Seminario de la tecnicatura superior en análisis de sistemas

Microscopía Holográfica Digital

Programación científica desde el enfoque de metodologías ágiles de desarrollo

Carlos M. Cabrera

Marzo de 2012

Instituto Dr. Facundo de Zuviria

Director
Lic. Andrea Carolina Monaldi

Tribunal
Prof. Balderrama, Marcelo
Prof. Moya, Mercedes
Prof. Arce, Anibal

Fecha de graduación
15/03/2012

a mis padres; por enseñarme a seguir adelante, por su gran corazón y capacidad de entrega

Índice general

Introducción

Con este trabajo evalúo la relación costo/beneficio de la adopción de metodologías ágiles iterativas de desarrollo de software en la creación de sistemas de aplicación técnico-científica. La documentación sobre metodologías aplicadas al desarrollo de software científico es realmente escasa. Este es un campo bastante descuidado por los teóricos de ciencias de la información y donde lo habitual es que los mismos científicos desarrollen su software de modo solitario o conformando pequeños equipos.

Téngase presente que nos hemos señalado ciertos límites precisos, que son este y el de más allá. En tales límites seguiremos el siguiente método...

Parte 1 intentará establecer la metodología más adecuada para nuestro problema y nuestras posibilidades. Para ello estudia la teoría aceptada y compara los distintos enfoques existentes.

Parte 2 documenta el proceso de aplicación del método Cristal Claro en el desarrollo del sistema objeto, desde la primera especificación de requerimientos hasta la implementación efectiva.

Parte 3 analiza los resultados obtenidos y los compara con los proyectados determinando el nivel de éxito de las decisiones metodológicas tomadas y planteando posibles mejoras.

Para limitar la incidencia de factores externos se mantendrá un constante control sobre el costo y el progreso durante todo el proceso. Para favorecer la revisión y reutilización de las conclusiones aquí obtenidas se hizo énfasis en la correcta aplicación de patrones de diseño orientado a objetos y buenas prácticas de programación y documentación. La documentación completa y el código escrito están disponibles para su reutilización bajo las licencias GPL3 y BSD[1].

[1]Puede descargarse de http://github.com/pointtonull/golsoft

Parte I.

Marco teórico

1. Desarrollo de software

1.1. Metodologías predictivas

Son las metodologías inspiradas en disciplinas ingenieriles como la ingeniería industrial, civil y electrónica.

1.1.1. Surgimiento

Hace poco más de cincuenta años las computadoras tenían nombre. Eran compradas con fines muy específicos, eran costosas y su adquisición y adecuación demandaban mucho tiempo. En este contexto nace la necesidad de desarrollar los primeros grandes sistemas. Hasta entonces los desarrollos se habían realizado sin planificación, a prueba y error.

El desarrollo de software que era una disciplina nueva y sin formulas propias imitó las metodologías de la planificación ingenieril. Estas metodologías dividieron el desarrollo de software en procesos estancos perfectamente diferenciados. Cada proceso depende del éxito del anterior. Estas metodologías son conocidas como de *desarrollo en cascada* (ver Figura 1.1).

1.1.2. Desarrollo en cascada

La analogía para con la ejecución de obras de ingeniería civil es llevada al extremo. Sus partidarios consideran al programador como un simple obrero en el proceso de fabricación de software.

El primer paso es el *análisis* del sistema existente[1]. Se hará mucho énfasis en la mejora de procedimientos desde un punto de vista Taylorista. Durante este paso se intentan capturar todas las posibles necesidades del cliente.

Durante el proceso de *diseño*, y a partir de la documentación producida por el análisis, los especialistas crean planos que determinan como será el nuevo sistema. Agotan las decisiones indicando con precisión los objetivos, el modo de conseguirlos y cuanto tiempo requerirá cada uno de ellos.

[1]La organización objeto bajo un estudio sistémico.

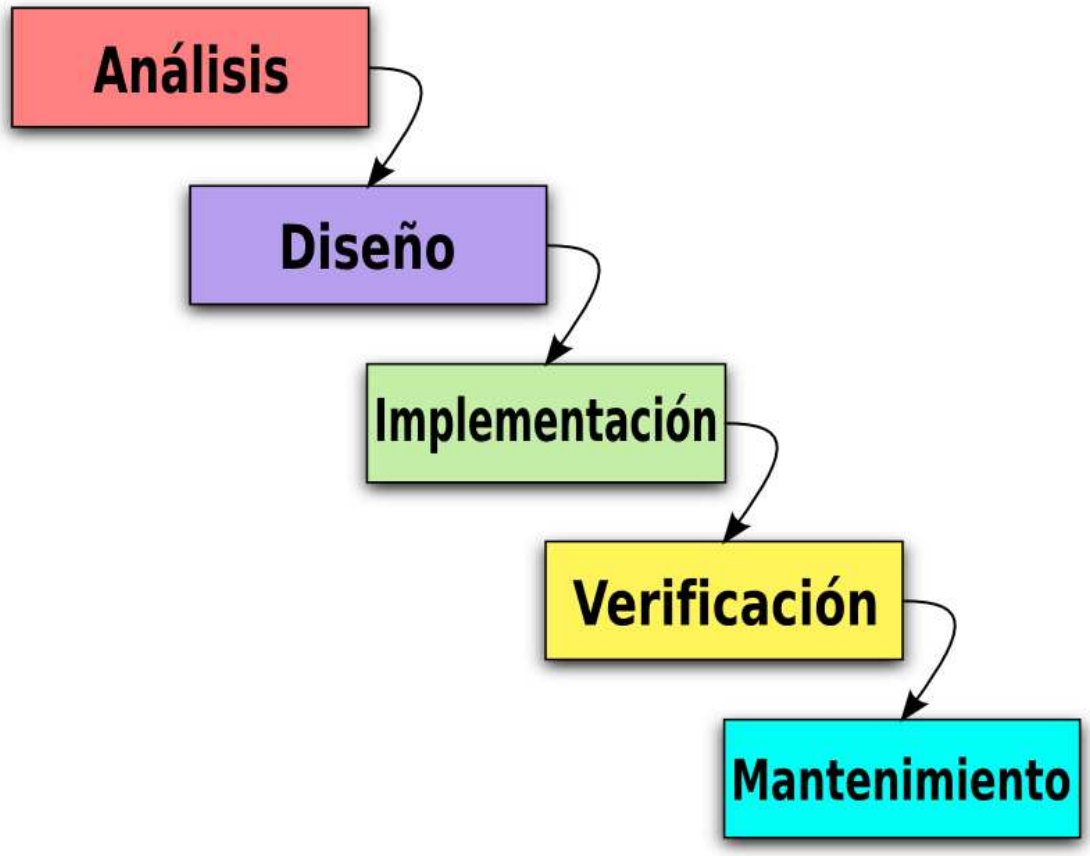

Figura 1.1.: Ilustración del flujo de trabajo en el desarrollo en cascada

Análisis y diseño pueden ser entendidos en conjunto como *planificación.* Es un ejercicio inherentemente creativo; sin plazos fijos ni garantías de éxito. Lo ejecuta personal altamente capacitado. Durante este periodo se toman todas las decisiones significativas.

El proceso de *implementación*, es muy predecible. Sus alcances y plazos han sido determinados durante el diseño. El es más largo y costoso pero se limita a la ejecución de un guión establecido y es reductible al análisis hombre/mes.

Durante la *verificación* se comprueba que el software resultante sea fiel reflejo de las especificaciones creadas durante la fase de diseño.

El *mantenimiento* se refiere a modificaciones posteriores a la entrega que son necesarias para corregir errores o para introducir nuevas características. En cada una de ellas se repiten los procesos de análisis, diseño, implementación y verificación.

Para el funcionamiento de esta metodología es crucial contar con una notación amplia y rigurosa. Solo una herramienta que permita especificar al detalle y sin ambigüedades todo el sistema permitirá la comunicación entre procesos aislados. Es por esto que se le otorga a los lenguajes de modelado, como UML, una enorme

importancia.

1.1.3. Crisis de la previsibilidad

El desarrollo en cascada funcionó bien durante mucho tiempo. Pero hoy, aunque sigue siendo la metodología más utilizada, pocos se niegan a aceptar su obsolescencia[Gro95].

Fracasa porque la analogía entre el desarrollo de software y la construcción de edificios ha caducado:

- Los requerimientos no son firmes como cimientos. La informática ya no está limitada a problemas matemáticos con requerimientos inmutables. Ahora es una actividad dinámica que debe satisfacer necesidades en constante evolución.
- A diferencia de la construcción de edificios, el desarrollo de software resuelve problemas inéditos[2]; sin antecedentes que sirvan como guía. Al enfrentar un nuevo desarrollo sólo se podrá especular muy vagamente sobre sus desafíos. Tomar todas las decisiones en ese momento maximiza los errores de análisis y diseño. Estos errores son responsables del 37 % de los fracasos[Gro95].
- El programador no es similar a un obrero. En la ingeniería civil el costo de la construcción es de un 90 % sobre el total mientras que en el desarrollo de software el costo de la programación es de sólo el 15 %[Fow]. Lo que es una inversión casi perfecta de las proporciones.
  Los metodistas modernos sostienen que la programación es parte activa del diseño y que el código no es sino la documentación de este proceso[Ree05]. La construcción (en la analogía con la ingeniería industrial) se corresponde, por sus características, con el proceso mecánico de compilación y ligado del código escrito.

De lo anterior se desprende que:

- las necesidades del cliente evolucionan junto con el sistema
- las decisiones tempranas suelen ser malas decisiones
- todo el desarrollo es un proceso creativo; imprevisible

Para solucionar estos problemas muchos intentaron impedir que los clientes realicen cambios en las especificaciones. Para eso recurrieron a la celebración de contratos con detalles de requerimientos. Esto desvirtuó los objetivos; ya no se busca satisfacer las necesidades del cliente, se busca cumplir con los términos del contrato firmado haciendo que el sistema y la realidad floten a la deriva.

[2] «Creative brains are a valuable, limited resource. They shouldn't be wasted on re-inventing the wheel when there are so many fascinating new problems waiting out there.» [Ray01, 1.2]

Los teóricos atribuyeren los problemas de especificaciones a ingeniería de requisitos pobremente aplicada. Lo que los llevó a incrementar la burocracia incrementando el coste de aplicar estas metodologías y retrasando los desarrollos.

También se intentó controlar y medir la productividad de los programadores. Pero es imposible medir la productividad en los procesos creativos y solo lograron desmotivar a los buenos programadores [Aus96].

Los resultados obtenidos no hicieron más que reforzar la idea de que el modelo ha sido agotado y se hizo evidente la necesidad de un replanteo completo.

1.2. Metodologías ágiles

1.2.1. Surgimiento

Como una reacción a los fracasos de los enfoques ingenieriles ha surgido un grupo de nuevas metodologías. Son las conocidas como metodologías ágiles. La diferencia más notable es que no son orientadas al documento; son más bien orientados al código[3]. Pero la falta de documentación es consecuencia de diferencias mucho más profundas:

Son adaptables en lugar de predictivas Las metodologías clásicas planean detalladamente para grandes plazos de tiempo (aspiran a planificarlo todo en el proyecto para toda su duración). Esto funciona siempre que los requisitos no cambien; se resisten al cambio. Para las metodologías ágiles el cambio es bienvenido; lo aceptan e intentan aprovecharlo, incluso fomentarlo.

Son orientados a la gente y no orientados al proceso La meta de los métodos ingenieriles es eliminar las variables impredecibles. Para eso deben tratar a los empleados como recursos y definir procesos que los hagan predecibles. En los procesos ágiles se reconoce el rol creativo de todas las personas involucradas en el desarrollo y se hace énfasis al desarrollo de las personas y a sus desempeños individuales y como equipo.

1.2.2. Programación Extrema

Más conocida como XP. Es, de todas las metodologías ágiles, la que ha recibido más atención. Esto se debe en parte a la notable habilidad de sus lideres, en particular Kent Beck, para involucrar nuevos talentos en su desarrollo.

Las raíces de la XP yacen en la comunidad de Smalltalk, y en particular la colaboración cercana de Kent Beck y Ward Cunningham a finales de los 1980s. Ambos refinaron sus prácticas en numerosos proyectos a principios de los 90s, extendiendo

[3]En rigor consideran que el código es el documento más importante.

sus ideas para el desarrollo adaptable de software y orientado a la gente. Ganó notoriedad cuando Kent intervino un proyecto de C3 para Chrysler que se encontraba en problemas. El desarrollo lo estaba haciendo en Smalltalk una compañía contratista. Kent, viendo muy baja calidad en la base del código, recomendó descartarlo todo y volver a empezar. El proyecto reinició bajo su dirección y se volvió el buque insignia y campo de entrenamiento de la XP.

Toda la metodología se funda en cuatro valores: *Comunicación*, *Retroalimentación*, *Simplicidad* y *Coraje*:

Comunicación:

- el código comunica mejor cuanto más simple sea
- las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de su funcionalidad
- programación en parejas: el código es revisado y discutido mientras se escribe
- el cliente forma parte del equipo de desarrollo; es él quien decide la prioridad de las características y debe siempre disponible para solucionar las dudas que puedan surgir

Retroalimentación: se logra a través de pruebas unitarias y su ejecución frecuentemente y automatizada, incluyendo pruebas de regresión. Esto posibilita:

- continua refactorización; las pruebas garantizan que con la reescritura no se han introducido fallos
- corrección de todos los errores antes de añadir nueva funcionalidad
- propiedad compartida del código: todo el personal puede corregir y extender cualquier parte del proyecto

Simplicidad:

- el código debe ser refactorizado frecuente
- documentación clara y sencilla orientada a los usos y características
- soluciones sencillas: programas creados para hoy y que deben ser adaptados mañana es mejor que programas con características sobre-diseñadas que nunca son usadas

Coraje:

- los desarrolladores deben sentirse cómodos reconstruyendo su código cada vez que sea necesario
- hay que poder quitar el código obsoleto sin importar cuanto tiempo y esfuerzo costó crearlo

- en la perseverancia; se puede permanecer sin avanzar en un problema complejo por un día entero, y luego se lo resolverá rápidamente al día siguiente, solo si persevera

XP construye sobre estos principios una docena de prácticas que los proyectos deben seguir. Teje estas técnicas como un sistema sinérgico dónde cada una fortalece a las demás. El resultado es un proceso que combina la disciplina con la adaptabilidad de una manera que indiscutiblemente la hace la más desarrollada entre todas las metodologías adaptables[BA04, Fow].

1.2.3. La Familia de Cristal de Cockburn

El doctor Alistair Cockburn es probablemente el más riguroso de los investigadores de metodologías. A través de sus frecuentes publicaciones a establecido las bases teóricas que sustentan gran parte del movimiento ágil. Entre sus aportes más universalmente aceptados se encuentran la definición de *Casos de Uso[Coc06b]* y el análisis del desarrollo de software desde la *Teoría de juegos*[Coc06a].

El fruto de sus investigaciones es la creación de la familia de metodologías Cristal. Es una familia porque define no una sino un conjunto de metodologías y los lineamientos generales que un proyecto debe seguir para adoptar la que le resulte más conveniente. Para ello dispone una escala donde se consideran las características de los proyectos a través de dos variables: la cantidad de personas involucradas en el proyecto, y las consecuencias de un posible fracaso (ver cuadro en esta página). Hay una metodología en la familia Cristal para cada combinación en la escala.

		Personas involucradas			
		1-6	**7-20**	**21-40**	**41-100**
Riesgo	**L**	L6	L20	L40	L100
	E	E6	E20	E40	E100
	D	D6	D20	D40	D100
	C	C6	C20	C40	C100

Cuadro 1.1.: *Escala de Cockburn* para clasificación de proyectos según la cantidad de personas que involucra y el riesgo que implica (que va desde confort (C) a vidas (L)).

Estas metodologías comparten con XP el enfoque orientado a las personas pero lo hacen de manera muy diferente. Cockburn considera poco realista exigir a todas las organizaciones y a todos los planteles la adopción de procesos muy disciplinados. En su lugar explora la aplicación de procesos menos estrictos que igualmente permitan obtener buenos resultados para determinados proyectos. De este modo las metodologías Cristal logran mejores resultados en termino medio ya que más proyectos serán capaces de aplicarlas exitosamente.

En la *Programación extrema* se define un proceso detalladamente y se enfatiza la importancia de su cumplimiento estricto. En cambio, en las metodologías *Cristal* se establecen lineamientos iniciales y se exhorta a cada organización a crear su propio proceso a partir de la modificación de estos lineamientos iniciales. Para ello establece procesos de revisión del proceso al final de cada iteración.

Entre las metodologías de la Familia Cristal resalta *Cristal claro*, que es considerada modelo de metodología ágil y liviana. Fácil de implementar y con numerosos casos de uso.

Está pensada para ser aplicada a equipos pequeños, de hasta 6 desarrolladores vecinos trabajando en sistemas no críticos. Se centra en las personas, no en los procesos o artefactos.

Cristal claro requiere las siguientes propiedades:

- Entrega frecuente de versiones útiles para el usuario
- Mejoras reflexivas, del sistema y de la metodología
- Comunicación osmótica, por proximidad e interrelación del equipo

Brinda estos atributos:

- Seguridad personal, es poder hablar cuando algo nos inquieta si miedo a ningún tipo de represalia
- Mantención del enfoque, los programadores enfrentarán desafíos de a uno
- Facilidad acceso para los usuarios expertos
- Pruebas automatizadas, gestión de configuraciones e integración frecuente

1.2.4. Código Abierto

Usted puede sorprenderse por este título. Después de todo el código abierto es un estilo de software, no tanto un proceso. Sin embargo hay una manera definida de hacer las cosas haciendo en la comunidad de código abierto, y mucho de su acercamiento es tan aplicable a los proyectos de código cerrado como a los de código abierto. En particular su proceso se engrana a equipos físicamente distribuidos, lo qué es importante porque la mayoría de los procesos adaptables exigen equipos locales. La mayoría de los proyectos de código abierto tienen uno o más mantenedores. Un mantenedor es la única persona a la que se le permite integrar un cambio en el almacén de código fuente. Sin embargo otras personas pueden hacer cambios a la base del código. La diferencia importante es que estas otras personas necesitan enviar su cambio al mantenedor que entonces lo revisa y lo aplica a la base del código. Normalmente estos cambios son hechos en forma de archivos de parches que hacen este proceso más fácil. El mantenedor así es responsable de coordinar los parches y mantener la cohesión en el diseño del software. Proyectos diferentes manejan el papel del mantenedor de diferentes maneras. Algunos tienen un mantenedor para el

proyecto entero, algunos lo dividen en módulos y tiene un mantenedor por módulo, algunos rolan el mantenedor, algunos tienen múltiples mantenedores sobre el mismo código, otros tienen una combinación de estas ideas. La mayor parte de la gente de código abierto son de tiempo parcial, así que hay una duda en qué tan bien se coordina un equipo así para un proyecto de tiempo completo. Un rasgo particular del desarrollo de código abierto es que la depuración es altamente paralelizable. Muchas personas pueden involucrarse en el depurado. Cuando encuentran un bug pueden enviar el parche al mantenedor. Esto es un buen papel para los no mantenedores ya que la mayor parte del tiempo se gasta en encontrar bugs. También es bueno para gente sin mucha destreza en programación. El proceso para el código abierto aun no se escribe bien. La referencia más famosa es el artículo de Eric Raymond The Catedral and the Bazar[citar], que aunque es una descripción excelente también es bastante informal. El libro de Karl Fogel sobre el almacén de código CVS también contiene varios buenos capítulos sobre el proceso de código abierto que incluso serían interesantes para aquéllos que no quieren hacer una actualización cvs.

1.2.5. El Desarrollo de Software Adaptable de Highsmith

Jim Highsmith ha pasado muchos años trabajando con metodologías predictivas. Él las desarrolló, instaló, enseñó, y concluyó que son profundamente defectuosas: particularmente para los negocios modernos. Su reciente libro se enfoca en la naturaleza adaptable de las nuevas metodologías, con un énfasis particular en aplicar las ideas que se originaron en el mundo de los sistemas complejos adaptables (normalmente conocida como teoría del caos.) No proporciona el tipo de prácticas detalladas como lo hace la XP, pero proporciona la base fundamental de por qué el desarrollo adaptable es importante y las consecuencias a los más profundos niveles de la organización y la gerencia. En el corazón del ASD hay tres fases solapadas, no lineales: especulación, colaboración, y aprendizaje. Highsmith ve la planificación como una paradoja en un ambiente adaptable, ya que los resultados son naturalmente imprevisibles. En la planificación tradicional, las desviaciones del plan son errores que deben corregirse. En un el ambiente adaptable, sin embargo, las desviaciones nos guían hacia la solución correcta. En este ambiente imprevisible se necesita que las personas colaboren de la mejor manera para tratar con la incertidumbre. La atención de la gerencia es menor en lo que tiene que hacer la gente, y mayor sobre la comunicación alentadora para que las personas puedan proponer las respuestas creativas ellos mismos. En ambientes predictivos, el aprendizaje se desalienta a menudo. Las cosas se ponen de antemano y entonces se sigue ese diseño.

«En un ambiente adaptable, aprender desafía a todos - desarrolladores y sus clientes - a examinar sus asunciones y usar los resultados de cada ciclo de desarrollo para adaptar el siguiente.» -[Highsmith]

El aprendizaje como tal es un rasgo continuo e importante, uno que asume que los planes y los diseños deben cambiar conforme avanza el desarrollo.

"El beneficio atropellado, poderoso, indivisible y predominante del Ciclo de Vida de Desarrollo Adaptable es que nos obliga a confrontar los modelos mentales que están en la raíz de nuestro auto-engaño. Nos obliga a estimar con realismo nuestra habilidad." -[Highsmith]

Con este énfasis, el trabajo de Highsmith se enfoca directamente en fomentar las partes difíciles del desarrollo adaptable, en particular cómo fomentar la colaboración y el aprendizaje dentro del proyecto. Como tal su libro ayuda a dar ideas para fomentar estas áreas "suaves" que hacen un buen complemento a los acercamientos basados en una práctica aterrizada como XP, FDD y Cristal.

1.2.6. Scrum

Scrum ha estado durante algún tiempo en los círculos orientados a objetos, aunque confesaré que yo no estoy muy al tanto de su historia o desarrollo. De nuevo se enfoca en el hecho de que procesos definidos y repetibles sólo funcionan para atacar problemas definidos y repetibles con gente definida y repetible en ambientes definidos y repetibles. Scrum divide un proyecto en iteraciones (que ellos llaman carreras cortas) de 30 días. Antes de que comience una carrera se define la funcionalidad requerida para esa carrera y entonces se deja al equipo para que la entregue. El punto es estabilizar los requisitos durante la carrera. Sin embargo la gerencia no se desentiende durante la carrera corta. Todos los días el equipo sostiene una junta corta (quince minutos), llamada scrum, dónde el equipo discurre lo que hará al día siguiente. En particular muestran a los bloques de la gerencia: los impedimentos para progresar que se atraviesan y que la gerencia debe resolver. También informan lo que se ha hecho para que la gerencia tenga una actualización diaria de dónde va el proyecto. La literatura de Scrum se enfoca principalmente en la planeación iterativa y el seguimiento del proceso. Es muy cercana a las otras metodologías ágiles en muchos aspectos y debe funcionar bien con las prácticas de código de la XP. Después de mucho tiempo sin un libro, finalmente Ken Schwaber y Mike Beedle escribieron el primer libro de scrum. Ken Schwaber también aloja controlChaos.com qué probablemente es la mejor apreciación global sobre SCRUM. Jeff Sutherland siempre ha tenido un sitio web activo sobre temas de tecnologías de objetos e incluye una sección sobre SCRUM. Hay también una buena apreciación global de las prácticas de Scrum en el libro PLoPD 4.

1.2.7. Desarrollo Manejado por Rasgos

El Desarrollo Manejado por Rasgos (FDD por sus siglas en inglés) fue desarrollado por Jeff De Luca y el viejo gurú de la OO Peter Coad. Como las otras metodologías adaptables, se enfoca en iteraciones cortas que entregan funcionalidad tangible. En el caso del FDD las iteraciones duran dos semanas. El FDD tiene cinco procesos. Los primeros tres se hacen al principio del proyecto.

- Desarrollar un modelo global
- Construir una lista de los rasgos
- Planear por rasgo
- Diseñar por rasgo
- Construir por rasgo

Los últimos dos se hacen en cada iteración. Cada proceso se divide en tareas y se da un criterio de comprobación. Los desarrolladores entran en dos tipos: dueños de clases y programadores jefe. Los programadores jefe son los desarrolladores más experimentados. A ellos se les asignan rasgos a construir. Sin embargo ellos no los construyen solos. Solo identifican qué clases se involucran en la implantación de un rasgo y juntan a los dueños de dichas clases para que formen un equipo para desarrollar ese rasgo. El programador jefe actúa como el coordinador, diseñador líder y mentor mientras los dueños de clases hacen gran parte de la codificación del rasgo. Hasta recientemente, la documentación sobre FDD era muy escasa. Finalmente hay un libro completo sobre FDD. Jeff De Luca, el inventor primario, ya tiene un portal FDD con artículos, blogs y foros de discusión. La descripción original estaba en el libro UML in Color de Peter Coad et al. Su compañía, TogetherSoft, también da consultoría y entrenamiento en FDD.

1.2.8. Método de Desarrollo de Sistema Dinámico

El DSDM (por sus siglas en inglés) empezó en Gran Bretaña en 1994 como un consorcio de compañías del Reino Unido que querían construir sobre RAD y desarrollo iterativo. Comenzó con 17 fundadores ahora tiene más de mil miembros y ha crecido fuera de sus raíces británicas. Siendo desarrollado por un consorcio, tiene un sabor diferente a muchos de los otros métodos ágiles. Tiene una organización de tiempo completo que lo apoya con manuales, cursos de entrenamiento, programas de certificación y demás. También lleva una etiqueta de precio, lo qué ha limitado mi investigación sobre su metodología. Sin embargo Jennifer Stapleton ha escrito un libro que da una apreciación global de la metodología. El método empieza con un estudio de viabilidad y negocio. El estudio de viabilidad considera si DSDM es apropiado para el proyecto. El estudio de negocio es una serie corta de talleres para entender el área de negocio dónde tiene lugar el desarrollo. También propone arquitecturas de esbozos del sistema y un plan del proyecto. El resto del proceso forma tres ciclos entretejidos: el ciclo del modelo funcional produce documentación de análisis y prototipos, el ciclo de diseño del modelo diseña el sistema para uso operacional, y el ciclo de implantación se ocupa del despliegue al uso operacional. DSDM tiene principios subyacentes que incluyen una interacción activa del usuario, entregas frecuentes, equipos autorizados, pruebas a lo largo del ciclo. Como otros métodos ágiles usan ciclos de plazos cortos de entre dos y seis semanas. Hay un énfasis en la alta calidad y adaptabilidad hacia requisitos cambiantes.

DSDM es notable por tener mucha de la infraestructura de las metodologías tradicionales más maduras, al mismo tiempo que sigue los principios de los métodos ágiles. Parece haber una pregunta en si sus materiales animan más de una orientación al proceso y más ceremonia de lo que me gustaría.

http://alturl.com/bxupf

- fase del pre-proyecto
- fase del ciclo de vida del proyecto
  - estudio de viabilidad
  - estudio de la empresa
  - iteración del modelo funcional
  - diseño e iteración de la estructura
  - implementación
- fase del post-proyecto

DSDM reconoce que los proyectos son limitados por el tiempo y los recursos, y los planes acorde a las necesidades de la empresa. Para alcanzar estas metas, DSDM promueve el uso del RAD con el consecuente peligro que demasiadas esquinas estén cortadas. DSDM aplica algunos principios, roles, y técnicas. No le encontré ventajas con respecto a RUP.

1.2.9. Scrum

Scrum es un modelo de referencia que define un conjunto de prácticas y roles, y que puede tomarse como punto de partida para definir el proceso de desarrollo que se ejecutará durante un proyecto. Los roles principales en Scrum son el ScrumMaster, que mantiene los procesos y trabaja de forma similar al director de proyecto, el ProductOwner, que representa a los stakeholders (interesados externos o internos), y el Team que incluye a los desarrolladores.

Durante cada sprint, un periodo entre una y cuatro semanas (la magnitud es definida por el equipo), el equipo crea un incremento de software potencialmente entregable (utilizable). El conjunto de características que forma parte de cada sprint viene del Product Backlog, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar. Los elementos del Product Backlog que forman parte del sprint se determinan durante la reunión de Sprint Planning. Durante esta reunión, el Product Owner identifica los elementos del Product Backlog que quiere ver completados y los hace del conocimiento del equipo. Entonces, el equipo determina la cantidad de ese trabajo que puede comprometerse a completar durante el siguiente sprint.2 Durante el sprint, nadie puede cambiar el Sprint Backlog, lo que significa que los requisitos están congelados durante el sprint.

Scrum permite la creación de equipos autoorganizados impulsando la co-localización de todos los miembros del equipo, y la comunicación verbal entre todos los miembros y disciplinas involucrados en el proyecto.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar de idea sobre lo que quieren y necesitan (a menudo llamado requirements churn), y que los desafíos impredecibles no pueden ser fácilmente enfrentados de una forma predictiva y planificada. Por lo tanto, Scrum adopta una aproximación pragmática, aceptando que el problema no puede ser completamente entendido o definido, y centrándose en maximizar la capacidad del equipo de entregar rápidamente y responder a requisitos emergentes.

Existen varias implementaciones de sistemas para gestionar el proceso de Scrum, que van desde notas amarillas "post-it" y pizarras hasta paquetes de software. Una de las mayores ventajas de Scrum es que es muy fácil de aprender, y requiere muy poco esfuerzo para comenzarse a utilizar.

1.2.10. AUP

Es una versión simplificada del Proceso Unificado de Rational (RUP) que busca librarlo de su exceso de artefactos y corregir su tendencia a la predictivilidad. Este describe de una manera simple y fácil de entender la forma de desarrollar aplicaciones de software de negocio usando técnicas ágiles y conceptos que aún se mantienen válidos en RUP. El AUP aplica técnicas ágiles incluyendo Desarrollo Dirigido por Pruebas (test driven development - TDD), Modelado Ágil, Gestión de Cambios Ágil, y Refactorización de Base de Datos para mejorar la productividad.

1.2.11. Lean

La metodología de desarrollo de software Lean es una translación de los principios y prácticas de la manufacturación Lean hacia el dominio del desarrollo de software. Adaptado del Sistema de producción Toyota, apoyado por una sub-cultura pro-lean que está surgiendo desde la comunidad ágil. El desarrollo Lean puede resumirse en siete principios, similares a los principios fabricación Lean:

- Eliminar los desperdicios: Todo lo que no añade valor al cliente se considera un desperdicio:
  - Código y funcionalidades innecesarias
  - Retraso en el proceso de desarrollo de software
  - Requisitos poco claros
  - Burocracia
  - Comunicación interna lenta

- Ampliar el aprendizaje: iteraciones cortas cada una de ellas acompañada de refactorización y pruebas de integración
- Decidir lo más tarde posible
- Reaccionar tan rápido como sea posible
- Potenciar el equipo
- Crear la integridad: El cliente debe tener una experiencia general del sistema.
- Mantener una visión global del sistema

1.2.12. OpenUP

Proceso mínimo y suficiente; solo el contenido fundamental es incluido. Por lo tanto no provee lineamientos para todos los elementos que se manejan en un proyecto pero tiene los componentes básicos que pueden servir de base a procesos específicos. La mayoría de los elementos de OpenUP están declarados para fomentar el intercambio de información entre los equipos de desarrollo y mantener un entendimiento compartido del proyecto, sus objetivos, alcance y avances.

- colaborar para sincronizar intereses y compartir conocimiento
- equilibrar las prioridades para maximizar el beneficio obtenido por los interesados en el proyecto
- centrarse en la arquitectura de forma temprana para minimizar el riesgo y organizar el desarrollo
- desarrollo evolutivo para obtener retroalimentación y mejoramiento continuo

El OpenUP está organizado en dos dimensiones:

método: los elementos del método (roles, tareas, artefactos y lineamientos) son definidos, sin tener en cuenta como son utilizados en el ciclo de vida del proyecto

proceso: los elementos del método son aplicados de forma ordenada Muchos ciclos de vida para diferentes proyectos pueden ser creados a partir del mismo método

2. Herramientas

2.1. Gestión del código fuente

2.1.1. Github

Es una red social para desarrolladores.

- repositorios GIT en servidores redundantes sin costo para proyectos de código abierto
- páginas Wiki para la documentación
- páginas web
- sistema para el reporte y gestión de fallas
- genera gráficos con información sobre la actividad de cada repositorio y sobre la incidencia de cada desarrollador en ella
- muestra los grafos de ramas de los proyectos
- sus características de red social favorecen la inserción de nuevos programadores al desarrollo

Figura 2.1.: Ejemplo de la representación gráfica de operaciones entre distintas ramas de un repositorio como se muestra en github.

2.1.2. GIT

GIT es un sistema de control de versiones que implementa el manejo de los estados discretos como endofunctores homeomorfos en un espacios de Girac. El trabajar sobre definiciones estrictas de dominio de la teoría de grafos le permite aplicar las

operaciones definidas en esta para gestionar complejas operaciones de versiones y ramas. Algunas de estas operaciones entre ramas se pueden ver en la figura en Figura 2.1

Desarrollo no-lineal rapidez en la gestión de ramas y mezclado de diferentes versiones. Incluye herramientas específicas para navegar y visualizar un historial de desarrollo no-lineal. Una presunción fundamental en Git es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

Gestión distribuida cada programador posee una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.

Gestión eficiente de proyectos grandes dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución. Todas las versiones previas a un cambio determinado, implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio (denominado autenticación criptográfica de historial).

2.2. Lenguajes de programación

El dominio del problema exige en lenguaje con buenas capacidades para el cálculo numérico. Y parte importante de los requerimientos es obtener un sistema que los mismos usuarios puedan mantener/extender.

- De dominio de los involucrados
- Eficiente para operaciones con matrices
- De sintaxis clara y documentación abundante
- Capacidades modernas; programación orientada a objetos/aspectos, duck-typing
- Multi-plataforma (debe poder correr en Windows y GNU/Linux)

Se decidió que no hay un lenguaje que cumpla con todos los requisitos por lo que se usará un único lenguaje para todo el proyecto sino que se aplicará el adecuado a cada problema. De momento se han utilizado los siguientes:

Python para la estructura general del sistema

Numpy paquete que extiende Python para permitirle el cálculo eficiente con matrices

Scipy librería construida a partir de Numpy que implementa muchos de los algoritmos frecuentemente utilizados en el desarrollo de aplicaciones científicas

Image librería para la creación y manipulación de imágenes

C/Cython para escribir las porciones de código que son criticas en rendimiento

Sh para la automatización de tareas de gestión y comunicación (auto-actualizaciones, publicación de reportes)

AWK para generación y modificación automática de de reportes

MatLab y pseudocódigo que no formarán parte del sistema final pero constituyeron una herramienta esencial para la comunicación precisa y fluida entre entre los involucrados

2.2.1. Librerías gráficas

Para la creación de interfaces de usuario se siguió la misma filosofía. Hay necesidades muy sencillas que pueden ser resueltas a través de herramientas de fácil aplicación. Y hay otras tareas que requieren la generación de espacios de iterativos de representación de gráficos tridimensionales.

TCL/TK es una librería para la creación de interfaces gráficas incluida en Python

Mayavi/VTK es un conjunto de librerías que permiten crear representaciones gráficas avanzadas e interfaces de control a partir de definiciones del código fuente

2.2.2. Herramientas de edición/depuración

La principal ventaja de trabajar con lenguajes auto-coherentes es poder mantener a la lógica del problema separada de las interfaces de usuario. Así el código fuente es sólo un fichero de texto que (gracias a la simplicidad del lenguaje) puede ser modificado con cualquier editor sin inconvenientes.

2.2.2.1. TKPipe

Windows carece de una consola que permita comandos ANSI e inserción de imágenes. Por ello, en las primeras etapas de desarrollo, se creó *TKPipe*, una librería que se puede usar como herramienta de prototipo y depuración independiente del sistema operativo.

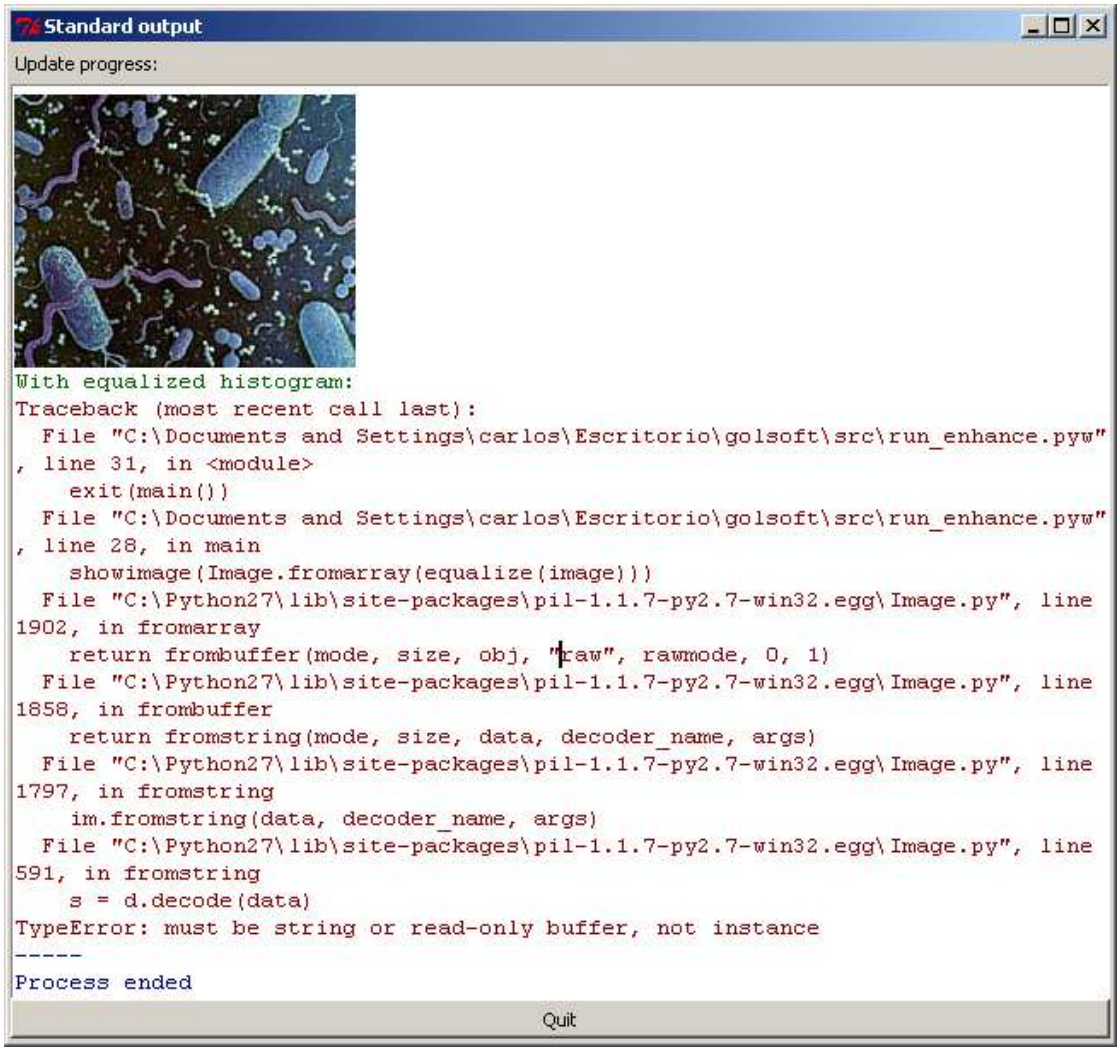

Figura 2.2.: Ejemplo de TKPipe usado en una aplicación para mostrar los resultados primero (en verde e imagen) y luego el traceback de un error encontrado en tiempo de ejecución (en rojo).

3. Conclusiones

Parte II.

Desarrollo del sistema objeto

4. Sobre la estructura del texto

Esta parte se irá escribiendo de forma progresiva e incremental conforme se lleva a cabo el desarrollo del sistema.

A cada capitulo corresponde una iteración. En él se detallan las tareas realizadas durante el correspondiente sprint con detalle de duración y nivel de satisfacción de requisitos. La tarea de organizar y plasmar esta información no es un perjuicio al proceso sino un complemento del mismo ya que conformará el proceso de auto-revisión recomendado en la metodología.

4.1. Los usuarios

Todos ellos tienen formación en, al menos, un lenguaje de programación y experiencia en la escritura de algoritmos de cálculo numérico.

Los principales operadores son:

Andrea C. Monaldi: Licenciada en física e investigadora del CONICET que se encuentra desarrollando su doctorado en física.

Graciela Romero: Doctora en física.

4.2. El problema

El Grupo de investigación de Óptica Laser, GOL, trabaja en el desarrollo te técnicas de microscopía holográfica digital. La microscopía holográfica digital utiliza la interferometría óptica y el procesamiento digital de señales para crear mapas tridimensionales de los objetivos.

El uso de fuentes coherentes de luz permite generar patrones de interferencia que codifican la información de fase. Un laser es un haz de luz coherente del que se pueden asegurar ciertas propiedades como la intensidad y la frecuencia.

La tercera dimensión es inferida de la información de fase y combina la distancia recorrida por el haz de luz y los índices de refracción de los medios que atraviesa.

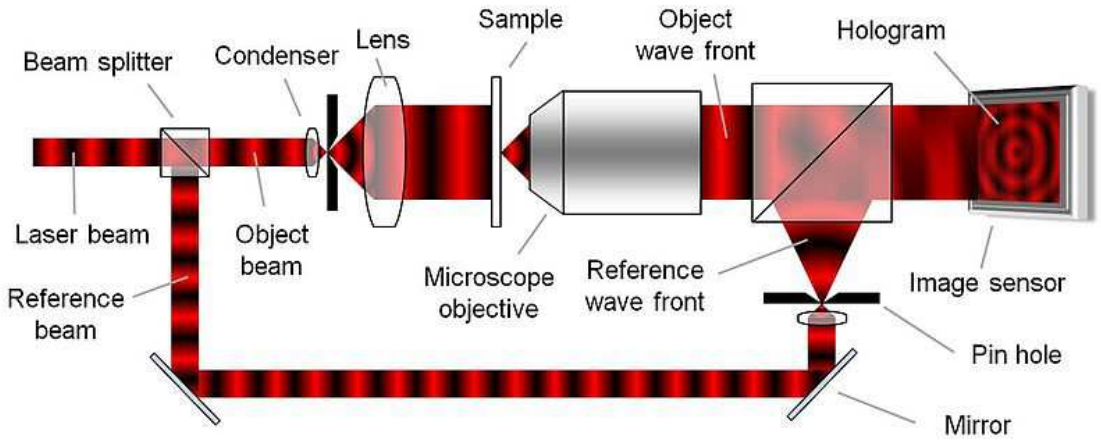

Figura 4.1.: Esquema de representación del montaje de un equipo de MHD.

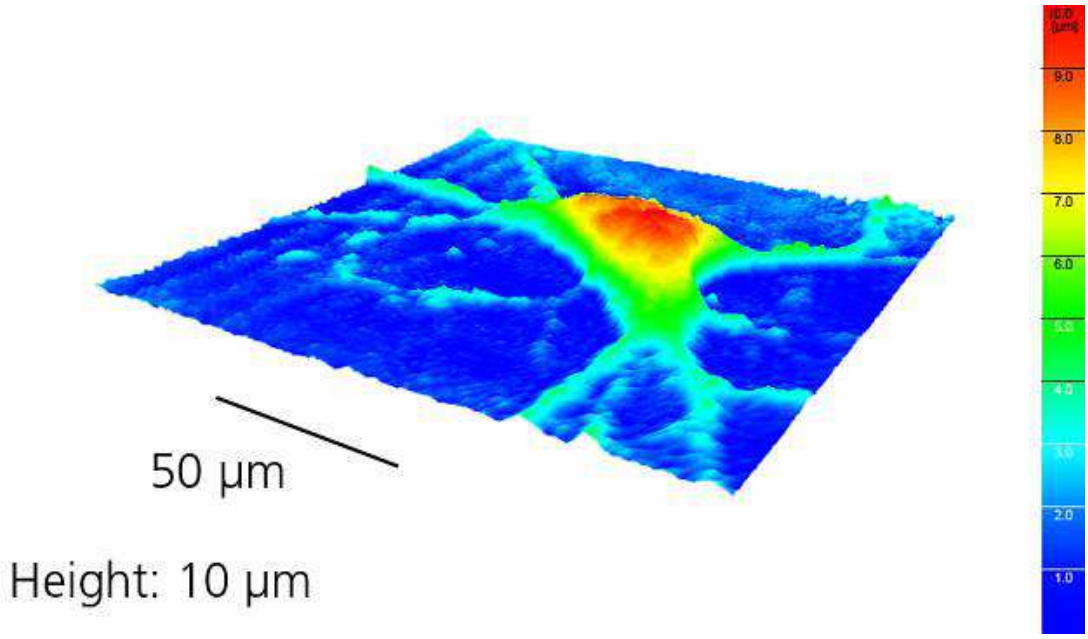

Figura 4.2.: Ejemplo de una imagen obtenida usando MHD.

5. Inicio/gestación

5.1. Sobre el inicio

5.2. Comprensión de los requisitos

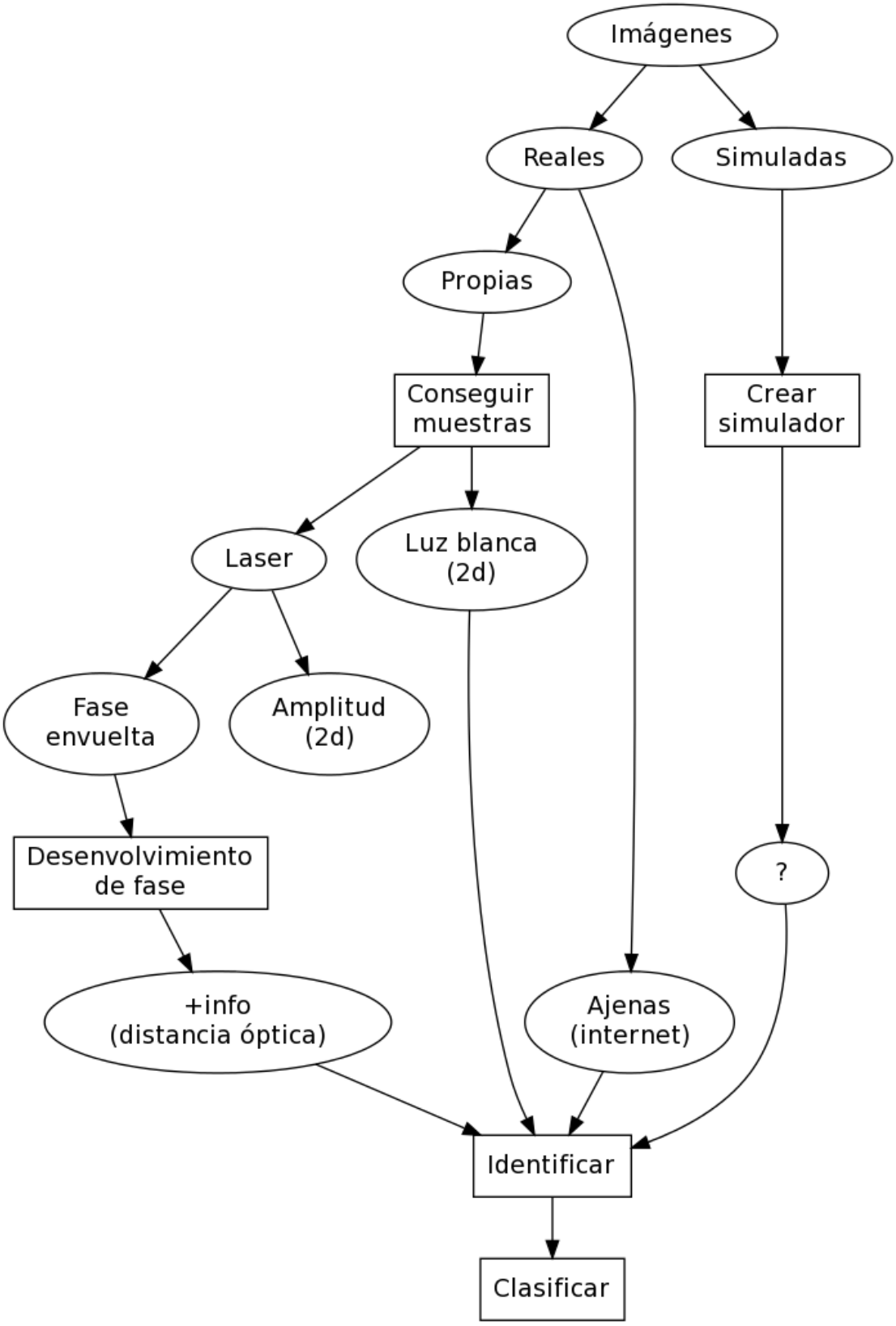

Figura 5.1.: Primer análisis de requisitos

6. Elaboración

6.1. Sobre la elaboración

6.2. Iteración 1

6.2.1. Demostración de avances

6.2.2. Actualización de requisitos

6.3. Iteración 2

6.3.1. Demostración de avances

6.3.2. Actualización de requisitos

6.4. Iteración 3

6.4.1. Demostración de avances

6.4.2. Actualización de requisitos

Al mostrar el programa que aplica los filtros este falló debido a cambios introducidos a ultimo momento mientras se intentaba acotar el problema de conversión de imágenes a partir de matrices de coma flotante de más de 32bits de precisión.

Se mostró la funcionalidad del programa de auto mascara y se explicó el funcionamiento de los algoritmos que aplica. Esto es de mucho interés para Andrea ya que necesita la certeza de que se podrá confiar en el programa independientemente de la imagen que le sea pasada como argumento.

Se decidió re-implementar todo lo que se pueda migrar del sistema anterior al nuevo. Él que seguirá siendo necesario para acceder al hard de captura. Esto actualiza largamente las tareas de desarrollo añadiendo (en resumen):

- Simulación de un haz de referencia que se usará para proyectar digitalmente el holograma.
- Aplicación de la transformada de Fourier a la imagen proyectada. Cuidando centrar el origen de las frecuencias.
- Obtener el mapa de intensidad del array complejo resultante de la transformada de Fourier.
- Sobre el mapa de intensidad operar el algoritmo de creación automática de la mascara de orden 1 o -1. Debe poder producir mascaras:
  - circulares
  - ventanas de Hanning
  - definidas por el usuario
- Aplicar la mascara sobre la matriz compleja obtenida de la FFT.
- Centrar los resultados al origen.
- Propagar los valores.
- Aplicar la IFFT a la matriz resultante.

El uso de TKPipe no fue tan sencillo como en las pruebas. El principal problema fue la política de seguridad de Windows que impide a los programas hacer forks. Estos problema abrieron la discusión de una posible interfaz gráfica. Los usuarios remarcan que la prioridad del sistema es que funcione y que los aspectos estéticos sos totalmente innecesarios. Pero los varios fallos producidos por las diferencias entre el entorne de pruebas y el del operador refuerzan el deseo de contar con un marco único de desarrollo.

Un poco de investigación sobre este aspecto mostró un conjunto de tecnologías que permitirían crear un entorno unificado multi-plataforma.

El primer boceto de interfaz gráfica se puede ver en las figuras Figura 6.1 y Figura 6.2.

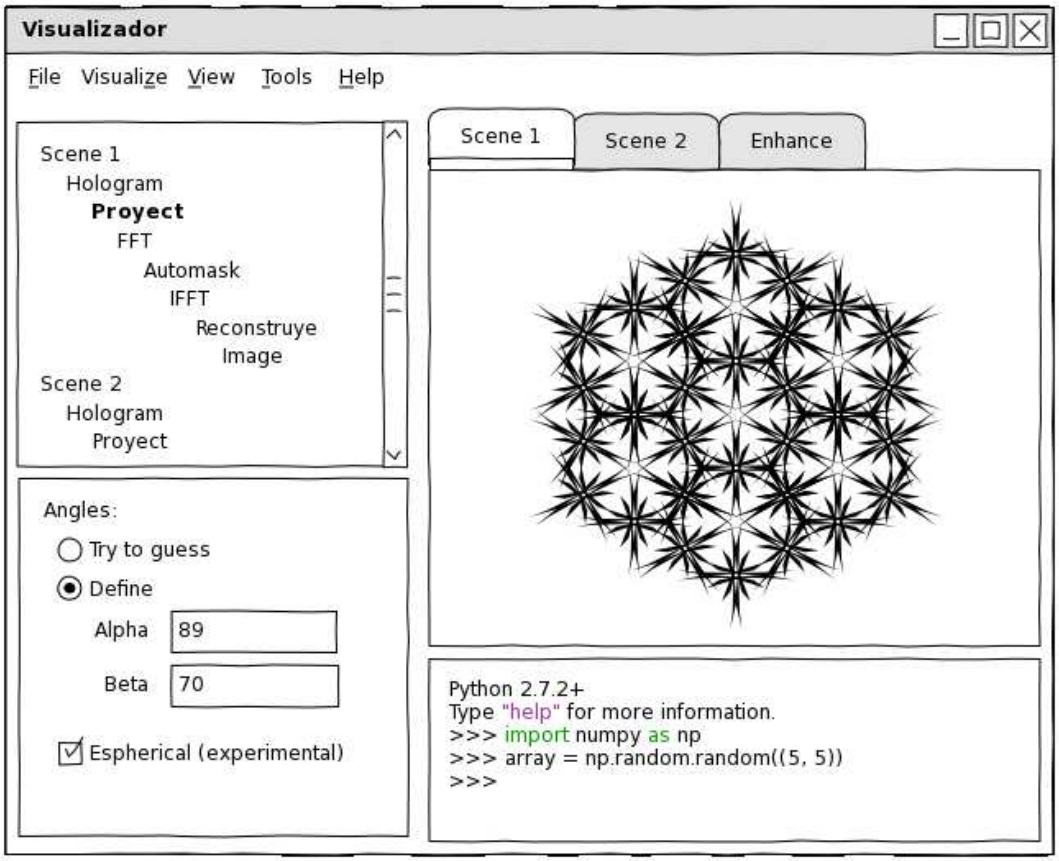

Figura 6.1.: Visualizador mostrando el editor de tubería (arriba a la izquierda), el editor de atributos (abajo a la izquierda), una pestaña de representación gráfica (arriba a la derecha) y la consola interactiva (abajo a la derecha).

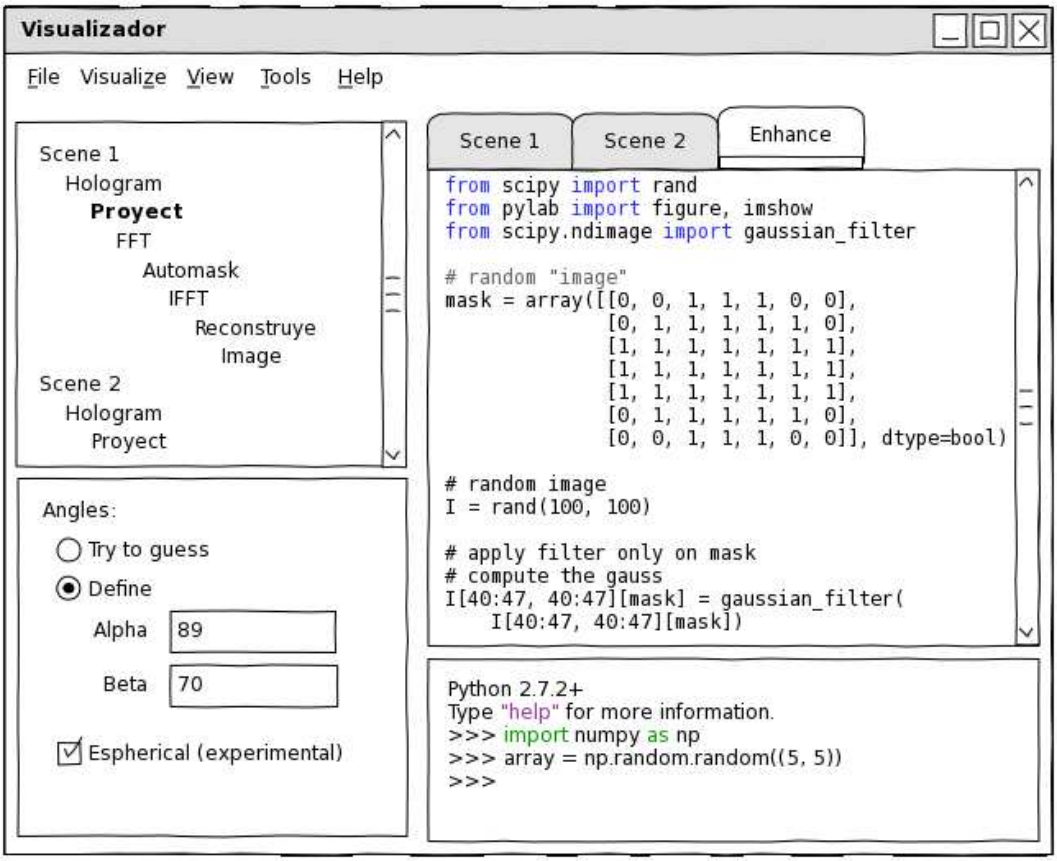

Figura 6.2.: Visualizador mostrando ahora una pestaña de edición de un script. En estos guiones se podrán importar los filtros diseñados para combinarlos en nuevos flujos de trabajo. Deberán poder acceder al entorno compartido de la aplicación.

7. Finalización

7.1. Documentación

7.2. Mantenimiento

Parte III.

Conclusiones

8. Objetivos conseguidos

9. Dificultades encontradas

10. Observaciones

Reconocimientos

Thank you, thank you, thank you

A. Lenguajes funcionales

A.1. Repaso

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla roughness parameter bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla, see.

A.2. Otros

Bibliografía

[Aus96] Robert D. Austin. *Measuring and Managing Performance in Organizations*. Dorset House Publishing, New York, 1996.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, 2Âª edition, 2004.

[Coc06a] Alistair Cockburn. *Agile software development: the cooperative game*. Agile Software Development. Addison-Wesley Professional, 2006.

[Coc06b] Alistair Cockburn. Use case fundamentals, 5/10/ 2006. http://alistair.cockburn.us/Use+case+fundamentals.

[Fow] Martin Fowler. La Nueva Metodología. *http://www.programacionextrema.org/articulos/newMethodology.es.html*.

[Gro95] The Standish Group. Chaos Report, 1995. http://www.cs.nmt.edu/cs328/reading/Standish.pdf.

[Ray01] Eric S. Raymond. How to become a hacker. *Retrieved December*, 4: 2005, 2001.

[Ree05] Jack W. Reeves. Code as design: Three essays. *Developer.**, 2005. What Is Software Design? originally published in C++ Journal, 1992.

Nomenclatura

ANSI	son secuencias de escape tratadas como carácteres embebidos in el texto usadas para controlar el formato, color y otras opciones de video en terminales de texto. Es ampliamente implementado en casi todos los sistemas operativos para mostrar las salidas estandares de los programas ejecutados.
Cascada	desarrollo en. También llamado modelo en cascada, es el enfoque metodológico que ordena rigurosamente las etapas de forma que el inicio de cada una espera a la finalización de la anterior.
duck-typing	En los lenguajes de programación orientados a objetos, se conoce como duck typing el estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica. El nombre del concepto se refiere a la prueba del pato, una humorada de razonamiento inductivo atribuida a James Whitcomb Riley que pudo ser como sigue: «Cuando veo un ave que camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato»
RAD	Desarrollo Rápido de Aplicaciones.
refactorizar	descartar el código antiguo y re-escribir un nuevo programa para la misma función pero evitando los antiguos errores de diseño.
Smalltalk	Lenguaje de programación que introdujo tempranamente las bases de la programación orientada a objetos y se considera su implementación de referencia. Es muy respetado en los ambitos academicos.
traceback	o retraza de pila: es una interfaz estándar para extraer, dar formato y presentar trazas de pila de progranas.
UML	Unified Modeling Language. Es el lenguaje de modelado más utilizado, permite visualizar, especificar, construir y documentar un sistema.