being reported as discouraging for many customers [HH09, Pul06].

- **Lack of motivation**. Shore et al. [SW07] state that developers should not underestimate the different tasks customers might need to cope with: set the appropriate priorities for the work, identify all the details that programmers will ask about, fit in time for customer reviews and testing, etc. This recognition of the customer's testing effort applies not only to the customer organization but also to developers that in many projects might be unaware of the long hours the customers are working [MBN10]. This results in a lack of membership and less responsibility on the customers' part for the success or failure of the project [Kup, Tes18].

- **Lack of knowledge**. Plenty of time and recognition do not yet guarantee good results. Customers might be used to traditional methodologies where UAT is conducted at the end of the development lifecycle. Thus, instead of the intensive participation demanded by Agile methodologies, some users might prefer to express requirements in one-shot meetings [HH08]. Besides, making customers perform testing on their own might make them feel intimidated [Joh09], and discourage them to completely throw themselves into their testing tasks.

Consequences

The difficulty to involve users in application testing might involve the following consequences:

- The **user testing feedback is given in traditional ways (text documents, phone calls, etc.) or in one-shot meetings** [HH08]. The need of a frequent communication between users and developers in Agile processes increases the importance of this feedback and demands more

dynamic and asynchronous methods. Also, it should be easy enough for users to prepare, and specific enough for developers to understand.

- When users do not have enough technical knowledge, they might **resort to QA personnel to help them to develop their acceptance tests** [CR08]. This might take up too many company resources (two people to perform one single test), as well as reduce the user's freedom to wander around the application by his own and detect errors.

- The time cost of testing and the lack of motivation of users might lead to a **reduced number of testers involved into the project** [HNM11]. Consequently, the number of tests performed also decreases, which might jeopardize the correctness and satisfaction of the final result. Many works highlight the importance of involving several people into UAT [Bec00, MBN10]. Having not enough people to test the system might result in a reduced quality of the final product [HD09].

Design problem

Having into account the aforementioned problem causes, this work aims at:

> *improving the acceptance testing of web applications*
> *by abstracting test scripts as mind maps*
> *satisfying learnability and replayability*
> *so as customers are empowered to create their own acceptance tests and give adequate feedback to the development team.*

This template assumes that, in the context of web application acceptance testing, end-users should be empowered to create their own acceptance tests and give adequate feedback to the development team. The designed artifact

must address a set of requirements. First, it needs to be learnable, so end-users are motivated to use it without difficulty. Second, it has to include replaying capabilities, so the created tests can be replayed by the development team in order to know exactly what user action or chain of actions triggered the error. This replayability also permits the tests to be checked as many times as needed in order to make sure that the error does not happen again, once it is solved. This thesis resorts to mind-maps and wikis as affordable approaches to perform UAT and share its results.

1.4 Research Methodology: Design Science Research

Design Science Research (DSR) is an approach to create artifacts that support people in developing, using, and maintaining IT solutions, in order to help these people to fulfill their needs and overcome their problems [JP14]. Thus, in Design Science (DS), artifacts are investigated as solutions to practical problems that people experience in real practices. Although Design Science is viewed mainly from an IT and information systems perspective, the principles underlying are applicable to many other areas [JP14]. The DS methodology includes five activities, depicted in figure 1.3.

- **Explicate the Problem.** In this activity, the problem is investigated and analyzed. This problem should be of general interest and be precisely justified by showing that it is significant for some practice.

- **Define the Requirements.** Here, the solution to the problem should be explicated in the form of a set of requirements for the artifact. This activity can be seen as a transformation of the problem into demands on the proposed artifact.

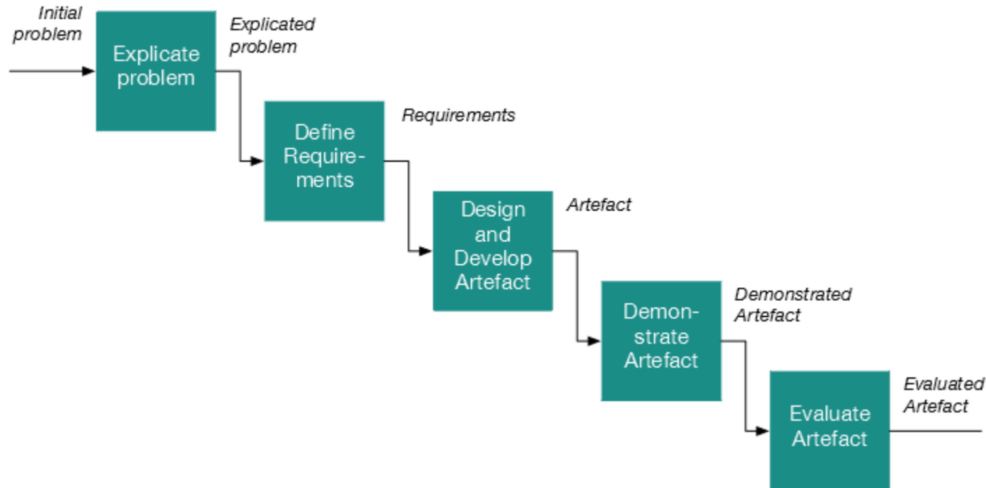


Figure 1.3: DSR methodology process model [JP14]

- **Design and Develop the Artifact.** This activity consists on designing and developing an artifact that addresses the explicated problem and fulfills the defined requirements.

- **Demonstrate the Artifact.** At this point, the developed artifact is used in an illustrative or real-life case, sometimes called a "proof of concept", thereby proving the feasibility of the artifact. The goal is to demonstrate whether the artifact actually can solve an instance of the problem.

- **Evaluate the Artifact.** This last activity determines how well the artifact fulfills the requirements and to what extent it can solve, or alleviate, the practical problem that motivated the research.

Although these activities might look sequential, a DS project is always carried out in an iterative way, moving back and forth between all the activities of problem explication, requirements definition, development, and evaluation. Thus,

the depicted activities should be seen as logically related through input-output relationships, and not as temporally ordered.

This dissertation has been developed along DSR hallmarks as it follows:

Explicate Problem. As for the problems, we identified two situations in the setting of Web Automation. For each one of them, we analyzed the causes that lead to the problem, as well as the consequences that they could provoke.

Define requirements. Once the problems where identified and analyzed, we translated each one of them into a set of requirements for an artifact that would solve the issue.

Design and develop the artifact. We designed and built two artifacts: Webfeeder (Chapter 3) and Testmind (Chapter 4), each one bringing a solution for one of the defined problems.

Demonstrate the artifact. Both presented artifacts where executed using an illustrative case, in order to prove their feasibility.

Evaluate the artifact. On one hand, we checked that Webfeeder fulfilled the requirements by comparing the effort required to use it against the effort required in situations where this tool is not used. On the other hand, Testmind was evaluated through a case study involving three real customers. Then, their experience was gathered through Likert forms and analyzed.

CHAPTERS

1	Introduction
2	Web Automation
3	End-User Development for web form filling
4	User Acceptance testing for Agile-developed web-based applications
5	Conclusions

Figure 1.4: Chapter map.

1.5 Outline

This section summarizes the contents of the Thesis. Figure 1.4 illustrates a map of the chapters in this dissertation. Below, a summary of each chapter is provided.

Chapter 2. This Chapter presents the context of Web Automation. It provides the reader with a background on two different web automation goals (namely *form autofilling* and *web testing*) and two different web automation techniques (*programming by demonstration* and *scripting*). It also includes works that tackled these goals and that are related to the proposals of this Thesis.

Chapter 3. This chapter tackles how to empower users to create their own form autofilling scripts from external sources. A recorder tool permits users to easily generate their autofilling scripts, while a web augmentation interface helps them to set the linkings between form fields and a data source (specifically, database tables). *WebFeeder,* is a plugin for *iMacros* that introduces *autofilling-script models* as first-class artifacts in *iMacros*. The synthesis, enactment and maintenance of these script models are handled without leaving the browser, minimizing users' cognitive load and involvement.

Chapter 4. This work introduces *Testmind*, a tool to empower customers to create their own acceptance tests and give appropriate feedback. Testmind is presented as a wiki-based approach where customers and developers asynchronously collaborate in order to perform UAT. Here, developers set the UAT scaffolding that will later shepherd customers when testing. To facilitate understanding, mind maps are used to represent UAT sessions. To facilitate engagement, a popular mind map editor, FreeMind, is turned into an editor for FitNesse, the wiki engine in which these ideas are then shared with the rest of the team.

Chapter 5. This chapter concludes the Thesis by resuming the main results, enumerating the limitations of the current solutions, and suggesting possible future lines of work.

1.6 Conclusions

This chapter gives an overview of the contents of this Thesis. Two problems are tackled: *how can end-users be empowered to generate their own form autofill-*

ing scripts that feed from external sources and *how can customers create their own acceptance tests and give appropriate feedback in an Agile development context*. The causes and potential consequences for the two proposed contributions (namely *Webfeeder* and *Testmind*) were listed. Both contributions were developed following a DSR methodology. Next Chapter delves into the context of Web Automation and into two of its main goals (*form autofilling* and *web testing*) and techniques (*programming by demonstration* and *scripting*).

2

Web Automation

2.1 Introduction

In recent years the Web has become an integral part of life for people. Many applications which formerly would have been designed for the desktop have gradually made a transition to the Web [AWDP17]. Nowadays, these Web applications are widely used by many people to create files, exchange information, consult data, and so on. Web applications work on top of a Web browser, which brings them some advantages compared to their desktop counterparts: they do not need to be installed on the user's computer; they can be accessed from different devices, such as a tablet or a mobile phone; and they are always up to date.

With the increase of users' technical knowledge, due to the extensive use of electronic devices, an interest to adapt and customize web applications has emerged. Different motivations guide this interest, for example, automating the execution of repetitive tasks (web automation), customizing the web interface (web augmentation), or joining information gathered on multiple applications (web mashup) [CDLN10].

Web automation focuses on automating repetitive tasks, such as navigating pages, filling in forms, and clicking on links [BWR+05]. This automation can be used to automate web navigation, perform web form autofilling, but also to test web applications by creating scripts that mimic users' interactions and then check the system status at different points of the execution. Web form filling and web user-acceptance testing are two cases in point. Section 2.2 dives more into detail on these two applications of web automation. While it is possible to write automation scripts using many programming languages, this option might be out of reach for many users. In these settings, web automation end-users might not have a technical background: while form filling

is largely performed by clerks, user-acceptance testing might be carried out by any potential user of a web application.

In fact nowadays, the vast majority of computer users are not professional programmers, and in most cases they lack of any training in programming languages. In order to empower these users to create their own scripts, they must be provided with appropriate tools that are easy to use and to learn [LPKW06].

The term "end-user programming" (hereafter EUP) was defined by Ko et al. as "programming to achieve the result of a program primarily for personal, rather public use" [KAB+11], and Lieberman et al. describe it as "a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact." [LPKW06]. Cypher et al. reduce this definition to "programming by people who are not professional programmers" [CDLN10].

To this end, different programming techniques have been proposed, such as programming by demonstration (PBD), spreadsheet programming, wizard-based programming or scripting languages [BCFP19]. Table 2.1 focuses on the goal and recorder and/or scripting language availability for some of the most well-known automation tools.

This section provides a brief about both goals and techniques to Web Automation. Specifically, Section 2.2 looks at two types of goals (i.e. form filling and web testing) and Section 2.3 presents two types of script creation techniques (i.e. programming by demonstration and scripting languages) that are of interest for the purpose of this dissertation (see Figure 2.1).

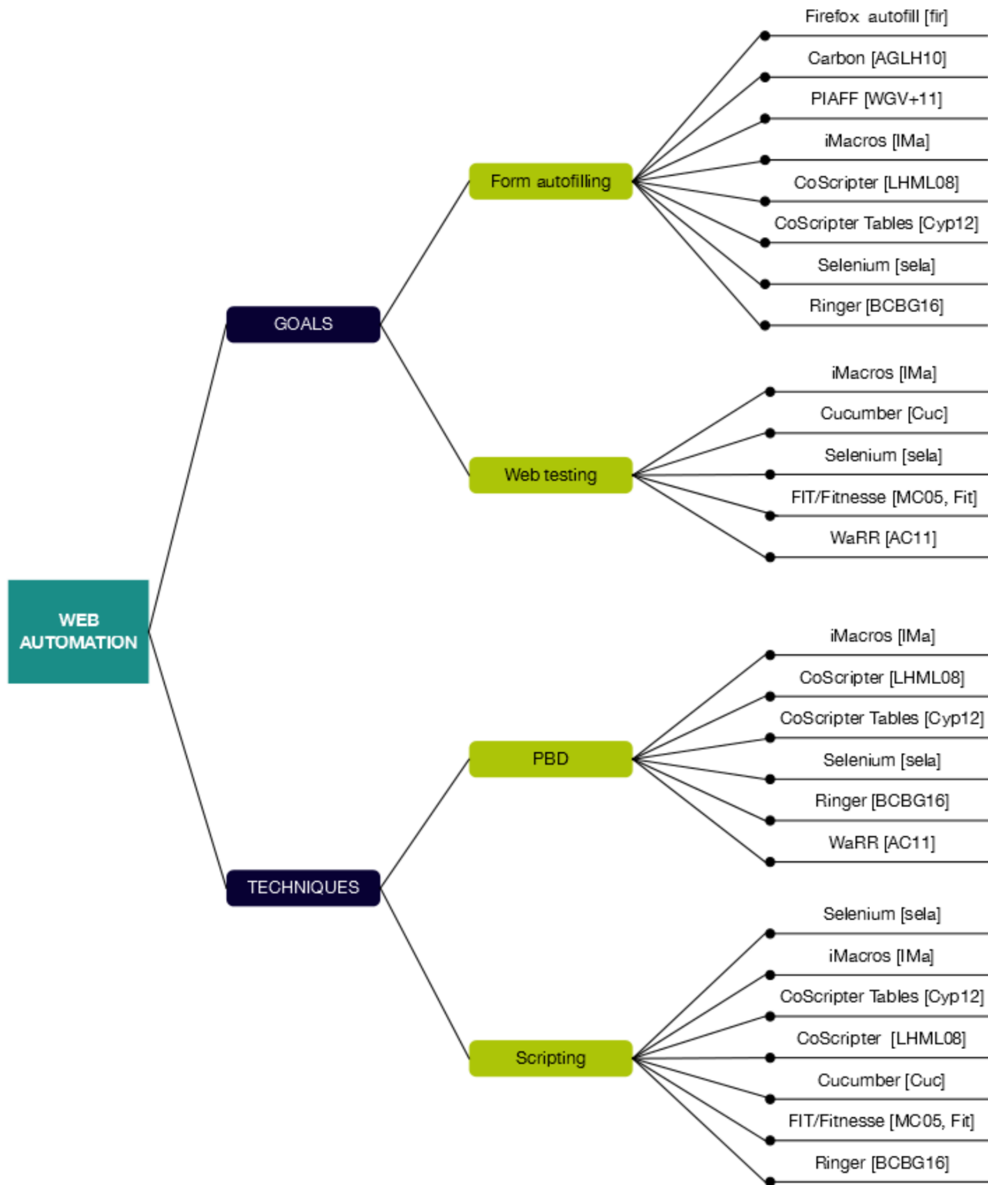


Figure 2.1: Web automation tools according to their goal and used techniques

Tool	Recorder	Scripting	Goal	Reference
Firefox Autofill	No	No	Form filling	[fir]
Carbon	No	No	Form filling	[AGLH10]
PIAFF	No	No	Form filling	[WGV+11]
iMacros	Yes	Yes	Form filling & web testing	[IMa]
CoScripter	Yes	Yes	Form filling	[LHML08]
CoScripter Tables	Yes	Yes	Form filling	[Cyp12]
Cucumber	No	Yes	Web testing	[Cuc]
FIT/Fitnesse	No	Yes	Web testing	[MC05, Fit]
Selenium	Yes	Yes	Form filling & Web testing	[sela]
Ringer	Yes	Yes	Form filling	[BCBG16]
WaRR	Yes	No	Web testing	[AC11]

Table 2.1: Recorder and scripting language availability on web automation.

2.2 Web automation goals: form autofilling and web testing

Web automation focuses on automating tasks, such as navigating pages, filling in forms, and clicking on links [BWR+05]. Web automation helps users complete tedious and repetitive interactions [BCBG16]. Different tools arose to create automation scripts, such as WaRR [AC11], iMacros [IMa], CoScripter [LHML08], CORSET [DDT13] and so on. These tools can be used to perform different tasks, among which we focus on web form autofilling and web application testing. Both tasks consist on mimicking user's interactions on the web, on one case to automatically fill in web forms, and on the other to check the application's behavior [LCRT13].

Form autofilling. Autofilling is defined as a feature of a computer program that allows filling in forms without requiring user intervention [AGLH10]. The main focus of autofilling tools has usually been the reuse of data from previously filled out forms. Filling data is asked to the user in advance (PIAFF [WGV+11]) or is collected while he navigates the web and fills in web forms containing some fields (e.g., name, address, zip code, etc.) (Firefox Autofill [fir], Carbon [AGLH10]). Once the tool has gathered the data, the autofilling can be triggered when the user visits a web page containing a form.

A slightly different use of this automation comes the same form needs to be filled more than once, but using different input data each time. For example, a company might need to introduce different projects into a governmental web page, or a student might need to fill each year the same form in order to ask for a University grant. These forms can be composed of different pages including many form fields, which makes form filling a tedious and time-consuming task, while it also augments the chance of introducing typos. Here is when PBD comes into play, offering the users an easy way of creating automation scripts without the need of any programming knowledge (see subsection 2.3).

Tools like iMacros, Selenium and CoScripter permit the recording and replaying of web macros through a recorder presented as a web browser extension [IMa, Selb, LHML08]. On its hand, Ringer aims at generating more robust scripts and reducing the chances of failure during replay by adding a trigger inference algorithm [BCBG16]. However, the macros recorded with these tools are usually too literal (i.e., they exactly replay the user's actions with the same input data) and scripts must be manually edited if the input data is to be extracted from an external data source, such as a database or a spreadsheet. In this line, *CoScripter Tables* permits the user to easily record macros that are fed from the data stored into a table [Cyp12]. This table is shown to the user during the recording, and permits to select table columns, rows and cells

without leaving the browser interface. However, this tool does not contemplate the use of more complex data structures, such as a database.

In Chapter 3 we propose Webfeeder, a tool that addresses the empowerment of end users to create externally-fed autofilling scripts from databases.

Web testing is "the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes" [HK07]. In the web domain, most testing tools exercise their testing through the user interface, by clicking buttons and introducing information mimicking a real user's interactions. These testing scripts also include assertions: special commands that can check the system status at some point of the execution (iMacros [IMa], Selenium [Selb]). Macro recorders are widely used to create testing scripts. These tools do not require any advanced programming or testing skill [LCRT13], which makes them suitable to end-users or professionals that need a quick way of creating tests. However, these tests tend to be quite fragile upon web changes, just like form autofilling macros. Maintaining and customizing the scripts require some scripting skills that are usually above the average end-users' knowledge [LCRT13] (see section 2.3).

Selenium [sela] is a suite of tools that are widely used to perform web testing. Among them, Selenium IDE is a macro recorder that permits to easily generate tests and replay them on the web browser [Selb]. Apart from Selenium, there exist a myriad of testing tools that can be used on web applications. For example, the WaRR tool is similar to Selenium but it adds support for more complex interactions such as drag-and-drop or Javascript event trigger-

ing [AC11]. As another example, Puppeteer[1] is presented as a testing tool focused on Chromium browsers, including the possibility to generate screenshots and PDFs from the tested pages. There also exist many commercial tools to achieve web testing that offer advanced interfaces to visualize test executions, results and analysis. Some tools, such as TestCraft[2], Subject 7[3] or Parasoft Selenic[4] make use of the Selenium engine to execute their tests, and they include improvements to increase test maintainability and robustness. In most cases, these improvements consist on an enhanced system to generate and fix web locators, one of the main source of problems when testing a web application by using its interface [AD17]. For example, Parasoft Selenic applies its heuristics to determine if a test failure is due to a real regression in the application, or if it's just a broken test. If the test is broken, Selenic heals the test at runtime. On its part, the TestCraft team states that their tool applies a unique machine learning algorithm to Selenium locators which enables the tool to automatically fix 97.4% of the changes that occur in the app, thus cutting maintenance time and resources. However, all of these solutions are strongly headed to people with testing skills, that is, professional testers that can get the best out of their advanced interfaces and analysis. From a regular user point of view, that is, without any testing skills, these tools might seem difficult to grasp, both to generate the tests and to visualize and understand their results [HH08].

[1]Puppeteer. https://github.com/puppeteer/puppeteer

[2]TestCraft. https://www.testcraft.io/

[3]Subject 7. https://www.subject-7.com/

[4]Parasoft Selenic. https://www.parasoft.com/products/selenic

While testing is a very important phase of a software development process, it becomes a crucial aspect when using Agile methodologies. These methodologies advocate for early delivery and continuous improvement through the iterative development of small sets of requirements in a short period of time, usually from 1 to 4 weeks (a.k.a iterations) [Mey14]. During these iterations, testing should be performed rapidly and efficiently, and at the end a runnable prototype of the final product is released [Mey14]. This prototype is next tested for customer satisfaction during **User Acceptance Testing (UAT)**. During the UAT phase the customer validates if the solution meets his specifications and exceptions [SH10], thus ensuring that the product is fully functional and meets his needs and expectations. In this phase customers and users are invited to check that the released product satisfies their needs by freely wandering around the system by themselves [HD09]. "The core business is what is verified and validated and who better to do it than the business owners and the customers" [See16].

Chapter 4 presents Testmind, a tool that permits to record tests and presents them in a mind-map structure. Here, testing is devised as a collaborative effort where tests are shared between developers and customers through a wiki.

2.3 Web automation techniques: Programming by demonstration and scripting languages

As stated before, scripting languages such as Javascript, allow expert users to manually create their own web automation scripts. Some tools offer their own scripting languages that limit their application to some specific domain ([Fit, Cuc, sela]). Although easier to use than general purpose languages, these scripting solutions might be out of reach for non-technical end-users. As the

results of the study of Barricelli et al. show, one of the more suitable techniques for generic users was Programming by Demonstration (**PBD)** [BCFP19]. PBD consists on the system automatically writing a program after gathering some input from the user, for example, after recording a sequence of user actions [CDLN10].

The rest of this section outlines the main works in the areas of PBD and scripting techniques (see Table 2.1)

Programming by demonstration

In PBD the input data is provided as user demonstrations of how the software is expected to behave [CDLN10]. These demonstrations can be presented as sets of input-output examples [LG14] or as recordings of user interactions [AC11]. PBD permits the user to write the program by giving examples of what the program should do by using its own user interface [Hal84, Lie01]. Many tools propose hybrid approaches that also provide a **scripting language**, allowing more technical users to manually edit and maintain the generated code [CH93].

Record and replay (R&R, a.k.a. capture and replay) is a basic form of PBD. R&R tools have proven useful for end-user programming of simple scripts [CBBG15]. R&R tools record users' interactions within a webpage, and replay them programmatically, facilitating users to create scripts to test a web or to automate their tasks. One of the main advantages of PBD is that it is "programming in the user interface" [Hal84]. That is, users generate the code by simply interacting with the program interface, with which they are already familiar to. They do not need to learn any programming language or any new software to get their scripts.

Unfortunately, the main problem of macro recorders is that they are too literal [CH93]. For example, if the user records a macro by filling in a form with his name and address, the macro will not be reusable by another person because the input data (name and address) is hardcoded into the macro script. This lack of parameterization and modularity mechanisms make so-generated scripts difficult to edit and maintain. To facilitate the edition and maintenance of the generated code, some tools provide also with scripting languages (e.g.: Selenese in the case of Selenium [Selb], ClearScript in the case of CoScripter [LHML08]). Next subsection dives into more detail on the scripting alternative of these tools.

PBD extends the idea of macro recorders by bringing a higher level of abstraction. In PBD, the program does not exactly replay the recorded actions, but it generalizes them by introducing variables, loops and conditional branches [CH93]. For example, instead of always introducing the string "project A" it creates a variable that takes the value of a table column "A" each time the script is executed. Or, instead of introducing 3 items on a list (i.e., the user recorded the input of 3 items), it loops through all the entries of a database and introduces them all.

Tools like iMacros, Selenium IDE or CoScripter include their own macro recorders [IMa, Selb, LHML08], that automatically create scripts that literally reproduce the users' interactions. On their hand, CoScripter Tables or Ringer extend the macro with parameterization and failure tolerance, creating more versatile and maintainable scripts [Cyp12, BCBG16].

In this thesis, Chapters 3 and 4 use R&R tools to permit the end user to easily generate the required scripts. In Chapter 3 the iMacros tool was reused and extended in order to support large form filling from databases. Chapter

4 uses Selenium to record simple test cases that are then converted to a mind map notation and exported to a wiki.

Scripting.

A scripting language is a "small, simple programming language whose vocabulary is specifically tailored to the objects and actions of a particular application domain" [CH93]. These languages are easier to use and to learn than the standard general-purpose programming languages. Scripting languages balance power and ease of use, restricting their use to a limited domain (such as web pages or spreadsheets) and offering usually a limited set of commands within that domain [CDLN10]. In the same line we find Domain Specific Languages (DSL).

DSLs are tailored to a specific domain application and bring a higher level of abstraction compared to general-purpose languages [MHS05]. Abstraction is a common mechanism to reduce the gap between technical and non-technical people. In web automation, DSLs are widely used for web testing. Toolkits such as Cucumber [Cuc] or FIT/Fitnesse [MC05, Fit] resort to DSLs for describing test stories which can later be converted into executable test cases [RPT+08].

CoScripter presents ClearScript, a pseudo-natural language that is both human-readable and machine-understandable [LHML08]. Ringer offers its own language that includes actions and triggers to control actions [CBBG15]. Selenium, a widely used tool for web automation and testing, uses Selenese, a specific language that holds a list of commands that interact with a webpage and allow to check its status (i.e., check the page title, check that a web element exists, etc.) [sela]. Similar to Selenese, iMacros holds a list of commands

that can be programmed as Visual Basic scripts using their scripting interface [IMa].

Chapters 3 and 4 make an internal use of the scripting languages of iMacros and Selenium. In both cases, the recorded scripts are internally abstracted into models in order to ease their manipulation and to increase their adaptability to other tools or scripting languages.

3

User-driven automation of web form filling

3.1 Overview

Form-intensive Web applications are common among institutions that collect bulks of data in a piecemeal fashion. European funding programs or income tax return illustrate these scenarios. Very often, most of this data is already digitalized in terms of documents, spreadsheets or databases. The task of manually filling Web forms out of these resources is not only cumbersome but also prone to typos. It does not benefit from the fact that the data is already in an electronic format.

Alternatively, externally-fed autofilling scripts can be programmed (e.g. using *iMacros* and *Visual Basic*) to code once, and enact many times. This approach is programming intensive and fragile upon upgrades in either the website or the structure of the external source. This moves these tools away from users with scarce programming skills.

This Chapter tackles how to empower users by abstracting the way feeding solutions are realized. Since external sources tend to be structured, they offer the chance to be abstracted in terms of models. Autofilling scripts can then be generated as weavings between the external data model and the website model. *WebFeeder,* is a plugin for *iMacros* that introduces *autofilling-script models* as first-class artifacts in *iMacros*. The synthesis, enactment and maintenance of these script models are handled without leaving *iMacros*, minimizing users' cognitive load and involvement.

This Chapter starts by identifying the problem of large form filling (section 3.2), then revising the related work (section 3.4) and giving a brief about *iMacros* (section 3.5). Section 3.6 outlines the approach to abstract from script code to script models, which is later detailed throughout Sections 3.7, 3.8, 3.9

and 3.10. Section 3.11 focuses on upgrades. A first evaluation of the tool is presented in section 3.12. Conclusions (section 3.14) end the Chapter.

3.2 Problem Definition

Websites can be classified based on the quantity of data they request. If the requested data is mainly personal and limited, then autofilling mechanisms are available to alleviate the tiresome task of periodically providing this information (personal data, card holder, visa number, etc.) [fir, goo]. On the other side of the spectrum, some institutional websites request a large quantity of data. We qualified websites as *"form-intensive"* when they account for numerous web forms spread along several pages. In these scenarios, the manual approach is not only cumbersome but also prone to typos. Neither does it help the use of traditional autofilling mechanisms (e.g. *Firefox autofill*) where the filling data comes from previously filled forms but they do not benefit from external data sources. Indeed, it is very common in form-intensive filling tasks for the required data to be already available within the organization. As an example, consider the application for R&D projects. The schedule, personnel, budget, etc, are all data that might well be prepared in advance and stored as spreadsheets, documents or databases (e.g. if a wiki or a document management system is used). In addition, the same institution (e.g. a University) might present different projects to the same funding agency, which results in navigating the same website many times.

So far, script-based approaches might offer a solution. Scripts can be programmed using *ad hoc* languages (e.g. *iMacros* [IMa], *Selenium* [Selb]). These tools offer a limited scripting language that permits users to code scripts for some specific domain (Web automation for iMacros, Web testing for Selenium).

However, manual coding of those scripts requires some basic programming knowledge, so it can be out of reach for many end-users. To help this type of users, some of these tools also provide with a record&replay device, that records the interactions of the user during a session in terms of a script. This script can next be replayed at user's will. However, if the input data is to be extracted from an external source, manual customization of the recorded scripts is required. In this case, the strategy rests on creating a program that consults the source (e.g. a spreadsheet), assigning the returned values to variables, and enacting the script which was previously parameterized with these variables. This permits to tap into existing data sources while automatizing repetitive data entries.

For form-intensive web sites, this approach offers a great potential. However, the creation of form-filling scripts is costly. This is mainly due to two main causes (see figure 3.1: causes):

- **it requires an important upfront investment.** Form-intensive websites necessarily lead to large scripts. This increases the chances of being affected by upgrades. Also, script development is programming intensive. The user has to code both the access to the external sources and the script using programming languages. In addition, this code shows external dependencies with the structure of both the external sources and the HTML pages. If upgrades are made on the structure on either the data source or the website, this code risks to fall apart [LHML08].

- **it requires scripting skills.** Form filling is a clerical work. Clerks manage the documents, spreadsheets and database applications that contain the data that will eventually feed the website. They know the site map, the possible flows for introducing the information as well as any directive concerning the feeding process. In other words, when it comes to form

feeding processes, they play the role of domain experts. Unfortunately, clerks do not usually have programming skills, so current scripting tools might not be affordable enough for them [KAB+11].

The difficulty to create form-filling scripts for form-intensive websites, along with their fragility upon changes on the website, result on form filling to be usually performed manually. In this scenario, two main problems might arise (see figure 3.1: consequences):

- The task of filling in the form **becomes a long and time-consuming task [Cyp12].**
- **Introduction of typos into the web form.** The long and tedious task of copying and pasting information from one support (e.g. Excel) to another (e.g. Web page) increases the chance of making human errors. For example, one piece of information might be pasted into the wrong form field, or even the wrong button or link can be clicked, forcing the user to go back and repeat the actions[AC11].

This work addresses **the empowerment of end-users to create externally-fed autofilling scripts** (hereafter just "scripts"). The approach rests on abstracting scripts from code to models. This stands for a decrease in human error and misunderstanding while improving efficiency and affordability. As a proof of concept, we describe how *iMacros* has been leveraged from managing script code to script models. Moving to the model realm improves *iMacros* affordability so that end-users can now synthesize, run and maintain script models without requiring programming skills. A video of the proposed tool is available at https://vimeo.com/173782588

In short, this work aims at:

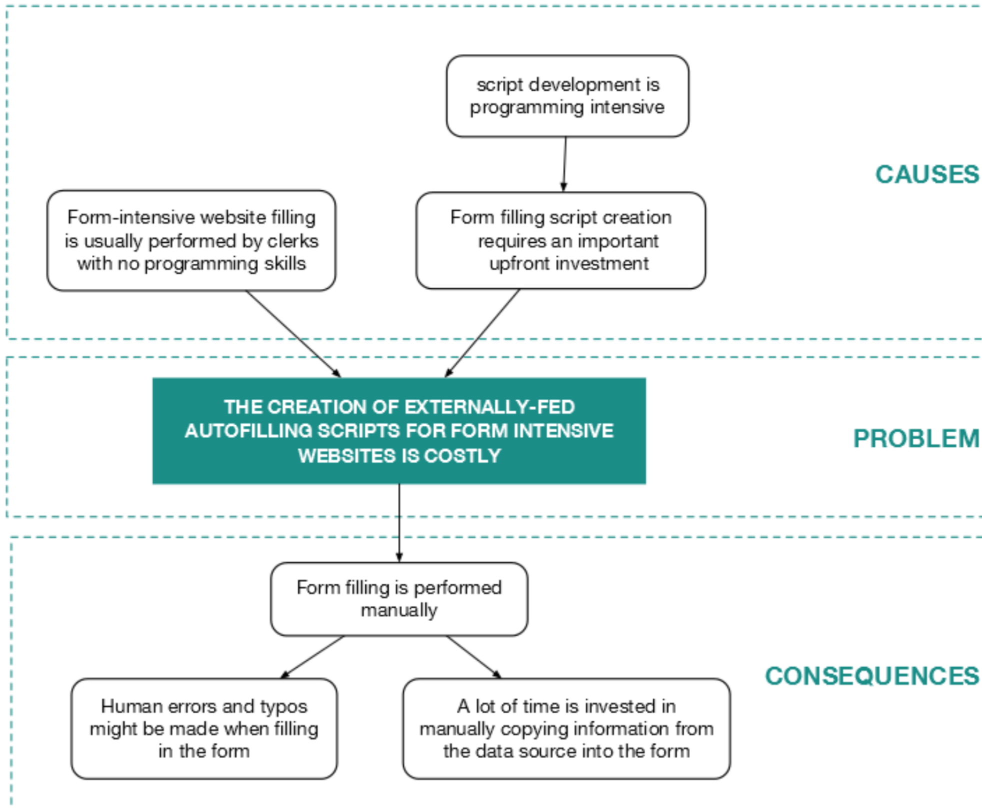


Figure 3.1: Root-cause analysis for form-intensive website filling

> *improving the creation of externally-fed autofilling scripts*
> *by abstracting the scripts from code to models*
> *satisfying efficiency and affordability*
> *so as end-users are empowered to create their own form-filling scripts.*

3.3 Meta-requirements for a solution

Achieving the empowerment of end-users to create their own externally-fed form autofilling scripts leads to the following proposed meta-requirements:

MR1 *Permit a non-technical person to establish a link between a form field and a database*

As stated before, at workplaces, usually are clerks who manage the documents that contain the data that will eventually feed the website. Unfortunately, those employees do not usually have programming skills. In most cases, they are not able to write a script that navigates through a web page while it extracts information from an external source. Here, the challenge is to present the user with an interface that is easy to grasp and efficient to generate form filling scripts.

MR2 *Offer an easy and intuitive script creation process*

End-users are not usually motivated to learn new tools or languages. Even if they do, the high cost of learning a new tool might jeopardize its use by non-technical people. Form filling scripts interact with a web application that the user already is familiar to. Thus, in order to increase the tool's affordability our proposal should extend but not change the way the user interacts with that web application.

	Mapping Approach	Data Origin	User In-volv.	Process Concerns
Firefox Autofill [fir]	string matching	previous fillings	none	filling
Carbon [AGLH10]	conceptual mapping	previous fillings	none	filling
iMacros [IMa]	script-based	external sources	high	filling & navigation
CoScripter [LHML08]	script-based	personal database (name/value pairs)	low	filling & navigation
CoScripter Tables [Cyp12]	script-based	spreadsheet	high	filling & navigation
WebFeeder [DOP13]	script-based	external sources	high	filling & navigation

Table 3.1: Autofilling approaches.

3.4 Related Work

Autofilling is defined as a feature of a computer program that allows filling in forms without requiring user intervention. Aside from providing personal information, the autofilling feature can be useful in a large number of scenarios [SCE+08]. Table 3.1 compares different solutions along four dimensions: mapping approach, data origin, user involvement and process concerns.

Mapping Approach. Autofilling implies a mapping between the data and the form fields. Three main solutions stand out to automatically infer this

mapping. First, *string matching* based on HTML field attributes (e.g. name, label or id), which consists on automatically deducing relationships by trying to match form field attributes with the data source labels. This approach is illustrated by Google Toolbar [goo] or Firefox Autofill Forms [fir]. Second, *semantic annotation* e.g. using Microformats [Kha06]. Microformats can make a website machine-readable by adding special markup to mark recognizable data items (such as events, contact details or geographical locations). Firmenich et al. propose the use of Microformats to set up the mapping between HTML rendered elements and the data source [FGG+12]. A similar approach but using HTML5 features rather than Microformats is introduced in [HG11]. Third, *conceptual mapping*. Unlike the previous techniques, now the mapping is achieved at the conceptual level. Form clues (e.g. id, label) are mapped to lexical words which are next compared with ontologies that contain synonyms and abbreviations, such as *WordNet* or *DBpedia*. *Carbon* [AGLH10] is a case in point. This application is extracts relevant metadata from the previously filled forms, semantically enriches it, and uses it for aligning fields between web forms. Previous approaches trade accuracy for user involvement. That is, they reduce the engagement of the user at the expense of less precise results. However, there exist many tools that hardcode the input data for each field on the script code. This is what we called *script-based mapping*, and it is the most usual solution for script recorders, such as *iMacros* [IMa] or *Selenium* [sela]. Once recorded, the script can replay the interactions at user's will, but it will always fill in the form fields with the same data.

Data Origin. Feeding data can be obtained from previous feeding processes or existing documentation. The former is illustrated by tools such as Google Toolbar [goo] or Firefox Autofill Forms [fir]. Alternatively, forms can be filled out from external sources. For example, *Safari* permits to tap into the *Address Book* [saf] while *CoScripter Tables* [Cyp12] uses spreadsheets. In

the commercial side, both *iMacros* and *Selenium* offer languages that permit creating scripts that populate web forms from databases and text files.

User Involvement. This dimension admits three values based on the contribution of the user: (1) *no-involvement*: no additional effort is required from the user as data is automatically collected from the filling of other forms, (2) *low involvement*: the user provides an example that is later used to fill out similar forms, and (3), *high involvement*: the user facilitates a script that can be parameterized and replay with different values. In this dimension we observe that the solutions that automatically fill in forms using previously gathered information (extracted from other form fillings) request no user involvement ([fir, AGLH10]), while the tools that need the user to record a script need a high involvement from the user's part ([IMa, Cyp12, DOP13]). A special case is CoScripter [LHML08]. Although this tool requires the user to record the automation scripts, it proposes a wiki platform for these scripts to be shared among the company. Thus, employees can reuse scripts recorded by some of their colleagues, and they will only need to provide the data that needs to be introduced into form fields (i.e., provide their personal database: a list of name/value pairs).

Process Concerns. The filling process can tackle different concerns. It can only focus on the *filling*, include *data validation*, or allow to be extended along different pages, that is, address navigation between pages as well. Tools like Firefox Autofill [fir] and Carbon [AGLH10] concentrate on the filling of one-page forms, that usually ask for user's personal data such as username, password or address. However, automation tools such as *iMacros [IMa], CoScripter [LHML08, Cyp12]* and *Selenium* [Selb] also support navigation through different webpages.

Table 3.1 frames our approach to related work. We focus on form-intensive websites, where feeding scripts from external sources offers an attractive solution to tap into existing data within the organization. In addition, user involvement should be reduced if we intend to make clerks self-sufficient. Moreover, automatically obtaining the mapping using semantic closeness also turned out to be difficult when addressing a whole website. This rules out the manual programming of the scripts. The main insight of this work is *to abstract the information structure of the external sources and guide the mapping process so as to make it affordable to clerks*. Therefore, the challenge is not so much about feasibility but affordability. Before delving into the details, we provide a brief on *iMacros*, the framework that underpins our approach.

3.5 A brief on iMacros

Figure 3.2 depicts a web form for project application at the CDTI website, a Spanish funding organization. Let us consider a scenario where: (1) the data has already been digitalized in terms of databases or spreadsheets, and (2) the organization (e.g. a university) applies for different projects, and hence, the very same forms need to be filled out over and over again. As in other software settings, repetitive tasks are worth being automatized through macros.

iMacros is an extension for the web browsers which adds record and replay functionality for user sessions on the web. The autofilling life cycle is conducted along two steps: (1) *record* the autofilling script as the user navigates throughout the site; and (2), *replay* the script at wish. Broadly, *iMacros* scripts are a sequence of navigation commands (e.g. the *URL()* command opens a new webpage) and automated interactions on the current page (e.g. the *TAG()* command performs some action on a web element). Autofilling wise, *iMacros*

iMacros script

```
1  VERSION BUILD=7401110
2  TAB T=1
3  URL GOTO=https://solicitudes.cdti.es/
-      GestionSolicitudes/Direcciones.aspx
4  TAG POS=1 TYPE=INPUT:RADIO ATTR=ID:Desarrollo_rbDirParticular
5  TAG POS=1 TYPE=SELECT ATTR=ID:Desarrollo_cboTipoVia
-      CONTENT=%178
6  TAG POS=1 TYPE=INPUT:TEXT
-      ATTR=ID:Desarrollo_txtDireccion
-      CONTENT={{STREET}}
7  TAG POS=1 TYPE=INPUT:TEXT
-      ATTR=ID:Desarrollo_ucLocalidadLugar_ltxtCodPostal
-      CONTENT={{ZIP}}
8  TAG POS=1 TYPE=SELECT
-      ATTR=ID:Desarrollo_ucLocalidadLugar_ldropProvincias
-      CONTENT=${{STATE}}
9  TAG POS=1 TYPE=INPUT:IMAGE
-      ATTR=ID:Desarrollo_ucLocalidadLugar_btnBuscar
10 TAG POS=1 TYPE=SELECT
-      ATTR=ID:Desarrollo_ucLocalidadLugar_ldropLocalidad
-      CONTENT=${{CITY}}
11 TAG POS=1 TYPE=INPUT:TEXT
-      ATTR=ID:Desarrollo_txtTelefono CONTENT={{PHONE}}
```

Visual Basic Script

```
1  sql = "select * from PROJECT inner join COMPANY on
-        PROJECT.main_company = COMPANY.id where PROJECT.id = 1"
2  set rs = rs.Execute(sql)
3  set iiml= CreateObject ("imacros")
4  iret = iiml.iimOpen("")
5  do until rs.eof
6     'Set the variable
7     iret = iiml.iimSet("STREET", rs.fields("street"))
8     iret = iiml.iimSet("ZIP", rs.fields("zip"))
9     iret = iiml.iimSet("STATE", rs.fields("state"))
10    iret = iiml.iimSet("CITY", rs.fields("city"))
11    set phone_number = Mid(rs.fields("phone"),1, 3)+"-"+
-                        Mid(rs.fields("phone"),4, 3)+"-"+
-                        Mid(rs.fields("phone"),7, 3)
12    iret = iiml.iimSet("PHONE", phone_number)
13    iret = iiml.iimPlay(mypath & "Macros\add_address.iim")
14    If iret < 0 Then
15         MsgBox "Error code: "+cstr(iret) + VbCrLf
16    End If
17   rs.movenext
18 loop
```

Figure 3.2: CDTI form example. Parameterized *iMacros* script (top) and its *VB* script configuration counterpart (bottom).

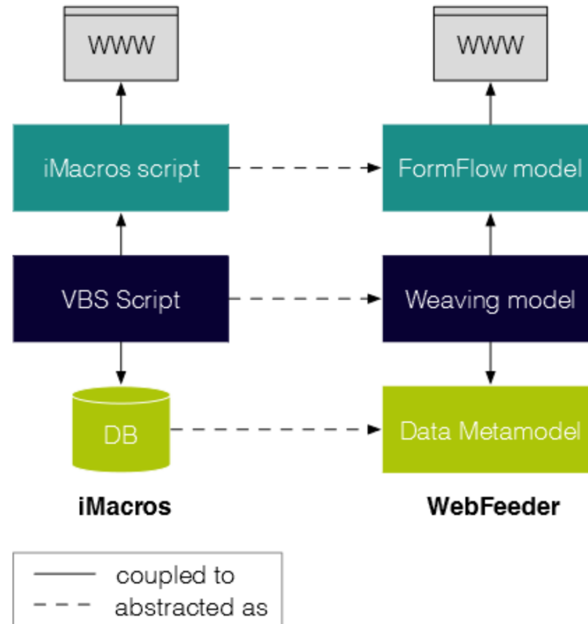


Figure 3.3: From script coding to script modeling.

admits three strategies: (1) the data is provided at *recording* time as a script constant; (2) the data is provided at *replaying* time by prompting the user; and (3), the data is obtained at *replaying* time by querying external sources. If data is provided at recording time, the autofilling follows the life cycle: *"record > play"*. If data is to be obtained from external sources, the life cycle is enlarged with two additional steps: *"record > parameterize > configure > play"*. Parameterize basically means to turn values for data input into variables, while configuration implies to code a program that instantiates these variables.

Figure 3.2 shows an example for the CDTI form sample. At the top, the *iMacros* script once parameterized as denoted by the expressions *{{variable}}*. At the bottom, the configuration step which is realized through *Visual Basic Script* (VBS) code. Configuration mainly involves four concerns: querying the

database (lines 1, 2), validating the data format (line 12), establishing database-to-script variable mappings (lines 8-9, 10-13) and enacting the *iMacros* script (line 15). By far, establishing the mapping is the most complex task. Notice that parametrization and configuration are not supported by *iMacros* but handled externally (e.g. using a *VBS* editor).

From a corporate perspective, the use of this solution for feeding form-intensive websites rises two issues. First, this solution might require an important upfront investment (e.g. the script for the CDTI case study took more than 20 hours to develop). This investment can be put in jeopardy if the structure of either the web pages or the external sources are upgraded. Second, affordability. Clerks are the domain experts as far as the feeding process is concerned. They are the ones that daily manage the documents, spreadsheets and database applications that contain the data that will eventually feed the website. They also know about the site map, the possible flows for introducing the information as well as any directive concerning the feeding process. However, they cannot set the solution by themselves: *VBS* is strange to them. Even a tiny change in the website (potentially breaking the script) makes them dependent on the availability of the always-busy IT department.

Configuration (i.e. the manual coding of the *VB* script) is the Achilles' heel of this solution. The question is whether this code can be abstracted in terms of a model. This would bring modelware benefits to the realm of Web autofilling: more simple development, lower required skills, faster delivery, etc [BCW12]. This grounds the development of *WebFeeder*.

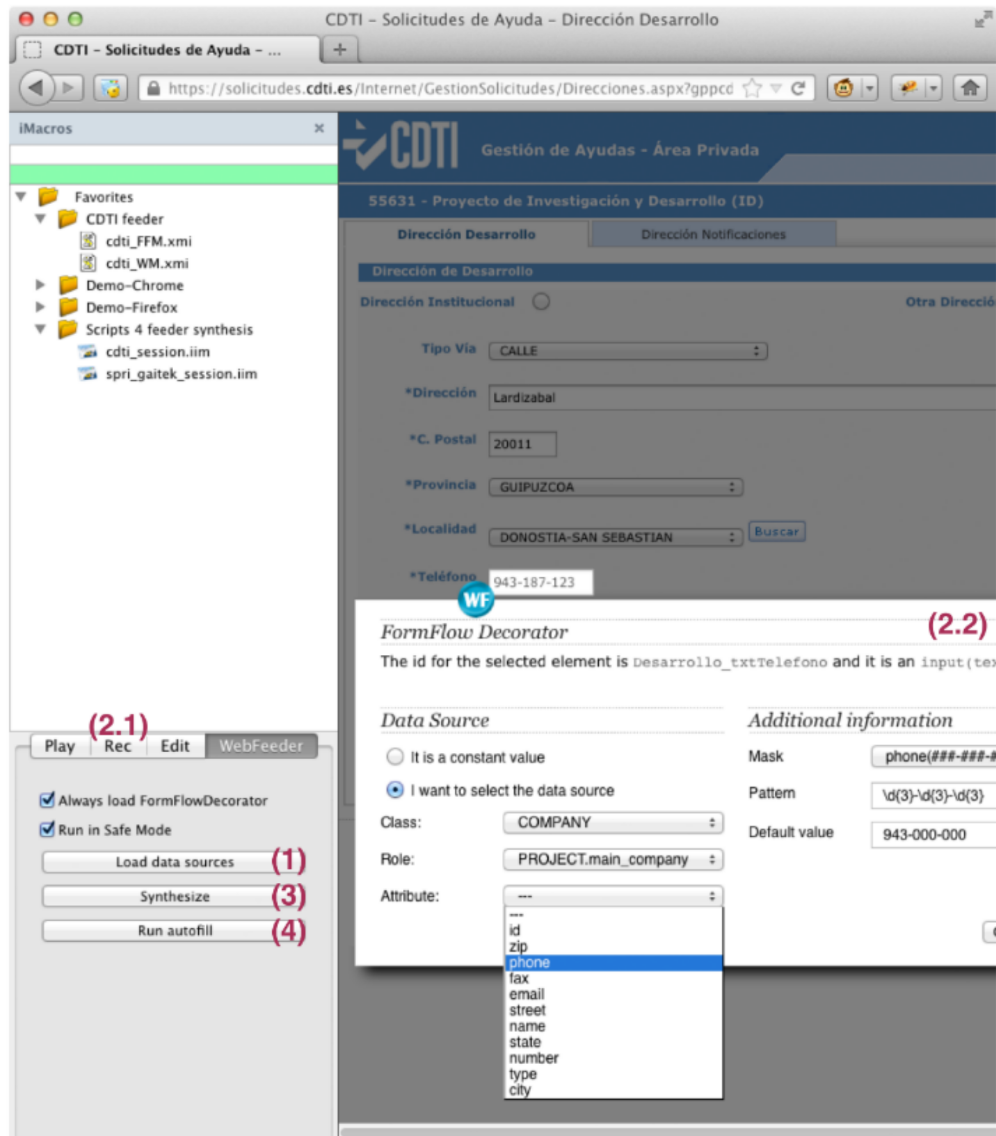


Figure 3.4: *WebFeeder* extends *iMacros* to support the life-cycle of *feeders*: **record** (2.1), **weave** (2.2), **synthesize** (3) and **run** (4).

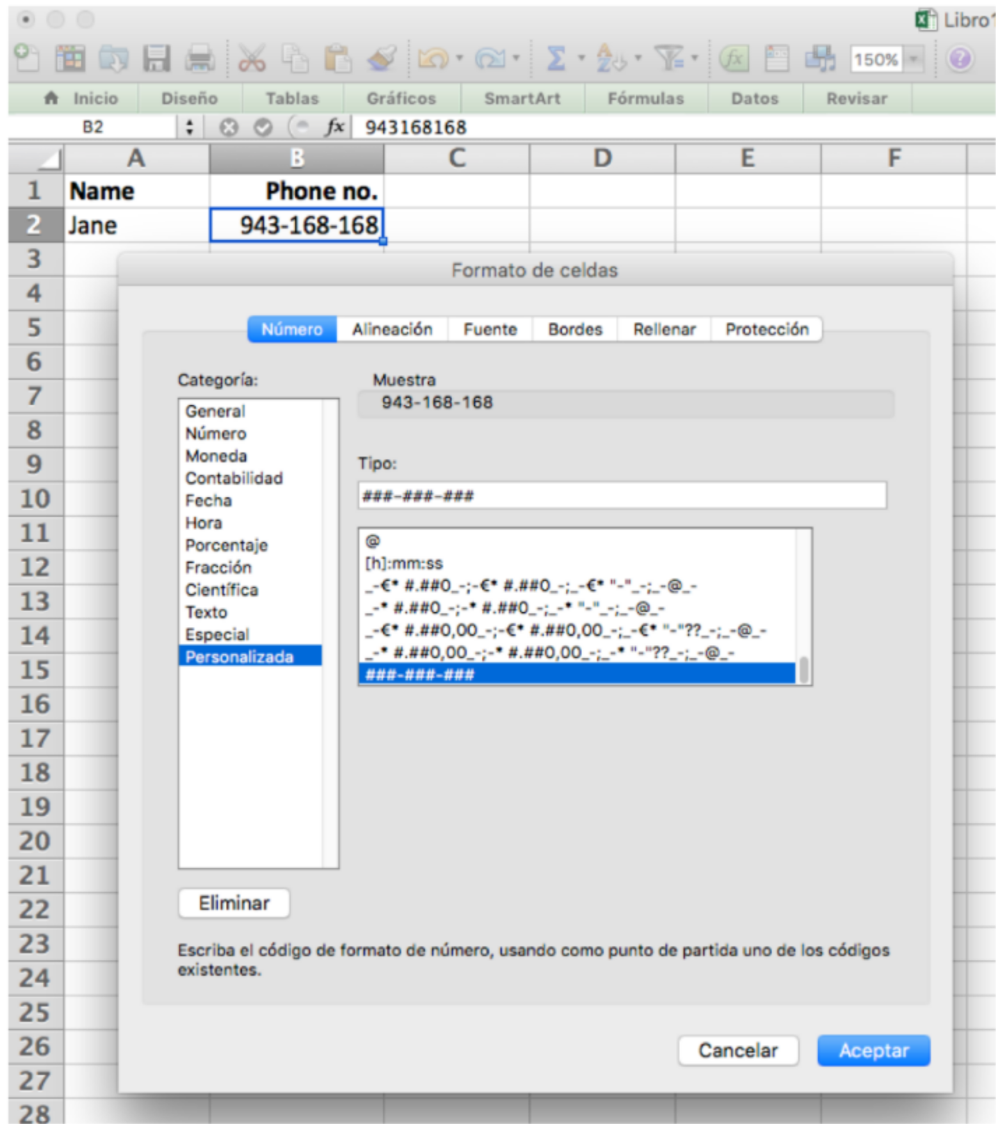


Figure 3.5: Use of *input masks* in Microsoft Excel

3.6 From iMacros to WebFeeder: from coding to modeling

WebFeeder is a plugin that works on top of *iMacros*. *iMacros* is realized as a sidebar where artifacts (i.e. *iMacros* scripts) can be *recorded*, *edited* and *played*. In this work, we extended *iMacros* with a second type of artifact: ***feeders*** (see Figure 3.4). *Feeders* are abstractions of form filling scripts (i.e. script models), which can be ***synthesized*** from *iMacros* scripts, and ***run***, i.e. transformed from models into macros, and next, enacted. This round-trip from scripts to *feeders* yields the very same script if all the input data is constant and provided by the user at recording time. However, we extended *iMacros* with a configuration parameter: the external data sources (Figure 3.4(1)). This permits *WebFeeder* to obtain a rudimentary conceptual model from the external source before recording. Then, at recording time (Figure 3.4(2.1)), when an input field is detected, the user is prompted to set the mapping between the entry field and the conceptual model. This prompt is realized using web augmentation techniques, that permit to enrich the actual webpage interface with new elements. In this case, a block with the data source options is presented to the user. Figure 3.4(2.2) shows this layered menu for the sample case. The *Teléfono* input field is mapped to the *phone* attribute of the *Company* class, when this class is playing the role of the main company of the project (*PROJECT.mainCompany)*. Furthermore, it is possible to set additional information for each input field, as we can see in figure3.4(2.2. *Additional Information*). Here, we have three different options:

- **mask**: This field permits to format the introduced data during replay. The dropdown list in figure 3.4(2.2. *Additional Information*) shows a set of predefined formatting templates. In Spain phone numbers are usually composed of 9 digits separated with hyphens, so in figure 3.4(2.2. *Ad-*

ditional Information) the chosen mask is *###-###-###*. The template specification was inspired by Microsoft Excel's input masks. In figure 3.5 we can see an example of the use of these masks, where the phone number 943168168 was formatted as 943-168-168.

- **pattern**: This field is used as a restriction, as it checks the validity of the data that is to be introduced into the text field during replay. The pattern is represented as a regular expression. The use of this restriction is complemented with the following field: the *default value*.
- **default value**: The default value is a piece of data that can be used in case there was no information on the database to fill in this field, or if the data extracted from the source is not valid (i.e., it does not match the *pattern* previously defined). The use of the default value is highly recommended in case of mandatory fields that can block the execution of the form filling script.

The important point to notice in this phase is that this mapping information (together with the *Additional Information*) is captured as part of the sample iMacros script being recorded.

At synthesis time (Figure 3.4(3)), a *feeder* is obtained from the recorded macro and stored as part of the *iMacros* artifacts (e.g. the *CDTI_feeder* folder in Figure 3.4). At run time (Figure 3.4(4)), the *feeder* is transformed back into an iMacros script where mapping links are resolved during the transformation process so that the resulting script is a totally valid (i.e. totally instantiated) *iMacros* script. The whole process goes on without programming nor leaving the *iMacros* side bar.

Implementation wise, three Ecore (meta)models are involved (see Figure 3.3(right)): (1) form filling scripts are abstracted in terms of *FormFlow* models,

(2) the structure of the data source (e.g. the database schema) is captured as a *Data* metamodel, and (3), the VBS script is mainly expressed as a weaving model between a *FormFlow* model and a *Data* metamodel. The aforementioned ***feeders*** are realized as pairs *(FormFlow* model*, Weaving* model*)* (see the *CDTI_feeder* folder in Figure 3.4).

Feeders can be ***synthesized*** and ***run***. Figure 3.6 depicts the processes triggered when pushing the namesake button in iMacros. During **synthesis**, injectors (i.e., programs that extract some data into a model) are used to obtain the *FormFlow* model (i.e. the platform-independent model (PIM)) and the *Weaving* model out of the sample iMacros script. At **run** time, the *feeder* is enacted, i.e, *(1)* references to external sources are resolved, *(2)* an *iMacros* model (i.e. the platform-specific model (PSM)) is generated merging the information contained into the *FormFlow*, the *Weaving* and the *Data* models, *(3)* this *iMacros* model is transformed into an *iMacros* script, and *(iv)* this script is run. Next sections introduce the main models that comprise the *WebFeeder* system and their extraction processes.

3.7 Abstracting the external sources

The Data metamodel. It stands for the elements and structure of the external source. If a database then, the *Data* metamodel captures the database schema as represented in the database catalogue. If a spreadsheet then, the *Data* metamodel denotes the tabular structure where data is ordered along different sheets and columns.

Injection. No matter the data source, the challenge is twofold. First, we need "a metamodel injector" that automatically obtains the *Data* metamodel out of the structure of the data source (e.g. the database schema, the spread-

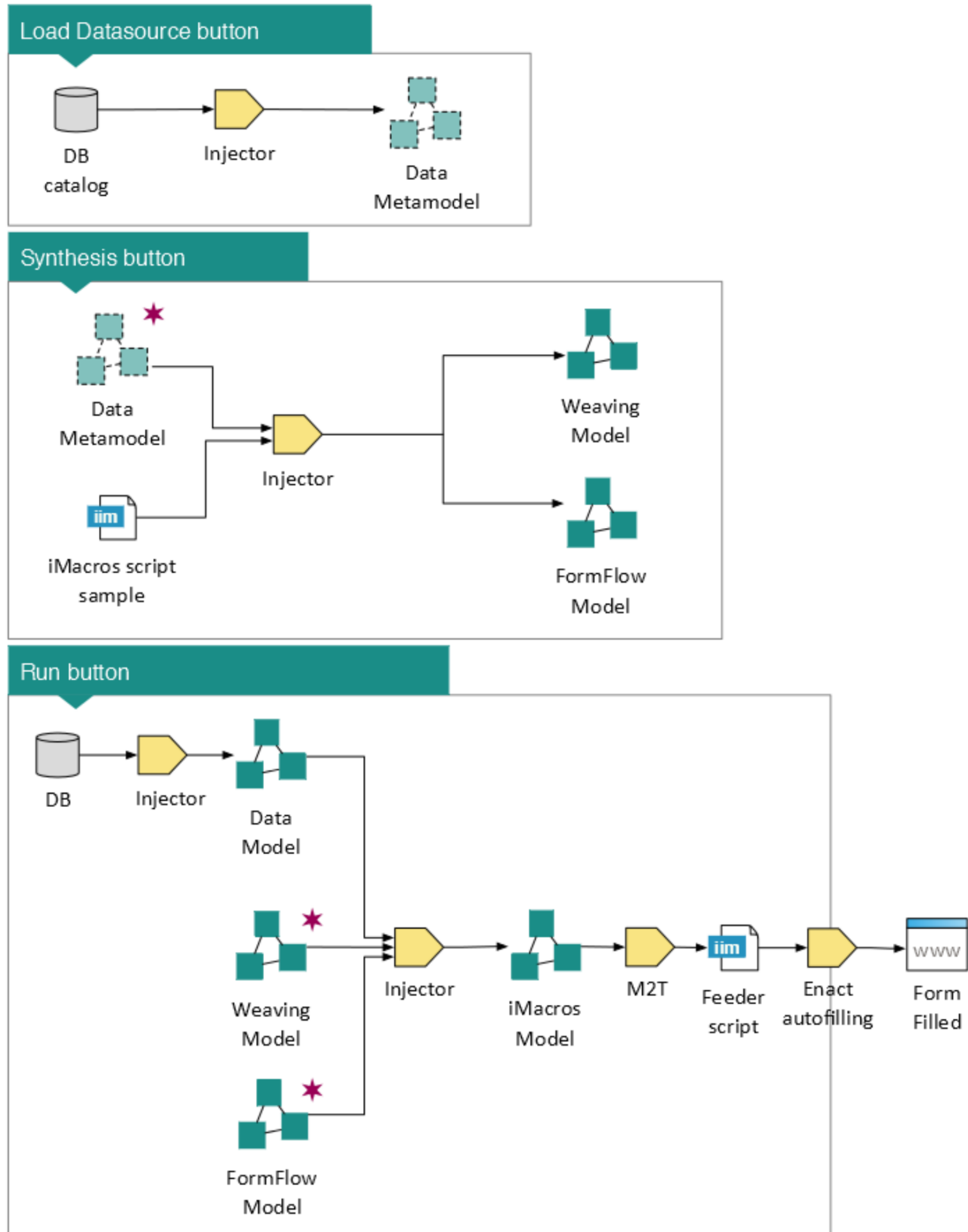


Figure 3.6: Pushing the buttons in iMacros: the **LoadDataSource** button (Figure 3.4(1)), the **Synthesis** button (Figure 3.4(3)), and the **Run** button (Figure 3.4(4)).

sheet file structure). Second, we require a "model injector" which harvests models out of a data source according to a given metamodel.

In a previous work Díaz et. al [DPCG13] studied model injection for databases, providing a language, *Schemol*, for defining database-to-model injectors. *Schemol* also permits to perform an automatic extraction of a metamodel based on a database schema. This feature, called *bootstrap,* transforms tables into metaclasses, columns into attributes, and foreign keys into references between metaclasses. This work permits to be extended to admit data sources other than databases. The approach is based on defining appropriate "drivers" that permit to conceptualize spreadsheets as databases, where sheets are the tables counterparts. The *Schemol* engine is the same but the driver changes. Figure 3.7(right) shows the *Data* metamodel automatically obtained by *Schemol* for our sample database (Figure 3.7(left)). Each project has a main company, a set of participating companies, a manager and a set of users. Meanwhile, each employee is associated with the company he works for. Both companies and employees can be associated with several projects.

3.8 Abstracting the form filling process

The FormFlow metamodel. A *FormFlow* model abstracts the process of filling in a specific web form. During this process a valid sequence of user interactions is obtained while the user performs the form filling. The model includes: (1) the elements that compose the web form (e.g., type of inputs, ids), (2) the order of the interactions (e.g., which input comes after another, when do we have to click a button, etc.) and (3) the existence of loops (i.e. a set of pages that can be filled more than once during the same process, e.g. adding many users to a project).

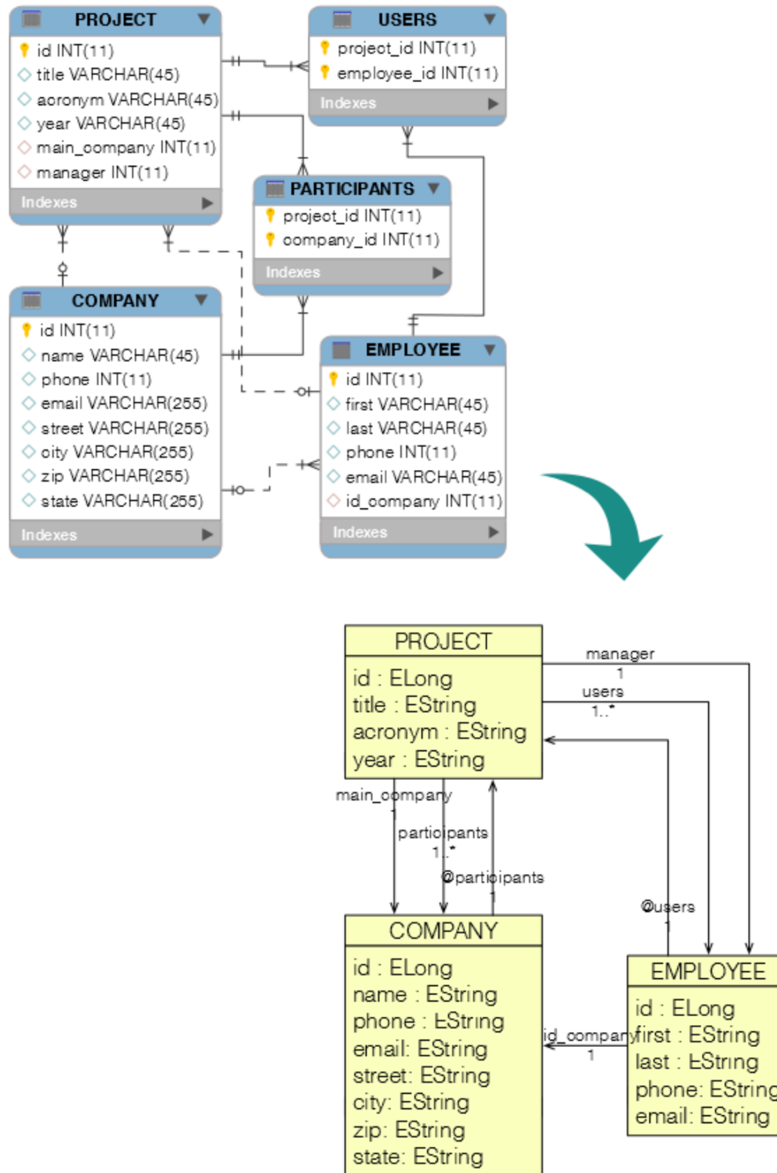


Figure 3.7: Harvesting the *Data* metamodel out of database catalogues.

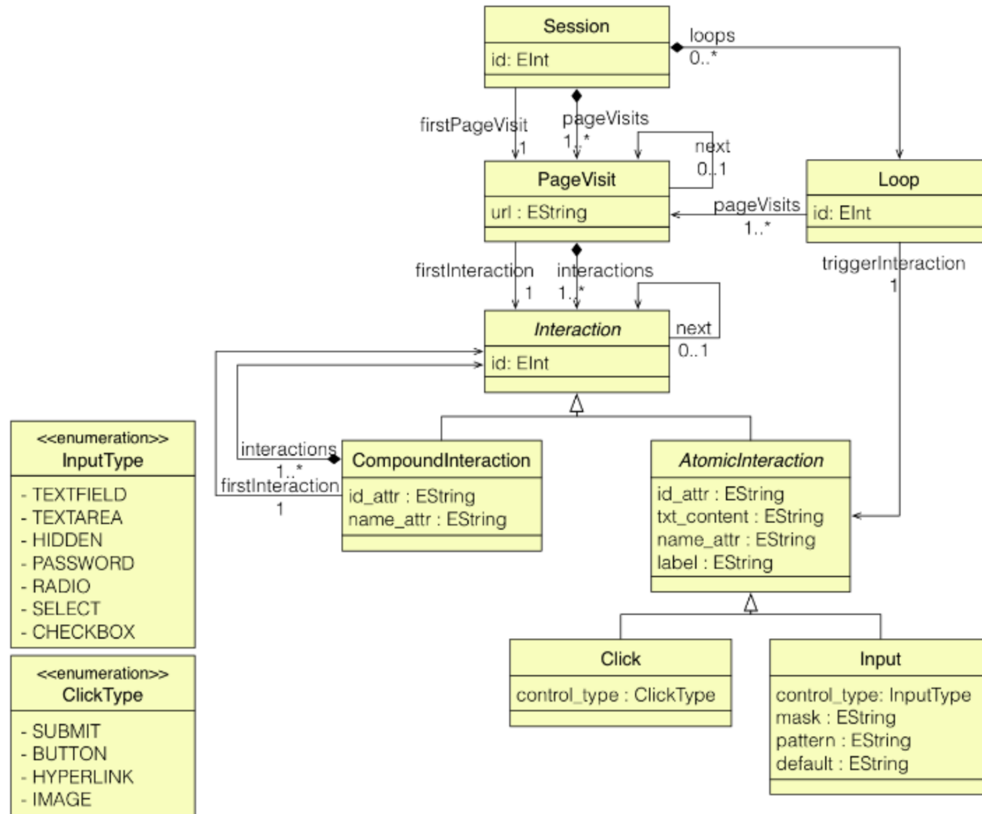


Figure 3.8: The *FormFlow* metamodel.

A *FormFlow* model abstracts a filling process through a ***Session*** class (see Figure 3.8). A *session* is a sequence of ***PageVisits*** which in turn, are conceived as sequences of ***Interactions***. Interactions can be ***AtomicInteractions*** or ***CompoundInteractions***.

Atomic interactions are classified as mouse ***Click*** or data ***Input***. Clicks act upon *Hyperlinks* or *Buttons*, and they are usually used for page navigation. *Input* elements require the user to introduce data; which can be formatted using a *mask*, checked against a *pattern* and overridden with a *default* value as seen

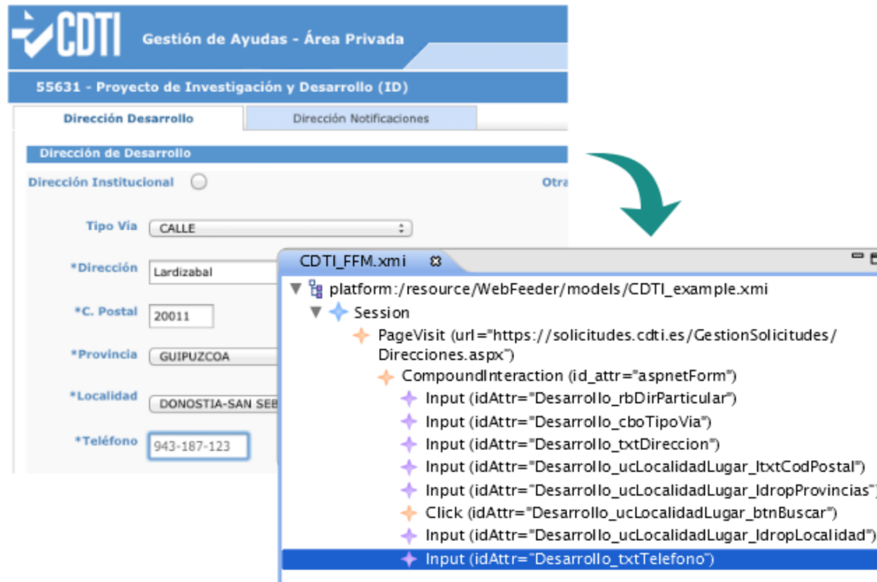


Figure 3.9: Harvesting the *FormFlow* model out of iMacros scripts.

previously in figure 3.4(2.2). On the other hand, a compound interaction stands for a meaningful unit of interactions (e.g. an HTML <form> element). ***Loops*** can be defined when many similar elements need to be introduced through the same section of the web form (e.g. adding users to a project). Each *loop* contains a reference to its trigger interaction (i.e. the button to access the first form in the loop) and the set of involved *pageVisits*.

Injection. *FormFlow* models can be *automatically* obtained from *iMacros* scripts. This process traverses the recorded macro and, for each action, it creates the corresponding *FormFlow* model element. For each *URL* or *TAG* command performed on a button or hyperlink, a *Click* element is created. On the other hand, when a TAG command is performed on an HTML input field, an *Input* element is generated. If the system detects that one command has been enacted in a different page than the previous one, a new *PageVisit* is

generated. Within each visited page, for each different HTML form element whose fields have been enacted a *CompoundInteraction* is created. Besides, if the transformation detects that a concrete set of pages have been filled out more than once using the same source of data (i.e., if during the form filling the user sets the mapping to the same database table), it automatically creates the corresponding *Loop* element. Figure 3.9 depicts the *FormFlow* model obtained from the *iMacros* script in Figure 3.2.

3.9 Abstracting the mapping

The weaving metamodel. A mapping sets a *feed* relationship from attributes in the *Data* metamodel to input fields in the *FormFlow* model. This mapping is captured as *"link"* elements along an AMW model [DBV06]. A link states a nexus (in this case, a *feed* relationship) that indicates which data is to feed what input field.

Injection. Weaving data is collected as part of the recording process. That is, we prompt the user for the weaving information when iMacros encounters an input field. This is achieved through the *FormFlowDecorator*, an interface added to the actual website using Web Augmentation techniques (see Figure 3.4(2.2)). When an input field is detected, the *FormFlowDecorator* pops up to collect the other participant in the *link* relationship: the corresponding attribute of the *Data* metamodel. In the example, the input field *C.Postal* is detected in a web form. This makes the decorator pop up, requesting its *Data* metamodel information counterpart. In this way, the *iMacros* script is leveraged with weaving data as part of the recording process. Optionally, the user can also provide the mask, the pattern and a default value for the input field at hand. Similar to the previous cases, an injector is defined to extract

```
1  VERSION BUILD=7401110
2  TAB T=1
3  URL GOTO=https://solicitudes.cdti.es/GestionSolicitudes/Direcciones.aspx
4  TAG POS=1 TYPE=INPUT:RADIO ATTR=ID:Desarrollo_rbDirParticular
5  TAG POS=1 TYPE=SELECT ATTR=ID:Desarrollo_cboTipoVia CONTENT=%178
6  TAG POS=1 TYPE=INPUT:TEXT ATTR=ID:Desarrollo_txtDireccion
-      CONTENT=Lardizabal
7  TAG POS=1 TYPE=INPUT:TEXT ATTR=ID:Desarrollo_ucLocalidadLugar_ltxtCodPostal
-      CONTENT=20011
8  TAG POS=1 TYPE=SELECT ATTR=ID:Desarrollo_ucLocalidadLugar_ldropProvincias
-      CONTENT=$Guipuzcoa
9  TAG POS=1 TYPE=INPUT:IMAGE ATTR=ID:Desarrollo_ucLocalidadLugar_btnBuscar
10 TAG POS=1 TYPE=SELECT ATTR=ID:Desarrollo_ucLocalidadLugar_ldropLocalidad
-      CONTENT=$DONOSTIA-SAN<SP>SEBASTIAN
11 TAG POS=1 TYPE=INPUT:TEXT ATTR=ID:Desarrollo_txtTelefono
-      CONTENT=943-187-123
```

- - - Information extracted from the Data model

Figure 3.10: A *feeder* generated for the form in Figure 3.2

the *Weaving* model out of the enriched iMacros script. No additional user intervention is required.

3.10 Generating the *feeder* script

A *feeder* script is an iMacros script where data is obtained from an external source. A *feeder* is generated from a *FormFlow* model, a *Data* model and a *Weaving* model, using a model-to-text transformation. This transformation contains a set of rules, one for each type of *FormFlow* element, that is, for each leaf element in this metamodel (i.e. *Click*, *Input* and *InteractionBlock*, see Figure 3.8). For instance, *Click* elements are directly transformed to the corresponding iMacros code for clicking the selected element. This generation is achieved by running a *MofScript* transformation [mof] that is governed by the structure of the *FormFlow* model: it traverses the *FormFlow* model through the *next* link, and gradually generates the iMacros instructions to mimic this navigation in the browser. In addition, the processing of *Input* elements in-

volves locating *links* in the *Weaving* model referring to the *Input* element at hand, and recovering its *Data* element counterpart. This process also entails potentially changing the format (applying the *mask*) and verifying the data (using the *pattern*).

Figure 3.10 shows the generated *feeder* for the sample page in Figure 3.2. Notice how the *feeder* combines information extracted from the *FormFlow* model with the data extracted from the *Data* model (dotted line).

3.11 Facing upgrades

Section 3.5 characterizes current solutions as being programming intensive and fragile. *WebFeeder* moves this endeavor from the programming realm to the modeling realm, and in so doing, reduces the effort and the skills required to obtain a solution. However, models (i.e. *feeders*) are still fragile. That is, upgrades on either the website or the structure of the data source can make the *feeder* break apart. This section addresses this issue.

Table 3.2 typifies some of the possible changes. Upgrades can be handled using corrective or preventive actions. A corrective action deals with an upgrade that has occurred, and a preventive action addresses the potential for an upgrade to occur. We opted for a preventive strategy for tackling upgrades. That is, for each page visit we first check whether an upgrade occurred before generating the filling script for that page. If the elements on the form changed, the user is prompted so as to reestablish the consistency between the *feeder* and the external dependencies (i.e. the website or the database schema). To this end, we introduce the **"safe mode"** for running *feeders*. This mode permits the FormFlow Model to be updated during replay, avoiding the generation of outdated scripts (see figure 3.11). Thus, when upgrades are expected (e.g., the

Change	Frequency	Contingency action
Create table	low	None
Drop table	low	Update Weaving model
Add column	high	None
Drop column	high	Update Weaving model
New Page	low	Regenerate the *feeder* from start
Delete Page	low	Regenerate the *feeder* from start
New Form	low	Update *FormFlow* model
Delete Form	low	Update *FormFlow* model
New Field	high	Update *FormFlow* model
Delete Field	high	Update *FormFlow* model

Table 3.2: A classification of upgrades for websites and database schema. Frequency is based on anecdotal evidences from the test case.

user notices the web was updated, or the feeder was created a long ago), *feeders* can be run in "safe mode".

Compared with the normal execution, this mode introduces two main differences as for the *feeder*-to-iMacros transformation, namely:

- the transformation is not enacted as a single shoot but it processes one *PageVisit* at a time. This introduces a kind of "lazy evaluation" where the transformation of *pageVisit* elements and the enactment of the resulting iMacros scripts are intermingled: *transform(pageVisit_1, scriptOutput_1), enact(scriptOutput_1), transform(pageVisit_2, scriptOutput_2), enact(scriptOutput_2)*, etc. The rationales are twofold. First, this permits to phrase *upgrade detection* in terms of model differences between the existing *pageVisit* model and the current *pageVisit* model as extracted from the current page. Second, *upgrade resolution* is also handled at the

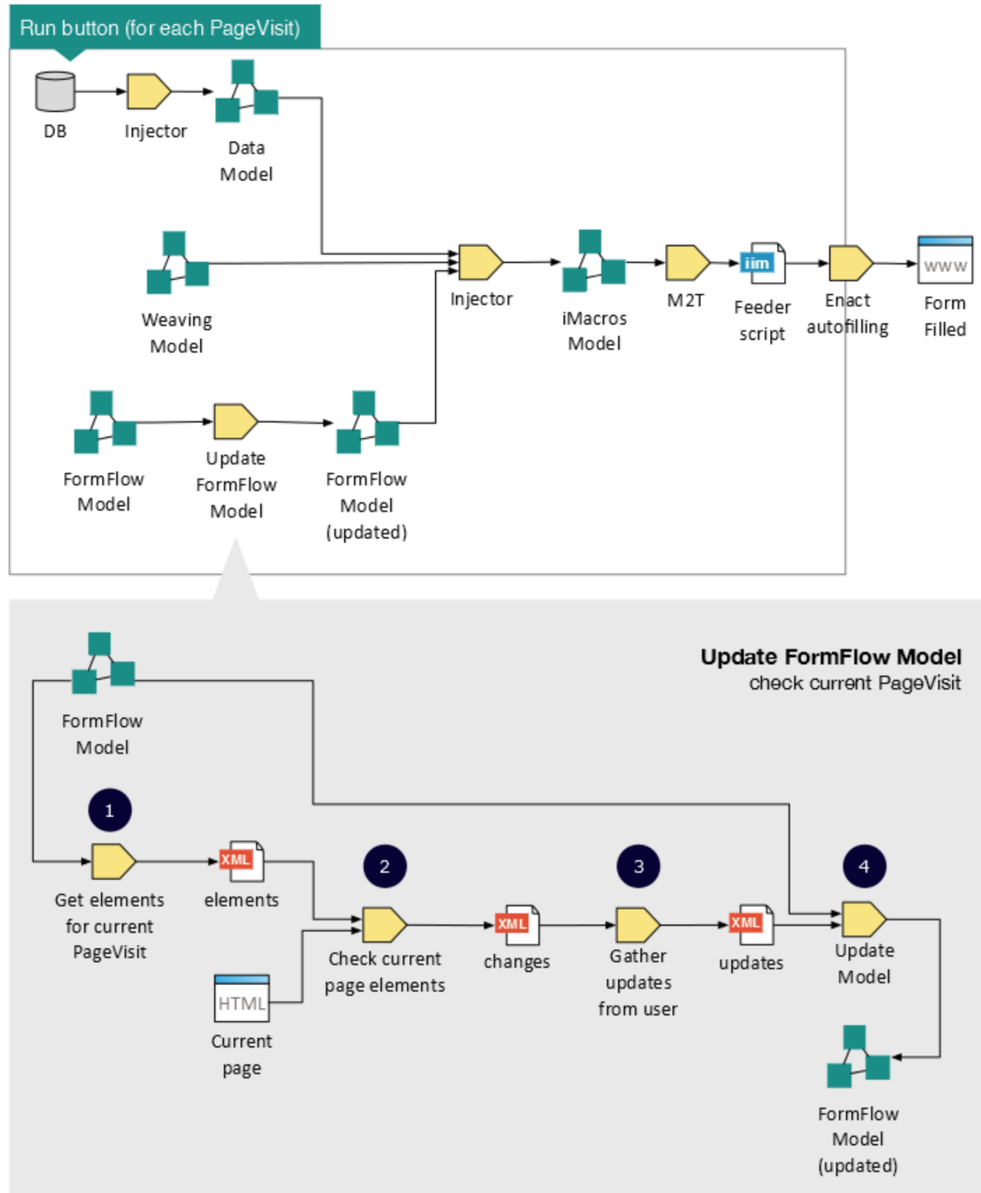


Figure 3.11: Updating the *FormFlow* model using the "Safe Mode" (see figure 3.6 for the normal mode)

page level, hence facilitating the intervention of the user at the time and at the place where the mismatch is detected (see next point).

- the transformation is leveraged with "caution clauses". Two types of caution clauses are introduced to handle each type of upgrades. For upgrades on web pages, a caution clause is introduced before the generation of the corresponding iMacros script. On loading, the clause transforms the current page visit information (the one stored into the existing *FormFlow* model) into an XML file (see figure 3.11(1)). The elements on this XML are then checked against the elements existing into the loaded page's DOM tree (see figure 3.11(2)). If mismatches, the *FormFlowDecorator* pops up for the user to restore the consistency. Once the information is provided, the FormFlow model is updated (see figure 3.11(4)) and the autofilling for the current page is enacted.

Figure 3.12 illustrates the case of upgrading the sample form with a new field. *WebFeeder* detects a mismatch between the current *FormFlow* model and the *pageVisit* model as extracted from the current page. The execution stops and the user is prompted to provide a data weaving for the new field (if appropriate). Alternatively, the user can ignore the addition of the new field if he considers it is an optional piece of information. Next, the execution resumes and goes to the next page.

The bottom line is that *feeder* co-evolution is handled using the very same mechanisms that those of *feeder* construction, hence minimizing the cognitive burdens. Clerks can cope with (small) upgrades by themselves without resorting to the technical staff. however, disruptive upgrades like introducing new tables or adding or deleting pages, require re-generating the *feeder* from scratch.

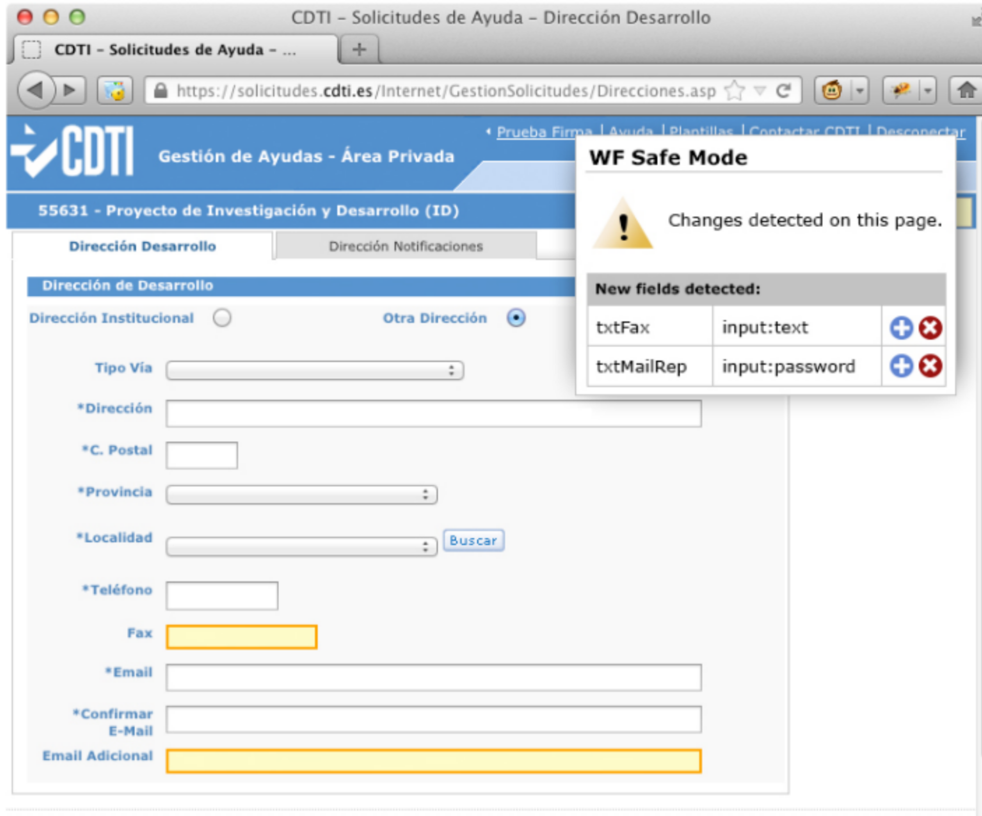


Figure 3.12: Upgrading the sample form with a new field.

3.12 Evaluation

Webfeeder is a tool aimed at helping clerks when filling in large web forms. This work was tested through two different use cases on the application for grants for R&D projects. A first use case was performed with a small set of input information, by filling in only the fields required to save a first draft of the project. The second use case was performed by filling in the web forms with information of a complete project.

Both use cases were tested on two different web platforms: SPRI and CDTI.

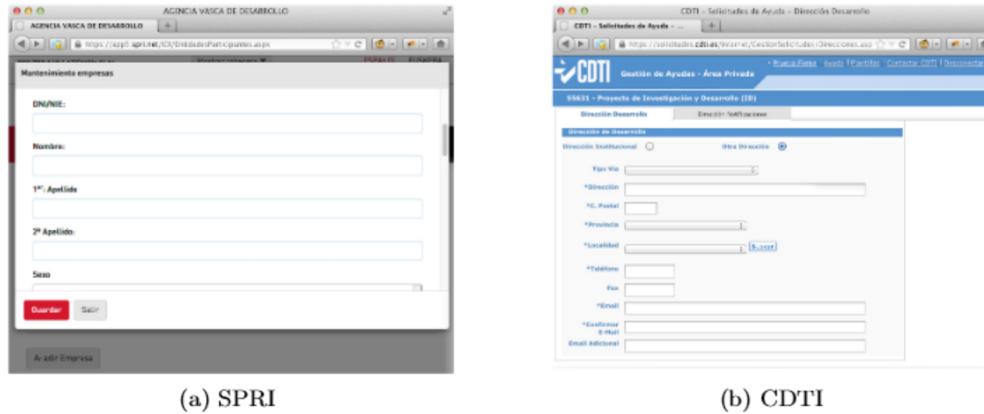
SPRI[1]**.** The SPRI Group is the Business Development Agency of the Basque Government. Its goal is to support and promote Basque companies through different programs and services. On their webpage we can find the list of subsidies offered for R&D projects and the links to their online tool in order to make the application for them (see figure 3.13a).

CDTI[2]**.** CDTI is the acronym for *Centro para el Desarrollo Tecnológico Industrial* (Centre for the Development of Industrial Technology). This organization is a Public Business Entity, answering to the Ministry of Science, Innovation and Universities of the Spanish Government, which fosters technological development and innovation activities of Spanish companies. CDTI also offers several subsidies for companies to carry on R&D projects, and the application can be made through a web form linked on their webpage (see figure 3.13b).

To make the comparison between the use cases, the number of fields that required manual filling were measured, that is, the number of times the user needs to introduce a field data by hand. In other words, we strive to measure the manual effort a user needs to make each time a web form has to be filled out.

The database used for this test was created based on the structure of a real R&D project management database that was used by ISG4, a company that

[1]**https://www.spri.eus/en/**
[2]**https://www.cdti.es/**



(a) SPRI (b) CDTI

Figure 3.13: The SPRI and CDTI web forms used for testing Webfeeder

collaborated on the early states of the Webfeeder project. Due to confidentiality restrictions, in this work we used fictitious project data.

Use case 1: Form filling with basic project information

A first test of the performance of Webfeeder on the SPRI and CDTI platforms was performed by filling in the basic information for one project, including information about the company that leads the project, contact data and technical details about the project. Only the required fields to allow saving a first draft of the project were filled in: 57 fields in the case of SPRI (see figure 3.14a) and 29 in the case of CDTI (see figure 3.14b) . As regards to the *Webfeeder decorator*, only the required fields were set, that is, the data source information (class, role and attribute).

We measured the number of fields that needed to be manually introduced by the user. In the manual method (Figure 3.14, 'M' markers), every field has

to be set by hand. The number of used fields increases each time the form is filled (i.e., each time we introduce a new project into the platform).

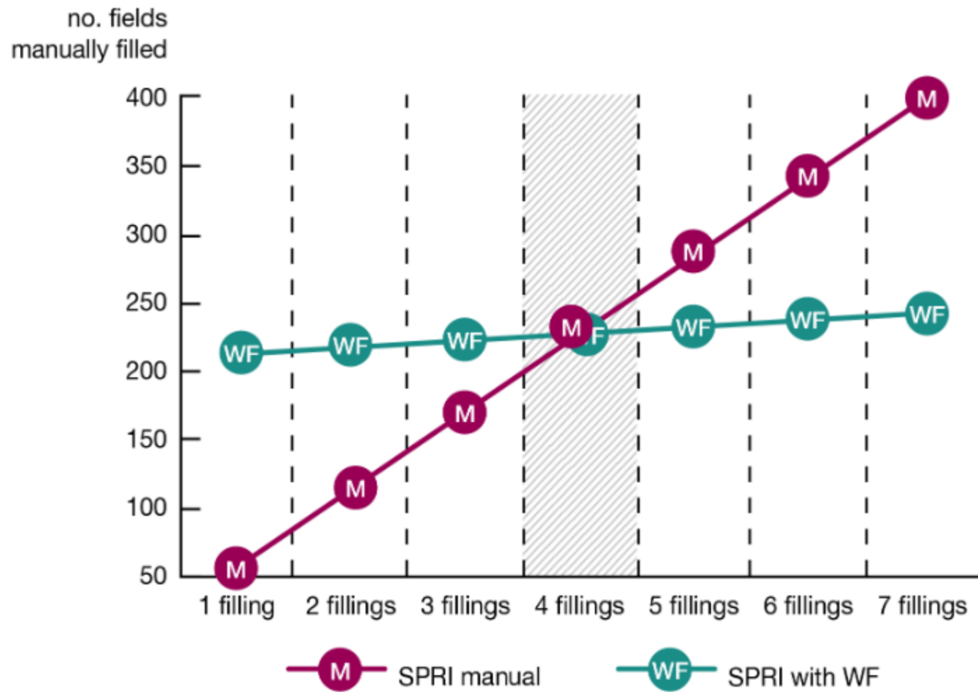
The first marker of the form filling using Webfeeder (Figure 3.14, 'WF' markers) corresponds to the recording phase. At this point, the use of the Webfeeder decorator increases the number of fields that need to be manually set. In this case the user has to fill in the same fields than in the manual method, plus the fields he needs to set up in the decorator. In fact, in the case of the SPRI the Webfeeder decorator increased in around a 274% the amount of form fields that needed to be filled in, while in the case of CDTI this increase was about a 280%.

However, as of the second form filling, Webfeeder automates most of the form fields (all of them except the ones that were marked as *"manual"* on the recording phase, that is, no link to any data source attribute was established). As we can see in figure 3.14 the initial investment is recovered after the fourth filling of the web form.

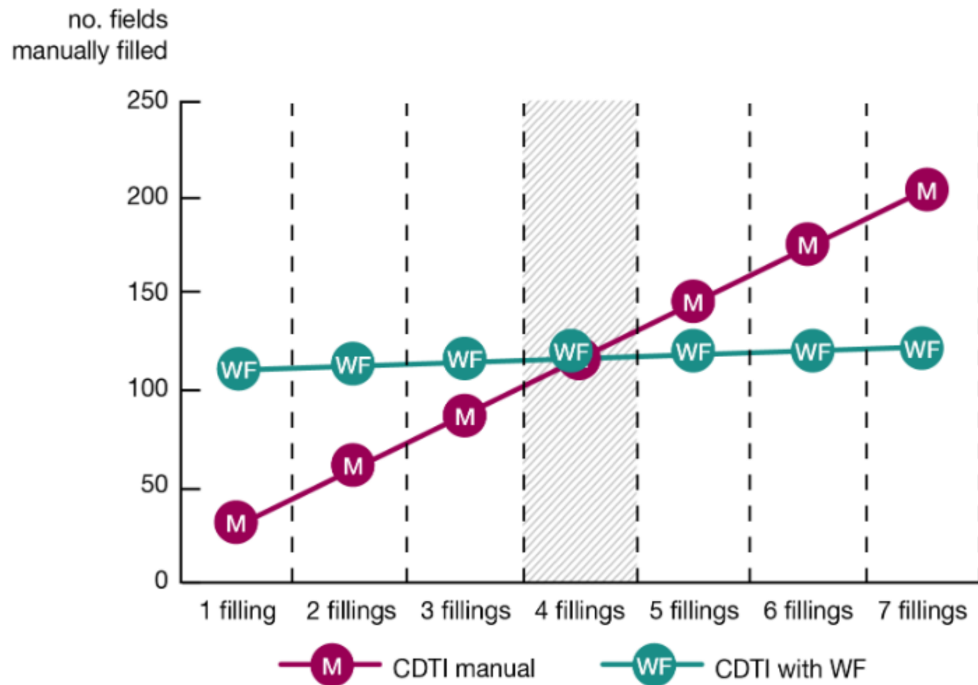
Use case 2: Form filling with full project information

After the simple test was performed, we carried on a full test by introducing all the data available in the database into the same two platforms (SPRI and CDTI). In this test, 349 fields were filled in for the SPRI case (see figure 3.15a), and 1221 fields for the CDTI (see figure 3.15b). Regarding to the Webfeeder decorator, in addition to the data source the optional fields (the mask, pattern and default value) were also set when possible.

In this case we also measured the number of fields that needed to be manually introduced by the user. As in *use case 1*, the manual method (Figure



(a) First approach for SPRI (57 fields filled in)



(b) First approach for CDTI (29 fields filled in)

Figure 3.14: Use case 1: break-evens for a simple form filling

3.14, 'M' markers), requires every field to be set by hand. The number of fields scales linearly each time the form is filled (i.e., each time we introduce a new project into the platform).

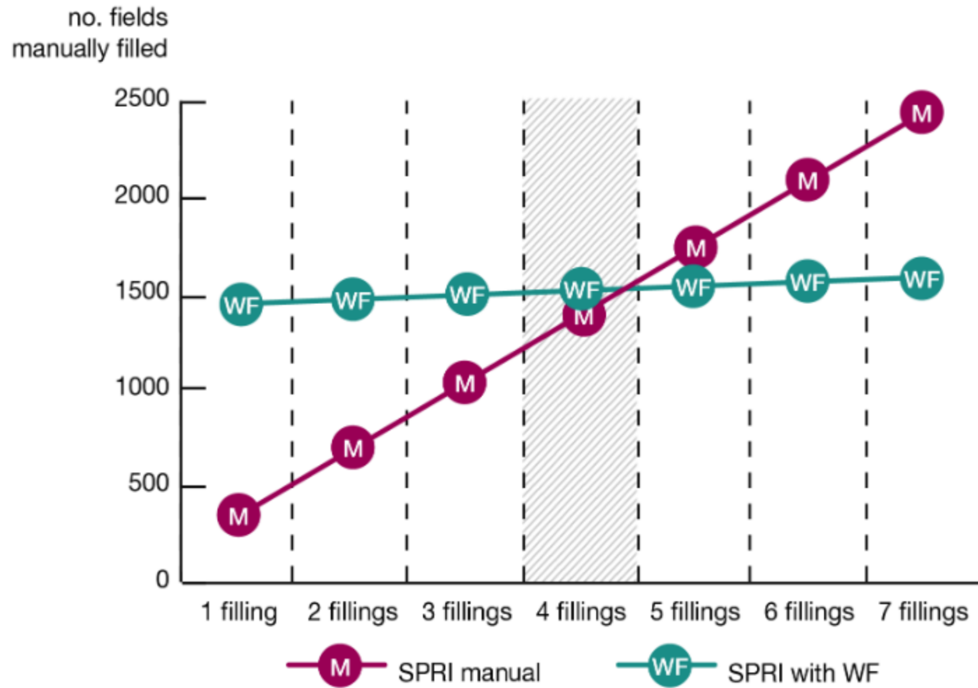
Using Webfeeder (Figure 3.14, 'WF' markers), the first form filling corresponds to the recording phase. At this point, the use of the Webfeeder decorator increases the number of fields that need to be manually set. Just like in *use case 1*, on large forms the initial investment is high: in the case of the SPRI the Webfeeder decorator increased in around a 316% the fields to be filled in, and in the case of CDTI this increase was of about a 304%.

Due to database limitations, some fields were marked as "manual" on the recording phase (that is, the required data was not available at the database and the user needs to manually provide it during replay). In the SPRI case, 23 form fields were set for manual filling, while in the CDTI case 97 manual fields were set. These 'manual' fields make the following Webfeeder executions not to be completely automated.

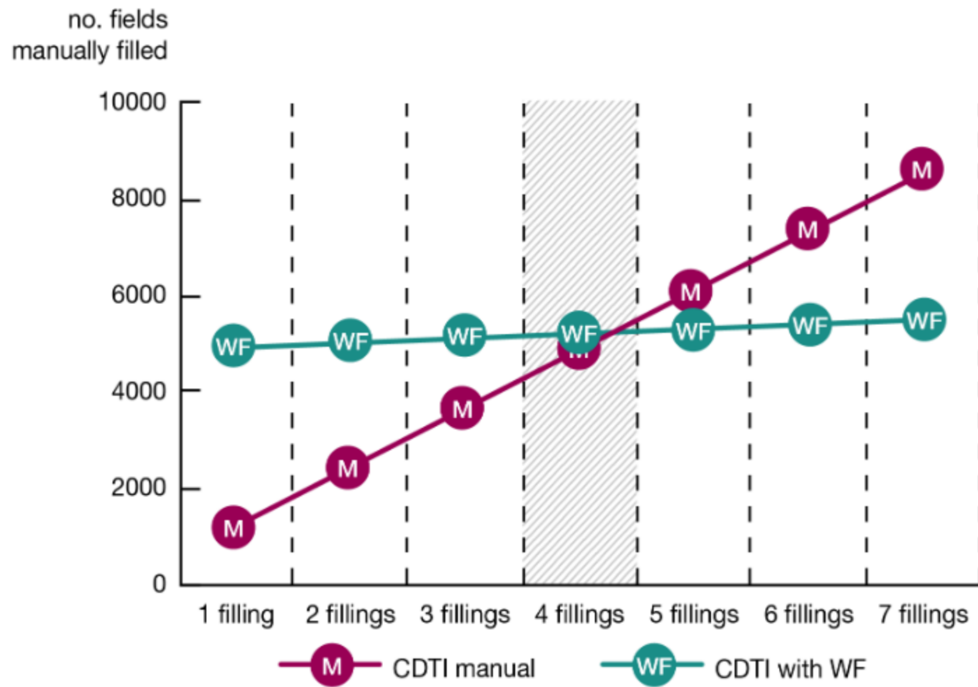
As of the second form filling, Webfeeder automates most of the form fields (all of them except the ones that were marked as "manual" on the recording phase). As we can see in figure 3.15 the initial investment is recovered between the fourth and the fifth web form filling. However, as the number of "manual" fields increases, the number of web forms required to recover the initial investment augments.

Evaluation conclusions

Results show that the initial investment required to use Webfeeder is quite high: in order to link form fields to the database at least 3 extra fields must be



(a) Full test for SPRI (349 fields filled in)



(b) Full test for CDTI (1221 fields filled in)

Figure 3.15: Use case 2: break evens for a complex form filling.

filled in (i.e., the decorator datasource fields: class, role and attribute). The larger the form, the higher is the amount of Webfeeder interactions required.

Although results show a high initial investment to start using Webfeeder, we observed that this effort is recovered around the fourth filling of the web form. This return of investment may vary depending on the use of the decorator: the more optional decorator fields are used, the later is this effort recovered. However, taking into account the maximum decorator fields that can be filled in for each form entry (that is 6 fields: 3 for the data source and 3 optional), the upper limit for this recovery is always of 7 form fillings in the worst case (that is, on the case we filled all of the 6 decorator fields for every entry). Nevertheless, reaching this limit is unlikely: in this first experiment we observed that around a 30% of the decorator fillings used at least one optional field, and less than a 5% used more than one of these options (in most cases only the default value was set).

The Webfeeder performance is also very bounded to the suitability of the database: if the data required to fill in the form fields is not available, it has to be manually introduced by the user on each replay. The more "manual" fields are set, the less automated are the *feeder* executions, which jeopardizes the benefits of using Webfeeder.

Feeders are very bound to the form structure and fields they were recorded for. However, we observed that the forms used for this evaluation are quite stable in their structure and fields (the SPRI web form was updated after 5 years, and the CDTI form stays the same nowadays since 2010). That increases the opportunities of *feeder* reuse before they need to be updated.

3.13 Future Work

During evaluation we run into several situations Webfeeder did not support. In all of these cases, the problematic fields needed to be set for manual filling. For instance, one issue was found when dealing with the SPRI platform: many form inputs are presented as text-areas that need to be filled with a composition of different fields on the database. For example, all the activities of a project and their milestones must be listed into a single text-area. We intend to add the option to compose several database entries into one single entry as future work.

In the CDTI case, the web architecture was more complex than the SPRI one and presented some semantic problems (e.g., some buttons were created using clickable paragraph elements '<p>' or image elements '<img>'. Click actions were enabled through the use of javascript code). These problems caused the original version of iMacros to work erroneously, as it does not record click actions performed on 'non-clickable' elements. In this work, this kind of problems were solved when found, and Webfeeder was progressively extended with new features that allow iMacros to record new types of interactions. However, a more detailed analysis of all the possible situations is needed, in order to adequate Webfeeder to a wider number of real web forms.

3.14 Conclusions

We address the feeding of form-intensive websites from external sources. Solutions such as *iMacros* are characterized as programming intensive and fragile, which moves these tools away from their more likely audience: clerks. We strive to empower clerks by abstracting the way at which feeding solutions are realized. Our approach abstracts the development effort from the coding of

iMacros scripts to the conception of models (i.e. *feeders*) from which these scripts are generated. In addition, *feeder* co-evolution (i.e. propagating website/data structure upgrades to the *feeder*) is handled using the very same mechanisms that those of *feeder* construction, hence minimizing the cognitive burdens. Clerks can cope with (small) upgrades by themselves without resorting to technical staff. The approach is realized through *WebFeeder*, a plugin for *iMacros*. *WebFeeder* introduces script models (i.e. *feeders*) as first-class artifacts in *iMacros*. *Feeder* synthesis, enactment and maintenance is handled without leaving *iMacros*.

First results show that the initial investment required to use Webfeeder is hard compared to the manual method, specially in large web forms. However, we observed that this effort is usually recovered around the fourth filling of the web form. That is, when the same form needs to be filled more than four times, the use of Webfeeder makes up for the time invested using the manual method.

4

User Acceptance Testing for Agile-Developed Web-Based Applications

4.1 Overview

Software development process includes different test levels such as unit testing, functional testing, integration testing or system testing. However, it is not until the user tests the application that its adequacy to the client's needs is checked. This is referred to as User Acceptance Testing (UAT).

UAT involves validating software in a real setting by the intended audience [Mey14]. The aim is not so much to check the defined requirements but to ensure that the software satisfies the customer's needs. Some Agile methodologies propose the idea of testing-first, that is, "never write code without first writing a test that exercises it" [Mey14]. Thus, tests are created at the start of each new Agile iteration, and then used throughout the development process to check whether the requirements are met. However, the evaluation of the user interface and its usability is usually performed when the new product release is executable and accessible, requiring to integrate users at the end of each iteration [HD09]. The feedback of user tests is very valuable, and it can result in design changes for the next iteration [HD09]. Due to the frequency at which the iterations are performed, providing this feedback through in-person meetings might not scale up well. Thus, alternative ways are needed to reduce the costs of developer-user collaboration during UAT.

This work introduces a wiki-based approach where users and developers asynchronously collaborate: developers set the UAT scaffolding that will later shepherd users during testing. To facilitate understanding, mind maps are used to represent UAT sessions. To facilitate engagement, a popular mind map editor, FreeMind, is turned into an editor for FitNesse, the wiki engine in which these ideas are borne out. To facilitate test creation, Selenium IDE test recorder [Selb] was integrated into the tool.

The approach is evaluated through a case study involving three real users. First evaluations are promising. Though at different levels of completeness, the three of them were able to complete a UAT. Users valued asynchronicity, mind map structuredness, and the transparent generation of documentation out of the UAT session.

In this chapter, section 4.2 introduces the practice, i.e. UAT within Agile methodologies. Section 4.3 identifies the causes and consequences of the problem that arises within this practice, i.e., poor user engagement. Section 4.4 draws the requirements to tackle this problem. Sections 4.5, 4.6 and 4.7 hold the main contributions of this work in terms of the process, notation and tooling to realize self-paced UAT. This approach is evaluated in Section 4.8 through a case study. Implementation details, related work and conclusions end the chapter.

4.2 The Practice: UAT in Agile processes

This work focuses on User Acceptance Testing (UAT) within Agile methodologies. One of the core principles of agile development is to work iteratively, producing frequent deliveries or shippable products [HD09]. Among all the existing Agile processes, Scrum is one of the most widely used [Mey14]. In Scrum, iterations are also called *sprints*. On each sprint the complete set of product requirements (a.k.a. product backlog) is grouped into smaller collections (a.k.a. sprint backlog). Then, a complete development process is performed: design, development and testing phases are completed into each one of the iterations [SW07, HD09]. The output of each sprint is a working product increment that accounts for the features in the iteration backlog.

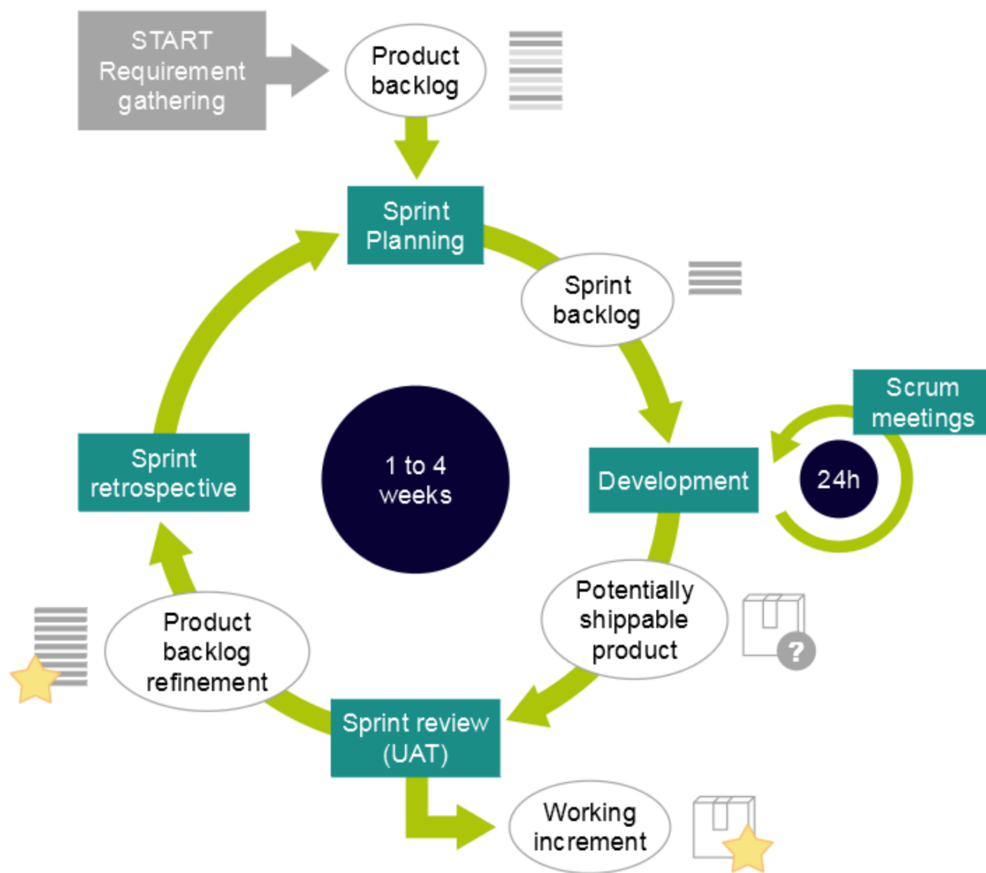


Figure 4.1: Process Outline.

In Scrum development, all management responsibilities are divided among three main roles: the Product Owner, the Team, and the Scrum Master. The *product owner* is the responsible of defining the properties of the product to be developed, being the representative of the customer and the users. This person selects the features to be developed at the start of each sprint, and evaluates the result once the sprint ends [HD09]. The *team* is composed of all the professionals that develop the product, including programmers and testers. On its part, the Scrum master is the *responsible for making sure a Scrum team lives by the values and practices of Scrum* [SB02].

Towards the end of each sprint, a *sprint review* meeting is performed. In this meeting, the team performs a task-by-task demonstration of the product in order to check whether it meets the requirements described by the product owner. In some cases, the product owner is able to perform an intensive free-hand use of the system, in order to reassure that it works as expected. Although the product owner represents the user group and is expected to evaluate the software product from the user perspective, in some cases this turns out to be ineffective [HD09]. Some works highlight the importance of the involving real users to check a system's usability, and propose to increase their implication and feedback [HD09]. Once the product owner accepts the released product, some selected users are invited to participate on a User Acceptance Testing phase, where they can freely wander around the system and give valuable feedback to the team. Their feedback might result in design changes for a future iteration (e.g., big design changes) , or in small changes added to the current iteration (e.g. bug fixes).

Figure 4.1 outlines this process. At the beginning, the features that compound the product are gathered into the product backlog. Next, the product development is carried out during a set of sprints. Each sprint planning includes selecting from the product backlog the subset of features to be developed (i.e.,

the iteration backlog). The development phase includes coding the new features and testing them through several techniques, such as: unit testing (i.e., check small pieces of code), regression testing (i.e., ensure that changes have not introduced bugs into the previously developed modules) or system testing (i.e., determine whether the application meets the requirements as defined on the backlog). The output is a potentially shippable product that accounts for the features in the sprint backlog. This product is next presented in a demo session to the product owner during the *sprint review* meeting, in order to reassure it behaves as expected. Once accepted by the product owner, the product is tested for user satisfaction during UAT. The feedback provided by users is reviewed, and the changes requested are incorporated into the product backlog. Finally, a sprint retrospective meeting is carried out by the scrum master and the team, in order to analyze and improve the entire development process [SW07].

UAT aims to "replicate the anticipated real-life use of the product to ensure that what the consumer or end user receives is fully functional and meets their needs and expectations" [Inv18]. It is important to note that UAT is not just acceptance testing, though UAT can take acceptance tests as a starting point. According to the Agile parlance, "an acceptance test is a formal description of the behavior of a software product, generally expressed as an example or a usage scenario" [All]. For many Agile teams, acceptance tests are the main form of functional specification to capture business requirements. However, some acceptance tests, such as checking the interface usability, should occur after a new product release has been released and are recommended to be performed by several real users, not only by the product owner [HD09, SH10]. As Dix et al. state, evaluation of user interfaces aims at assessing the extent of system functionality while the user interacts with and gains experience with the system, and identifies specific problems related to it [DDF+03]. This user-centered approach for UAT brings an increased quality of the final product,

as quality is more than just error-free code, but it also implies that the users' expectations are met [SH10].

The underlying assumption is that new functionalities have already been tested during development against the initial requirements defined at the beginning of the project. UAT is a new phase where system users check that the software satisfies their needs, regardless of whether these needs were previously defined or not. So "the core business is what is verified and validated and who better to do it than the business owners and the customers" [See16].

4.3 Problem definition

Software development process includes different test levels: unit testing, functional testing, integration testing or system testing. However, it is not until the customer or potential users use the application that its adequacy to the client's needs is checked. This is referred to as User Acceptance Testing (UAT). During UAT, the software is tested in a real setting by the intended audience [Mey14]. Using the software itself as a means of communication and basis for discussion helps to make sure that as little as possible is misunderstood between the customer's expression of need and what actually gets built. Agile presents similar insights under the principle "customer collaboration over contract negotiation" [agi].

In an Scrum process, customer collaboration is realized as face-to-face meetings that tend to be held every 1 to 4 weeks to validate a runnable prototype. On the upside, this high frequency facilitates an earlier adjustment of expectations between developers and customers as for the final outcome. On the downside, it requires a larger customer involvement as compared to traditional methodologies where customer participation tends to be limited to the begin-

ning and the end of the lifecycle. No wonder customers are not always ready to follow this quick pace. This makes developers complain about the lack of engagement and part-time dedication of customers [CdL12]. So much so that the lack of customer engagement is regarded as a major stumbling block among Agile practitioners [LAF13, HNM11, HH08, II15]. Adverse consequences include "pressure to over-commit, problems in gathering and clarifying requirements, problems in prioritizing requirements, problems in securing feedback, loss of productivity, and in extreme cases, business loss" [HNM11]. Stober and Hansmann highlight the importance of the product being checked by more people in addition to the product owner. They propose the delivery of a beta program after each iteration in which only selected customers can participate in order to gather their feedback [SH10].

UAT should give users the chance to interact with the software on their own, and find out if everything works as it should. Nonetheless, those users that were not completely involved throughout the development process need to be tutored, as they might not fully understand how the new software should be tested. The bottom line is that UAT excels when becoming a truly collaborative endeavor between developers and customers [CdL12]. Shaye pointed out the increased need to have a good communication and coordination between developers and customers in agile processes [Sha08].

FitNesse [Fit] is a wiki platform facilitates team members to edit and execute tests. Before development, acceptance tests can be set to describe the expected behavior of the software. After the development of a product release, UAT is performed to validate and accept the software. Unfortunately, the experience so far is for FitNesse to be still used as a mere test repository rather than a place where tests are collaboratively created [CdL12]. Overcoming this situation requires customers to be able to work both on their own and at their own pace.

We advocate for the use of mind maps as an accessible notation for describing UAT sessions (hereafter referred to as “test maps”). However, conducting UAT goes beyond specifying UAT. Conducting UAT involves assisting customers in coming up with their test maps. We need to elaborate on the tasks involved in UAT, from test setting to feedback documentation. In between, customers are assisted in different activities i.e. roaming, test coverage, enacting and commenting. This process originates a test map. A video of a UAT session along this approach is available at https://vimeo.com/157943880. This vision is evaluated through a case study involving three real web projects.

Contributions of this work are threefold. First, we propose a process that combines mind mapping and wikis to empower customers to conduct UAT on their own (i.e. self-paced testing). Second, we introduce a DSL for UAT whose concrete syntax is realized in terms of mind maps. Third, self-paced testing is realized using FitNesse as the wiki, and FreeMind as the mind map editor. These two platforms are bridged through TestMind, conceptually conceived as an editor for FitNesse that permits to capture UAT sessions as test maps. Self-paced testing is then devised as an iterative and collaborative effort where developers set the testing scaffold through the wiki, while customers build the test map that ends up as FitNesse pages.

The Problem: Causes and consequences

Ideally, UAT should be based on actual customer use, and performed by real users. Unfortunately, the lack of customer involvement is being identified as a major jeopardy in Agile [CdL12]. We searched the bibliography in order to detect the causes of this problem. Then, we grouped the causes along three main axes, namely: lack of time, lack of motivation and lack of knowledge (see Figure 4.2). However, this bibliographical review was not systematic.

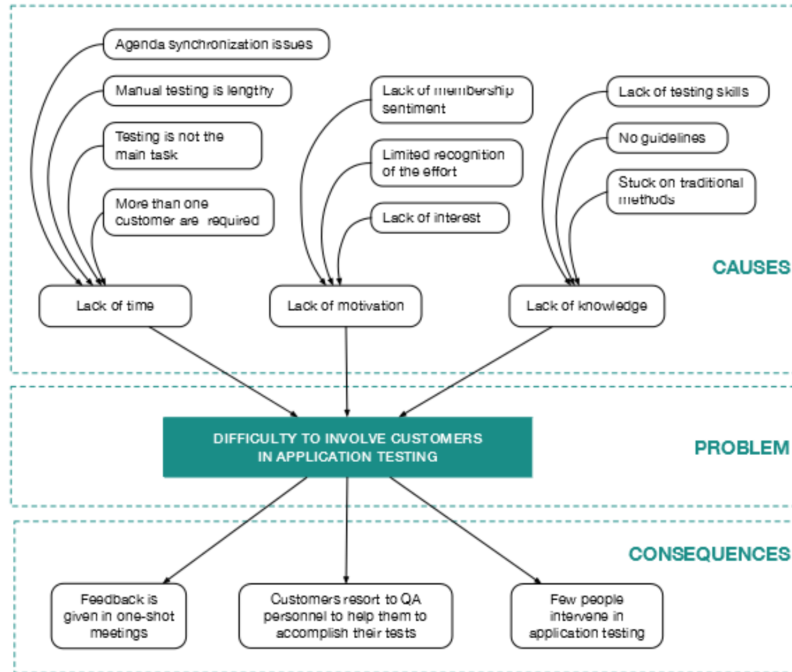


Figure 4.2: Root-cause analysis for the lack of customer involvement.

Hence, other issues might have been reported that were overlooked in our work. We didn't aim at conducting an exhaustive analysis of the rationales behind limited customer involvement, that would be a full-fledged research work on its own right in terms of a Systematic Literature Review. Here, we collect some evidences about the problem while our focus resides on attempting to solve it.

Lack of time. UAT testing goes beyond covering the defined requirements and lets customers freely wander along the application to achieve certain tasks. This implies customers to dedicate significant time away from their daily roles to participate [Sha08]. However, getting users to invest their time on testing is not easy. As Hoda et al. reported, in many cases developers complained

about the customers not having enough time to adequately participate on the Agile process [HNM11]. The literature points to three main causes: agenda issues, task issues and priority issues. The former is concerned with in-person meetings. The distance between the development team and their customers is an important factor leading to lack of customer involvement [HNM11]. This situation aggravates as the number of people involved increases. The initial books about XP describe the on-site customer as a single person. However, the bias that a single customer representative can introduce has been documented in the literature as a cause of bringing about systems that will fail to satisfy many users [RCFH11]. Also, as stated in [Mey14], "if management is so willing to assign to your project a supposed expert of the application domain, taking him or her away from tasks in that domain, you may wonder whether the person is really the most qualified". Beck also highlights that listening to just one person you risk of getting a system this person wants, but that won't be suitable for anyone else [Bec00]. Today, there is a greater acknowledgement within the community that more than one customer is needed due to the large set of issues to be handled [MBN10]. This seems also to be the conclusion in [Lin16]: "I hear teams complain about product owners being insufficiently available and unable to make decisions on a short notice. Depending solely on the product owner appears to conflict with the team-based way the Agile process promotes". Yet, the more people intervene, the more difficult is to find a time slot that fits everyone's agenda. This problem exacerbates in globally distributed environments. Here, physical as well as temporal proximity between participants is even more challenging [HNM11].

Another issue is the UAT being conducted manually. If the time is scarce, we should strive to find ways to assist this process as much as possible. A totally manual UAT approach can become tedious and time consuming which is being reported as discouraging for many customers [HH09, Pul06].

Lack of motivation. Any time is a lot if you are not committed to UAT. Shore et al. [SW07] state that developers should not underestimate the different tasks customers might need to cope with: set the appropriate priorities for the work, identify all the details that programmers will ask about, fit in time for customer reviews and testing, etc. This recognition of the customer's testing effort applies not only to the customer organization but also to developers themselves as reported by Martin et al.: "most of the programmers were unaware of the long hours the customers were working. This situation appears to be unsustainable, and so constitutes a great risk to agile projects, especially in long duration or high-pressure projects" [MBN10]. This results in a lack of membership and less responsibility on the customers' part for the success or failure of the project [Kup, Tes18].

Lack of knowledge. Plenty of time and recognition do not yet guarantee good results. Customers might be used to traditional methodologies where UAT is conducted at the end of the lifecycle. This might bring an inertia when this modus operandi changes in Agile processes [HNM11]. Instead of the intensive participation demanded by Agile methodologies, they prefer to express requirements in traditional ways or in one-shot meetings [HH08]. Besides, making customers perform testing on their own might make them feel intimidated. For instance, in [Joh09] a director testing the product confessed: "I don't know what to do. I don't know what to test and I don't know how to test." In some cases, the lack of technical knowledge lead to resort to QA personnel to help customers to develop their acceptance tests [CR08].

Having into account the aforementioned problem causes for UAT of web applications developed using Agile methodologies, this work aims at:

> *improving the acceptance testing of web applications*
> *by abstracting test scripts as mind maps*

	Lack of Time	Lack of Motivation	Lack of Knowledge
MR1_async	✔	✔	
MR2_R&R	✔		✔
MR3_scaf			✔
MR4_maps			✔
MR5_mapping		✔	✔
MR6_feedback	✔	✔	
MR7_learnability	✔		✔

Table 4.1: Issues and addressing meta-requirements.

> *satisfying learnability and replayability*
> *so as customers are empowered to create their own acceptance tests and give adequate feedback to the development team.*

4.4 Meta-requirements for a solution

This section draws some meta-requirements extracted from the three limitations identified in the bibliography. For each requirement, we look at existing solutions, if any. We ground the derived meta-requirements on research on Software Engineering. Table 4.1 maps the issues and their addressing meta-requirements.

Firstly, traditional face-to-face meetings constraint UAT to a defined time frame, and hence lessen the chances of the UAT participants to get distracted. However, some customers might not be able to spend two to four hours away from their desks for each sprint. This sets our first meta-requirement:

MR1_async: Account for asynchronous collaboration through wikis

Being a common place for knowledge management [LDP+12], wikis are known to be beneficial in the workplace for groups requiring a collaborative medium. This is a common scenario in software development [Lou06] where wikis are proposed for improving software documentation [FdS08], supporting collaborative requirements engineering [HLBCF16], fostering software reuse [RBH07] or sustaining acceptance tests [Fit].

In an Agile setting, wikis are proposed to support the application development lifecycle [SH10]. Story planning, backlog management, and acceptance testing find their way as wiki pages. A powerful exponent is FitNesse [Fit], a wiki platform geared to testing. This framework is based on Ward Cunningham's Fit (Framework for Integrated Test) [MC05], an open source framework that permits "to define acceptance tests as spreadsheets and then execute them using different languages". In FitNesse, Test pages capture tests as a collection of input-output pairs, navigation narratives or other forms. Test pages can be enacted by simply clicking a "Test" button at the top of the page. This testing framework is unique in using a wiki for creating and maintaining test cases. Here, the wiki becomes a repository for product requirements [Cri]. One of the drivers behind FitNesse is maintenance. Test automation projects often fail because tests are not designed for maintainability, and the overhead to keep the tests up to date becomes overwhelming. FitNesse helps to prevent this problem by using declarative though executable test specifications which are defined jointly with stakeholders.

Unfortunately, customers do not perceive FitNesse as an standalone communication and collaboration tool: off-line discussions are conducted prior to changes made in the wiki [HH11]. This suggests that wiki asynchronicity is

not enough to achieve self-paced UAT. This moves us to the second meta-requirement:

MR2_R&R: Offer customers a head-start using Record&Replay facilities

UAT moves away from predefined scenario scripts to more exploratory testing, finding out about the application, what it does and what it does not. Record&Replay (R&R) tools permit to automatically generate browsing scripts as the customer freely navigates throughout the application. For instance, Selenium IDE offers this feature as a Firefox and Chrome plug-in [Selb][1]. Although other recording tools exist, in this work we selected Selenium because of the big amount of help and documentation available, and because it works for all major web browsers. However, leaving customers to unrestrictedly wander throughout the application might end up in lengthy, unfocused scripts. This might be damaging in Agile. The short deadline for each sprint makes it necessary to keep focus on the new developed features. This leads to our third meta-requirement:

MR3_scaf: non-intrusively guide customers during "exploratory" UAT through "testing scaffolding"

If customers are not tutored, they might not fully understand how the new software should be tested. Developers should set "a scaffold" that assists customers during UAT. To this end, we propose the use of Kickoffs and Hints. Kickoffs set

[1] Selenium scripts can be enacted on FitNesse using the Xebium fixture. Available at http://xebia.github.io/Xebium/

the web application under test (WAUT) at a state ready to be validated. This might imply setting up a database or running a browsing script in order to set the application at some specific point, saving the customer the intermediate navigation to reach the functionality to be checked. Notice that Kickoffs are not only a question of relieving customers from initializing the application but also of getting a state that might not be possible for the customer to achieve on his own due to e.g., lack of credentials (e.g. setting the company's holiday days).

However, customers are not professional testers. They might not be aware of good testing practices, e.g. checking not only the most common scenarios but also the faulty ones. Besides, demotivated customers might focus on getting things done with simplicity, without being too much involved in analyzing the different casuistry. This behavior can jeopardize UAT as the chances of overlooking functionalities increase. To face this situation, developers can set testing hints, that is, short messages that aim at avoiding some aspects of the WAUT to go unnoticed. Hints are realized as notification messages that show up on the browser when the customer reaches specific application pages (see Section 4.5).

Scaffolding might help customers to keep focus. But once on focus, customers are left on their own. A notation is needed for customers to capture their UAT sessions. This moves us to the fourth meta-requirement:

MR4_maps: Capture UAT sessions as mind maps

Different tools and languages have been proposed to permit the definition of acceptance tests. Toolkits such as Cucumber [Cuc] or FIT/Fitnesse [MC05, Fit] offer Domain Specific Languages (DSLs) to allow the description of so-called test-stories which can later be converted into executable test cases. These test

cases can be used to measure and precisely describe the level of progress in the implementation of the requirements [RPT+08]. The use of DSLs to define test cases improves their readability, making them understandable to different stakeholders, from developers to business experts.

Based on this insight, this work investigates the use of a DSL for UAT based on mind maps. A mind map is a diagram to visually organize information. These maps are often created around a single concept, represented as a node in the center of a blank page, to which associated representations of ideas such as images or words are added [BB06]. Rationales for proposing the use of mind maps are twofold: familiarity and structuredness. As for the former, mind maps are a common notation among organizations for brainstorming and idea forming [Buz14]. Chances are customers have already been exposed to this notation, hence facilitating adoption. In addition, mind maps bring an structure where ideas are radially disposed around a root node. For instance, test maps capture actions as leaf nodes of the map, while intermediate nodes serve to group actions based on the page where the action took place. This might well facilitate understanding in comparison with current FitNesse tables where actions are displayed as table rows, and no grouping exists.

But conducting UAT goes beyond UAT specification. Mind maps might provide a more familiar notation for customers. Yet, coming up with these test maps is not obvious. This results in the fifth meta-requirement:

MR5_mapping: Rephrase UAT as mind mapping

UAT processes are rephrased in terms of mind mapping. In other words, common gestures in drawing a mind map (e.g. adding a child node) should now be overloaded with UAT semantics (e.g. test coverage).

Mind maps might well engage customers but the final consumers of this information are developers. For developers to make the best out of customers' UAT sessions, they should be able to reproduce them so as to ease the understanding of the test process specially in asynchronous collaborations. This brings us to the sixth meta-requirement:

MR6_feedback: Customers should be able to provide appropriate feedback documentation for developers to reproduce UAT sessions

This poses stringent demands on feedback documentation quality which is at odds with the lack of both time and knowledge. Here, we investigate the extent to which this documentation can be automatically generated out of test maps. Specifically, test map structure helps obtaining test scripts that can be directly enacted from FitNesse. The aim is for developers to be able to replay the scenario conducted by the customer, and to reproduce the setting where potential mismatches arise.

Even though the use of mind maps might ease the test definition task, the customer role tends to be transient, i.e. it is frequently played by different employees depending on the application at hand. That is, there might not be a second time for an employee to play the role of the customer in another application. Hence, the UAT learning effort should pay off for a single application. Therefore, the last meta-requirement consists on reducing the learning barrier for the UAT solution:

MR7_learnability: The solution should be intuitive enough for customers to be able to grasp it rapidly

To conclude, we advocate for an asynchronous collaborative approach to UAT. The instantiation of this theory is then threefold: a process that describes how collaboration takes place (Section 4.5), a notation to capture UAT sessions (Section 4.6), and a tool that supports this collaboration (Section 4.7).

4.5 The process: iterative and collaborative UAT

Wikis are widely used in industry for teamwork and collaboration [LDP+12]. Wiki pages might respond to different purposes. For instance, Wikipedia holds Article pages but also Help pages, Template pages, etc. Different questions arise about: (1) what type of pages a UAT-aimed wiki would contain; (2) when are they created with respect to the Agile cycle, and (3), who creates them (i.e. developers vs. customers). Figure 4.3 outlines an answer by spreading out the UAT box. Next paragraphs delve into the details. A Calendar application is used as the running example.

Before Development. Developers first start by defining **Sprint pages**, i.e. pages that hold information about the Agile iterations, such as the list of features included in the sprint backlog. Each feature is fully described through a dedicated **Feature page**. These pages hold a list of the customers assigned to testing and a brief description of the feature to be developed. This information is later enriched with links to the Test pages where this feature is being tested out (see an example in Figure 4.4 (3)).

After Development. Once a new product release is available, developers set a scaffold to assist customers during UAT. To this end, we propose the use of **Kickoff pages** and **Hint pages**. Kickoffs set the web application under test (WAUT) at a state ready to be validated. As an example, consider the Calendar application. Calendar permits adding new events as well as modify-

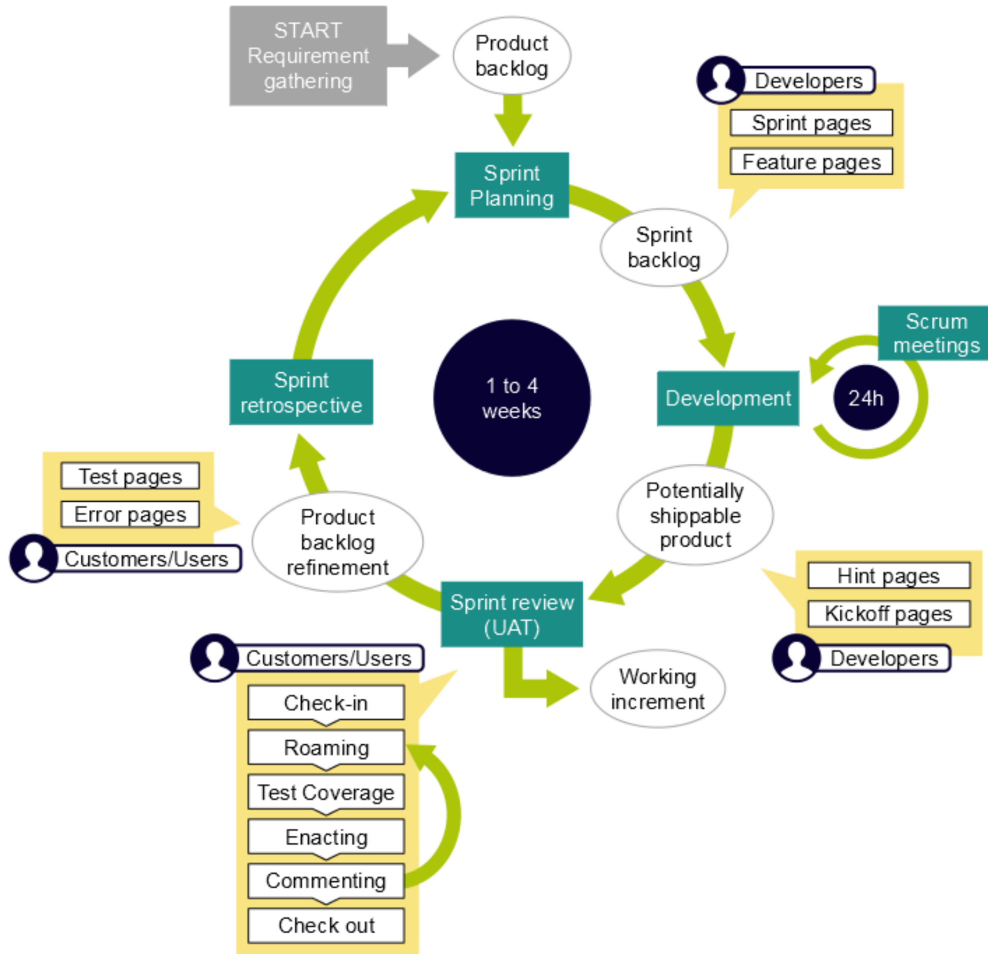


Figure 4.3: Agile process with a focus on UAT activities. Customers and developers collaborate through the wiki pages.

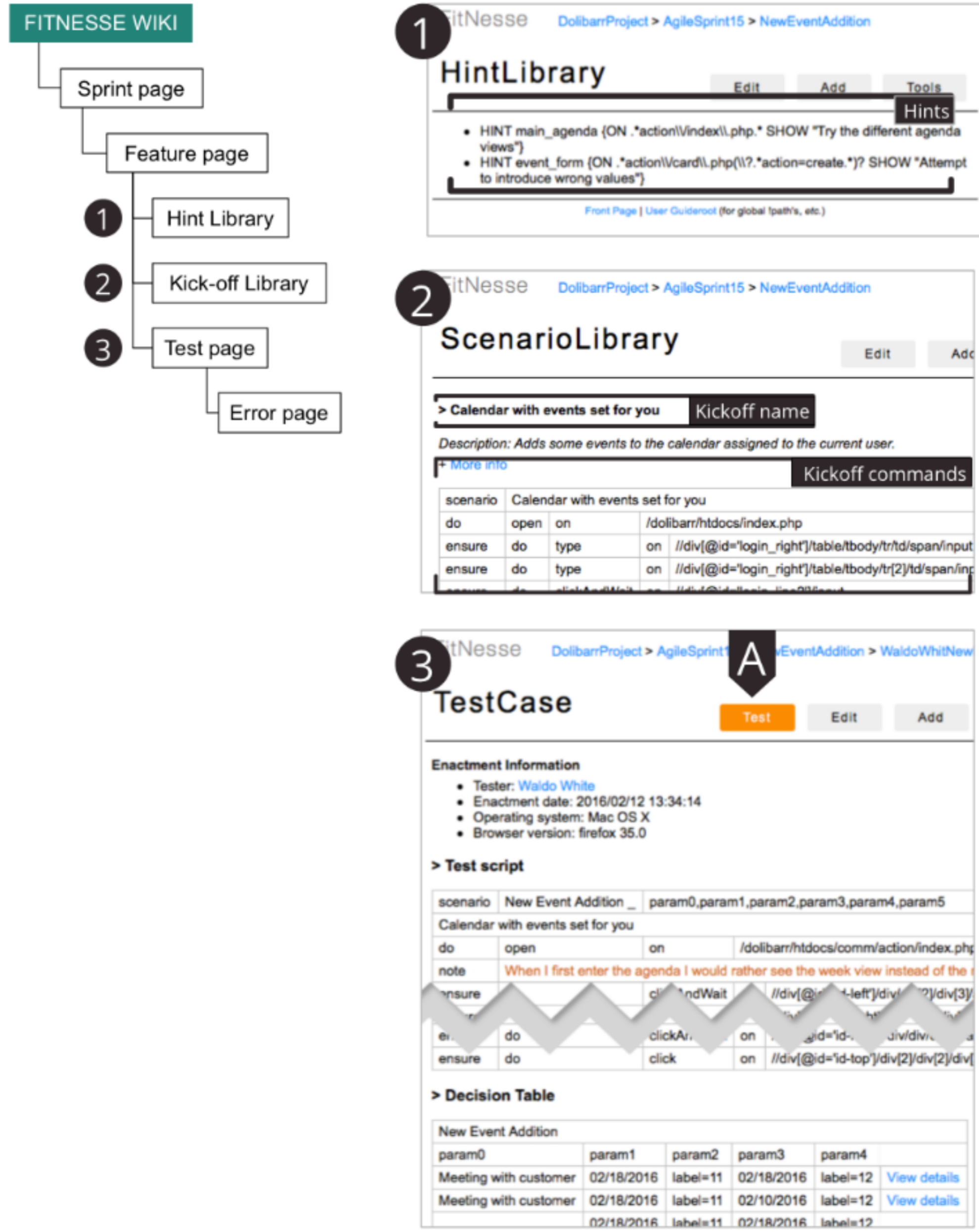


Figure 4.4: Wiki structure and distinct scaffolding pages for the *Calendar* application: *Hint* page (1), *Kickoff* page (2), and *Test* page (3). Test pages are enacted through the *"Test"* button (A).

ing the existing ones. Rather than starting with an empty calendar, Kickoffs might set the calendar in a stage ready for UAT, e.g. calendar with events set for you, calendar with public holidays already set, calendar with events jointly arranged with your colleagues, etc. Figure 4.4 (2) displays the case of the Kickoff "Calendar with events set for you". Rather than starting with a empty calendar, this Kickoff initializes the calendar with some events before the UAT session starts. Kickoffs are realized as scripts, specifically a sequence of Selenium commands that achieve the desired initial state. Kickoff pages might provide a more-information hyperlink to extend the information (i.e. kickoff description, usage suggestions, screenshots of the point where the kickoff leads to, etc).

As for Hint pages, they aim at suggesting customers to perform some testing practices they might not be aware of (e.g., "attempt to introduce wrong values"), or getting their attention to some functionality aspects that might be overlooked (e.g., "try the different agenda views"). Hint pages describe testing hints as pairs (context, message) where context refers to the point in the browsing navigation where the message should pop up. For the sample case, two hints are defined (Figure 4.4 (1)): "if the navigation reaches the URL action/index.php, then suggest the customer to check different agenda views", and "if the page action/card.php?action=create is loaded, then inform about the possibility of using wrong input data". Implementation wise, this is supported through a Selenium IDE plug-in. The plug-in tracks the context. When the context is reached, the message is visualized in the browser's hint bar (see Figure 4.7 (2)). Notice, hints should not give any clue about how to achieve tasks as this is part of the UAT itself.

During UAT. It is now the turn for customers to provide **Test pages**. Test pages are not produced out of the blue but after iterating along the following tasks: roaming (i.e. freely wandering around the application), test coverage

(i.e. adding different data sets), enacting (i.e. running the application with the provided data sets), and commenting (i.e. providing remarks about test outputs). This cycle is framed within check-in, where the UAT session is initialized, and check-out, where the UAT session is finally documented as a Test page on the wiki.

A Test page contains: the testing setting (e.g., operating system, browser version, etc.), a script table and a decision table (see Figure 4.4(3)). Script tables hold a sequence of executable commands which stand for the test case. Clicking on the "Test" button automatically replays the script, i.e. a Firefox window pops up and the user interactions are reproduced. In this way, FitNesse permits developers to easily replay the scenario conducted by the customer. Script tables also hold the customer's comments risen during the testing. Comments are captured as note rows (e.g. "When I first enter the agenda ..."). In addition, decision tables hold data sets with which the test case should be checked out. If the test did not meet the customer expectations for a specific data set, then the corresponding row might hold a hyperlink to an **Error page**. Error pages document the script results: the customer expectations about the test outcome and the screenshots taken.

Figure 4.3 highlights this collaborative nature of UAT. Customers can facilitate Test pages at any time, provided the testing deadline is not over. All these Test pages are saved under the Feature page representing the requirement that is being tested. Once the testing deadline is over, developers can go back to FitNesse and replay the customers' Test pages. In this way, developers can make a more informed decision about whether the current development really meets customers' expectations. At this point, additional hints and Kickoffs can be defined, and the UAT deadline be extended for another UAT round. The bottom line is that UAT is not a one-shot effort but a collaborative and continuous endeavor between developers and customers. This feedback might

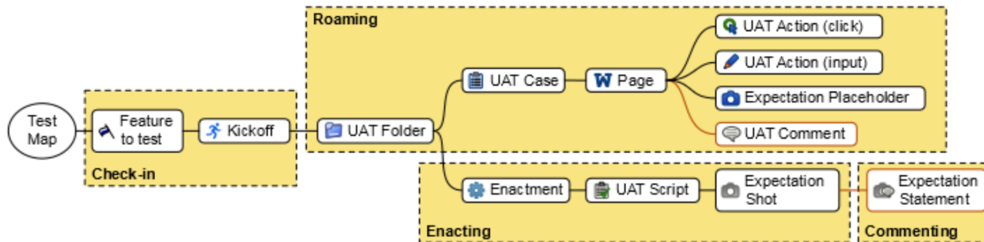


Figure 4.5: *Test maps*: mind maps to capture UAT concerns. Map nodes are overloaded with UAT significance. Their operational semantics is defined through a Domain Specific Language (see Appendix A).

lead to changes on the product backlog, which is updated after all the users have finished testing.

However, this vision has a main stumbling block: test scripts are outside most customers' competences. The question is then

> *how can Test pages be created by customers?*

This rises two main issues: a notation for capturing UAT sessions (Section 4.6), and assisting customers in coming up with test cases using this notation (Section 4.7).

4.6 The notation: test maps

The formalism used to capture tests very much depends on the kind of tests. In acceptance testing, Cucumber is a popular tool [Cuc]. As another example, Selenium provides a different notation (called Selenese) that permits to define test cases in terms of a sequence of user interactions with the application UI

[Selb]. No matter the notation used, an important requirement is for the test cases to be executable since users other than the author might need to run the test. However, manually defining test cases following a restricted notation implies some complexities from which customers should be sheltered. Here, we advocate for the use of mind maps. The insight is that mind maps might provide a balance between usability (i.e. visual and familiar notation) and formality (i.e. to be structured enough to be enactable).

Broadly, UAT sessions collect three main types of data: (1) information provided by developers to set up the customer testing (scaffolding); (2) test case definitions generated by customers (UAT Cases), and (3), test case enactments (UAT Scripts). To capture these concerns, we resort to mind maps. Figure 4.5 depicts the different kind of nodes involved and their structural arrangement. Each node holds a pair (icon, label). The icon conveys the semantics, i.e. it indicates what the node stands for. The label names the concrete realization. The use of these nodes along this arrangement leads to a test map.

Therefore, test maps are mind maps, but not all mind maps are test maps. That is, test maps restrict the structure and kind of participating nodes. In short, test maps become the graphical representation of a Domain Specific Language (DSL) for UAT. Coming up with a DSL implies identifying the main concerns of the UAT domain, providing a metamodel that captures the main relationships (a.k.a. abstract syntax), and finally, facilitating a graphical notation to specify the DSL expressions (a.k.a. concrete syntax) [MHS05]. This Section focuses on the expressiveness of test maps while other design considerations are left for Appendix A. Readers can follow the process of coming up with a test map in Figures 4.6, 4.8 and 4.9.

The expressiveness of test maps

For the purpose of this work, UAT sessions are framed by some information provided by developers (e.g., the list of features to test, the list of available Kickoffs) called the Scaffolding. The goal of UAT is to validate a backlog feature by defining a UATCase and checking out the result of different enactments (UATScript) using several input values. Next paragraphs delve into the distinct concerns.

Scaffolding. From a user's perspective, this refers to the sprint features to be tested and the available kickoffs that set up the UAT scenario. Broadly, the feature to test becomes a node of the map from where its different kickoffs hang up (see Figure 4.5).

UAT Cases. Test cases are defined as "sets of test inputs, execution conditions, and expected results" [IEE10]. In our approach, test cases represent a sequence of steps denoted as "UATActions", i.e. commands that mimic a user interaction with the web interface (e.g. click, data input). To ease customer understanding, UATActions are grouped into the Page where these actions happen. As for the expected results, they are defined as test oracles, that is, methods for checking whether the WAUT has behaved correctly on a particular execution [BY01]. In scripting languages, these oracles are realized as assertions, i.e. checking actions to be performed in order to validate the UAT case result (e.g. check the content of an element, check that a specific page is loaded). Traditionally, assertions are boolean expressions set by developers at a specific point in a program which will be true unless there is a bug in the application. Assertion definition commonly implies two concerns:

- **When**, which indicates the application state on which the checking should be performed. It is described in terms of the content of the application's variables, the page being rendered, etc (e.g., "when I fill out the form", "when I click this hyperlink").

- **What**, which stands for the condition to be satisfied (e.g., "when I fill out the form (when), the missing fields should be highlighted (what)", "when I click this hyperlink (when), a specific page should be loaded (what)").

UAT cases are test cases, and hence, they can be similarly described. The difference stems from both the aim (validation vs. verification) and the audience (customers vs. developers). Being a validation technique, we consider UAT to be more a confirmatory practice than an investigative practice, in the sense that the customer knows "what the good result is and is trying to find proof that the product conforms to that result" [Kan04]. This basically means that the customer acts as the test oracle. We then rephrased the When and the What as follows,

- **When** refers to the specific points in the browsing navigation where the customer might have some expectations about the UI rendering (referred to as ExpectationPlaceholders),

- **What** holds the customer's comments about whether or not the UI met those expectations (referred to as ExpectationStatements).

While ExpectationPlaceholders are set during the UATCase definition, ExpectationStatements are captured after the test execution (see next). In addition, customers can include UATComments (e.g., "The font is too small", "The logo is not well positioned on this page") to transmit ideas arisen during the UATCase creation.

UAT Scripts. These scripts are the executable artifacts generated from UATCases. Each UATScript exercises a UATCase using different input data sets to check out the behavior of the application in different scenarios. For each set of input data, the customer might have a different expectation (e.g., "if I introduce 16/02/02, the application should pass", "if I introduce, 16/42/42, the application should throw an error"). As stated in the previous section, the customer manually defines this expectation based on application screenshots that are taken during the test execution (ExpectationShots). Once checked, ExpectationStatements can be created in order to point out whether the test passed (i.e., the application worked as desired) or failed (i.e., the application did not behave as expected).

4.7 The tool: Assisting users to come up with test maps

Notation plays a key role in facilitating user involvement. But notation alone will not succeed unless appropriate assistance is put in place. This moves usability at the forefront. ISO defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [IEE10]. For our purpose, this can be rephrased as the ease with which customers can use test maps for achieving UAT. We understand "achieving UAT" as creating the final Test pages that will be shared on FitNesse. We then elaborate on the tasks that end up with a Test page. These tasks are listed in Table 4.2: check-in, roaming, test coverage, enacting, commenting and check-out. This section describes how these tasks are conducted through TestMind, a plug-in for the FreeMind mind-map editor. A video of the process is available at https://vimeo.com/157943880.

Task	Description	GUI Gesture
Check-in	Setting the context (e.g. the backlog feature to be tested, the Kickoff to be used)	Click on "check-in" button
Roaming	Wandering around the application in a free-way	Tab on Kickoff node
Test Coverage	Working out different data sets	Bottom canvas editing with UAT Case node selected
Enacting	Running the application with different data set	Tab on Enactment node
Commenting	Providing remarks about application outputs	Tab on ExpectationShot node
Check-out	Collecting UAT session insights as a Test page	Click on "check-out" button

Table 4.2: Tasks involved during UAT.

FreeMind is a popular editor for mind mapping [Fre]. Figure 4.6 displays the main screen regions for this editor: the upper canvas to edit the mind map structure; the lower canvas to extend the information related to the node being selected (in the Figure the selected node is "Calendar has evens set for you - Kickoff"); the toolbar on the left for adding icons; and the toolbar on the top for managing the mind map.

TestMind customizes FreeMind for test map specification. Screen wise, the difference with the standard FreeMind stems from the two icons on the left of the toolbar (see Figure 4.6(1)). The check-in button initializes the definition of a test map by downloading from FitNesse the features to be tested by the current user. The check-out button saves the canvas content as a Test page in the same FitNesse installation. In between, customers need to elaborate

the test map along the stages detailed in Table 4.2. On each stage different nodes are gradually added to finally obtain a complete test map (see Figure 4.5). Basically, customers' main tasks are threefold: creating the UAT Case, providing data sets, and confirming expectations after the execution of the test scripts. Moving from one stage to the other is achieved either via standard node creation in FreeMind (TAB key pressing[2]) or via selecting toolbar buttons (see Table 4.2). The rest of this section describes the details.

Check-in

It could have been possible for customers to directly access FitNesse to get informed. However, customer disorientation in accessing the wiki content and the lack of appropriate access control mechanisms put this option aside. We then explore an alternative where FitNesse is accessed from a mind mapping tool through a plug-in called TestMind.

At the time TestMind is installed, users are prompted for their credentials to access FitNesse. At check-in time, these credentials are used to connect to the FitNesse installation, and to recover UAT duties for this customer. Figure 4.6 shows the outcome for the customer Waldo. Waldo can see how his participation is required in the testing of two features: EventModification and NewEventAddition (Figure 4.6 (2)). Roughly, FreeMind's canvas depicts

[2]TestMind overrides FreeMind's default daemon for node creation. Based on the kind of node selected, the daemon invokes the corresponding task. For instance, the daemon associated with Kickoff nodes invokes Selenium IDE so as to be able to create a child UAT Case node. This is part of the TestMind plug-in functionality.

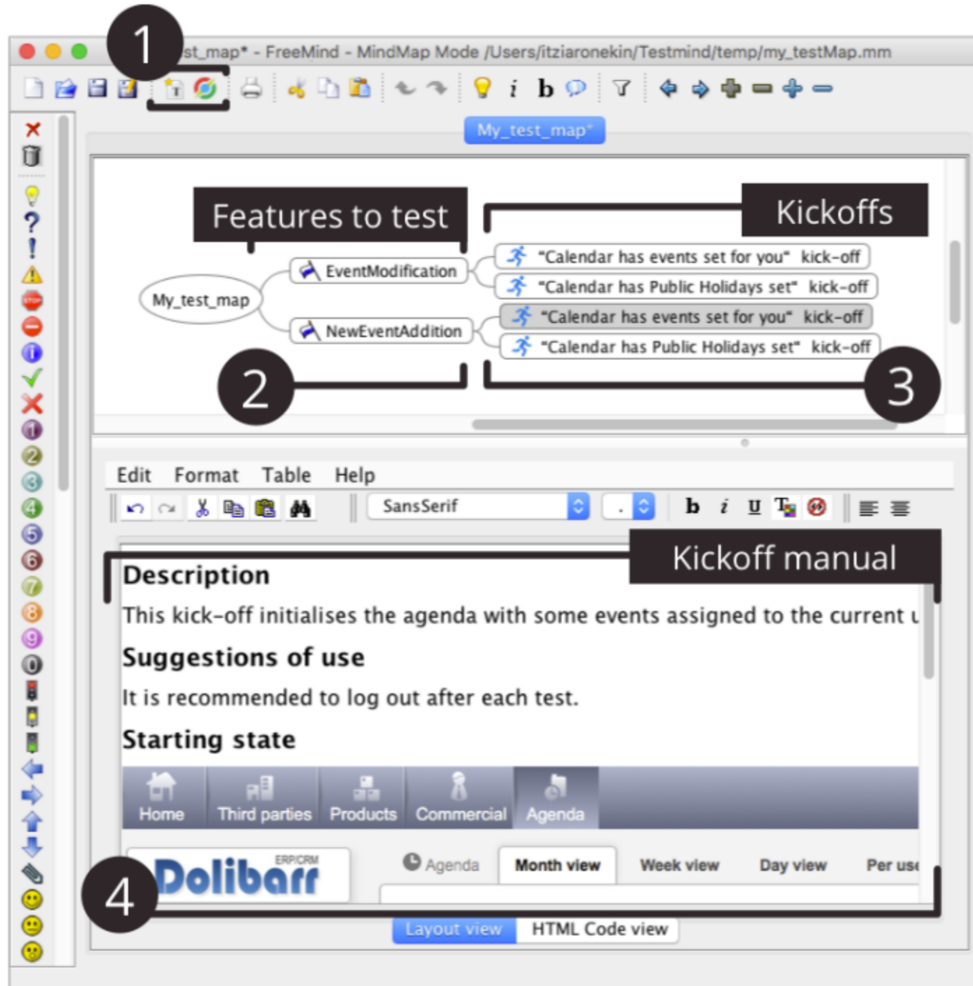


Figure 4.6: **Check-in.** Customer attention is drawn to two features: *EventModification* and *NewEventAddition* (2). For each feature, Kickoffs are displayed as sub-nodes (3). Select a *Kickoff* node for a description to pop up in the lower canvas (4).

the mind map counterpart of the FitNesse pages set on the Scaffolding. The mapping between both representations is up to TestMind.

Roaming

Now, Waldo is ready to provide a new UATCase. He selects a Kickoff scenario (e.g. "Calendar has events set for you"), and uses the FreeMind gesture to create a new child node: TAB key press. This gesture moves the control from FreeMind to Firefox. A new window is opened, the Web application is loaded, and the customer is positioned at the Kickoff point. From then on, the customer can wander freely. Using Record&Replay tools (specifically, Selenium IDE[3]), customer actions start being recorded. Three enhancements are however introduced:

- comment as you go (see Figure 4.7 (1)). Customers can introduce comments as they browse. Comments are automatically accompanied with a screenshot of the current page. No need for the customer to manually take screenshots.
- testing hints (see Figure 4.7 (2)). Notifications can be triggered on the browser bar when the right context is reached. The aim: getting customer attention to good practices on testing or to application aspects that could go unnoticed. In the example, TestMind suggests the customer to try different calendar views.

[3]Selenium IDE is an open source plug-in for Firefox that enables recording user actions and replaying them for testing purposes.

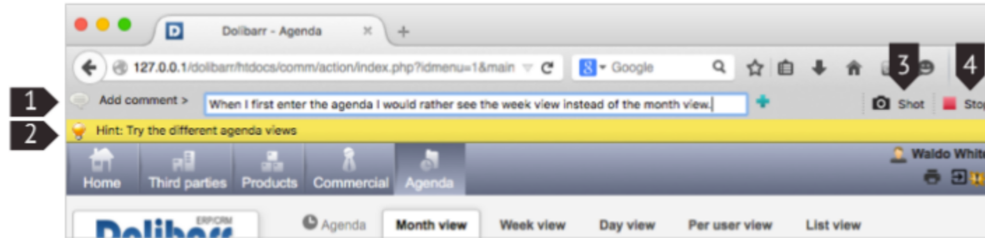


Figure 4.7: **Roaming**. Obtaining *UATCases* through Selenium IDE. Enhancements to this IDE include: (1) comment bar; (2) hint bar; (3) expectation-placeholder bookmark; (4) stop recording.

- setting ExpectationPlaceholders (see Figure 4.7 (3)). These are points in the application that might rise some expectations as for their UI outcome. They are captured through the camera button. By clicking, the current navigation step is turned into an ExpectationPlaceholder. These placeholders are later used to present the user the screenshots taken when the application is replayed with different input data sets (see Subsection 4.7).

Clicking the "Stop" button ends the recording (see Figure 4.7 (4)). This closes Firefox, and passes the control back to FreeMind whose canvas now displays a new UATCase node (see Figure 4.8 (1)). This node is automatically generated after the recorded Selenium session: clicks, comments, and expectation placeholders find their way as mind map nodes.

The mind map now represents Waldo's performed interactions with the WAUT. Next, Waldo can select a node in the mind map and see its content in the lower canvas. The upper canvas and the lower canvas are synchronized so that selecting a different node in the mind map makes the lower canvas to be accordingly refreshed. This is useful in two scenarios:

- to ease user orientation in UATCases. UATCases can comprise a large number of nodes. By including a screenshot of the page where each action was conducted the customer can better pinpoint where actions fit in the test by simply looking at the lower canvas,

- to manage test data. The input data sets are rendered as an HTML table on the lower canvas for UATCase nodes. In Figure 4.8 (2), the lower canvas summarizes the data being provided for the page at hand: Title, Start date, aphour, End_dat, p2hour and event_assigned_to. Column names are derived from web elements' available attributes, such as the id, the title or the label.

The readability of the test map very much depends on the web application. For example, if form fields are not correctly labeled, the corresponding node's text might not be sufficiently representative. This can reduce the testability of the WAUT[4].

The customer can create more UATCase nodes at will by simply pressing the TAB key on the Kickoff node of choice. Each test case is reflected as an additional child of the Kickoff node at hand.

[4]ISO defines testability as "the effort required to test software"[IEE10]. As HTML labels become identifiers for nodes in test maps, the testability of the application using TestMind greatly depends on whether HTML elements are meaningfully labelled or not. Hence, customers' understanding very much depends on developers creating an accessible application, where labels and ids are not randomly defined, but based on the HTML element's semantics. In general, W3C WCAG 2.0 recommendations should be followed [W3C].

Test Coverage

Data sets are traditionally held in Excel tables or CSV files. Next, testing programs map this data to the appropriate test function parameters. By contrast, TestMind resorts to test maps as a situated way to introduce additional data at the point this data is requested. Data is requested by input actions. Action nodes hang from Page nodes which, in turn, hang from UATCase nodes. Hence, UATCase nodes aggregate the data from their underlying Page nodes which, in turn, gather the data from their Action children. This tree-like structure allows to see the data set at different levels. Users can have a whole view of the data set by selecting a UATCase node (see Figure 4.8 (2)). Alternatively, if a Page node is selected, the bottom canvas will limit the display to the data used in this page.

As an example, consider a UATCase which visits pages P1, P2 and P3 which hold entry forms F1, F2 and F3, respectively. The customer can provide new data sets for forms F1, F2 and F3, by placing himself into the UATCase node. Alternatively, he might only focus on F1 by selecting the corresponding Page node (P1), and provide new data combinations just for this specific form. In this case, TestMind completes F2 and F3 with the default values (i.e. those obtained during recording). Figure 4.8 displays how the UATCase previously recorded is supplemented with additional data sets. In this way, customers can easily add new data by simply creating new rows on this table[5]. The expectations are for this situated data provision (i.e. providing test data at

[5]Consistency is maintained among the different aggregation views so that data inserted in a Page node propagates both upwards (to its UATCase node) and backwards (to its Action node).

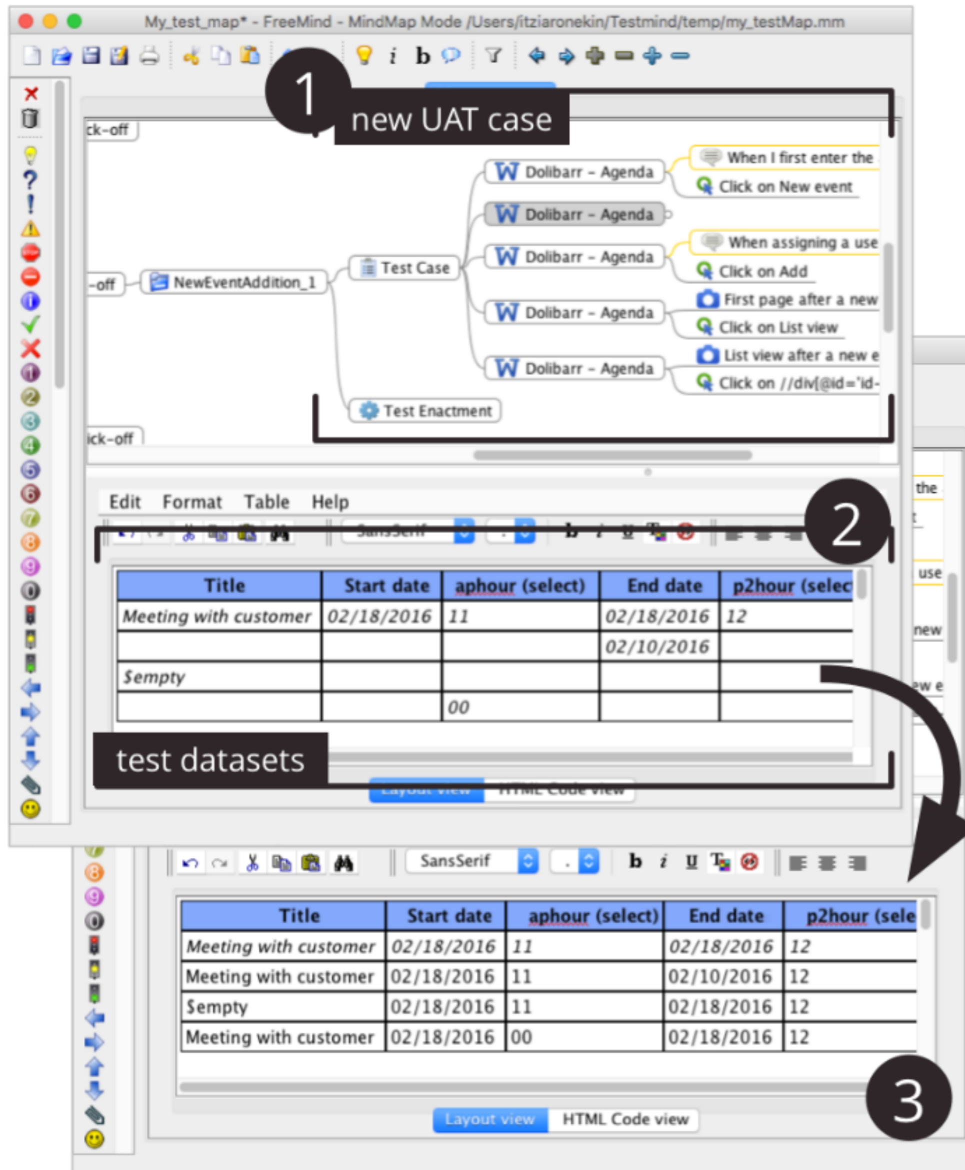


Figure 4.8: **Test Coverage.** When a page node is selected on the test map, the lower canvas shows a table where new input data sets can be introduced. Here three scenarios are added (2). *TestMind* complements the empty cells as follows: blanks are replaced by default values (taken from the first row) while the *$empty* keyword is used to deliberately leave a blank value. The resulting table is depicted in (3).

the place where the data is going to be consumed) to facilitate customers' understanding and orientation.

Enacting

Once new input data sets are provided, the customer positions himself on the Enactment node of the corresponding UATCase and presses the TAB key. Behind the curtains, TestMind generates UATScripts, enacts so-generated scripts, and adds the outcome as map nodes for the customer to check. Figure 4.9 shows the output for the data coverage in the previous subsection: four data rows give raise to four UATScript nodes. In addition, the test map is enriched with the ExpectationShots taken at the ExpectationPlaceholders defined by the customer while browsing the WAUT.

Commenting

At the end of the enacting stage, the test map depicts a UATScript node for each input data row (e.g. Test case row 190). The customer can now go through the different ExpectationShots, inspect them, and report whether his expectation is fulfilled or not (i.e. add an ExpectationStatement node). By now, the customer might have a better understanding about the application. He can decide to go back to provide additional test data, or even try a new wandering in order to create new UATCases. Once finished, the UAT session is to be exported to FitNesse. This moves us to the check out.

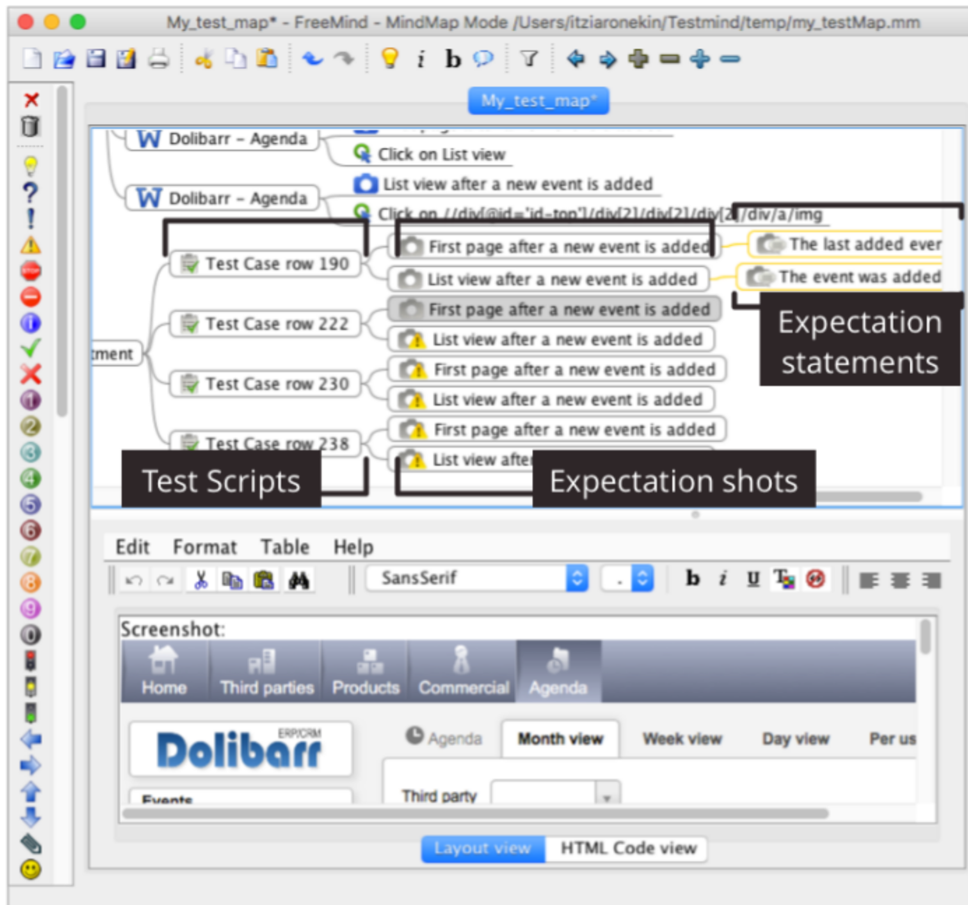


Figure 4.9: **Commenting**. Customer adds two *ExpectationStatement* nodes for *Test Case row 190.*

Check out

Test maps are stored as FitNesse pages. Each UATCase node results into a Test page. A Test page contains: the testing setting (e.g., operating system, browser version, etc.), a script table and a decision table (see Figure 4.4 (3)). In short, TestMind relieves customers from the burden of generating feedback documentation. No more need to write emails or text documents, freeing up time to explore different task flows and input data set combinations.

4.8 Evaluation

This Section looks at whether TestMind is an effective process and tool for self-paced UAT. We believe UAT is not only a technical but also a social issue. Limited customer involvement is frequently the result of poor motivation and limited membership sentiment. This mixture of technical and social issues advocates for the use of a case study to evaluate TestMind.

Case studies in software engineering are "an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified" [JP14]. A case study differs from a laboratory experiment in its focus, based on depth and context [JP14]. A laboratory experiment reduces complexity by controlling, even eliminating, factors that can interfere with the experimental results. In contrast, complexity is essential for a successful case study, as it investigates multiple factors, events, and relationships that occur in a real world setting. We are interested in measuring TestMind effectiveness for conducting UAT, together with the reactions it

produces on its environment: how the customer's colleagues react, the level of interruptions that arise while testing, the level of customer procrastination, and, finally, whether the customer's perception about testing changes after using TestMind. UAT is predominantly about the customer. Developers play an ancillary role as the definers of the UAT scaffolding. Hence, this evaluation focuses on the customer, leaving for future work the study of the developers' perspective.

A case-study design includes the methodology (subsection 4.8) and the subject selection (subsection 4.8). The rest of the subsections introduce the research questions and the results. Results were gathered using different data collection methods, namely subject observations, interviews and questionnaires [RHRR12].

Methodology

Testing sessions were conducted at the customers' places and supervised by one researcher who played both the roles of facilitator and observer. A FitNesse installation was prepared for each of the customers' web applications: sprints, features, kickoffs and hints were accordingly defined. At the beginning of the evaluation session, the customer was informed about the objectives of the test and she was interviewed about her previous testing experiences, putting a special focus on detecting the problems or limitations so far. A first questionnaire was then filled out by the subjects, so as to gather demographic information and basic data about their technological background. Then, customers were introduced to TestMind with a basic explanation of about 30'.

After this introduction, the control over TestMind was handed over to the customer. The researcher accompanied customers along the usage of TestMind,

Figure 4.10: Diverging Stacked Bar Chart for the Satisfaction Questionnaire using Likert scales. The "3" on the left means the three customers, i.e. C1, C2 and C3, *Strongly Disagree* while "3" on the right corresponds to all *Strongly Agree*.

observing their actions, solving questions and taking notes. Also, we encouraged users to express their sensations and impressions during the evaluation, based on the guide on Appendix B. Once the testing session was finished, participants were asked to fill in a questionnaire without the monitorization of the researcher. This questionnaire rates different aspects of TestMind through Likert scales (see Figure 4.10).

Notes about customer sensations were taken by the researcher, along with data about the number of tests correctly generated and their complexity (number of comments and ExpectationPlaceholders added). For result triangulation, the data gathered during the evaluation was verified against the answers given on the Likert questionnaire [RHRR12]. Each interview lasted for about one hour.

Evaluation subjects

TestMind was evaluated on three web projects at Labox, a Spain-set web development company. Next, we describe each case in terms of three variables: the customer, the organization, and the web application under test (WAUT). Subjects were already playing the role of "customer" for the application at hand. Each unit of analysis was conducted at the customer's place. Table 4.3 profiles each subject along the main stumbling blocks identified in Section 4.3.

Unit of Analysis 1

- The customer (C1): she has no programming knowledge. She is used to diagramming tools and to test form-intensive Web applications. She works for a quite large company where she is usually asked to test new application interfaces. However, she does not like testing applications and complains about her work not being valued. She usually performs the testing on her own.
- The organization: small-medium enterprise. The customer regretted her company not considering testing as a first-class duty as long as testing is not scheduled as part of her regular workload. As for testing prac-

	Lack of Time	Lack of Motivation	Lack of Knowledge
C1	- no time assigned for testing - manual feedback generation is lengthy	- feels the effort of testing is not recognized - feels like a waste of time - feels she could be bothering developers	- no programming skills - low testing skills
C2	- very busy	- difficulty in describing problems to technical people	- no programming skills - no testing skills - phobia to technical concerns
C3	- manual UAT documentation is cumbersome	- very motivated	- low programming skills - low testing skills - used to mind mapping

Table 4.3: Subject profiling along risen issues.

tices, when an error is found she has to write an incidence describing the problem and send it through a web form to the development team. The customer confesses that the procedure was so tiresome that she occasionally let go some errors only not to create a new incidence. Her workplace and the testing place are 1,5 kilometers away.

- The WAUT: a human-resource management program. It took around 6 months to develop. At the onset, new versions were launched every month, but at the end of the project they had to test a new release every week. The application is for internal use. The potential number of users is expected to be around 100 people.

Unit of Analysis 2

- The customer (C2): she has no programming skills. She finds it difficult to use new software, resorting to friends when facing "the deviltries" of her PC. However, she is the owner of the application. This makes her very conscious about the financial costs of bad testing. She is determined to make the application work since her business is going to go online. She claims to be always very busy.
- The organization: a Pharmacy. Testing feedback was based on making phone calls to developers when problems arose. The customer admitted having difficulties in explaining the problem, and that developers usually asked her to send screen captures to clarify the error. The Pharmacy is around 3 kilometers away from the testing place, i.e. the development company.
- The WAUT: an online store. It was developed in 2 months, having a new version available to test each week.

Unit of Analysis 3

- The customer (C3): she has no technical schooling, though she is familiarized with HTML and CSS. She uses FreeMind on a daily basis. She is very interested in technical concerns and she has extensive experience in testing Web projects. She finds testing very important, and she usually plays also the role of project manager. She admitted being very meticulous when checking, specially in visual details (e.g., element position, colors) and Web forms (e.g., wrong phone numbers, wrong email addresses). She spends a lot of time preparing accurate feedback documentation for developers.

ISSUES	Lack of Time	Lack of Motivation	Lack of Knowledge
MR1_async	✔	??	
MR2_R&R	??		✔
MR3_scaf			‰
MR4_maps			✔
MR5_mapping		‰	✔
MR6_feedback	✔	??	
MR7_learnability	??		??

Table 4.4: Validating meta-requirements as effective (✔), partially effective (‰) or unconclusive (??) as for the issue at hand.

- The organization: two freelance people. Their office is around 50 kilometers away from the testing place.

- The WAUT: a website offering online services to promote tourism. Her last project was developed in a hurry, lasting only 2 weeks. During these weeks she had to test new releases every day while the development team continued working against the clock.

Next subsections revise the causes of limited customer involvement in the light of the combined use of wikis and mind maps. Table 4.4 outlines the main constructs of the theory. For understanding sake, evaluation results are simultaneously presented.

RQ1. Does TestMind alleviate the lack of time?

Hypothesis

TestMind aims at easing the communication between customers and developers by asynchronously sharing the testing information via a wiki (i.e. MR1_async). On the upside, asynchronicity relieves the team from the burden of trying to coincide in time and space. Furthermore, the customer might feel more free to wander around the application without being overseen by developers. On the downside, leaving the customer on his own might eventually lead to procrastination, more to the point if testing is felt to be an ancillary and not specially rewarding endeavor. In addition, R&R facilities (MR2_R&R) and automatic wiki page generation from test maps (MR6_feedback) might help to streamline the UAT process. Due to the short duration of the sessions and to the presence of the researcher, the learnability (MR7_learnability) was not measured on this first evaluation approach.

Result

Asynchronicity was highly valued. Though distance was not such a big issue, subjects valued the fact of "playing with the application on their own without the surveillance of developers". As a subject herself put it: "it never hurts to have a first go on my own, and if any doubt, I still can go back to direct contact. This, at least, saves me two or three trips". This seems to suggest TestMind to complement rather than substitute face-to-face meetings.

An interesting fact brought about by C2 was the fear of being "an annoying person". C2 is used to calling developers frequently. The fear of interrupting

developers discourages her from notifying "some tiny issues that might not be worth the attention". Phone calls were regarded as an aggressive way of interacting, dissuading customers, and hence reducing the possibility of spotting improvement opportunities. C2 and C3 specially appreciated the use of testing hints to avoid important application spots to go unnoticed.

C1 and C3 found very useful UAT sessions being automatically exported as FitNesse pages. Traditionally, they spend much time trying to describe the encountered problems. C1 was particularly eager about this feature. The fact of testing tasks having poor recognition makes her unwilling to dedicate too much time to documenting. Indeed, C1 recognized that the effort for reporting mismatches was greatly reduced.

RQ2. Does TestMind alleviate the lack of motivation?

Hypothesis

TestMind facilitates testing to be conducted at the customer's workplace (MR1_async). This might well promote the visibility of testing, the spontaneous collaboration of the colleagues next door, and a better replication of the context in which the application will be used. By conducting testing at the customer's place, chances are that the customer's colleagues become aware of the testing effort. On the downside, this practice might be hindered by interruptions. Testing is not just running the application but also foreseeing different scenarios of usage. This requires focus to come up with different input data combinations or alternative navigation narratives. If the workplace carries a high likelihood of being interrupted, the benefits of in-place testing might be at jeopardy. In addition, reducing the burden of documentation (MR6_feedback) and rephrasing UAT

as mind mapping (MR5_mapping) can make UAT a more enjoyable activity, hence fighting back the boredom that customers might feel with traditional UAT.

Result

This case study was not particularly appropriate to check for motivation. C2 and C3 were the owner and the project manager of their projects, respectively, so they were deeply committed. Only C1 expressed dissatisfaction for conducting test tasks right from the beginning. That said, TestMind can bring about "playability". C1 enjoyed "playing" with different data sets, and seeing the outcome in seconds. This was felt as more attractive that the boring task of running the application and next, reporting the experience by typing incidents into a Word document. This leads us to tick with a cautious ‰ the impact of MR5_mapping in Table 4.4. Questions 16 to 21 of the questionnaire (see Figure 4.10) attempt to get some insights on whether customer perception about testing itself changed as a result of using TestMind. When compared with respect to their previous face-to-face experiences, customers regard the TestMind experience as more effective (number of bugs caught, feedback usefulness, product quality impact). This could well motivate their engagement. So far, however, this has not been evaluated and hence, no claim is made about TestMind tackling the lack of motivation.

RQ3. Does TestMind alleviate the lack of knowledge?

Hypothesis

TestMind aims at empowering customers to conduct UAT on their own. Without their actions being supervised, customers might feel more relaxed and spend time exploring rarer data sets or different click streams. However, freedom also implies more involvement on the customer's side. This is a critical issue since the customer role tends to be transient, i.e. it is frequently played by different employees depending on the application being developed. Hence, the time dedicated to learn TestMind should pay off for a single application.

Result

Reducing the learning barrier was a main driver during TestMind development. Strategies include: (1) the use of mind maps as the notation for capturing testing sessions (MR4_maps); (2) resorting to a popular mind map editor, FreeMind, to increase the chances of customers being already familiarized with the interface; (3) sticking to FreeMind gestures to handle test maps in order to reduce the cognitive overload (MR5_mapping); (4) providing a head-start by obtaining test cases through R&R (MR2_R&R); and (5), the use of test hints and Kickoffs as a means to gently guide customers (MR3_scaf). Although results were promising, a more extensive evaluation would be required so as to ascertain the general learnability of the proposal (MR7_learnability).

None of the subjects presented big trouble understanding test maps. The radial disposition of nodes and the use of screenshots associated with page nodes was recognized as helpful for localization purposes (MR4_maps). More-

over, subjects appreciated the test recording utility (MR2_R&R). C1 and C3 edited the mind map for data set and comment addition. By contrast, C2 understood the parameterization and test reproducibility benefits, but she did not feel comfortable doing it. Rather, she would have liked what she called "TestMind for dummies" with the functionality being limited to wander, comment and export to FitNesse. She also believes on developers to check form validations better than herself.

The three of them enjoyed the comment-addition toolbar on the browser. They liked being able to add in-place suggestions during the recording without having to swap to another program.

Letting customers introduce their own input data sets might provide important cues to developers who can later complement those sets with extra data. The notion of "expectation" proved to be an adequate way to informally capture testing assertions. Nevertheless, some subjects experimented problems in the two-stage definition of expectations: the ExpectationPlaceholder (at Roaming phase) and the ExpectationStatement (at Commenting phase). Some subjects were impelled to introduce the oracle (i.e. whether the current rendering matches expectations) at recording time, rather than waiting till replaying.

One shocking result was the poor valuation of Kickoffs (MR3_scaf). One customer strongly disagreed on Kickoff selection being a duty of the customer. Rather, she considered the application should be ready to test from the onset without forcing her to ascertain which Kickoff to choose. Alternatively, one subject expressed her willingness to participate in the design of the Kickoffs. So far, developers designed Kickoffs without any customer involvement.

Threats to validity

We follow here the recommendations of Runeson et al. as for analyzing the extent to which the results are true and not biased by the researchers' subjective point of view [RH09].

Construct validity refers to the appropriateness of inferences made on the basis of observations or measurements (often test scores), specifically whether a test measures the intended construct [RH09]. In this work, this mapping goes as follows:

1. questions 1 - 3 provide feedback as for asynchronicity (MR1_async)
2. question 4 collects information as for automating documentation generation (MR6_feedback)
3. questions 5 - 10 provide evidences on lowering the UAT effort (MR2_R&R, MR4_maps, MR5_mapping)
4. questions 11 - 15 supply insights for in-place assistance (MR3_scaf)

Internal validity is a matter of concern when causal relationships are examined. It depends upon whether the observed change in a dependent variable is, indeed, caused by a corresponding change in an independent variable and not by other factors. Here, the ability to conduct UAT at the customer's location and at their most suitable time, both have a high internal validity with regard to lessen the lack of time. So does lowering the learning bar and UAT assistance with regard to lessen the lack of knowledge.

However, it should be noted that experiments were conducted in a single session, scheduled in advance. Customers were not free to allocate the UAT

at their most convenient time. Thus, procrastination risks were not assessed. Subjects' timely response was, probably, facilitated by the excitement of the novelty together with the presence of the researcher. In a real situation both boosters will not be there. Nevertheless, leaving customers largely on their own with the only pressure of a deadline might be risky, more to the point if UAT is regarded as an ancillary activity. Fortunately, developers might track customer testing progress through the wiki. If there is a delay, developers can resort to email or phone calls to remind customers of their testing duty.

Another issue for self-paced UAT is focus loss. UAT sessions should focus on the sprint features to be tested. However, our experience is that customers are very often too tempted to wander around the application, leading to large UAT sessions that hinders developers from grasping the real intention of the customer. TestMind does not prevent wandering around but might warn about it. A proposed improvement is to count the number of Action nodes in the UATCase and, if a certain threshold is exceeded, show a warning message. Both issues, i.e. procrastination and focus loss, are left for future evaluations.

Evaluation was performed by only one researcher. To minimize the possible bias, customer sensations were gathered through two different means: the researcher's notes and a Likert questionnaire. The researcher's notes were checked against the scale ratings obtained in the questionnaire so as to verify the customer impressions.

External validity is concerned with the extent to which the problem and the findings can be generalized. A common criticism of case studies is that their results only apply to the instance being studied. We tried to reduce this threat by selecting diverse customer profiles (different technical background, occupation, web application sizes). Nevertheless, the three evaluated cases involve a single customer. We have not evaluated the case where the same

feature is tested by several customers. However, we do not envision much difference in the results unless customers gather together to conduct the test. Nevertheless, a larger group of customers is still required to sustain the results of this work.

Reliability is concerned with the extent to which the data and the analysis are dependent on the specific researchers. Hypothetically, if another researcher conducted the same study at another time, the result should be the same. We reduce this threat by describing the subjects of the studies in terms of customers, organizations and WAUT application, as well as presenting the questionnaire. In this way, other researchers can recreate this experiment in the same setting and check the results.

4.9 Architecture

Figure 4.11 outlines the TestMind architecture. Broadly, TestMind follows a model-view-controller architecture:

- the TestMap model (4). This model captures the UAT session data along the meta-model depicted in Appendix A.2.
- the view (1). Test maps are visually rendered as mind maps in Freemind.
- the controller. It achieves a double aim. First, it keeps the FreeMind canvas in sync with the underlying TestMap model. This is achieved through the FreeMind manager (2), which hooks the TestMind plug-in to FreeMind. In addition, it listens to child creation interactions (e.g., TAB key pressing) and customizes these interactions to account for the

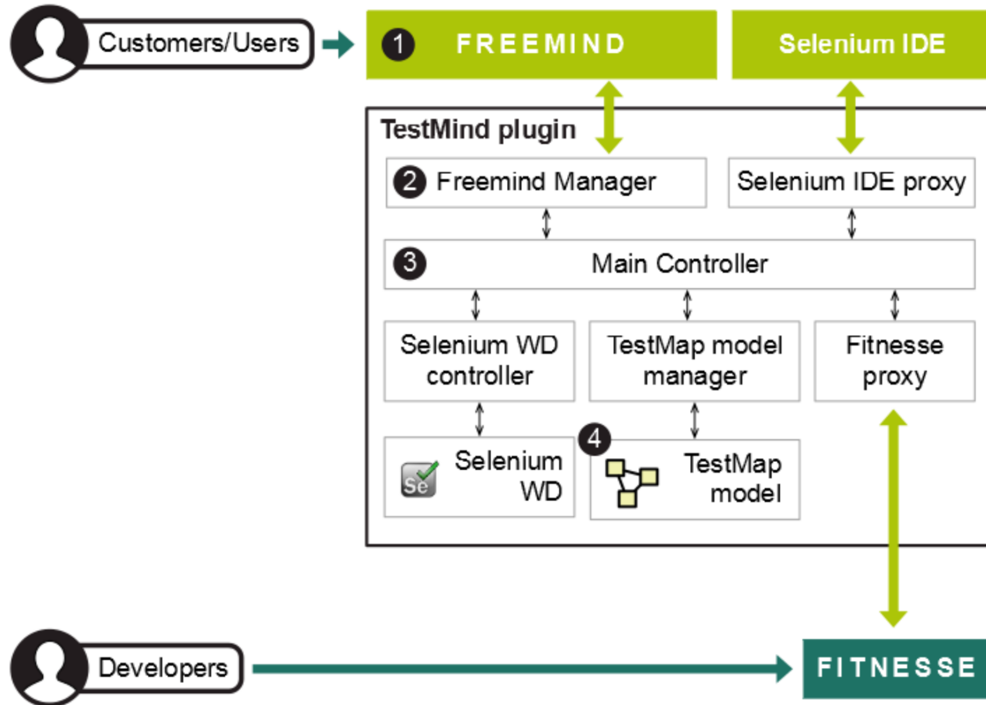


Figure 4.11: *TestMind* architecture

test map semantics. This customization might trigger interactions with FitNesse or Selenium that happen through the Main Controller (3).

Installation guidelines for TestMind can be found in Appendix C.

4.10 Related Work

Different approaches and techniques have been devised to foster customer participation in testing. This section reviews some of these works w.r.t. TestMind.

Domain Specific Languages (DSLs). Abstraction is a common mechanism to reduce the gap between technical and non-technical people. The use of DSLs to define test cases improves their readability, making them understandable to different stakeholders, from developers to business experts. Toolkits such as Cucumber [Cuc] or FIT/Fitnesse [MC05, Fit] resort to DSLs for describing test stories which can later be converted into executable test cases [RPT+08]. Within FIT, the Telling TestStories tool integrates test generation techniques (such as model-based testing) for the test and test data specification to be described in a tabular form [FZFB10]. Häser et al. [HFB16] propose the integration of business domain concepts into testing DSLs. They conclude that including these concepts permit a faster creation of test case specification. TestMind aligns with this research as for the use of DSLs, specifically, the use of mind maps (the DSL's graphical concrete syntax) for UAT.

Live demos. They allow developers to show the application to customers, collecting their opinions and suggestions [SW07]. This approach might fail to cover all scenarios since demos themselves tend to be led by developers, hence representing their mindset on what the application is about, rather than the customers'. Hence, live demos are recommended to be jointly used with real releases where customers are left on their own "playing with the application", i.e. manual testing [SW07, HD09].

Manual testing. This is a widely accepted practice for UAT [LAF13]. Here, users wander through the application coming with examples and action sequences not necessarily foreseen by developers. Traditionally, feedback is given via emails or phone calls. In this line, JIRA Capture [Atl] is a tool that permits users to annotate application screenshots while navigating. Then, JIRA incidences are automatically created from them. In most cases, the qual-

ity of the feedback greatly depends on customer's knowledge and motivation. In some cases, it may be difficult for developers to reproduce the UAT scenario in order to clarify the root of the problem [LAF13]. This sustains the vision of UAT as a collaborative endeavor between developers and consumers, and underlines the importance of the testing scaffold. This is aligned with Shaye's vision of customers developing and executing automated tests by themselves without resorting to the technical staff [Sha08]. TestMind shares this mindset.

Record&Replay tools. They target non-technical customers to generate tests out of browsing sessions. Unfortunately, the lack of parameterization and modularity mechanisms make so-generated test scripts difficult to maintain. In these scripts, input data is hardcoded while the test structure is strongly coupled with the web interface. The latter makes these scripts fragile upon changes to the application GUI [LCRT13]. TestMind departs from R&R tools by providing a context. In TestMind, Selenium IDE [Selb] is not launched in a vacuum but as part of a larger UAT session that aims at obtaining a test map. A test map can include different Selenium scripts that are complemented with input data sets, screenshots, comments and expectations, all arranged along a mind map structure, hence facilitating location and experimentation.

Test automation. It is "the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes" [HK07]. TestMind is a software that helps the capture of Selenium scripts and their enactment, and from this perspective, it can be framed within the test automation effort. However, the aim of TestMind is not so much about automation but about asynchronous collaboration. Tests need to be enactable not so much from being later automated but to be replayed by developers and hence, to get a better insight of customers' UAT session. That said, test maps could be used as a sort of regression tests for next sprints. In a survey conducted in [GM16], regression testing

was the most frequently mentioned factor for test automation decisions. This makes sense in an Agile setting where previously developed backlog features need to be verified to check whether they still perform correctly on the next sprint's release. Unfortunately, small changes on the application UI can make recorded test scripts to fall apart. This backward-compatibility requirement in GUI evolution might be too stringent in some scenarios, and might increase the maintenance cost of keeping test maps up and running.

4.11 Conclusions

Agile methodologies put stringent demands on UAT, if only for the frequency at which it needs to be conducted. In-person meetings might need to be complemented with asynchronous ways for customers and developers to collaborate during UAT. We coin the term "self-paced UAT" to denote asynchronous sessions where customers perform UAT on their own using a scaffolding previously set by developers. Test scaffolding helps customers to effectively perform UAT (keeping the focus through testing hints) and efficiently (automatically setting customers in ready-to-go scenarios through kickoffs). In addition, mind maps are proposed to give structure and context to UAT sessions. In this way, Record&Replay is not launched in a vacuum but framed within a test map.

First evaluations are promising. Subjects specially valued the chance of conducting UAT at their own pace. No travel, no agenda sync problems. They all prized the opportunity to add comments during test recording (the best rated feature in Likert scales) and to report feedback with a single click. Test parameterization was specially appreciated by the subjects who usually checked form-intensive websites.

5

Conclusions

5.1 Overview

This work tackles two issues on the way to leverage end-users to take advantage of web automation. Chapter 3 looks at the creation of automation scripts for the filling of web forms from databases, specifically, the filling of form-intensive websites. The proposal consists on an abstraction of the form filling scripts to models. This abstraction aims at end-users being able to create complex form filling scripts without writing a single line of code. On the other hand, Chapter 4 focuses on User Acceptance Testing (UAT) for web applications. This other proposal entails the use of mind mapping, where tests are recorded and presented to the user using this popular notation. This chapter reviews main results as well as suggests future developments.

5.2 What is being done

This section reviews main highlights along the different requirements introduced in previous chapters.

Automation of web form filling

This research effort aims at improving the creation of externally-fed autofilling scripts by proposing the following:

RQ1 *Provide a model-based solution with a web augmentation interface.*

Abstracting the data sources as models permits to then transform this information and present it to the user in a visual way. We proposed a web augmentation interface that presents the database structure as a set of drop-down lists. These lists are added to the web application interface so they can be edited whenever a form field is used. Internally, we propose to represent the database structure as a model. Form filling scripts are also abstracted into models, so we are able to parameterize them and link them to the database model. This linking permits to establish a relationship between each form field and a database registry.

RQ2 *Offer the solution as an extension to be added to the user's web browser.*

In order to ease the task for end-users, the proposed tool is presented as a browser extension. These extensions work on top of well–known web browsers (e.g., Google Chrome, Mozilla Firefox). Thus, the user is not required to leave the browser window in any moment: every task (script creation, script execution) is performed through sidebar buttons.

Automation of web testing

This research effort aims at improving the problem of self-paced UAT by proposing the following solutions to the metarequirements:

RQ1 *Account for asynchronous collaboration through wikis*

Wikis are known to be beneficial in the workplace for groups requiring a collaborative medium [LDP+12]. In this work we proposed the integration of our

tool with Fitnesse [Fit], a wiki platform created for the storage and execution of test suites. *FitNesse* permits to keep test cases as wiki pages (called *test pages*), and enact them by simply clicking a *"Test"* button at the top of the page. By using this tool all the development team, along with the clients, can access to the created tests and replay them.

RQ2 *Offer customers a head-start using Record&Replay facilities*

Record&Replay (R&R) tools permit to automatically generate browsing scripts as the customer freely navigates throughout the application. For instance, Selenium IDE offers this feature as a Firefox plug-in [Selb]. In this work, we reused the Selenium IDE extension and its script recorder, in order to empower any user to easily generate his own tests, even if he lacks of programming skills.

RQ3 *Non-intrusively guide customers during "exploratory" UAT through "testing scaffolding"*

In order the user to be guided through the testing process we proposed the use of *Kickoffs* and *Hints*. Kickoffs are used to set the web application at a state ready to be validated. This might imply, for example, running a script in order to set the application at some specific point, thus saving the customer the intermediate navigation required to reach the functionality to be tested. During the test recording, the user is helped through Hints: short messages that aim at avoiding some aspects of the web application to go unnoticed. These messages might entail the user to pay attention on some special aspect of the application, or they also might provide ideas about the possible interactions on the current web page.

RQ4 *Capture UAT sessions as mind maps*

In this work we investigated the use of a DSL for UAT based on mind maps: structures where ideas are radially disposed around a root node. Rationales for proposing the use of mind maps are twofold: familiarity and structuredness. As for the former, mind maps are a common notation among organizations for brainstorming and idea forming [Buz14]. Representing user tests as mind maps make them easier to understand and to manipulate than raw script code or tables.

RQ5 *Rephrase UAT as mind mapping*

Since tests are going to be represented as mind maps, UAT processes were rephrased in terms of mind mapping. In other words, common gestures in drawing a mind map should now be overloaded with UAT semantics (e.g. the "adding a child node" interaction might become a "creating a new test" action).

RQ6 *Customers should be able to provide appropriate feedback documentation for developers to reproduce UAT sessions*

The aforementioned mind map structure helps users to obtain test scripts. Through a new option we added to the mind map's root node, all the information gathered by the user (recorded tests, comments, screenshots) can be uploaded to the wiki with a single click. The aim is for developers to be able to replay the scenario conducted by the customer, and to reproduce the setting where potential mismatches might arise.

RQ7 *The solution should be intuitive enough for customers to be able to grasp it rapidly*

To conclude, we advocate for an asynchronous collaborative approach to UAT. Through the combined use of R&R and mind mapping tools, we aimed at obtaining a complete solution to help users to intuitively perform their acceptance testing tasks.

5.3 Publications

Part of the work presented in this Thesis has been already presented and discussed in distinct peer-reviewed forums. The publications that endorse this Thesis are the following:

- Otaduy, I., & Diaz, O. (2017). User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. *Journal of Systems and Software.*[OD17]. Related to Chapter 3 of this Thesis.
- Diaz, O., Otaduy, I., & Puente, G. (2013). User-Driven Automation of Web Form Filling. *In F. Daniel, P. Dolog, & Q. Li (Eds.), Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings (pp. 171–185).* Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-39200-9_16 [DOP13]. Related to Chapter 4 of this Thesis.

5.4 Research stage

The realization of a Thesis is not only the result of a single person's hard work. During this journey, there have been many opportunities to know other profes-

sionals' research, and to collaborate and learn from them. The author of this Thesis had the change to stay for three months at the Hochschule of Neu-Ulm (HNU), in Germany, under the supervision of Professor Philipp Brune. Apart from the enriching experience that supposed working at a foreign country, the author had the opportunity to visit for several days two companies that were interested on the work that was being carried out: eXXcellent Solutions (*Ulm*, Germany) and NTT Data (*Ettlingen*, Germany). These companies introduced the author into Agile methodologies, which they used both to develop and to manage their projects. They also shared their concerns and most usual problems, which where mostly related to the communication between them and their clients. Thanks to this experience, we developed the work presented in Chapter 4.

5.5 What is left

This subsection introduces some limitations that open the door to further research for both of the issues being risen by this Thesis.

Automation of web form filling

- **Extend the evaluation in a real setting.** In this work, a first evaluation was performed, aimed at comparing the time cost of filling a form manually versus the time cost of filling it using the proposed tool: Webfeeder. Although the results sound promising, presenting a high saving of time from the fourth form filling onwards, a more extensive evaluation should be performed. On one hand, a larger set of different forms should be checked, in order to detect the limitations of this first

version of the tool (e.g., different entry fields, different source data). The results of these evaluations might bring interesting information about the future lines of work to be followed in order to improve the tool.

- **Evaluate the safe mode.** As stated before, a first approach to evaluate the performance of Webfeeder was carried out. However, the *safe mode* option was not included in this evaluation. This option permits the user to execute the form filling one page at a time, checking for each page whether the form fields included into the filling script coincide with the actual fields on the page. If they do not coincide, the system permits the user to delete the fields that were not found and to add new ones into the form filling script. This execution mode slows down the filling process, and it is aimed at being executed occasionally, depending on the frequency of changes on the web application. It would be interesting to perform a qualitative evaluation with real end-users, in order to check the safe mode interface and whether the increase of time on the form filling is perceived as countervailing or not.

- **Adaptation to different data sources, such as spreadsheets or pdfs.** This work presents a tool that permits the user to fill in forms from information extracted from databases. However, in some cases the information might be held into different data sources, such as spreadsheets or pdfs. The model-based structure of Webfeeder permits to be easily adapted to new data sources, as long as these data sources can be abstracted in terms of models (the transformation from the data source to the data model needs to be programmed).

Automation of web testing

- **Reuse *test maps* as regression tests for future *sprints*.** In this work, we addressed the phase of UAT, where clients check if the functionalities developed on the current *sprint* fulfill their needs. Once the sprint finishes, the created test maps are discarded, and a new blank test suite is created for the next sprint. It would be interesting to investigate the extend to which the created *test maps* can be reused as regression tests on future *sprints*. Regression tests are used to ensure that the previously developed and tested features still work on subsequent sprints. In this setting, it would be worth investigating how test maps could be adapted to work as regression tests, creating a regression suite that would increase on each *sprint*. Here, we will have to deal with the locator problem, as web application elements might change their attributes between different versions of the product, thus making the tests to break apart.

- **Check the suitability of the solution on different Agile processes**. We presented a tool, called Testmind, that helps to perform UAT in the setting of a Scrum process. However, it will be interesting to investigate to which extent this solution fits into different Agile processes, such as XP, Lean or Kanban.

- **Perform additional evaluation.** Three use cases were proposed to check the usefulness of the proposed tool: *Testmind*. Those use cases, performed on a real setting with three real customers, served to check the viability of the tool, and the adequateness of the use of mind maps as a visual representation of a test suite for end-users. However, a more extensive evaluation would be worth performing, involving a larger number of customers and different web applications. In addition, it would be interesting to delve into how procrastination can jeopardize the agility of UAT, and the impact of self-paced UAT on customers' motivation.

5.6 Conclusions

In this Thesis we followed a Design Science Research (DSR) approach to identify and solve problems that raise when trying to empower end-users to take advantage of web automation techniques. Specifically, we aimed at empowering end-users to automate **web form filling** and **web testing**.

Following DSR practices we provided, for each work: (1) the context and key definitions, (2) the root-cause analysis of the problem to be solved, (3) the design problem formulated along Wieringa's template, (4) the set of meta-requirements to be addressed, and (5) the contributions. The content of this Thesis has already been published in distinct peer-reviewed publications. Furthermore, limitations of the presented work and new areas for future research were listed in this chapter.

The development of this Thesis gave the author the opportunity to get to know new technologies and tools that look for helping people at their workplaces. Although the development of the solutions presented in this work is still at a prototype level, the author was able to apply part of the technology and the acquired knowledge at her actual workplace. These contributions were embraced by her colleagues, and showed the real benefits that even the smallest application of these new ideas can bring to a company. Not only technical skills were gained during these years, but also a growing interest on learning new things and assessing the hard work performed by the research community. A Thesis is about being curious and willing to discover innovative ways that can improve, or help any other researcher to improve a specific situation. This discovering is not easy, and sometimes the wrong path must be traversed only to find out that it was not the solution to the problem. Accepting that failure is an option, and looking at it as an opportunity to improve and not surrender,

is one of the most valuable lessons a PhD acquires during the development of his Thesis. A priceless lesson that can be applied not only to the research work, but to life itself.

Bibliography

[AC11] Silviu Andrica and George Candea. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 403–410, jun 2011.

[AD17] Inigo Aldalur and Oscar Diaz. Addressing Web Locator Fragility: A Case for Browser Extensions. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '17, pages 45–50, New York, NY, USA, 2017. ACM.

[agi] Manifesto for Agile Software Development. http://www.agilemanifesto.org. Last accessed March 2020.

[AGLH10] Samur Araujo, Qi Gao, Erwin Leonardi, and Geert-jan Houben. Carbon : Domain-Independent Automatic Web Form. In *Proceedings of the 10th international conference on Web engineering*, pages 292–306, Vienna, Austria, 2010. Springer-Verlag.

[All] Agile Alliance. What is Acceptance Testing? http://guide.agilealliance.org/guide/acceptance.html. Last accessed March 2020.

[Atl] Atlassian. Capture for JIRA. https://es.atlassian.com/software/jira/capture. Last accessed March 2020.

[AWDP17] Iñigo Aldalur, Marco Winckler, Oscar Díaz, and Philippe Palanque. *Web Augmentation as a Promising Technology for End User Development*, pages 433–459. Springer International Publishing, Cham, 2017.

[BB06] Tony Buzan and Barry Buzan. *The mind map book*. Pearson Education, 2006.

[BCBG16] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. Ringer: Web Automation by Demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 748–764, New York, NY, USA, 2016. ACM.

[BCFP19] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software*, 149:101–137, 2019.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[Bec00] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[Bou99] Niels Olof Bouvin. Unifying strategies for Web augmentation. In *Proceedings of the tenth ACM Conference on Hypertext and hypermedia: returning to our diverse roots: returning to our diverse roots*, pages 91–100, 1999.

[Buz14] Tony Buzan. *Mind maps for business : using the ultimate thinking tool to revolutionise how you work.* Pearson, Harlow, 2014.

[BWR+05] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C Miller. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, UIST '05, pages 163–172, New York, NY, USA, 2005. ACM.

[BY01] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2:9, 2001.

[CBBG15] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. Browser Record and Replay As a Building Block for End-User Web Automation Tools. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 179–182, New York, NY, USA, 2015. ACM.

[CdL12] Eliane Figueiredo Collins and Vicente Ferreira de Lucena. Software Test Automation practices in agile development environment: An industry experience report. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 57–63, 2012.

[CDLN10] Allen Cypher, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols. *No code required: giving users tools to transform the web.* Morgan Kaufmann, 2010.

[CH93] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration.* MIT press, 1993.

[CR08] Lan Cao and Balasubramaniam Ramesh. Agile Requirements Engineering Practices: An Empirical Study. *IEEE Software*, 25(1):60–67, jan 2008.

[Cri] Lisa Crispin. FitNesse: A Tester's Perspective. http://www.methodsandtools.com/tools/tools.php?fitnesse. Last accessed March 2020.

[Cuc] Cucumber. https://cucumber.io/. Last accessed March 2020.

[Cyp12] Allen Cypher. Automating Data Entry for End Users. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 23–30, 2012.

[DBV06] Marcos Didonet Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe*, volume 2006. Citeseer, 2006.

[DDF+03] Alan Dix, Alan John Dix, Janet Finlay, Gregory D Abowd, and Russell Beale. *Human-computer interaction.* Pearson Education, 2003.

[DDT13] Oscar Díaz, Josune De Sosa, and Salvador Trujillo. Activity fragmentation in the web: empowering users to support their own webflows. In *Proceedings of the 24th ACM Conference on Hypertext and Social Media*, pages 69–78, 2013.

[Día12] Oscar Díaz. Understanding web augmentation. In *International Conference on Web Engineering*, pages 79–80. Springer, 2012.

[DOP13] Oscar Diaz, Itziar Otaduy, and Gorka Puente. User-Driven Automation of Web Form Filling. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings*, pages 171–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DPCG13] Oscar Díaz, Gorka Puente, Javier Luis Cánovas Izquierdo, and Jesús García Molina. Harvesting Models from Web 2.0 Databases. *Software & Systems Modeling*, 12(1):15–34, 2013.

[FdS08] David Ferreira and Alberto Rodrigues da Silva. Wiki supported collaborative requirements engineering. In *Wikis4SE 2008 Workshop, Porto, Protugal*, 2008.

[FGG+12] Sergio Firmenich, Vincent Gaits, Silvia Gordillo, Gustavo Rossi, and Marco Winckler. Supporting Users Tasks with Personal Information Management and Web Forms Augmentation. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering*, volume 7387 of *Lecture Notes in Computer Science*, pages 268–282. Springer Berlin Heidelberg, 2012.

[fir] Firefox Autofill Forms plugin. https://addons.mozilla.org/es/firefox/addon/autofill-forms-webextension/. Last accessed March 2020.

[Fit] FitNesse. http://www.fitnesse.org. Last accessed March 2020.

[Fre] Freemind. http://freemind.sourceforge.net/wiki/index.php/Main_Page. Last accessed March 2020.

[FZFB10] Michael Felderer, Philipp Zech, Frank Fiedler, and Ruth Breu. A Tool-Based Methodology for System Testing of Service-Oriented Systems. In *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*, pages 108–113, aug 2010.

[GHS12] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet Data Manipulation Using Examples. *Commun. ACM*, 55(8):97–105, aug 2012.

[GJ03] Robert Godwin-Jones. Blogs and wikis: Environments for online collaboration. *Language learning & technology*, 7(2):12–16, 2003.

[GM16] Vahid Garousi and Mika V Mäntylä. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, 76:92–117, 2016.

[goo] Google toolbar. https://www.google.com/toolbar/ie/index.html. Last accessed March 2020.

[Hal84] Daniel Conrad Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[HD09] Orit Hazzan and Yael Dubinsky. *Agile software engineering*. Springer Science & Business Media, 2009.

[HFB16] Florian Häser, Michael Felderer, and Ruth Breu. Is business domain language support beneficial for creating test case specifications: A controlled experiment. *Information and Software Technology*, 79:52–62, 2016.

[HG11] Matthias Heinrich and Martin Gaedke. WebSoDa: A Tailored Data Binding Framework for Web Programmers Leveraging the WebSocket Protocol and HTML5 Microdata. In Sören Auer, Oscar Díaz, and George Papadopoulos, editors, *Web Engineering*, volume 6757 of *Lecture Notes in Computer Science*, pages 387–390. Springer Berlin / Heidelberg, 2011.

[HH08] Børge Haugset and Geir Kjetil Hanssen. Automated Acceptance Testing: A Literature Review and an Industrial Case Study. In *Agile, 2008. AGILE '08. Conference*, pages 27–38, 2008.

[HH09] Geir Kjetil Hanssen and Børge Haugset. Automated Acceptance Testing Using Fit. In *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, pages 1–8, 2009.

[HH11] Børge Haugset and Geir Kjetil Hanssen. The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse. In *Agile Conference (AGILE), 2011*, pages 97–106, 2011.

[HK07] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management.* Wiley-IEEE Computer Society Press, 2007.

[HLBCF16] Irit Hadar, Meira Levy, Yochai Ben-Chaim, and Eitan Farchi. *Using Wiki as a Collaboration Platform for Software Requirements and Design*, pages 529–536. Springer International Publishing, Cham, 2016.

[HNM11] Rashina Hoda, James Noble, and Stuart Marshall. The impact of inadequate customer collaboration on self-organizing Agile teams. *Information and Software Technology*, 53(5):521–534, 2011.

[IEE10] IEEE. Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, 2010.

[II15] Tomayess Issa and Pedro Isaias. *User Participation in the System Development Process*. Springer London, London, 2015.

[IMa] IMacros. Browser automation, data extraction and web testing. https://imacros.net/.

[Inv18] Investopedia. Acceptance Testing. http://www.investopedia.com/terms/a/acceptance-testing.asp, 2018. Last accessed March 2020.

[Joh09] Karen N Johnson. Tips for better user acceptance testing. http://www.informit.com/articles/article.aspx?p=1431821&seqNum=5, 2009. Last accessed March 2020.

[JP14] Paul Johannesson and Erik Perjons. *An introduction to design science*. Springer, 2014.

[KAB+11] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.*, 43(3):21:1—-21:44, apr 2011.

[Kan04] Cem Kaner. The ongoing revolution in software testing. In *Software Test & Performance Conference*, volume 8, 2004.

[Kha06] Rohit Khare. Microformats: the next (small) thing on the semantic Web? *IEEE Internet Computing*, 10(1):68–75, jan 2006.

[Kup] Jonathan Kupersmith. Putting the User Back in User Acceptance Testing. http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/Art the-User-Back-in-User-Acceptance-Testing.aspx. Last accessed March 2020.

[LAF13] Grischa Liebel, Emil Alegroth, and Robert Feldt. State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 17–24, sep 2013.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 272–281. IEEE, 2013.

[LDP+12] Ioanna Lykourentzou, Foteini Dagka, Katerina Papadaki, Giorgos Lepouras, and Costas Vassilakis. Wikis in enterprise settings: a survey. *Enterprise Information Systems*, 6(1):1–53, 2012.

[LG14] Vu Le and Sumit Gulwani. FlashExtract: A Framework for Data Extraction by Examples. *SIGPLAN Not.*, 49(6):542–553, jun 2014.

[LHML08] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[Lie01] Henry Lieberman. *Your wish is my command: Programming by example.* Morgan Kaufmann, 2001.

[Lin16] Ben Linders. Requirements with the Agile process management. http://searchsoftwarequality.techtarget.com/tip/Requirements-with-the-Agile-process-management, 2016. Last accessed March 2020.

[Lou06] Panagiotis Louridas. Using wikis in software development. *IEEE Software*, 23(2):88–91, mar 2006.

[LPKW06] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-User Development: An Emerging Paradigm*, pages 1–8. Springer Netherlands, Dordrecht, 2006.

[MBN10] Angela Martin, Robert Biddle, and James Noble. *Agile Software Development: Current Research and Future Directions*, chapter An Ideal C, pages 111–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[MC05] Rick Mugridge and Ward Cunningham. *Fit for developing software: framework for integrated tests*. Pearson Education, 2005.

[Mey14] Bertrand Meyer. *Agile!* Springer International Publishing, Cham, 2014.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*, 37(4):316–344, dec 2005.

[mof] MofScript. https://marketplace.eclipse.org/content/mofscript-model-transformation-tool. Last accessed March 2020.

[OD17] Itziar Otaduy and Oscar Diaz. User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. *Journal of Systems and Software*, 2017.

[Pay00] Gordon W Paynter. Automating iterative tasks with programming by demonstration. 2000.

[PT15] Pallavi Pandit and Swati Tahiliani. AgileUAT: A framework for user acceptance testing based on user stories and acceptance criteria. *International Journal of Computer Applications*, 120(10), 2015.

[Pul06] Michael Pulcio. How not to do agile testing. In *Agile Conference, 2006*, pages 7 pp.—-314, 2006.

[RBH07] Jörg Rech, Christian Bogner, and Volker Haas. Using Wikis to Tackle Reuse in Software Projects. *IEEE Software*, 24(6):99–104, nov 2007.

[RCFH11] Rasmus Rasmussen, Anders S Christensen, Tobias Fjeldsted, and Morten Hertzum. Selecting users for participation in {IT} projects: Trading a representative sample for advocates and champions? *Interacting with Computers*, 23(2):176–187, 2011.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, apr 2009.

[RHRR12] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[RPT+08] Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, Mariano Ceccato, and Corrado Aaron Visaggio. Are fit tables really talking? In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 361–370, may 2008.

[saf] Safari - Autofill. http://www.apple.com/safari/. Last accessed March 2020.

[SB02] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[SCE+08] Christopher Scaffidi, Allen Cypher, Sebastian Elbaum, Andhy Koesnandar, and Brad Myers. Using Scenario-based Requirements to Direct Research on Web Macro Tools. *Journal of Visual Languages & Computing*, 19(4):485–498, 2008.

[See16] Swati Seela. What is User Acceptance Testing (UAT): a complete guide. https://www.softwaretestinghelp.com/what-is-user-acceptance-testing-uat/, 2016. Last accessed March 2020.

[sela] Selenium. https://www.selenium.dev/. Last accessed March 2020.

[Selb] Selenium Projects. Selenium IDE. Last accessed March 2020.

[SH10] Thomas Stober and Uwe Hansmann. *Best Practices for Large Software Development Projects*. Springer, 2010.

[Sha08] Susan D Shaye. Transitioning a Team to Agile Test Methods. In *AGILE*, pages 470–477, 2008.

[SW07] James Shore and Shane Warden. *The art of agile development*. O'Reilly Media, Inc., 2007.

[Tes18] Bjørnar Tessem. What causes positive customer satisfaction in an ineffectual software development project? A mechanism from a process tracing case study. *IJISPM-International Journal of Information Systems and Project Management*, 6(4):83–98, 2018.

[W3C] W3C. HTML and XHTML Techniques for WCAG 2.0. https://www.w3.org/TR/WCAG20-TECHS/html.html. Last accessed March 2020.

[WGV+11] Marco Winckler, Vicent Gaits, Dong-Bach Vo, Firmenich Sergio, and Gustavo Rossi. An Approach and Tool Support for Assisting

Users to Fill-in Web Forms with Personal Information. In *Proceedings of the 29th ACM international conference on Design of communication*, SIGDOC '11, pages 195–202, New York, NY, USA, 2011. ACM.

[Wie14] Roel J Wieringa. Research Goals and Research Questions. In *Design Science Methodology for Information Systems and Software Engineering*, pages 13–23. Springer, 2014.

EN

Appendix

Designing Test Maps

Test maps are mind maps but not all mind maps are test maps. That is, test maps restrict the structure and kind of participating nodes. In short, test maps become the graphical representation of a Domain Specific Language (DSL) for UAT. Coming up with a DSL implies identifying the main concerns of the UAT domain (i.e. the feature model). Next, providing a metamodel that captures the main relationships (a.k.a. abstract syntax), and finally, facilitating a graphical notation to specify the DSL expressions (a.k.a. concrete syntax). We begin with the feature model.

The feature model In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered [MHS05]. Broadly, the output consists basically of domain-specific terminology and semantics in more or less abstract form, being feature models a the main asset. A feature model captures "the commonalities and variabilities of domain concepts and their in-

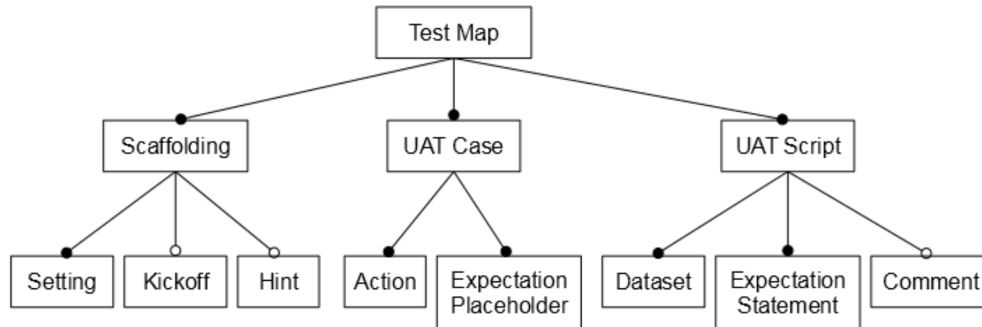


Figure A.1: UAT Feature Model.

terdependencies" [MHS05]. In this case, "the domain" is a UAT session. Figure A.1 depicts the main UAT concerns for our purposes. This diagram states that a *test map* captures a UAT session in terms of the available **scaffolding** set for guiding the session together with a **test case** and different enactments of this test, i.e. **test scripts**. These features where already described in Section 4.6. Notice that the aim is to identify the main concerns and their variabilities. How these concerns are to be expressed is postponed till the abstract syntax is specified.

Abstract Syntax Concerns risen during DSL analysis should now find their way into the DSL's abstract syntax. The abstract syntax describes the concepts of the language, the relationships among them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules. This is expressed as the DSL metamodel (see Figure A.2). A TestMap model (i.e. a *test map*) includes five main classes, namely:

- the *Setting* class, which indicates the name and email of the customer performing the testing, and a deadline for conducting the UAT. This class also contains the features to be tested by this specific customer.

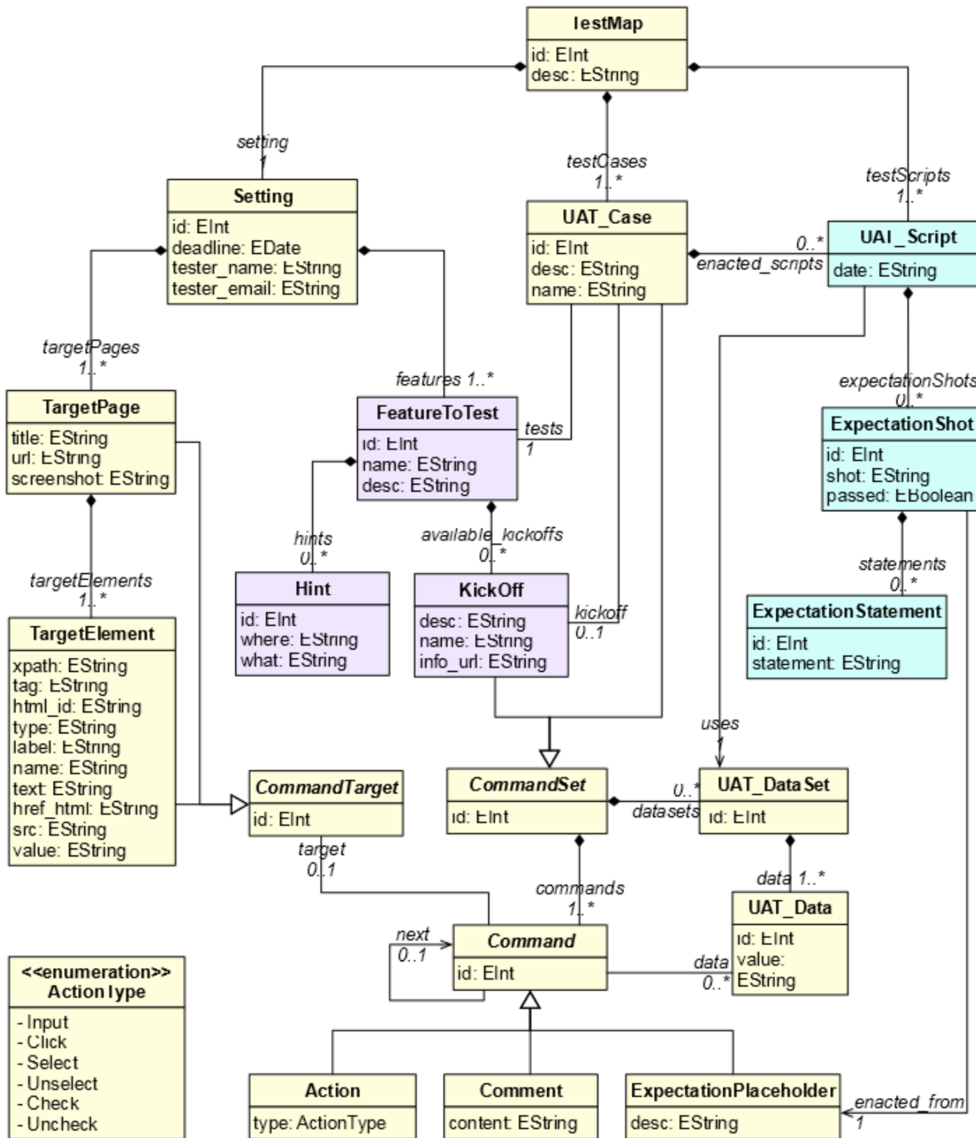


Figure A.2: TestMind Abstract Syntax.

- the *Kickoff* class, which holds the different Kickoff scenarios for each feature.

- the *Hint* class, which describes tooltip notes to be displayed during test case recording for a specific feature.

- the *UAT_Case* class, which stands for a test case. Test cases are meant to be provided through R&R tools. Hence, this class's expressiveness is based on the R&R tool to be used: Selenium IDE. Test cases are defined as a set of commands that mimic the interactions the customer performed during the test recording. Every command might interact with a target element (*CommandTarget*), that can be either a web element or the whole web page. Commands might be of three types: *Actions*, *Comments* and *ExpectationPlaceHolders*.

- the *UAT_Dataset* class, which collects a set of input data (*UAT_Data*) to be used on each test case execution in order to check the test case on different scenarios.

- the *UAT_Script* class, which stores the result of the execution of a *UAT_Case* using the provided *UAT_Datasets*. This class includes the *ExpectationShots* taken for the defined *ExpectationPlaceholders* on each execution.

Concrete syntax The concrete syntax comprises a mapping between the metamodel concepts (i.e., the abstract syntax) and their textual or visual representation. While the abstract syntax addresses expressiveness, the concrete syntax cares for usability as for the target audience. Our target audience are customers with low or no technical knowledge. We then resort to mind maps as a notation of test maps. However, though *test maps* are mind maps, not all mind maps are *test maps*. That is, we need to restrict the expressiveness

of mind maps to limit the kind of nodes and the structure that *test maps* can exhibit.

Limiting the kind of nodes. Mind mapping supports the notion of *Node*. This general notion needs to be specialized into the different concerns risen during UAT. Hence, we do no longer have just generic nodes but *Kickoff* nodes, *Action* nodes, *UATCase* nodes and so on. The type of each node is denoted through an icon (see Figure 4.5). Icons play the same role than profiles in UML. In this way, classes in the abstract syntax are mapped as nodes with specific icons. Concepts are mapped to nodes where the node's icon stands for the type and the label and associated notes accommodate the rest of the properties.

Limiting the structure. The arrangement of the previously defined node types should follow some rules to maintain the *UATCase* structure. For example, *Action* nodes can hang from *Pages* but not vice versa, *UATScript* nodes can hold *ExpectationShot* nodes but not *Kickoff* nodes, etc [1].

---

[1]This is supported by extending the FreeMind's XML Schema with additional sub-types and restrictions along the TestMind's abstract syntax

B

Testmind interview Guide

1. Was the subject alone when conducting the test? Were other colleagues around?
2. How quiet was the place when conducting the test? (phone calls, WhatsApp sounds, etc.)
3. How many interruptions happened when conducting the test? (phone ringing, door slams, chatting staffers, etc)
4. What technical resources were there? (Screen size, wifi, broad band, ...)
5. Was the subject capable of creating valid tests?
6. Did he introduce any error into the test?
7. How many tests did he generate?
8. How many comments did he add during the recording?

9. How many ExpectationPlaceholders did he use?
10. Where did he find most problems?
11. What benefits would he highlight from TestMind?
12. Is the subject easily finding the features?
13. Is the subject feeling comfortable using the tool?
14. Does the subject feel eager to create tests with this tool?
15. Does the subject understand the mind map view of the tests?
16. Will the subject prefer a different visualization?
17. Will the subject change anything on the tool?

TestMind installation guide

TestMind installation goes along three steps:

1. install *FreeMind*[1]

2. install *TestMind* for Freemind[2] and configure it for your *FitNesse* installation

3. install *Selenium IDE*[3] and its *TestMind* plug-in[4] to enhance *Selenium IDE* with the *hint bar*, i.e. a bar where testing hints are showed; and the

[1]http://freemind.sourceforge.net/wiki/index.php/Download accessed 29-Feb-16.

[2]http://www.onekin.org/testmind accessed 15-Nov-16.

[3]http://www.seleniumhq.org/projects/ide accessed 29-Feb-16.

[4]http://www.onekin.org/testmind accessed 15-Nov-16.

commenting bar, i.e., a bar where user comments can be added during navigation.

TestMind has been checked with *FreeMind* v1.0.1, *Firefox 37, Selenium IDE* v2.9.1, and *FitNesse* v20130530.

Summary

Many applications which formerly were designed for the desktop have gradually made a transition to the Web. Accordingly, an increasing number of tasks can now be conducted through the Web. As a result, opportunities arise to achieve a higher level of automation that the one being previously possible with proprietary, OS-anchored desktop applications. This results in an emerging interest in empowering users to check, adapt and customize the way they navigate and make use of these applications. Web automation, Web augmentation, or Web mashups are performant approaches that pursuit this aim.

This work explores the possibility of end-user involvement in two tasks, namely, Web-form filling and Web Application User-Acceptance Testing. In both cases, the challenge rests on abstracting scripting code into platform-independent models that permit the notion of scripting to be hidden into more affordable representations.

For *Web-form filling*, this work tackles the problem of repetitive form-filling from external sources. The solution is realized through WebFeeder, a plugin for iMacros that introduces autofilling-script models as first-class artifacts in iMacros.

As for *User Acceptance Testing*, we tackle the issue of the need for the regular physical presence of stakeholders in Agile methodologies. In this case, we resort to mind-maps as the model representation. These ideas are fleshed out in TestMind, an editor that permits to capture UAT sessions as test maps.

Summing it up, the bottom line is that WebFeeder and TestMind showcase the benefits that Model-Driven Engineering can bring to Web Automation. By moving away from code to high-level models, we reduce the entry barrier for the participation of end-users.

Itziar was born in Vitoria-Gasteiz, where she studied Technical Engineering in Computer Science. In 2008 she moved to San Sebastián where, in 2010, she got her degree on Computer Engineering.

Her hobbies include hand drawing, graphic design and watching animation films. She never refuses to have a good pint of beer with her friends.