

Trabajo de Grado



Instituto Universitario Aeronáutico.

Facultad de Ingeniería.

Ingeniería en Sistemas.

Calidad en el Desarrollo de Software Científico.

Coenda Francisco Javier.

Asesor: Maller Patricio.

Derecho de Autor

Calidad en el Desarrollo de Software Científico by Francisco Javier Coenda is licensed under a Creative Commons Atribución-CompartirDerivadasIgual 3.0 Unported License.



Para ver una copia de esta licencia, visitar <http://creativecommons.org/licenses/by-sa/3.0/>

Dedicatoria

Este proyecto esta dedicado a mi padres y hermana.

Agradecimiento

Agradezco a mi familia por haberme acompañado a lo largo de este proyecto en el cual me embarqué con la finalidad de forjar un futuro para mí. También a mis amigos y compañeros por haber formado parte de este viaje. Y agradezco a Patricio Maller por haber sido un soporte y guía en esta etapa que me encuentro transitando.

Calidad en el Desarrollo de Software Científico.

***Aprobado por el Departamento de Sistemas en cumplimiento de los
requisitos exigidos para otorgar el título de Ingeniero de Sistemas
Al Señor Coenda Francisco Javier – DNI 31.838.068.***

Revisado por:

.....
Mg. Maller Patricio
Tutor de Trabajo

.....
Lic. Salamon Alicia.
Directora Dpto. Sistemas

.....
Ing. Héctor Carlos Riso
Director Dpto. Desarrollo Profesional

Tribunal Examinador

.....
Lic. Salamon Alicia
Presidente del Tribunal Examinador

.....
Piccolotto Pablo
Vocal del Tribunal Examinador

Córdoba, 23 de Noviembre de 2011

Resumen

Los científicos se enfrentan a un cuello de botella en sus investigaciones, debido a demoras en los tiempos que necesitan para llevar a cabo el desarrollo de software científico. Esto se traduce, inevitablemente en una reducción del tiempo asignado para llevar adelante la investigación. Además, se suma el hecho de que sufren cada vez más presión con los tiempos, para entregar resultados visibles con mayor celeridad y con reducción de costos en las investigaciones.

A través de una investigación teórica, se pretende demostrar cuáles son las problemáticas que afrontan los científico en el desarrollo de software científico, al momento de llevar a cabo dicho proceso en las investigaciones científicas. También se buscó determinar qué rol ocupa la Ingeniería de Software, dentro del desarrollo de software científico en las investigaciones.

Actualmente, existen en el mercado de desarrollo de software, sistemas ágiles que permiten llevar un buen ritmo de desarrollo de productos software. A través de la selección de algunos patterns de software que ofrece SCM, se busca ofrecer una pequeña solución que mejore la situación que atraviesan los investigadores científicos al llevar a cabo el desarrollo de programas científico. También, se busca dejar planteado un camino a seguir para futuras investigaciones que puedan contribuir a mejorar la problemática abordada en este proyecto.

Índice

Derecho de Autor.....	i
Dedicatoria.....	ii
Agradecimiento.....	iii
Resumen.....	vi
Índice.....	7
Índice de Ilustración.....	10
1. Introducción.....	13
1.1. Antecedentes.....	13
1.2. Situación Problemática.....	14
1.3. Problema.....	15
1.4. Objeto de Estudio.....	15
1.5. Campo de Acción.....	16
1.6. Objetivos.....	16
1.6.1. Objetivo general.....	16
1.6.2. Objetivos específicos.....	16
1.7. Idea a Defender / Propuesta a Justificar / Solución a Comprobar.....	17
1.8. Delimitación del Proyecto.....	17
1.9. Aporte Práctico.....	18
1.10. Aporte Teórico.....	18
1.11. Métodos de Investigación	19
2. Marco Contextual.....	21
2.1. Low Hang Fruit y SCM.....	21
2.2. Análisis de los problemas observados.....	23
2.3. Antecedentes de proyectos similares.....	33
3. Marco Teórico.....	35
3.1 Patterns.....	35
3.2. Patrones a Patrones de SCM.....	39
3.3. Patrones Seleccionados.....	41
3.3.1. Active Development Line.	41

3.3.1.1. Contexto.....	41
3.3.1.2. Problema.....	43
3.3.1.3. Posibles Soluciones.....	43
3.3.1.4. Solución.....	44
3.3.1.5. Consideraciones.....	45
3.3.1.6. Beneficios Esperados.....	46
3.3.2. Integration Build.....	46
3.3.2.1. Contexto.....	47
3.3.2.2. Problema.....	48
3.3.2.3. Solución.....	48
3.3.2.4. Consideraciones.....	50
3.3.2.5. Beneficios Esperados.....	51
4. Modelo Teórico.....	52
4.1. Active Development Line.....	52
4.1.1. Nota de Liberación.....	53
4.1.2 Tags.....	54
4.2. Integration Build.....	55
4.2.1. Workspace Integrador.....	57
4.2.2. Tiempo de Integración	58
5. Concreción del Prototipo.....	59
5.1. Implementación de una línea activa de desarrollo.....	59
5.2. Implementación de Integration Build.....	68
6. Conclusiones.....	72
7. Referencia Bibliográfica.....	73
8. Bibliografía.....	75
9. Glosario.....	80
10. Anexo.....	81
10.1. Soluciones.....	81
10.1.1. Inicialización del repositorio.	81
10.1.2. Recomendaciones para utilizar Git.....	91

10.1.3. Rollback con Git.....	93
10.1.4. Requisitos para implementar Integration Build.....	96
10.1.5. Recomendaciones al trabajar con Integration Buil.....	96
10.2. Presentación.....	96
10.2.1. Speech para el Workshop.....	96
10.2.2. Pasos de la Demostración del Workshop.....	98
10.2.3. Filminas de la presentación del Workshop.....	99
10.3. Validación de la Propuesta.....	114

Índice de Ilustración

Ilustración 1 – Distribución del tiempo que posee un Desarrollador Científico.....	13
Ilustración 2 – Cómo se manifiesta la carencia de herramientas en el desarrollo de software.....	14
Ilustración 3 – Actividades que componen la Gestión de Configuración.....	15
Ilustración 4 – Comunidad Científica.....	18
Ilustración 5 – Transferencia de conocimiento desde Ingeniería de Software al Campo de Investigación Científica.....	19
Ilustración 6 – Gráfica de Criticidad y Esfuerzo.....	21
Ilustración 7 – Posicionamiento de SCM en la gráfica de Criticidad - Esfuerzo.....	22
Ilustración 8 – Distribución de requisitos según desarrolladores científicos.....	23
Ilustración 9 – Distribución de requisitos según Ingeniería de Software.....	24
Ilustración 10 – Dominio del Problema comprendido solamente por los científicos.....	25
Ilustración 11 – Ámbitos según científicos	26
Ilustración 12 – Herramientas usadas por los científicos.....	27
Ilustración 13 – Curva de Aprendizaje.....	28
Ilustración 14 – Falta de Reutilización de componentes y otros elementos en el Software Científico.	29
Ilustración 15 – Test que realizan los científicos.....	30
Ilustración 16 – Cómo entienden la performance los científicos.	31
Ilustración 17 – Software descartado cuando finaliza la investigación científica.....	32
Ilustración 18 – Formas en que los científicos adquieren conocimientos para el desarrollo de software.....	33
Ilustración 19 – Complementación de Pattern.....	35
Ilustración 20 – Generación de Pattern Language.....	35
Ilustración 21 – Problema Recurrente.....	36
Ilustración 22 – Pattern Language y Patterns seleccionados.....	37
Ilustración 23 – Formas de aplicar los patterns.....	38
Ilustración 24 – Cómo trabaja el Lenguaje de Patterns.....	39
Ilustración 25 – Modificación y rotura del main line.....	41

Ilustración 26 – Puntos de Sincronización en el main line.....	42
Ilustración 27 – Main Line con múltiples ramificaciones.....	43
Ilustración 28 – Balance entre estabilidad y progreso.....	44
Ilustración 29 – Gráfica de una Active Development Line.....	45
Ilustración 30 – Complicación al momento de realizar una integración.....	47
Ilustración 31 – Nuevo componente que genera un problema en el main line.....	48
Ilustración 32 – Proceso de Integration Buil.....	49
Ilustración 33 – Marcadores de las construcciones realizadas del proyecto.....	50
Ilustración 34 – Error en el Main Line.....	52
Ilustración 35 – Notas de liberación o lanzamiento.....	53
Ilustración 36 – Tag de los puntos OK de chequeo.....	54
Ilustración 37 – Error al momento de integrar, al main line, los componentes.....	56
Ilustración 38 – Workspace que contiene los componentes a ser integrado.....	57
Ilustración 39 - Cómo funciona comando git add.....	59
Ilustración 40 - Tomar una snapshot del archivo modificado.....	60
Ilustración 41 - Cómo funciona comando git commit.....	61
Ilustración 42 - Editor de texto del commit.....	62
Ilustración 43 - Opciones de guardado del commit.....	63
Ilustración 44 - Fin del proceso de commit y reporte del commit que se ha realizado.....	64
Ilustración 45 - Utilización del comando git tag.....	64
Ilustración 46 - Creación de tag y corroboración del tag creado.....	65
Ilustración 47 - Utilización de git merge.....	65
Ilustración 48 - Ejecución de la operación merge.....	67
Ilustración 49 - Corroboración de que la operación merge.....	68
Ilustración 50 - Se inicializa el repositorio.....	82
Ilustración 51 - Mensaje al finalizar la inicialización del repositorio y comprobación de la creación de éste.....	83
Ilustración 52 - Inicialización. Comando git add.....	84
Ilustración 53 - Snapshot inicial del proyecto.....	84
Ilustración 54 - Inicialización. Comando git commit.....	85

Ilustración 55 - Commit del snapshot inicial del proyecto.....	86
Ilustración 56 - Inicialización. Comando branch.....	86
Ilustración 57 - Creación de la rama trabajo.....	87
Ilustración 58 - Corroboración de que se creó la rama y sobre qué rama se está posicionado.....	89
Ilustración 59 - Cambiar de rama y comprobación de que se cambió de rama.....	90
Ilustración 60 - Agregando archivos modificados por separados. Imagen 1.....	92
Ilustración 61 - Agregando archivos modificados por separados. Imagen 2.....	93
Ilustración 62 - Ejecución del rollback de una modificación.....	94
Ilustración 63 - Usando comando log para determinar la versión y valor del commit.....	94
Ilustración 64 - Ejecutando el rollback.....	95
Ilustración 65 - Corroborando que se retornó a la versión anterior.....	95

1. Introducción

1.1. Antecedentes

Este proyecto sobre metodología de desarrollo de software en el ámbito científico, surge de necesidades puntuales en grupos de Investigación y Desarrollo en el Instituto Universitario Aeronáutico, que podría canalizarse a través de un trabajo de grado.

Este proyecto abordará dificultades típicas en el desarrollo de software científico, con las particularidades que lo diferencian del desarrollo de software tradicional.

Los grupos de Investigación y Desarrollo que producen software científico, deben destinar parte de su valioso tiempo de investigación (1) al desarrollo, mantenimiento y modificación del software científico (2), careciendo de las técnicas e infraestructura (3) para hacerlo al nivel de perfección y calidad del software comercial.

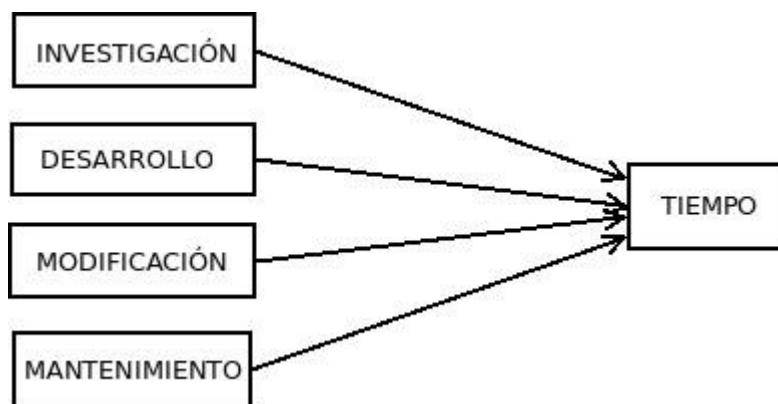


Ilustración 1 – Distribución del tiempo que posee un Desarrollador Científico.

A su vez, estos grupos experimentan presiones crecientes en presupuesto y exigencias (4), para mejorar tiempos de entrega y calidad de los resultados.

No se pretende brindar una solución de fondo a la problemática planteada, sino introducir mejoras a través de las herramientas de la Ingeniería de Software, para la situación que atraviesa la Investigación Científica en este ámbito.

1.2. Situación Problemática

El desarrollo de software científico se ve afectado por una serie de factores, los cuales llevan a que se generen problemas de distinta índole en esta área.

El desarrollo de software científico es llevado adelante por los propios científicos/investigadores (2), detectándose una escasa integración de los ingenieros de software en los equipos de trabajo (3).

Los científicos, al no poseer conocimientos formales sobre Ingeniería de Software (3) (5), carecen de las herramientas para afrontar los requerimientos de mejora en tiempos de entrega, que necesitan para continuar con sus actividades. La carencia de instrumentos se manifiesta en software monolíticos, difíciles de mantener en el tiempo y complicados de modificar si hiciere falta.

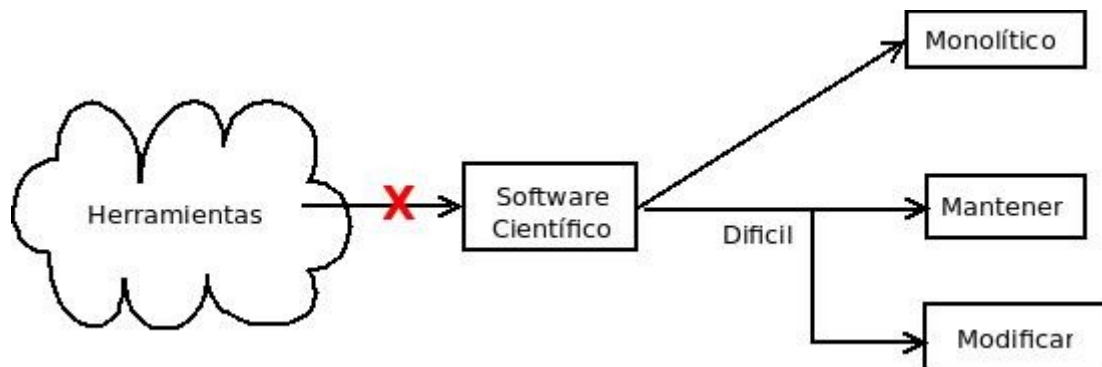


Ilustración 2 – Cómo se manifiesta la carencia de herramientas en el desarrollo de software.

Además, se detectó un escaso uso de herramientas de desarrollo de software, siendo los IDEs la única herramienta de mayor uso extendido entre los desarrolladores de software científico (6).

La pobre o nula existencia de documentación formal (6) del software científico, dificulta la continuación de éste en el tiempo con equipos de trabajo cambiante.

Tras un resumen de los distintos factores que afectan el desarrollo de software científico, quedan de manifiesto los distintos problemas que se presenta en dicha área de desarrollo.

1.3. Problema

Los científicos cuentan cada vez con menos tiempos para entregar resultados visibles y operativos a la comunidad (4). Esto, sumado a la falta de herramientas de Ingeniería de Software que poseen los desarrolladores de software científicos, hace necesario la introducción de cambios que mejoren el proceso de desarrollo de software.

1.4. Objeto de Estudio

El objeto de estudio para este proyecto es la Gestión de Configuración en el ámbito del desarrollo de software científico.

La Gestión de Configuración es una actividad protectora que se aplica a lo largo del proceso de software. Esta se compone de un conjunto de actividades que se han desarrollado, a fin de gestionar el cambio a lo largo del ciclo de vida del software.



Ilustración 3 – Actividades que componen la Gestión de Configuración (18).

La Gestión de Configuración se considera como una actividad de aseguramiento de la

calidad que se aplica a lo largo del proceso de desarrollo, con retorno en la inversión inmediata y una amplia aplicabilidad en diferentes dominios (7).

1.5. Campo de Acción

El campo de acción se centra en el estudio de patrones del Software Configuration. Específicamente, se selecciona un patrón del Software Configuration, que, aplicado al desarrollo de software científico, mejore calidad y tiempos de entrega.

1.6. Objetivos

1.6.1. Objetivo general

El objetivo que se persigue con este proyecto es explorar la aplicación de la disciplina Ingeniería de Software al desarrollo de software científico, con la finalidad de identificar patrones de configuración que podrían ayudar a mejorar el tiempo de desarrollo de software científico y la productividad de los desarrolladores de software científico.

1.6.2. Objetivos específicos

- Identificar las necesidades metodológicas en el desarrollo de software científico.
- Caracterizar el desarrollo del software científico.
- Identificar los diferentes aspectos de la Gestión de Configuración que podrían utilizarse en el desarrollo de software científico.
- Analizar patrones de configuración.
- Seleccionar un patrón de configuración.
- Analizar dicho patrón de configuración en el contexto del proyecto.

-
- Adaptar dicho patrón al desarrollo de software científico en el contexto del proyecto.
 - Verificar el patrón propuesto con una implementación.
 - Analizar los resultados.
 - Generar conclusiones.

1.7. Idea a Defender / Propuesta a Justificar / Solución a Comprobar

En el presente trabajo se trata de definir y aplicar un patrón de configuración al desarrollo de software científico. El objetivo que se persigue no solo es mejorar la eficiencia del desarrollo, sino también mejorar la productividad de los desarrolladores de software científico. De esta manera, se intenta una mejora en los tiempos de desarrollo en software científico.

1.8. Delimitación del Proyecto

Se realiza un estudio del estado del arte en la temática y una implementación de una solución en el ámbito científico – técnico utilizando herramientas, como repositorios de datos y scripts entre otros. Queda fuera del alcance de este proyecto la verificación empírica de la propuesta.

1.9. Aporte Práctico

Se espera que los resultados de este proyecto tengan un impacto directo dentro y fuera de la comunidad del I.U.A., beneficiando a los desarrolladores de software científico.



Ilustración 4 – Comunidad Científica.

Se pretende mejorar la eficiencia de los equipos de desarrollo de software científico, así como también incrementar el nivel de productividad de éstos.

1.10. Aporte Teórico

Transferencia de conocimientos de la disciplina Ingeniería de Software al desarrollo de software científico. Transferir conocimientos que sean sencillos de aplicar y que posean un alto impacto en la eficiencia del desarrollo de software científico, así como en los niveles de productividad.

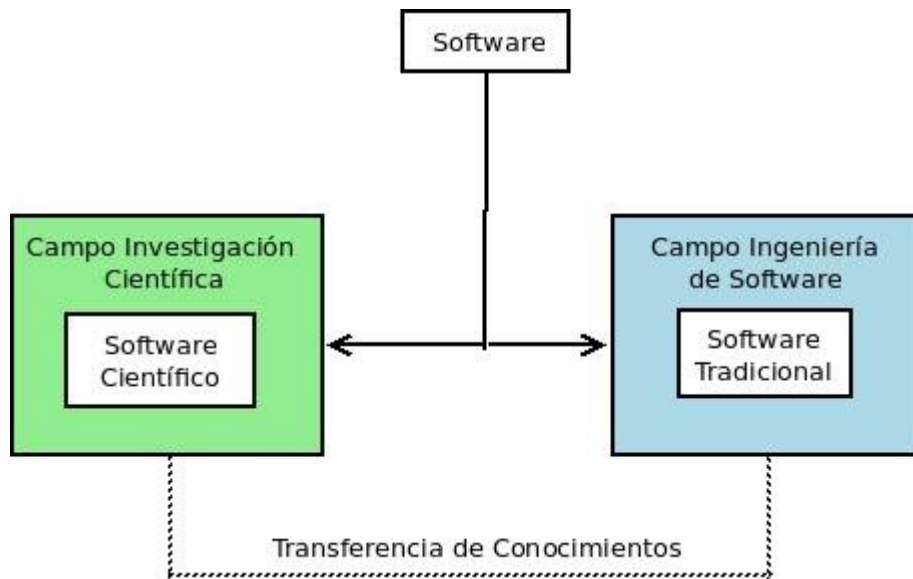


Ilustración 5 – Transferencia de conocimiento desde Ingeniería de Software al Campo de Investigación Científica.

Se plantea la aplicación de algunos patrones seleccionados al desarrollo de software científico. Por lo tanto, se deja la puerta abierta para seguir realizando investigaciones, ya sea sobre esta misma línea de trabajo o sobre otra línea de la Ingeniería de Software.

1.11. Métodos de Investigación

La metodología de investigación utilizada es un análisis teórico, con una validación de la propuesta a través de una demostración.

1. Exploración de papers, que permitan determinar el estado del arte en el dominio del desarrollo de software científico – técnico, para la identificación de los factores problemas.
2. Exploración de posibles patrones de configuración.
3. Elaboración de la solución.
4. Prototipado de solución.
5. Demostración de la solución construida.

6. Relevamiento y análisis de datos.

2. Marco Contextual

2.1. Low Hang Fruit y SCM.

El desarrollo de software científico reporta una multiplicidad de problemas ya resueltos, en el marco de la Ingeniería de Software (documentación, testing, gestión de configuración, entre otros). Estos problemas pueden clasificarse rápidamente por su criticidad y esfuerzo requerido para solucionarlos, es decir, por el retorno en la inversión requerida para solucionarlos.

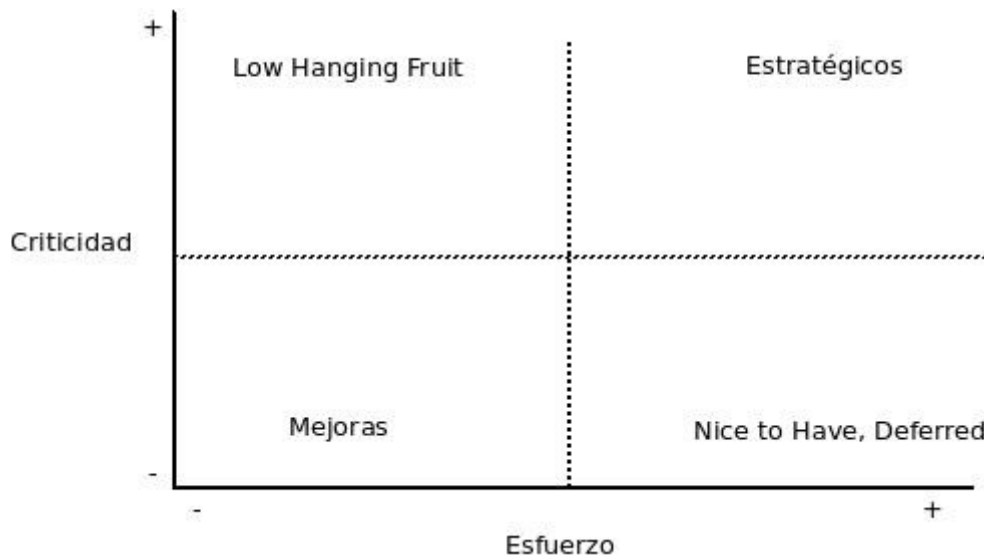


Ilustración 6 – Gráfica de Criticidad y Esfuerzo.

SCM es una de las tantas soluciones (herramienta) que brinda la Ingeniería de Software y que permite atacar muchos de los problemas que reporta el desarrollo de software científico, con poco esfuerzo.

SCM se basa en un conjunto de prácticas, muchas veces no usadas por los desarrolladores de software, a pesar de que algunas de ellas existen desde hace tiempo en el ámbito de la Ingeniería de Software, que ayudan a incrementar la productividad con muy poco

esfuerzo.



Ilustración 7 – Posicionamiento de SCM en la gráfica de Criticidad - Esfuerzo.

Muchas organizaciones suelen implementar incorrectamente SCM o directamente no lo hacen, creyendo que su implementación va a insumir grandes esfuerzos o costos monetarios.

Para poder usar correctamente SCM, en primera instancia, se debe tomar en cuenta que está compuesto de técnicas y procesos. Por lo tanto, al momento de implementar SCM o algunas técnicas o procesos de SCM, se debe considerar que puede llegar a verse influenciada por algunos aspectos, tales como la estructura de la organización, la arquitectura o las herramientas, entre otras. Si esto no es tomado en cuenta, la aplicación de SCM fracasa o genera serios problemas tanto en el desarrollo de software como a nivel organizacional.

SCM es una herramienta de suma utilidad que se aplica al desarrollo de software, especialmente en ambientes de desarrollos ágiles, ya que requiere de muy poco esfuerzo para ser implementada y permite resolver problemas críticos.

2.2. Análisis de los problemas observados.

El desarrollo de software científico se encuentra influenciado por una serie de factores que dan lugar a problemas de diversa índole. Esto hace necesario que se deba buscar cuáles son estos factores y de qué forma impactan en el desarrollo del software científico.

Al analizar el desarrollo de software científico se observa que carece de documentación, formal o informal, donde se detallan los requisitos que se desarrollaron en el pasado hasta el presente y los que vendrán. Esto se debe a que, a medida que la investigación avanza, van surgiendo distintas necesidades. Algunas de estas dan lugar a los requisitos de software (8).

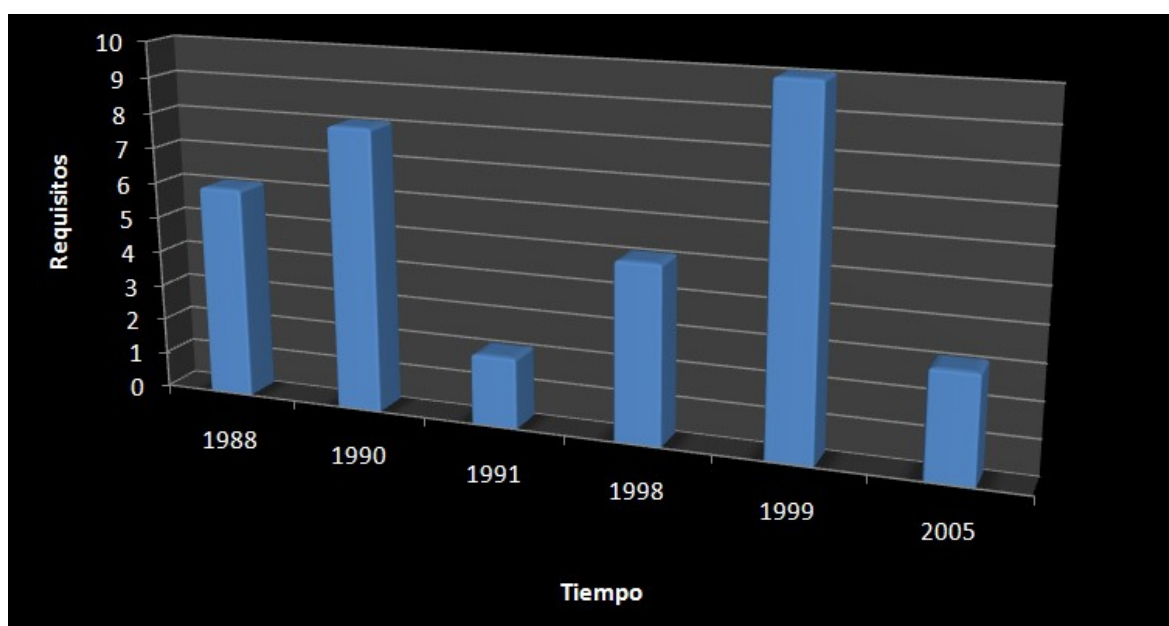


Ilustración 8 – Distribución de requisitos según desarrolladores científicos.

Esta forma que poseen los desarrolladores de software científico de manejar los requisitos de software, contrasta con la forma de manejarlos en la Ingeniería de Software. Esta establece que, antes de empezar el desarrollo de software, es necesario tener documentados los requisitos de software.

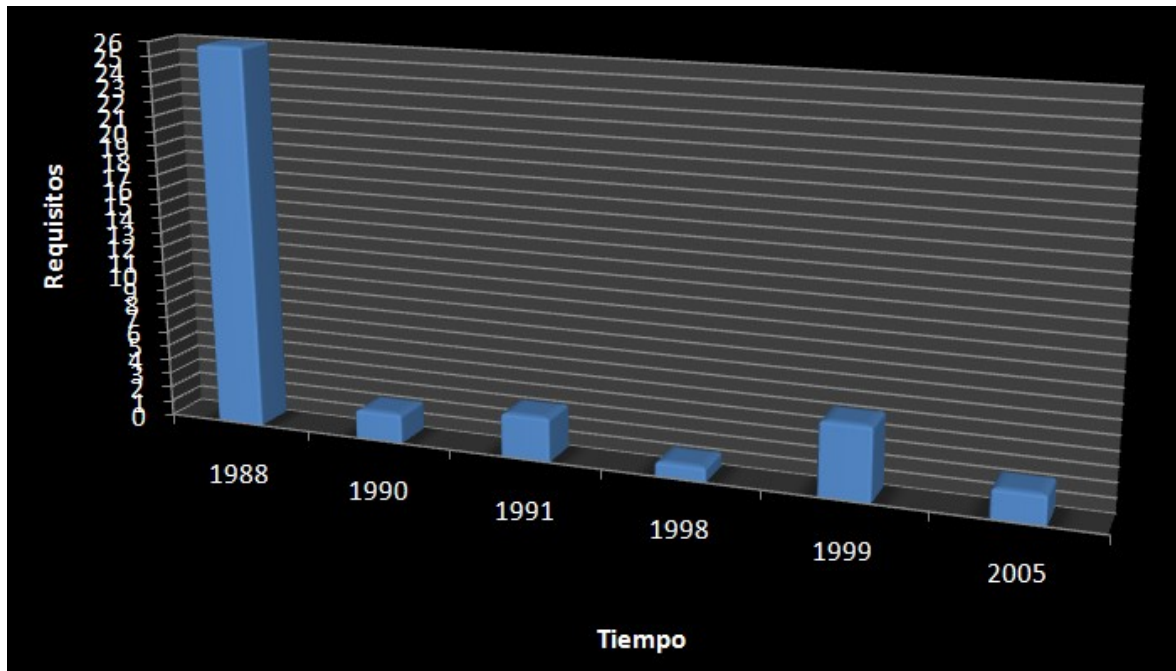


Ilustración 9 – Distribución de requisitos según Ingeniería de Software.

La falta de documentación sobre las necesidades de software, hace que los desarrolladores de software científico no sepan con exactitud qué requisitos se han implementado y cuáles permanecen pendientes (8).

Tampoco existe una descripción del dominio de trabajo. Esto hace que solamente aquellas personas que se encuentran en contacto directo con éste, sepan cuáles son las necesidades concretas (8) del software científico. En este caso, serían los desarrolladores de software científico.

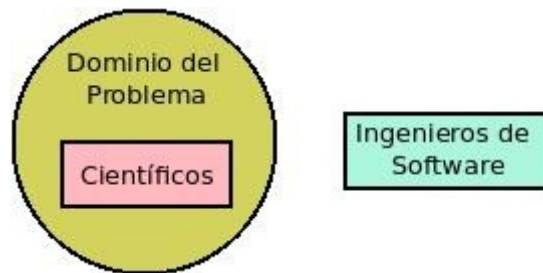


Ilustración 10 – Dominio del Problema comprendido solamente por los científicos.

Una de las razones por la que los desarrolladores de software científico obvian la documentación, es que llevar a cabo dicha tarea consume tiempo y esfuerzo. Por otra parte, poseen la apreciación de que el software que ellos desarrollan no sale del ámbito académico (9).

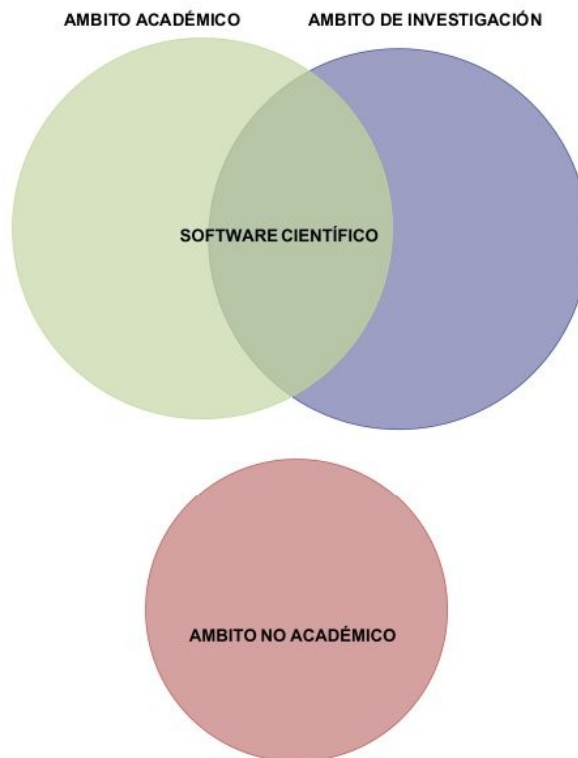


Ilustración 11 – Ámbitos según científicos .

Los desarrolladores de software científico, normalmente no utilizan las herramientas que ofrece la Ingeniería de Software, siendo los IDEs la herramienta de mayor uso entre ellos.

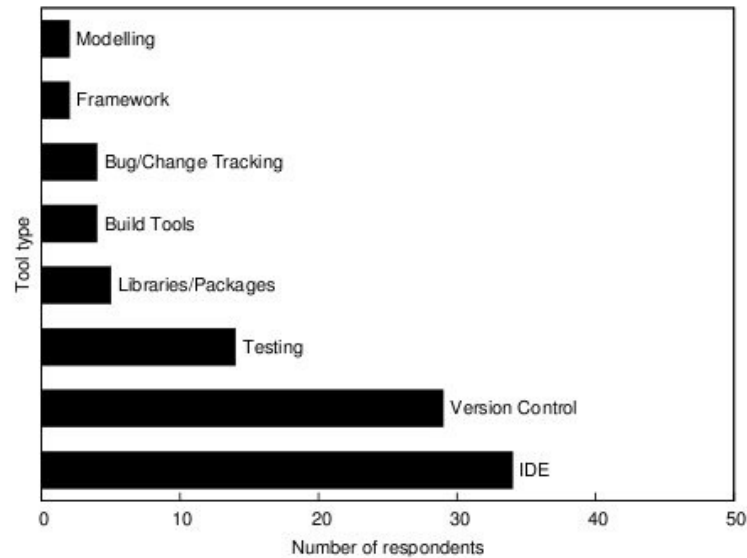


Ilustración 12 – Herramientas usadas por los científicos. (6)
(Encuesta realizada a una pequeña muestra de científicos)

Sin embargo, tratan de evitar su uso, debido a que consideran que los IDEs son rígidos en su estructura de trabajo y, por lo tanto, limitan las tareas de desarrollo (9).

Debido a que el período de vida útil del software científico es largo, los desarrolladores de software científico son reacios a usar nuevas herramientas o prácticas de la Ingeniería de Software. Esto se debe a que no tienen la certeza de que dichas herramientas o prácticas perduren en el tiempo, como el código de software científico que ellos están desarrollando (11).

Para los desarrolladores de software científico, existe una curva de aprendizaje por cada nueva herramienta o práctica de Ingeniería de Software que implementan. Esto hace que deban destinar parte del tiempo que poseen al aprendizaje del nuevo elemento a implementar.

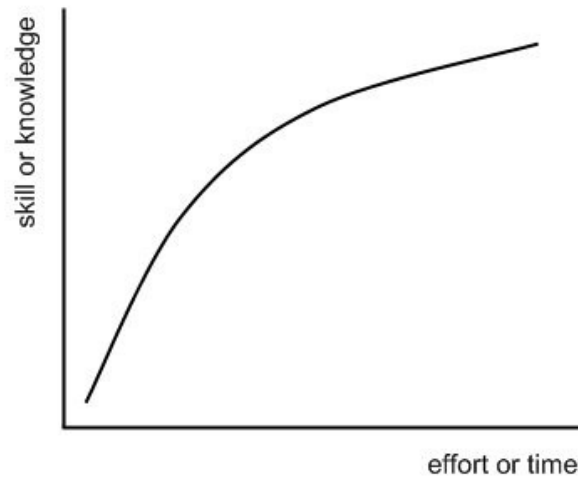


Ilustración 13 – Curva de Aprendizaje. (19)

Los desarrolladores de software científico reconocen la importancia de la reutilización. A pesar de esto, no lo aplican en los proyectos de desarrollo de software. En su lugar, desarrollan sus propios frameworks, una arquitectura totalmente nueva o el código del software científico.

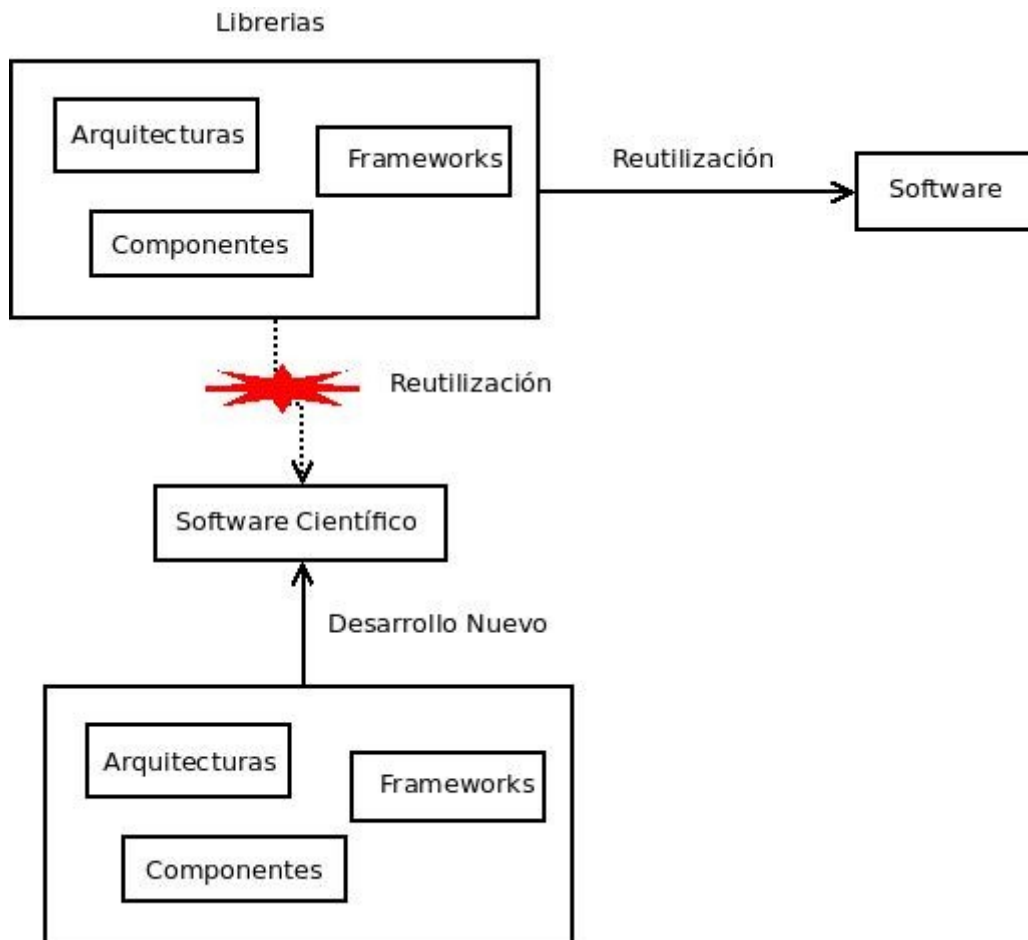


Ilustración 14 – Falta de Reutilización de componentes y otros elementos en el Software Científico.

Esto hace que los desarrolladores de software científico deban destinar parte de su tiempo a desarrollar una arquitectura, un framework o una pieza de código totalmente nuevos, siendo esta una actividad que requiere de un mayor esfuerzo que la reutilización (11).

Los investigadores hacen escaso uso de los testings. El test unitario es la actividad de testing más extendida entre los desarrolladores científicos, tendiente a comprobar el buen funcionamiento del software científico desarrollado. A pesar de la situación descrita, ellos llevan a cabo una serie de test en los software científicos. Sin embargo, estos test están destinados a verificar hipótesis, modelos de investigaciones y resultados de las distintos trabajos que llevan a cabo los científicos.

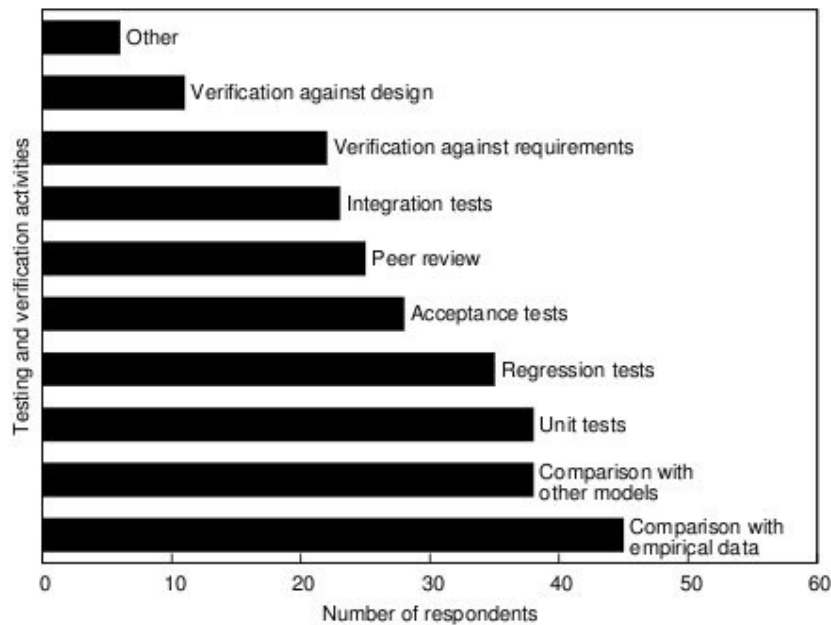


Ilustración 15 – Test que realizan los científicos. (18)
(Encuesta realizada a una pequeña muestra de científicos)

Otro tipo de verificación que los desarrolladores de software científico llevan adelante, es el de mostrar el producto desarrollado en funcionamiento a los interesados o usuarios finales (9), solamente exponiendo que produce la salida correcta.

La performance en el software científico es crucial. Sin embargo, los científicos entienden por performance no solamente un ahorro de tiempo, sino que también implica la obtención de resultados más fidedignos (11). Esta forma de comprenderla, plantea un serio problema para los desarrolladores de software científico, ya que cada mejora que logran en los tiempos, se traduce en una solicitud por parte de los científicos para que se agregue mayor funcionalidad al software científico, a fin de incrementar la fidelidad de los resultados que éste produce (11).

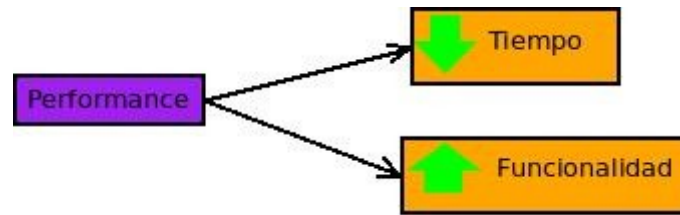


Ilustración 16 – Cómo entienden la performance los científicos.

Por este motivo, los desarrolladores de software científico se vuelcan a la utilización del lenguaje FORTRAN (11), en busca de que el software científico posea una buena performance. De esta manera, los desarrolladores de un programa científico pueden centrarse en funcionalidades que agreguen mayor fidelidad a los resultados que produce el software científico. También se encuentra un uso extendido del lenguaje C o C++ (11), por el mismo motivo descrito anteriormente.

Los desarrolladores de software científico no utilizan programación orientada a objetos, por lo tanto, el programa científico que desarrollan es monolítico (12). Esto hace que el software científico sea complejo y difícil de mantener en el tiempo. Por otra parte, estos programas científicos son prototipos o versiones betas (12). Esto se debe a que los científicos ven al software científico como una herramienta más, cuyo único propósito, al igual que el resto de herramientas que utilizan en sus investigaciones, es el de permitirles alcanzar los objetivos de sus investigaciones (10). Es por esto que normalmente el software científico suele ser descartado (12), ya que una vez finalizada la investigación, carece de utilidad.



Ilustración 17 – Software descartado cuando finaliza la investigación científica.

A pesar de la situación que se describe, muchas veces, dicho software científico sale del ámbito académico o de investigación en el que se encuentra circunscripto y termina convirtiéndose en un producto que necesita mantenimiento. Este seguimiento solamente puede ser dado por los desarrolladores de programas científicos, debido a que son los únicos que conocen cómo se ha desarrollado el mismo. Por otra parte, cuando el software científico es sacado del marco de trabajo para el cual fue creado, pierde utilidad (10). Esto se debe a que el programa científico es útil únicamente en el contexto de origen.

El hecho de que el software científico sea considerado por los científicos como algo descartable, hace que los ingenieros de software sean reticentes a trabajar con los científicos y desarrolladores de software científicos. Esto se debe a que los investigadores no reconocen la participación y aporte que los ingenieros de software realizan a la/s investigación/es que se llevan adelante (10).

Los desarrolladores de software científicos carecen de los conocimientos formales de Ingeniería de Software, más allá de que algunos de ellos han estado codificando por más de 20 años en diversos proyectos de investigación (11). La forma más común en que los desarrolladores de un programa científico adquieren los conocimientos para realizar desarrollo de software, es a través de otros desarrolladores de software científico que han estado codificando desde hace tiempo. También son autodidactas (11) (12). Esta forma de

adquirir los conocimientos contrasta mucho con la forma en que lo hacen los ingenieros de software, los cuales, a través de una educación formal, adquieren los conocimientos para llevar a cabo el desarrollo de software. Otra forma que poseen los desarrolladores de software científico para adquirir los conocimientos, es a través de cursos de capacitación. Sin embargo, esta última forma, no es ampliamente usada por ellos.

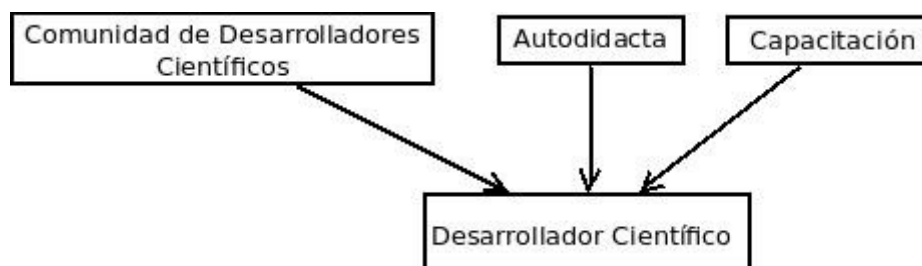


Ilustración 18 – Formas en que los científicos adquieren conocimientos para el desarrollo de software.

A través de un análisis se puede corroborar cuáles son los distintos factores que impactan en el desarrollo de software científico y que dan lugar a una serie de problemas, que han sido resueltos por la Ingeniería de Software.

2.3. Antecedentes de proyectos similares.

En el siguiente apartado, se explica brevemente un proyecto llevado adelante en el Reino Unido y que utiliza SCM.

La UK Met Office, en el año 2004, empezó a desarrollar el sistema Flexible Configuration Managment o FCM, que fue finalizado en el año 2006. Este proyecto fue llevado a cabo en forma conjunta por los científicos de la UK Met Office y un grupo de expertos en IT convocados por la UK Met Office. Los científicos de la UK Met Office, ya venían utilizando algunas prácticas de Ingeniería de Software, como así también venían aplicando algunos aspectos de SCM en los distintos proyectos de investigación que llevaban adelante. A pesar de haber implementado estas prácticas, surgían problemas de diversa índole. Esto se debía a que los distintos grupos de investigaciones que trabajan en la UK

Met Office, utilizaban distintos procesos y procedimientos cuando llevaban a cabo sus investigaciones y desarrollo de software, lo que dificultaba mucho el traspaso de información o la rotación de los científicos entre los distintos grupos de investigaciones. Esta situación entorpecía mucho el ingreso de nuevos miembros o la participación de colaboradores externos a los grupos de investigación que poseía la UK Met Office, ya que para poder ingresar, tenían que aprender los distintos procesos que existían en estos grupos de investigación, es decir, existía una curva de aprendizaje.

Ante esta situación, las autoridades de la UK Met Office decidieron desarrollar el Flexible Configuration Management. Este sistema simplifica la tarea de desarrollo de software actuando como una interface entre los científicos y los distintos software que se utilizan para llevar a cabo el desarrollo del software de simulación científica. Con este sistema, se logra mejorar los tiempos de desarrollo de software y unificar las distintas herramientas. Se mejoraron así los ciclos de lanzamiento y se logró reducir la curva de aprendizaje (22).

3. Marco Teórico

3.1 Patterns.

Los patterns son soluciones a problemas concurrentes que se presentan en un determinado contexto. Cada uno de estos pattern, individualmente, complementa a otro pattern.

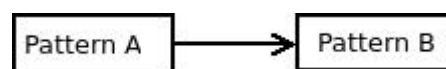


Ilustración 19 – Complementación de Pattern.

Este proceso, en el que los patterns se van complementando, da lugar a que se genere un lenguaje de patterns.

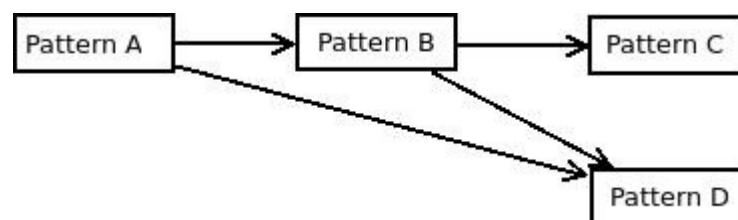


Ilustración 20 – Generación de Pattern Language.

La idea de los patterns y lenguaje de patterns, proviene de los trabajos del arquitecto Christopher Alexander. Este arquitecto se refiere a los patterns de la siguiente manera: “Los patterns describen problemas que ocurren una y otra vez en nuestro entorno. Estos patterns definen la solución a dicho problema y que puede ser aplicada un millón de veces, sin hacerlo dos veces de la misma manera.” (13).

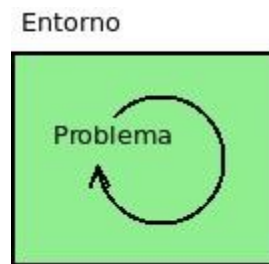


Ilustración 21 – Problema Recurrente.

La meta de los patterns es crear un lenguaje que ayude a los desarrolladores de software a resolver problemas que son recurrentes (14) en el desarrollo de software. El lenguaje de patterns, nos permite concatenar los distintos patterns a fin de poder obtener un mayor beneficio de la aplicación de ellos, en especial, para documentar la arquitectura de software (13) (14).

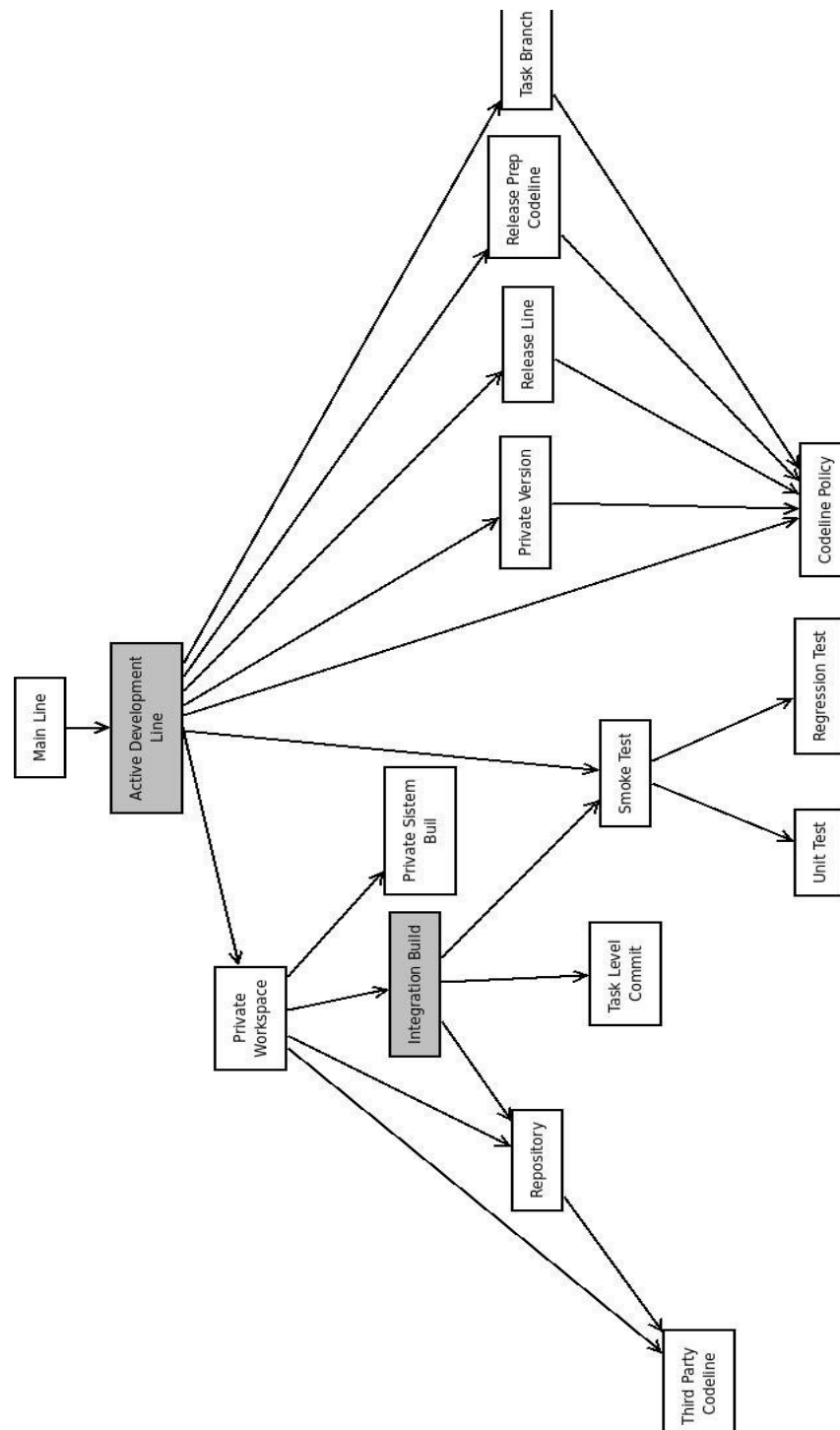


Ilustración 22 – Pattern Language y Patterns seleccionados.

Una buena arquitectura de software es aquella, que es adaptable y resiliente a los cambios (14). Los patterns son totalmente independientes de la herramienta que se utilice; hay herramientas que los soportan, mientras que otras no lo hacen, o solamente soportan algunos patterns. Aún aplicando los patterns de forma manual o rudimentariamente, se aportan beneficios al proyecto (13).

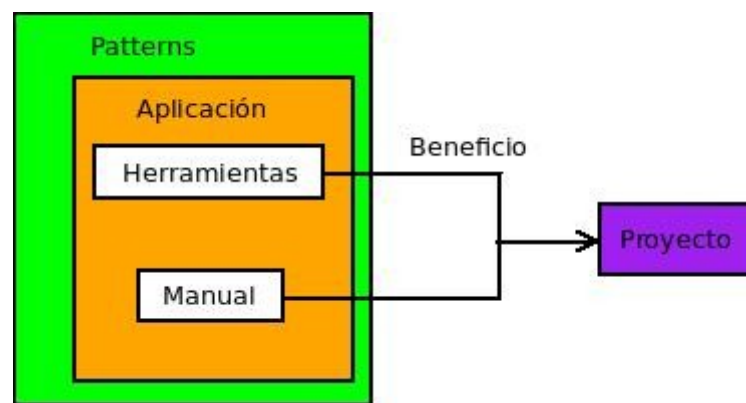


Ilustración 23 – Formas de aplicar los patterns.

Los patterns reciben nombres significativos, los cuales no dan una idea conceptual de cuáles son los problemas que abordan (14). Un buen pattern tiene los siguientes componentes:

- Resuelve un problema.
- Es un concepto probado.
- La solución que ofrece no es obvia.
- Describe una relación.
- Posee un componente significativamente humano (14).

A estas características que debe reunir un buen pattern, se le deben sumar los siguientes conceptos:

- Útil (nos muestra cómo esta instancia del pattern se materializa en el mundo real).
- Que se pueda usar (transforma la literatura del pattern en un concepto que reside en nuestra mente).

- Usado (de esta forma se aplica al mundo real) (14).

El lenguaje de patterns es una colección de patterns que trabajan en conjunto para resolver un problema complejo. Esto sería imposible de realizar utilizando solamente un pattern.

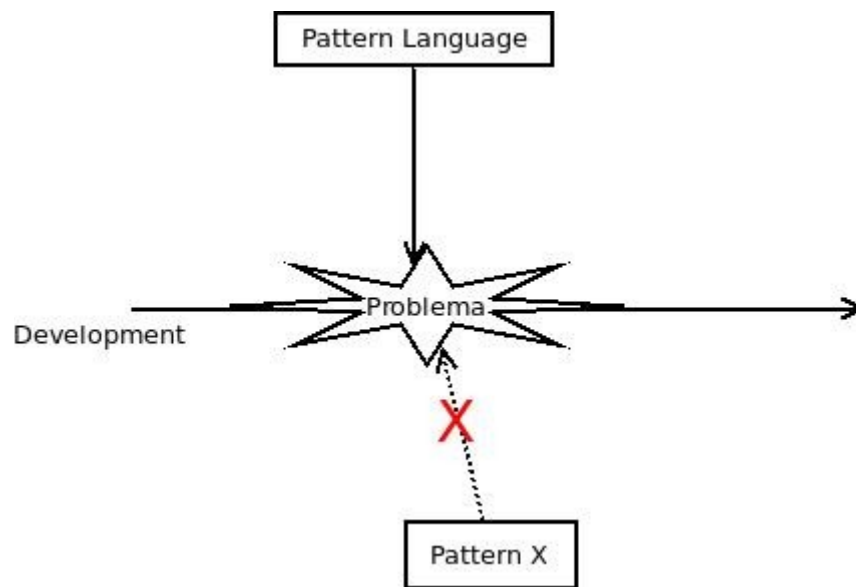


Ilustración 24 – Cómo trabaja el Lenguaje de Patterns.

El lenguaje de pattern incluye reglas y guías, que indican cuándo y cómo deben ser aplicados los patterns (14).

3.2. Patrones a Patrones de SCM.

Con el advenimiento de las metodologías ágiles, dentro de la ingeniería de software, el uso de patrones ha excedido el ámbito de la arquitectura de software para pasar a ser utilizados como herramientas para representar procesos de desarrollo, teamwork, build management, review, entre otros. Esto ha llevado a que, a pesar de que las herramientas simplifican mucho los procesos, no sea suficiente para lograr organizaciones exitosas en el desarrollo de software, y se hace necesaria la aplicación de patrones para SCM.

Básicamente, en cualquier área donde existan procesos y comportamientos que involucran

soluciones repetitivas y en distintos grupos de trabajo, los patrones son una poderosa herramienta de comunicación que permite mejorar la eficiencia de los grupos de trabajo. Esto se debe a que los patrones son particularmente buenos para describir procesos y determinados aspectos de ciertas prácticas de ingeniería de software, con el fin de que sean utilizadas eficientemente por los grupos de trabajo al momento de aplicar SCM.

En el año 2002, Steve Berczuk y Brad Appleton, publicaron un libro fundacional en cuanto a patrones de SCM, que pasó a convertirse en una de las bases de la Ingeniería de Software, especialmente en los ambientes de desarrollo ágiles. Este libro presenta el concepto de patrones, desde su surgimiento hasta nuestros días, pasando por autores que ya habían comenzado a realizar algunos planteamientos sobre la temática. Esta bibliografía nos presenta 16 patrones de SCM, que van desde la utilización de repositorios, hasta las estrategias de release management, pasando por test, espacios de trabajo, etc. A su vez, se explica cómo surgieron los patrones, los distintos trabajos que contribuyeron a que se crearan, así como conceptos de Ingeniería de Software que son base fundamental para el entendimiento de los patrones de SCM.

La estructura de cada patrón, que se encuentra desarrollada en este libro, se compone de la siguiente forma:

- Un título.
- Imágenes.
- Una descripción de contexto del patrón.
- El problema que resuelve el patrón.
- Un resumen de la solución aportada.
- Una descripción detallada de la solución.
- Una descripción de los problemas no resueltos y cómo pueden ser resueltos.

A través de este texto, escrito por Steve Berczuk y Brad Appleton, se deja plasmados a los patrones de SCM como representaciones de procesos. Estos procesos son actividades de Gestión de Configuración que venían utilizándose en el desarrollo de software y que han sido expresadas en el libro escrito por S. Berczuk y B. Appleton.

Tomando como referencia esta obra, se ha procedido a seleccionar dos patrones de SCM,

con una aplicabilidad directa a algunos de los problemas que se describen en el capítulo Marco Contextual. Específicamente, se busca mejorar la eficiencia y productividad en el desarrollo del software científico, adaptando estos patrones, de acuerdo al marco contextual en el cual van a ser implementados, que es el campo de Investigación Científica.

3.3. Patrones Seleccionados.

3.3.1. Active Development Line.

En cualquier proyecto de desarrollo de software, el programa que se está creando, sufre cambios. Estas modificaciones implican el riesgo de que el código se rompa o que se produzca un conflicto con el código. Para evitar esto, el pattern active development line, permite encontrar un balance entre estabilidad y progreso.

3.3.1.1. Contexto

Cualquier cambio que se lleve adelante sobre el software o componente que se está desarrollando, inevitablemente implica el riesgo de que el sistema se rompa o que dichas modificaciones que se realizan generen conflicto/s con otro/s componente/s del software que se está desarrollando en el proyecto (13).



Ilustración 25 – Modificación y rotura del main line.

Básicamente, cuando se desarrolla software, intervienen muchas personas (en los ambientes científicos, intervienen desarrolladores de software científicos, investigadores, docentes, alumnos, etc.), con la finalidad de llevar adelante un desarrollo concurrente del

proyecto. Cuanto mayor sea la cantidad de miembros que intervengan en el desarrollo del proyecto, mayor será la necesidad de comunicación (en la producción de software científico, los desarrolladores muchas veces suelen encontrarse en distintas localizaciones) entre los miembros del grupo de desarrollo. Esta necesidad de comunicación, lleva a que se tengan que establecer puntos de sincronización a lo largo del proyecto, con la finalidad de que el desarrollo del software o los distintos componentes software, converjan en dichos puntos.

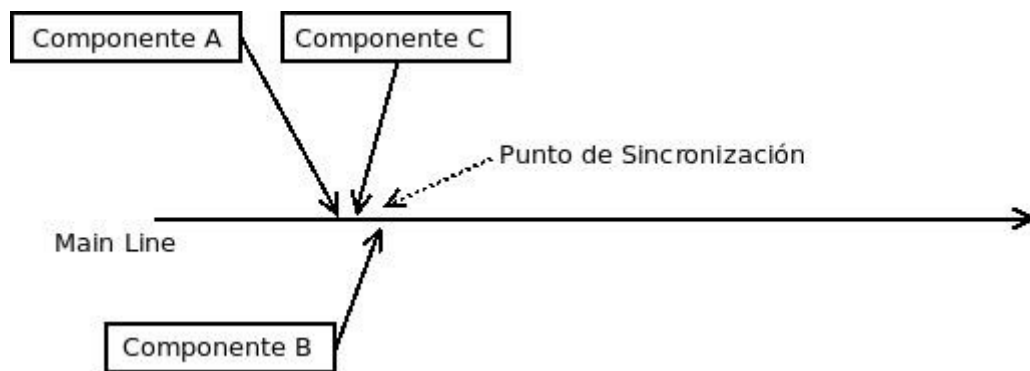


Ilustración 26 – Puntos de Sincronización en el main line.

Cuando se establecen estos puntos de sincronización, existe el riesgo de que se generen puntos muertos en el proyecto o un bloqueo (en los desarrollos de software científico, muchas veces, los proyectos quedan a cargo de otro científico o nuevos integrantes del grupo de investigación, ya que suele haber alto índice de rotación), si es que no se coordinan adecuadamente los esfuerzos de trabajo (13).

3.3.1.2. Problema

Por los motivos mencionados anteriormente, se recomienda que cuando se desarrollan nuevas funcionalidades, no se trabaje sobre el main line u otras líneas estables, ya que éstas han sido testeadas. Esto se debe a que, para poder realizar esa tarea, se requiere que la modificación que se realizará posea cierto nivel de testing. Llevar a cabo dichos tests insume tiempo, y esto genera demoras en el desarrollo de software, como así también lo hacen los componentes defectuosos (13).

3.3.1.3. Posibles Soluciones

Podrían implementarse semáforos; sin embargo, con la implementación de este sistema, solamente una persona por vez puede realizar testing y checking de los cambios realizados, con lo cual el progreso del trabajo en el proyecto se ve reducido drásticamente.

También se pueden generar nuevas ramas para las nuevas funcionalidades.

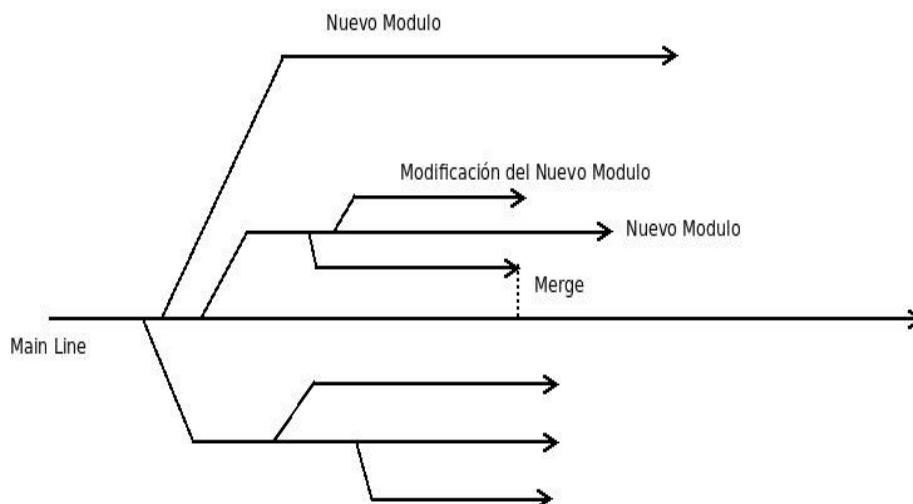


Ilustración 27 – Main Line con múltiples ramificaciones.

No obstante, al generar nuevas ramas para las nuevas funcionalidades se agrega una mayor complejidad al manejo del proyecto.

Incluso se puede dejar libre acceso al código del proyecto, para que cada desarrollador de software realice las modificaciones y checking que sean necesarios. Sin embargo, esto hace que el código del proyecto pierda valor y carezca de utilidad. Otro recurso que se puede utilizar es el de modificar la arquitectura del software, aunque sigue latente la posibilidad de que se llegue a romper el código del proyecto (13).

3.3.1.4. Solución

Para hacer frente a la problemática expuesta, se utiliza el pattern active development line. El mismo permite obtener un balance entre estabilidad y progreso en un proyecto de desarrollo de software (13).



Ilustración 28 – Balance entre estabilidad y progreso.

Un active development line sufre cambios frecuentes. Básicamente, se estructura de forma similar al main line, pero con una serie de políticas que aseguran que el código posee cierto nivel de calidad (15). Esto se traduce en que se poseen algunos puntos de chequeo que garantizan que el código está OK, y otros que son lo suficientemente bueno o aceptable para los desarrolladores de software que necesitan realizar una modificación o prueba sobre la línea de desarrollo (active development line). Para poder implementar un active development line, es necesario determinar qué tan bueno debe ser el código, para lo cual se debe realizar un proceso similar al análisis de requisitos (13).

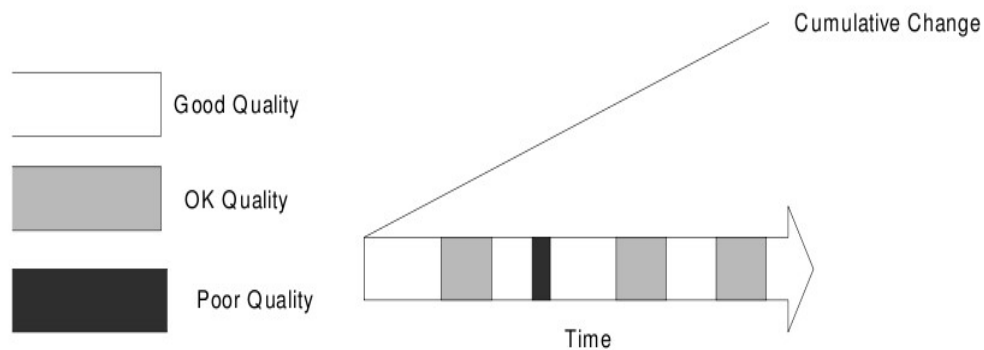


Ilustración 29 – Gráfica de una Active Development Line (16).

3.3.1.5. Consideraciones

Uno de los aspectos fundamentales que se debe entender del active development line, es el ritmo del proyecto. Éste es definido por Kane y Dikel, como los cambios recurrentes y predecibles de los productos de trabajo, con una arquitectura grupal y que va desde los clientes hasta los desarrolladores de software. Por este motivo, es necesario un buen ritmo para cualquier proyecto de desarrollo de software que sea concurrente y posea dependencias. Cabe aclarar que hay que tomar en cuenta que una buena estructura de control influye en cómo se ejecuta el ritmo del proyecto, mientras que la cultura organizacional ayuda a definir qué tipo de ritmo es necesario para el proyecto.

Si el proyecto posee buenos tests unitarios y de regresión, entonces los errores no van a persistir por mucho tiempo. Por lo tanto, el énfasis debe ser puesto en acelerar los checking de los cambios realizados en el proyecto. Hay que considerar que, si no existe una buena infraestructura de testing, se debe proceder con mucho cuidado en la ejecución de estas tareas, hasta que la misma se encuentre bien desarrollada y madura (13).

Por último, se debe establecer un criterio de cuánto es lo que se va a testear, a fin de poder determinar cuánta estabilidad es realmente necesaria para el proyecto (13).

3.3.1.6. *Beneficios Esperados.*

Con la introducción de un active development line, se preserva el main line de los errores, ya que cualquier modificación que se realice, se efectúa en el active development line. También permite que los desarrolladores de software científico puedan retornar a estados previos, en caso de que la modificación no produzca los resultados esperados, sin realizar el rollback manualmente. De esta forma, se logra incrementar la eficiencia del desarrollo de software científico. Además, a través de los puntos de chequeo, permite asegurar la calidad, lo que permite agregar esas modificaciones al main line, sin que éste sufra fallos.

A través del Active Development Line, se obtiene estabilidad y progreso en el desarrollo de software, a la vez de que se garantiza y preserva el main line del proyecto de desarrollo de software. Aparte, el código desarrollado a través del active development line, posee distintos niveles de calidad que marcan su utilidad y permiten agregarlo al main line, de ser necesario, garantizando siempre el nivel de calidad de éste.

3.3.2. **Integration Build.**

En un proyecto de desarrollo de software, cada desarrollador trabaja en su propio workspace, realizando modificaciones. Sin embargo, todos esos cambios deben integrarse y el sistema tiene que poder construirse correctamente. Para esto, el pattern integration build proporciona los mecanismos para llevar adelante la integración y construcción del sistema.

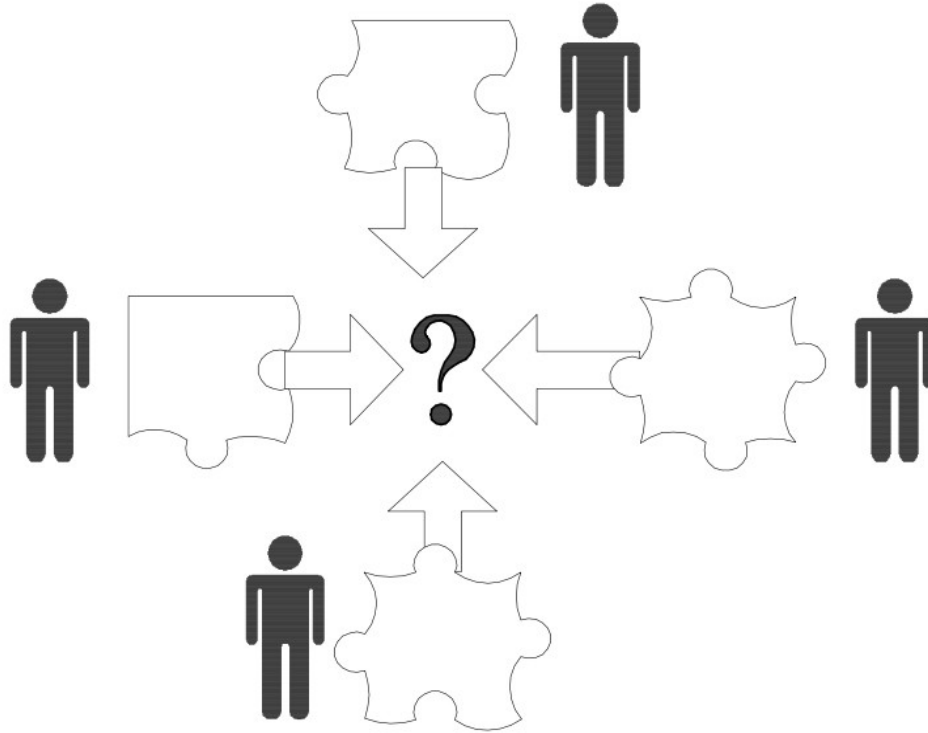


Ilustración 30 – Complicación al momento de realizar una integración (13).

3.3.2.1. Contexto

En los proyectos de desarrollo de software, intervienen muchos desarrolladores, los cuales realizan cambios constantemente (en el desarrollo de software científico, el código del software científico se va modificando en el tiempo, para mejorar o ampliar las funcionalidades de éste) al código del proyecto. Por esto, es imposible asegurar que el sistema se va a construir correctamente, después de integrarlo al main line (13). Éste problema sucede debido a que el programa sigue evolucionando con el transcurso del tiempo. Por lo tanto, el código con el que empezó a trabajar el desarrollador al principio, ha cambiado (17).

3.3.2.2. Problema

El llevar adelante una construcción completa y limpia del sistema toma tiempo; sin embargo, ese tiempo es mucho menor que el tiempo que pierde el grupo de desarrollo de software en resolver los problemas (en el desarrollo de software científico, se debe destinar parte del periodo de investigación, a la resolución de los problemas en el código) que se generan, porque los cambios que se realizan terminan por romper la construcción del sistema (13).

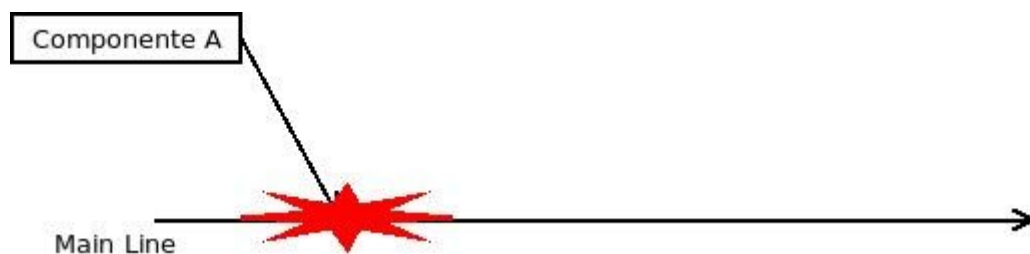


Ilustración 31 – Nuevo componente que genera un problema en el main line.

Algunos desarrolladores de software tienden a desarrollar sus códigos sobre otros componentes (códigos) y, una vez que finalizan con el desarrollo, recién proceden a trabajar en la integración de la construcción.

Además, suele ser frustrante para los desarrolladores el tener que rastrear aquellos cambios inconsistentes (los componentes software que se producen en el ámbito científico, al momento de integrarse producen problemas) que se encuentran presentes en el software desarrollado (13).

3.3.2.3. Solución

Cuando los desarrolladores de software realizan checking de los cambios, existe la posibilidad de que se introduzcan errores en las construcciones, por lo tanto, es mejor realizar una construcción completa del sistema que se está desarrollando.

Para llevar adelante esto, la construcción del sistema debe efectuarse en un espacio de

trabajo que contenga todos los componentes a integrarse para armar la build (13).

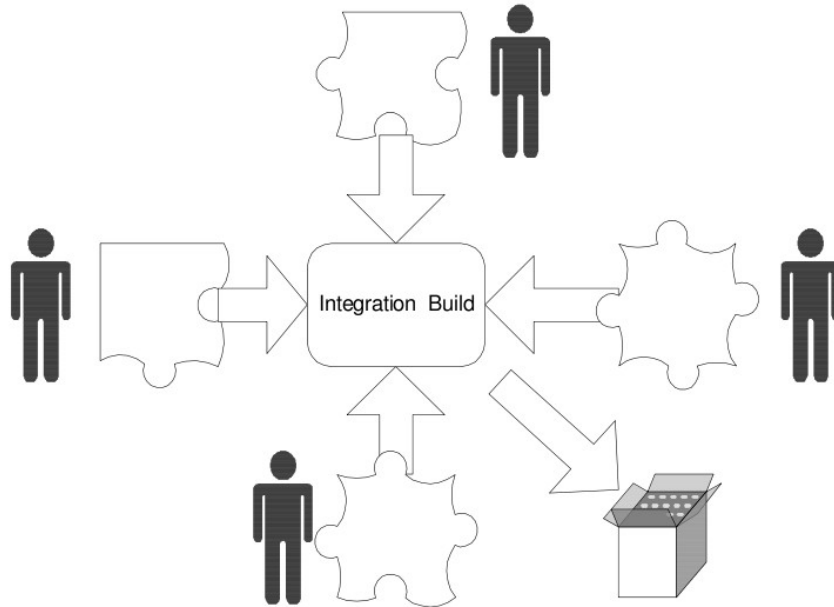


Ilustración 32 – Proceso de Integration Buil (13).

Para determinar cada cuánto es indispensable realizar la integración de la construcción, es necesario determinar:

- Cuánto tiempo toma construir el sistema.
- Cuán rápido se producen los cambios.

Si la construcción del sistema toma mucho tiempo, entonces se debe considerar realizar una integración diaria. En cambio, si la construcción del sistema es rápida, se puede realizar una integración cada vez que se realiza checkin del código del proyecto.

La integración de construcción tiene por finalidad atrapar las fallas que puedan llegar a emerger en la construcción del sistema. Si dichas fallas persisten, se pueden agregar más pre - checking, como pasos previos de verificación.

Los checking de las modificaciones se realizan en el repositorio. El sistema de control de recursos que posee el repositorio, responde ante los checking, extrayendo todos los

archivos y construyendo el sistema. Cualquier error que se produzca durante el armado del sistema, es reportado a la construcción maestra, así como a la persona responsable de haber ejecutado dicho cambio que se llevó a cabo (13).

3.3.2.4. Consideraciones

Cuando se usa este pattern, es necesario asegurarse de que todos los cambios y dependencias son construidos usando un proceso central de construcción integrador. Éste debe ser:

- Reproducible.
- Lo más cercano posible a la versión final del producto.
- Automatizado o requerir de la menor intervención posible de trabajo manual.
- Poseer notificaciones o algún otro mecanismo que permita identificar errores e inconsistencias.

Si el sistema de control no serializa adecuadamente los cambios que se realizan al sistema, se puede llegar a construir un sistema con fallas, debido a las inconsistencias que presenta el código. Es por este motivo que se recomienda identificar con marcadores las construcciones que se realizan.

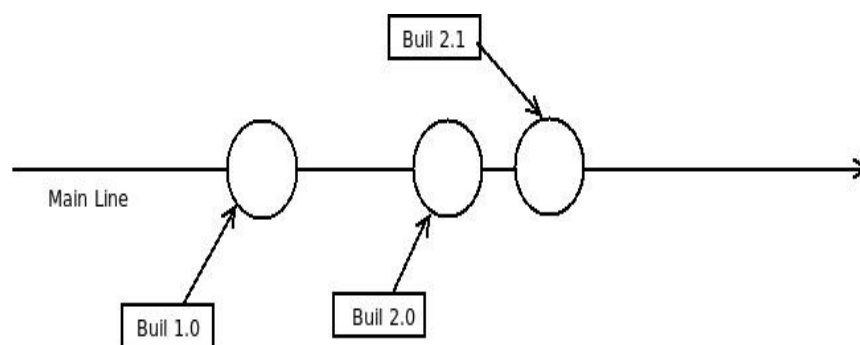


Ilustración 33 – Marcadores de las construcciones realizadas del proyecto.

También es bueno tener en cuenta que la integración de construcción debe ser repetida en

todas las plataformas que soporte el sistema (13).

3.3.2.5. *Beneficios Esperados*

Con la integration build, los desarrolladores de software científico mejoran la eficiencia del desarrollo de software científico y de la investigación científica. A través de la integration build, los desarrolladores de software científico se aseguran de que los componentes y la construcción cuentan con el nivel de calidad adecuado. Esto se debe a que se realizan testing previos, con el fin de verificar que los componentes cuentan con el nivel de calidad adecuado para integrarlos al main line. También se certifica que la construcción del sistema se realiza correctamente, al comprobar que se cuenta con la última versión de todos los componentes para llevar a cabo la construcción. Se establecen los tiempos que demora la integración y, en base a esos tiempos, se define cuándo se realiza el proceso de integración (una o varias veces al día). De ésta forma, se mejora la eficiencia del desarrollo de software científico, ya que los desarrolladores de software científico no pierden tiempo realizando integraciones innecesarias o esperando que estas finalicen.

A través de la Integration Build, se pueden integrar las distintas modificaciones al sistema, con la seguridad de que cualquier falla que se produzca, será atrapada y solucionada a tiempo, evitando la rotura del sistema.

4. Modelo Teórico

4.1. Active Development Line

El software va cambiando con el tiempo. Esto se debe a que se amplía o mejora la/s funcionalidad/es que dicho software ofrece, debido a que las necesidades de los usuarios van modificándose con el tiempo, como también lo hacen las piezas hardware y los demás software (programas, SO, etc.). Para evitar problemas al momento de llevar a cabo las modificaciones, en vez de trabajar sobre la línea principal de desarrollo, se utiliza un Active Development Line, a fin de preservar el main line de posibles errores.

El software científico sufre cambios constantes, ya sea porque los modelos de simulación evolucionan o porque es necesario ampliar las funcionalidades que ofrece. Pero los desarrolladores de software científico realizan estas modificaciones sobre el main line, con lo cual se introducen errores en el código produciéndose fallos en el main line, llevando a que el software científico falle.

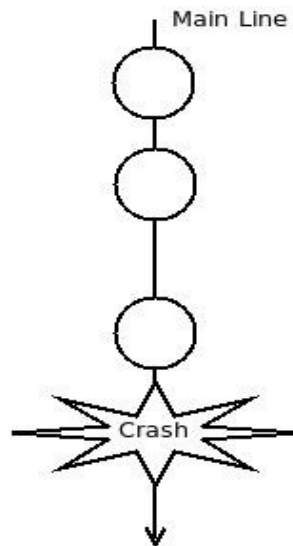


Ilustración 34 – Error en el Main Line.

Por lo tanto, como solución, se recomienda la utilización del pattern Active Development Line, con las siguientes adaptaciones:

4.1.1. Nota de Liberación.

A lo largo del Active Development Line, se encuentran puntos de chequeos. Por cada uno de estos puntos de chequeo, se va a agregar una nota de liberación o lanzamiento.

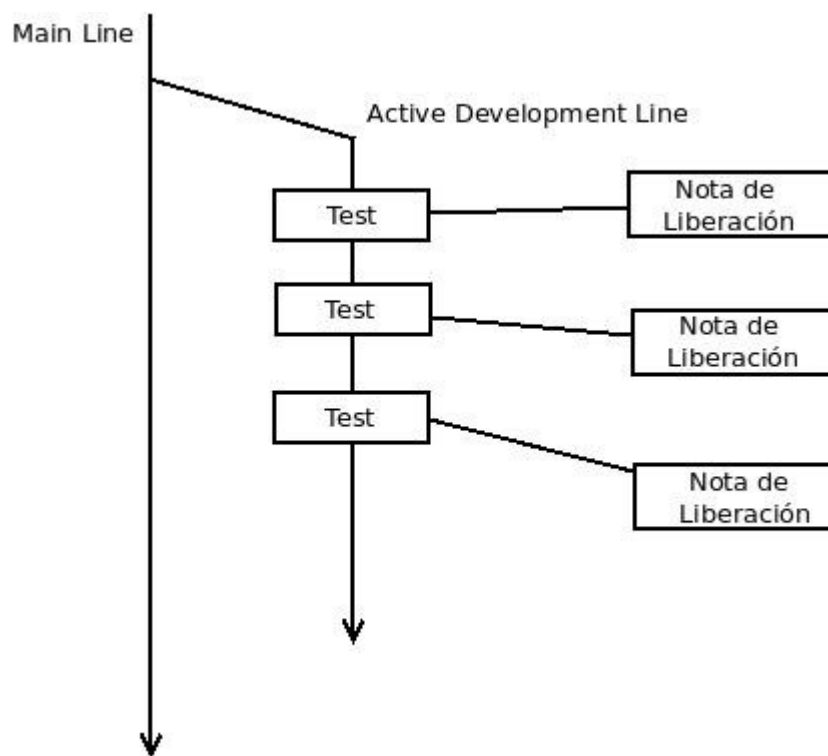


Ilustración 35 – Notas de liberación o lanzamiento.

Cada nota de liberación o lanzamiento va a especificar:

- Fecha de la Modificación
- La versión de la modificación realizada.
- Los bugs corregidos hasta ese punto/versión.

A través de estas notas de liberación, se brinda información específica de lo que se ha

hecho hasta ese punto de chequeo, del Active Development Line. Con estas notas de liberación, los desarrolladores de software científico van a poder determinar cuál de esos puntos de chequeo es de utilidad para seguir adelante con el desarrollo de componentes que vienen realizando. Básicamente, con las notas de liberación se puede evaluar si el último punto de chequeo posee utilidad o si se deben remitir a un punto de chequeo previo en el active development line.

4.1.2 Tags

Cada punto de chequeo que se encuentra OK, va a ser marcado con un tag, con la finalidad de marcar hasta qué modificación se va a agregar al main line, sin que éste se vea afectado, es decir, que no se produzcan fallos en él. Para que los puntos de chequeo sean tageados, éstos deben pasar los tests satisfactoriamente.

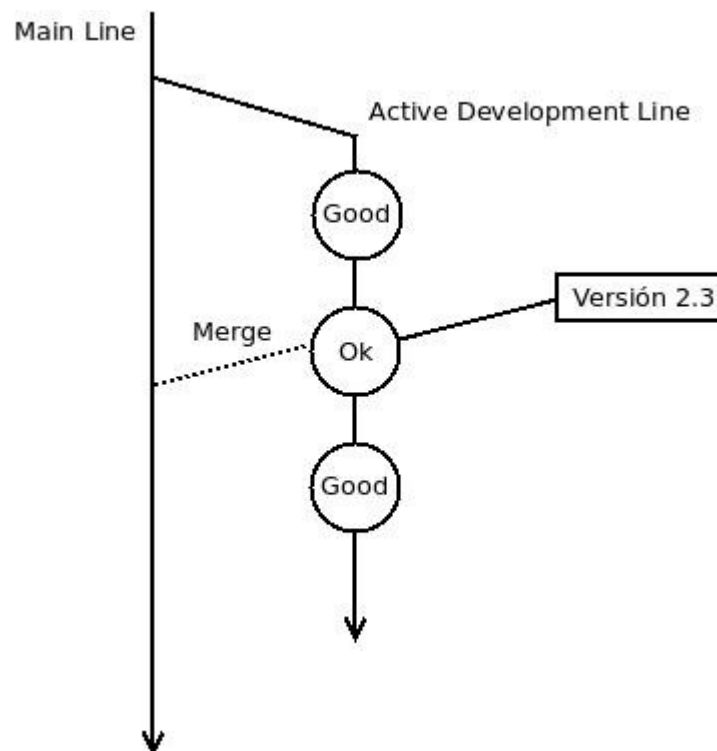


Ilustración 36 – Tag de los puntos OK de chequeo.

Estos tags van a poseer el número de versión del punto de chequeo. Con estos tags, lo que se busca es marcar aquellos puntos OK del active development line, para establecer hasta qué punto de chequeo se va a someter a una operación merge, con el fin de agregar las modificaciones al main line. De esta forma, se garantiza que dichos puntos de chequeo poseen el nivel de calidad adecuado para ser agregados al main line, sin afectar la calidad y estabilidad de éste en el proyecto.

Las notas de liberación y tags, mejoran el desarrollo de software científico, ya que evitan que el main line sufra los errores que se generan por las modificaciones que llevan adelante los desarrolladores de software científico.

4.2. Integration Build.

El software sufre modificaciones a lo largo del tiempo con la finalidad de solucionar bugs o agregar mayor funcionalidad al software. Cada desarrollador de software trabaja en un componente de software existente o desarrolla un nuevo componente, pero esto lo hace en su propio workspace. De esta forma, se evitan los potenciales conflictos entre los componentes que se están desarrollando. A pesar de esto, más adelante, dichos componentes deben integrarse al main line para construir el sistema. Por lo tanto, no existen garantías de que los componentes puedan funcionar correctamente cuando se realice la integración con el main line. Esto da como resultado que se produzcan errores en el sistema o que directamente el sistema falle (se rompa).

La situación que se describe en el párrafo anterior, sucede con frecuencia en el desarrollo de software científico, puesto que los desarrolladores de software científico trabajan de forma distribuida, con lo cual, al momento de integrar los componentes, surgen errores en la construcción.

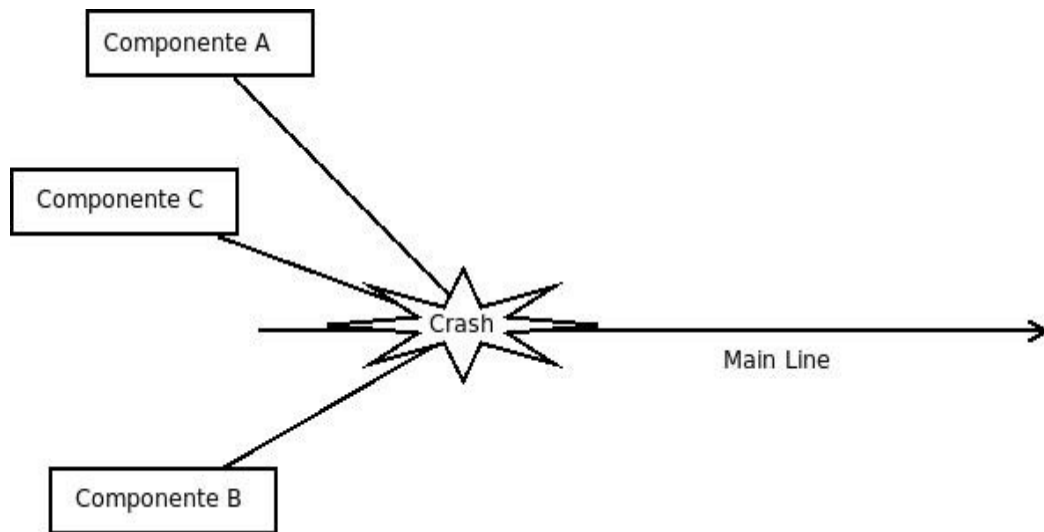


Ilustración 37 – Error al momento de integrar, al main line, los componentes.

Por dicho motivo, se propone la utilización del pattern Integration Build, con las siguientes adaptaciones:

4.2.1. Workspace Integrador.

Se va a establecer un workspace, donde los distintos componentes a ser integrados.

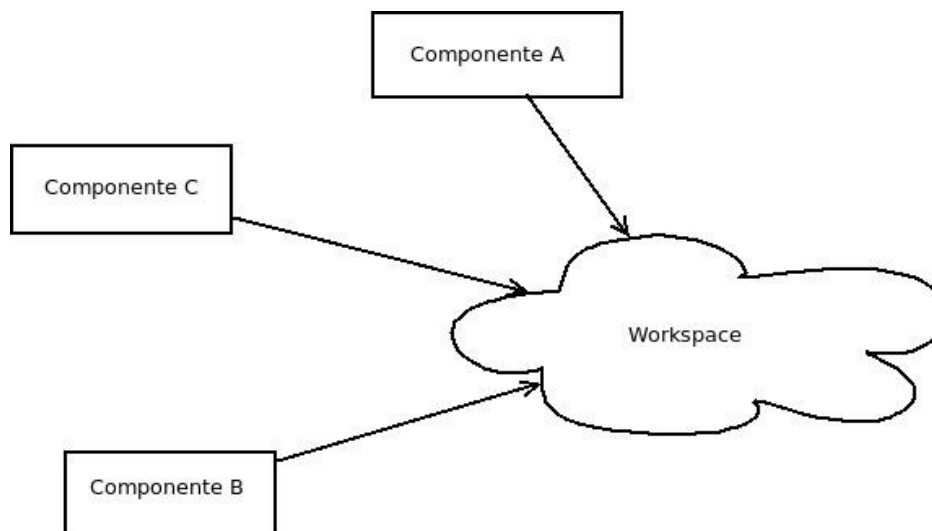


Ilustración 38 – Workspace que contiene los componentes a ser integrado.

Estos componentes deben encontrarse actualizados (ser la última versión) y haber pasado los test correspondientes. Con esto, se busca evitar que se produzcan errores al momento de realizar la integración, ya sea por usar distintas versiones de los componentes o por errores que posean alguno de ellos.

Con la modificación propuesta, los desarrolladores de software científico van a asegurarse de que cuentan con todos los componentes necesarios para llevar a cabo la integración. También, van a tener la certeza de que los componentes importados son la última versión estable. Con esto, en caso de producirse fallos en la integración, el desarrollador de software científico tiene la seguridad de que esos fallos provienen de su propio componente y no de los componentes que importó a su workspace.

4.2.2. Tiempo de Integración

Se definirá cada cuánto tiempo se va a realizar el proceso de integración. Para esto, es necesario establecer los tiempos que toma realizar la integración. Si el proceso de integración lleva menos de 2 minutos, entonces se puede integrar después de cada checking que se realice del componente desarrollado. En cambio si es superior a los 2 minutos, la integración se va a realizar una vez por día.

Con esta modificación, el desarrollador de software científico puede mantener un buen ritmo de desarrollo, evitando pérdidas de tiempo.

Un workspace que concentra los componentes a ser integrados y la definición de los tiempos de integración, permite obtener estabilidad en el desarrollo de software científico, a la vez que se asegura que el nivel de fallos se vea reducido.

5. Concreción del Prototipo.

5.1. Implementación de una línea activa de desarrollo.

A través de un sistema de control de versión, se pueden generar distintas ramas de trabajo. Estas ramas que se generan, pueden ser consideradas como una línea activa de desarrollo. Cada vez que se realicen modificaciones al/los archivo/s, se procede a tomar un snapshot de los archivos con el comando `$ git add <nombre del archivo>`.

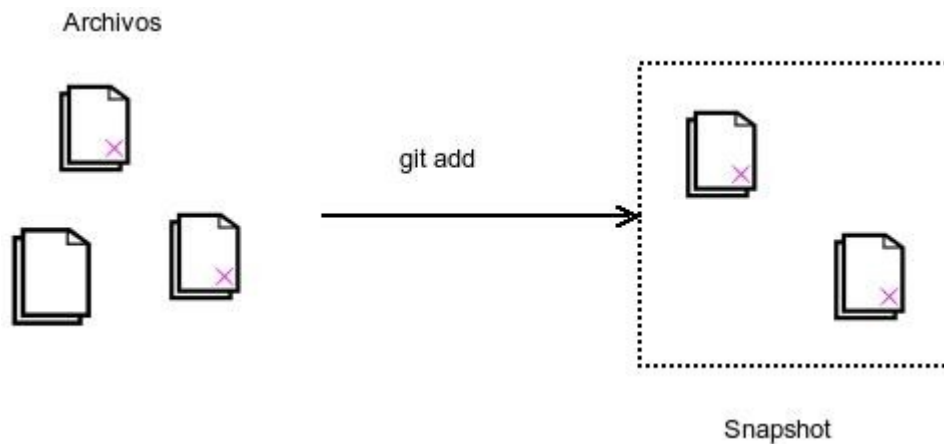
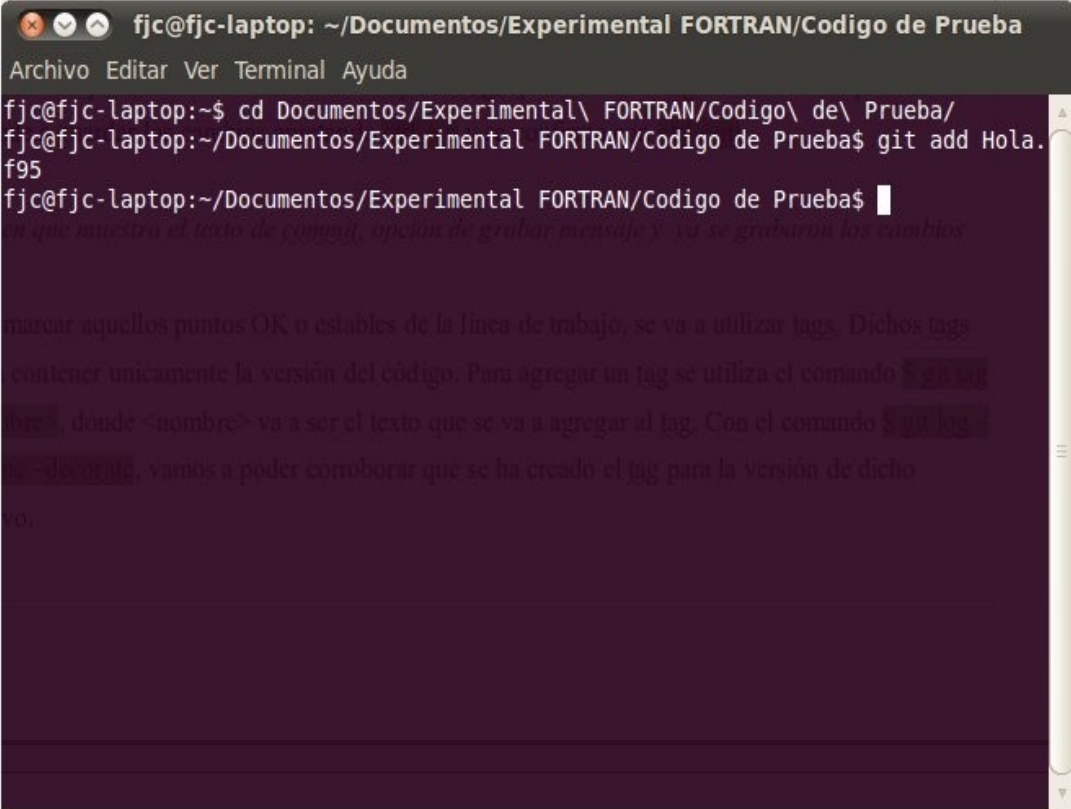


Ilustración 39 - Cómo funciona comando git add.

Cada vez que se realicen modificaciones y pruebas, se procede a agregar el archivo modificado con el comando `$ git add <nombre del archivo>` (20).



The screenshot shows a terminal window titled "fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba". The terminal displays the following commands and output:

```
fjc@fjc-laptop:~$ cd Documentos/Experimental\ FORTRAN/Codigo\ de\ Prueba/
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add Hola.f95
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Below the terminal output, there is a block of text in Spanish explaining the next steps in the Git workflow:

en que muestra el texto de commit, opción de grabar mensaje y ya se grabaron los cambios

marcar aquellos puntos OK o estables de la línea de trabajo, se va a utilizar **tags**. Dichos **tags** contener únicamente la versión del código. Para agregar un tag se utiliza el comando **\$ git tag <nombre>**, donde <nombre> va a ser el texto que se va a agregar al tag. Con el comando **\$ git log** se **describiendo**, vamos a poder corroborar que se ha creado el tag por la versión de dicho

vo.

Ilustración 40 - Tomar una snapshot del archivo modificado.

Una vez que se toma la snapshot, se procede a realizar el commit, para agregar el snapshot del archivo modificado a la línea de prueba. Para esto se va a ejecutar el comando **\$ git commit** (20) .

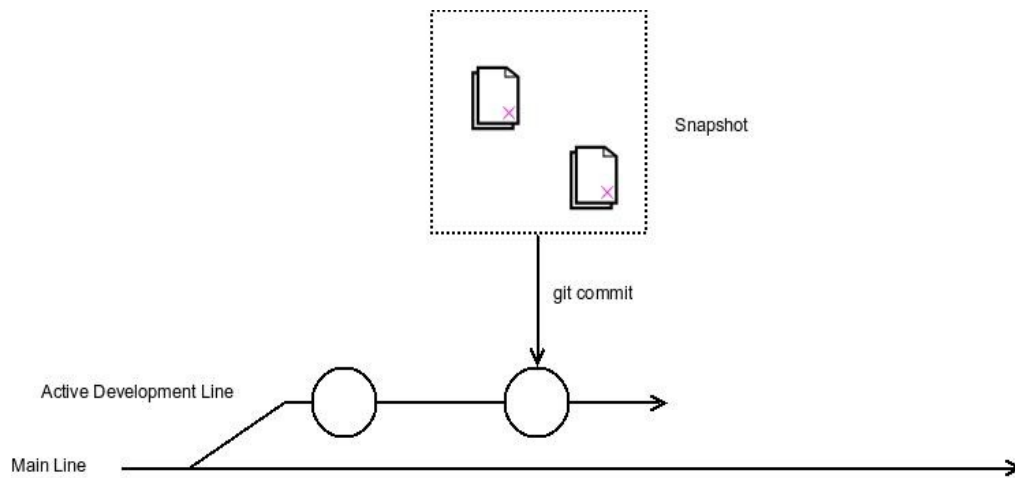
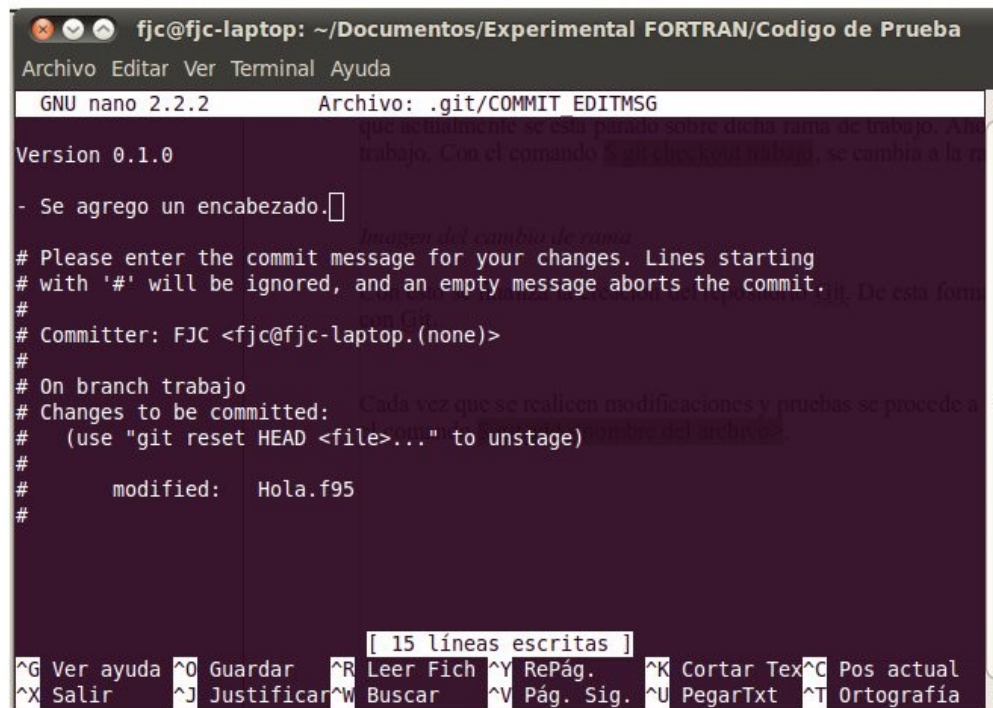


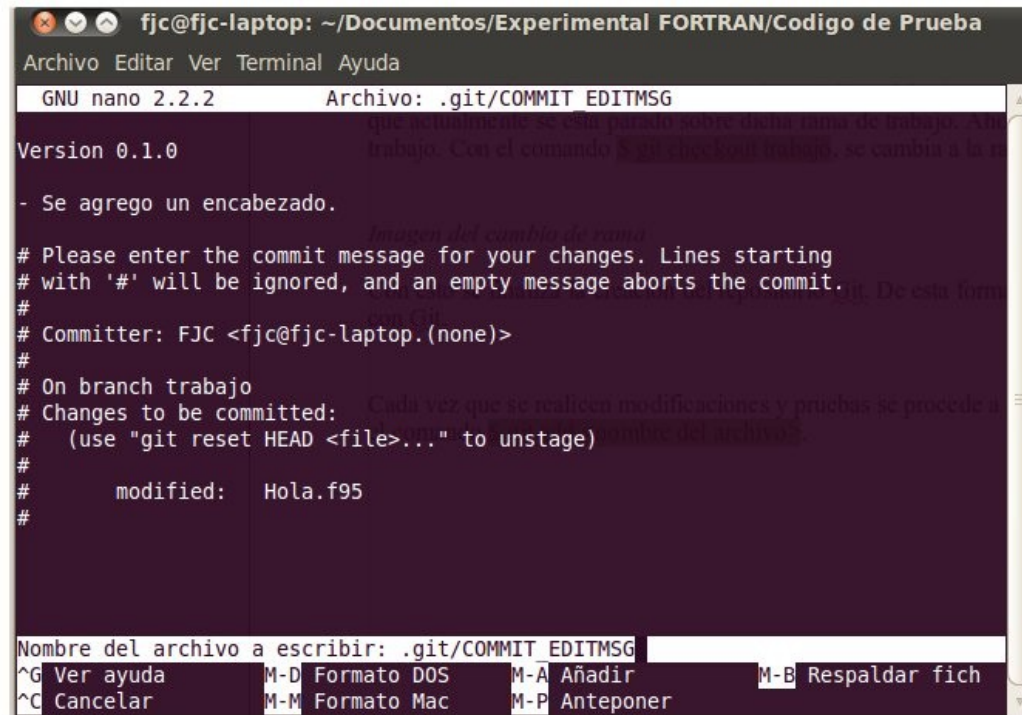
Ilustración 41 - Cómo funciona comando git commit.

Con la ejecución de este comando, se abre un editor de texto, donde se escribe la versión del archivo y los bugs corregidos hasta ese punto. Para guardar el texto escrito, primero se deben presionar las teclas **ctrl + o**, luego se confirma presionando las teclas **ctrl + m** y por último, para que se ejecute el commit, se debe salir presionando la combinación de teclas **ctrl + x** (20).



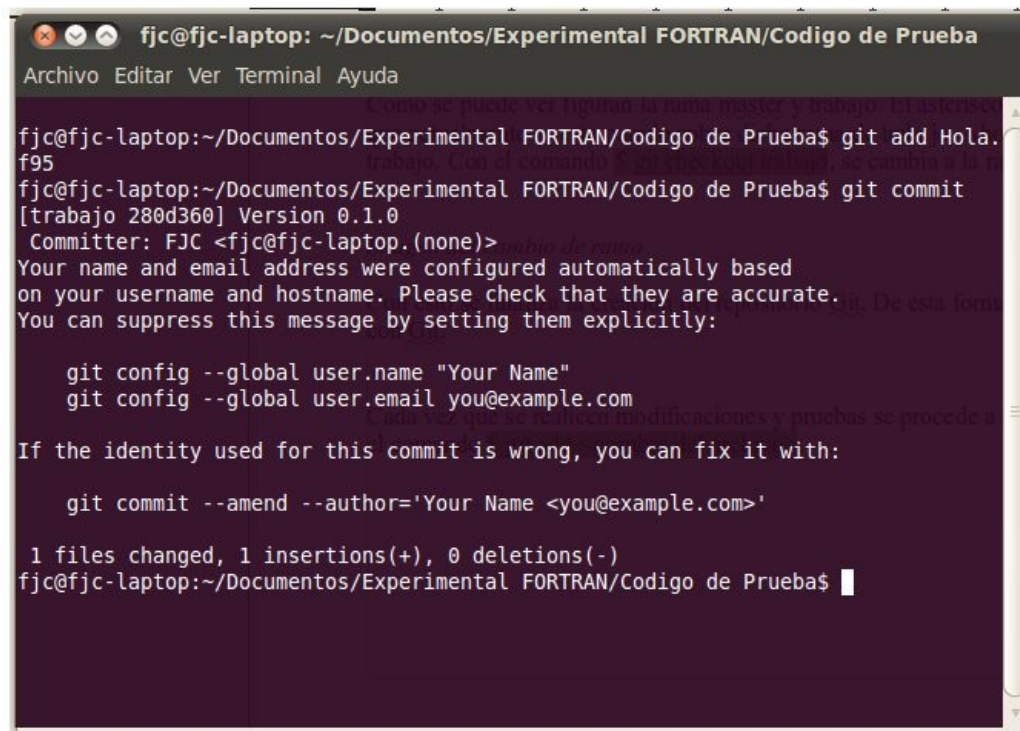
```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
GNU nano 2.2.2 Archivo: .git/COMMIT_EDITMSG
Version 0.1.0
- Se agrego un encabezado.
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: FJC <fjc@fjc-laptop>.(none)>
#
# On branch trabajo
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Hola.f95
#
[ 15 líneas escritas ]
^G Ver ayuda ^O Guardar ^R Leer Fich ^Y RePág. ^K Cortar Tex ^C Pos actual
^X Salir ^J Justificar ^W Buscar ^V Pág. Sig. ^U PegarTxt ^T Ortografía
```

Ilustración 42 - Editor de texto del commit.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
GNU nano 2.2.2 Archivo: .git/COMMIT_EDITMSG
Version 0.1.0
- Se agrego un encabezado.
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: FJC <fjc@fjc-laptop.(none)>
#
# On branch trabajo
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Hola.f95
#
Nombre del archivo a escribir: .git/COMMIT_EDITMSG
^G Ver ayuda      M-D Formato DOS  M-A Añadir       M-B Respalda fich
^C Cancelar      M-M Formato Mac  M-P Anteponer
```

Ilustración 43 - Opciones de guardado del commit.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add Hola.f95
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git commit
[trabajo 280d360] Version 0.1.0
Committer: FJC <fjc@fjc-laptop.(none)>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

1 files changed, 1 insertions(+), 0 deletions(-)
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 44 - Fin del proceso de commit y reporte del commit que se ha realizado.

Para marcar aquellos puntos que se encuentran OK/estables en la rama de trabajo, se utilizarán tags. Los mismos contienen la versión del archivo modificado. Para agregar tags a los puntos de chequeo OK, se utiliza el comando `$ git tag <nombre>` (20), donde <nombre> va a ser la versión del archivo modificado.

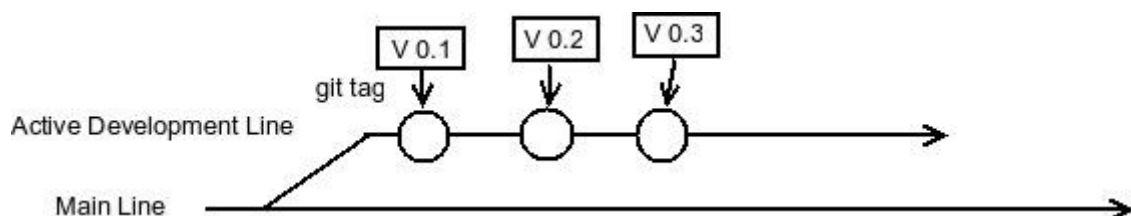


Ilustración 45 - Utilización del comando git tag.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git tag v0.1.5
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git log --one
line --decorate
74fa45b (HEAD, v0.1.5, trabajo) Version 0.1.5
b62d3be (v0.1.4, master) Version 0.1.4
cbale84 Version 0.1.3
c4dd016 (v0.1.2) Version 0.1.2
81cd4e9 Version 0.1.1
280d360 Version 0.1.0
bf7407e Imagen inicial del proyecto
```

Ilustración 46 - Creación de tag y corroboración del tag creado.

Para agregar los cambios realizados en la rama trabajo (active development line) a la rama master (main line), se usa el comando merge. Previo a ejecutar la operación merge, es necesario estar posicionado en la rama master. Esto lo hacemos con el comando `$ git checkout master`. Una vez posicionado en la rama master, se procede a ejecutar la operación merge con el comando `$ git merge trabajo` (20).

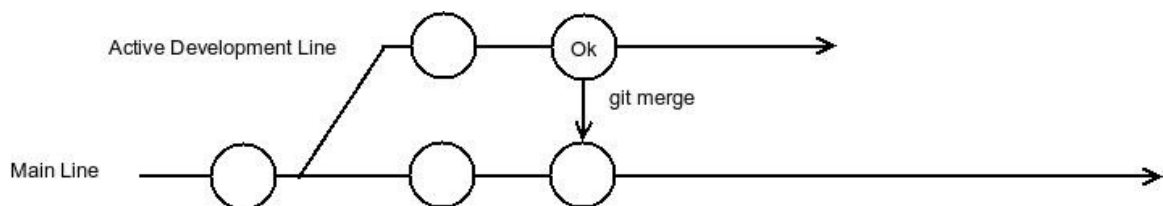


Ilustración 47 - Utilización de git merge.

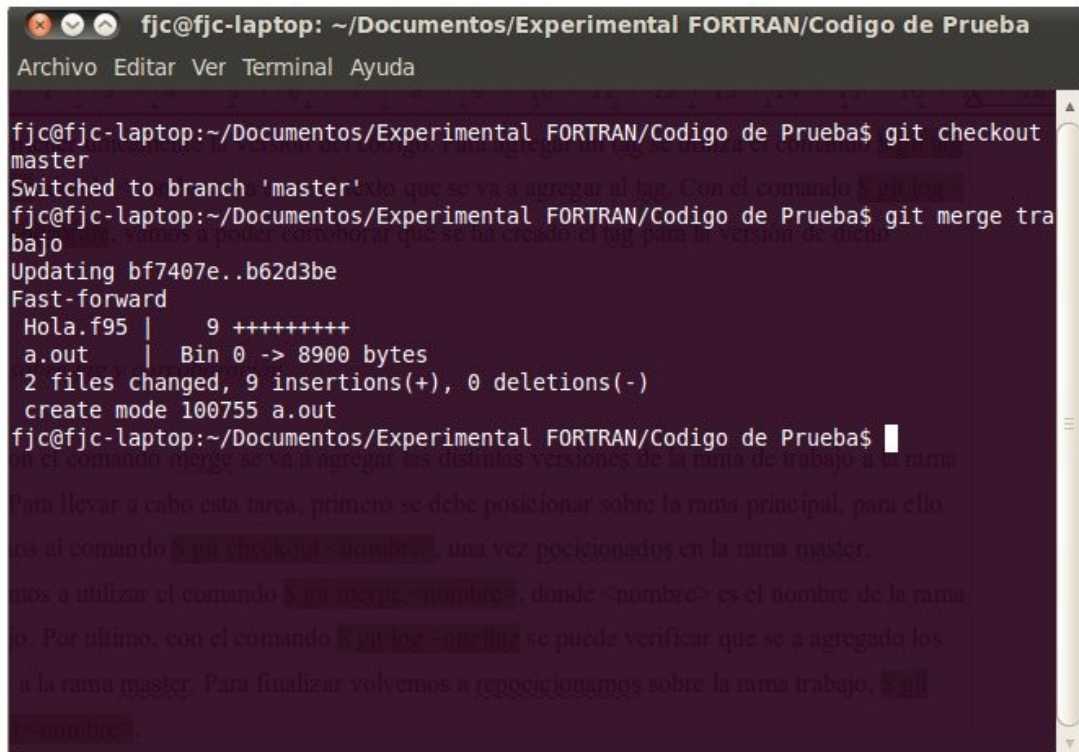
Para verificar si se han agregado las distintas versiones, se utiliza el comando `$ git log - -oneline`. Por último, se retorna a la rama trabajo con el comando `$ git checkout trabajo`.

Ejemplo de script que ejecuta la operación merge:

```
#!/bin/bash

# Posicionamiento sobre el directorio donde
# se encuentra el repositorio
cd /home/fjc/Documentos/Experimentos/Codigo

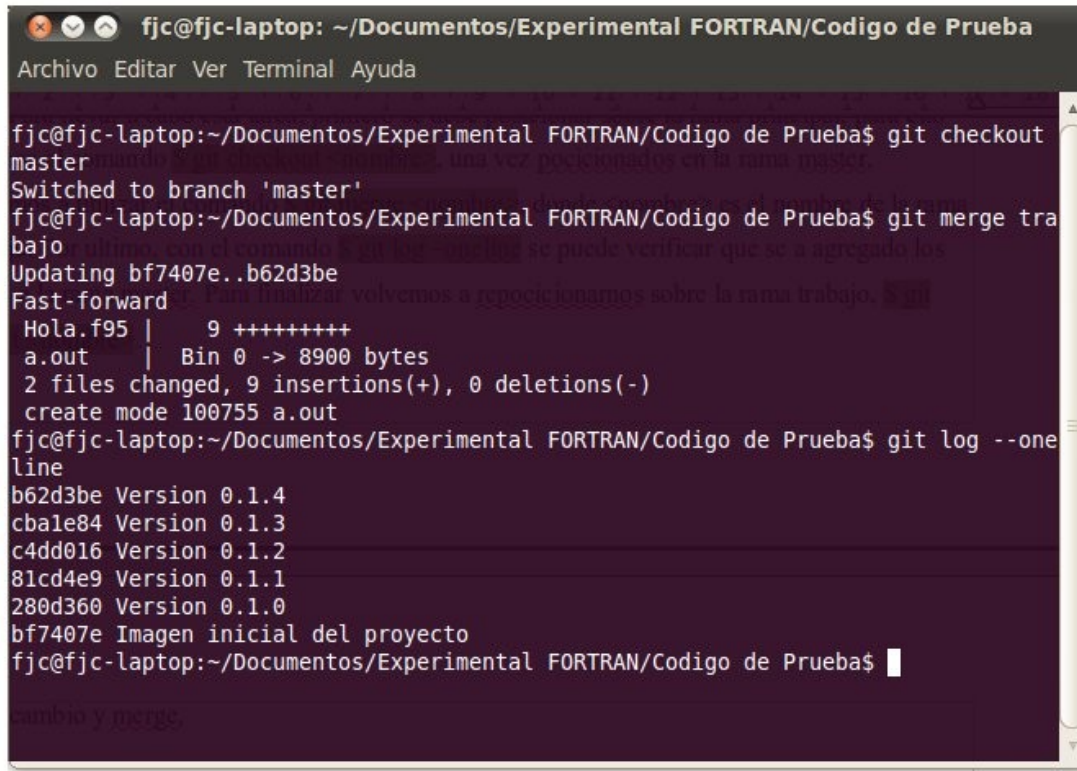
#Ejecución del comando merge
git checkout master
git merge t
git checkout t
```



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo  Editar  Ver  Terminal  Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git checkout
master
Switched to branch 'master'
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git merge tra
bajo
Updating bf7407e..b62d3be
Fast-forward
  Hola.f95 |    9 ++++++++
  a.out    | Bin 0 -> 8900 bytes
  2 files changed, 9 insertions(+), 0 deletions(-)
  create mode 100755 a.out
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 48 - Ejecución de la operación merge.

A screenshot of a terminal window titled 'fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba'. The terminal shows the following commands and output:

```
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git checkout master
Switched to branch 'master'
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git merge trabajo
Updating bf7407e..b62d3be
Fast-forward
  Hola.f95 | 9 ++++++++
  a.out     | Bin 0 -> 8900 bytes
  2 files changed, 9 insertions(+), 0 deletions(-)
  create mode 100755 a.out
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git log --one
line
b62d3be Version 0.1.4
cbale84 Version 0.1.3
c4dd016 Version 0.1.2
81cd4e9 Version 0.1.1
280d360 Version 0.1.0
bf7407e Imagen inicial del proyecto
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 49 - Corroboración de que la operación merge.

Con la creación de una rama en el repositorio, se logra implementar el active development line (trabajo) y el main line (master), permitiendo la utilización del pattern active development line.

5.2. Implementación de Integration Build

Para que la integración de las modificaciones sea correcta, es fundamental llevar a cabo dos procesos de integración, uno local para corroborar su correcta construcción y otro con el main line para verificar que se integran perfectamente.

Primero se debe verificar que se poseen todos los componentes necesarios en nuestro

workspace (base de datos, librerías, etc). Después de corroborar esto, se ejecuta la compilación y armado de la build correspondiente a nuestro workspace, para continuar con la realización de los test planificados a fin de constatar que la build se ha construido exactamente y posee el grado de calidad necesario para ser agregada al main line.

Luego, terminada la integración de nuestro workspace, se va a importar a él la última versión del código y demás componentes que se encuentren en el main line. Concluida esta, se inicia el merge de los componentes importados con nuestra rama de desarrollo, a fin de poder realizar la compilación y armado de la build.

Posteriormente, se somete la build a los test, con el objetivo de verificar que se ha integrado perfectamente con los componentes del main line.

Finalmente, si pasa satisfactoriamente los test, se procede a realizar el commit de la build y luego se ejecuta el merge con el main line (21). En caso de no pasar los test, se deberá revisar y corregir el código a fin de poder obtener una build en condiciones, para ser integrada al main line.

Ejemplo de un script que realiza la integration build:

```
#!/bin/bash
```

```
# Nos posicionamos
```

```
cd /home/fjc/Documentos/Experimentos/Codigo/Codigo\ Fortran/
```

```
# En este apartado va la verificación de que se tienen
```

```
# todos los componentes necesarios para realizar la
```

```
# integración.
```

```
#
```

```
# Se procede a realizar la compilación del código fuente y armado del
```

```
# ejecutable o build.
```

```
read -p "Ingrese el nombre del codigo fuente con extensión incluida: " FUENTE
```

```
read -p "Ingrese el nombre que desea darle al archivo ejecutable: " NAME
```

```
gfortran $FUENTE -o $NAME.exe
```

```
# En este apartado se llevan a cabo los test tendientes a evaluar
```

```
# la build
```

```
#
```

```
# En este apartado se procede a importar la última versión del
```

```
# main line
```

```
#
```

```
# Se procede a realizar nuevamente la compilación del código fuente
```

```
# y armado del ejecutable o build.
```

```
read -p "Ingrese el nombre del codigo fuente con extensión incluida: " FUENTE
```

```
read -p "Ingrese el nombre que desea darle al archivo ejecutable: " NAME
```

```
gfortran $FUENTE -o $NAME.exe
```

```
# En este apartado se procede a realizar los test pertinentes a la
```

```
# build
```

```
#
```

La integration build nos permite obtener una construcción del sistema, con la seguridad de que posee el nivel de calidad adecuado para ser integrado al main line, además de incrementar el nivel de eficiencia y productividad del grupo de desarrollo de software científico.

6. Conclusiones

Este proyecto surgió de la necesidad de determinar los factores que afecta el tiempo de desarrollo del software científico. Éste es uno de los pilares de las investigaciones científicas e impacta en la eficiencia de las investigaciones.

A lo largo del proyecto, se corrobora la escasa integración que existe entre el campo de Ingeniería de Software y Desarrollo de Software Científico. Esto lleva a que se generen falencias en el desarrollo del software científico, que repercute en el tiempo de investigación.

Para mejorar esta situación, a través de este trabajo se propone la utilización de algunos patrones de Software Configuration Managment. Esta herramienta de Ingeniería de Software permite llevar a cabo desarrollos ágiles, posibilitando mejorar el tiempo de desarrollo del software científico.

Como resultado de este trabajo, se elaboro una solución que fue implementada a través de un prototipo, el cual fue mostrado en el “Primer WorkShop: El Desarrollo de Software en ambientes Científicos y Técnicos”.

Por ultimo, con la información recolectada en la demostración y obtenida en el marco de investigación realizada en este trabajo, queda plasmado un camino para seguir realizando investigaciones que contribuyan a mejorar la integración entre la Ingeniería de Software y el Desarrollo de Software Científico.

7. Referencia Bibliográfica

- (1) Diane Kelly y Rebecca Sanders. Assessing the Quality of Scientific Software. In First International Workshop on Software Engineering for Computational Science & Engineering (2008).
- (2) David Woollard, Chris A. Mattmann, Daniel Popescu y Nenad Medvidovic. KADRE: Domain-Specific Architectural Recovery For Scientific Software Systems. ACM 2010.
- (3) Diane F. Kelly. A Software Chasm: Software Engineering and Scientific Computing. IEEE Computer Society 2007.
- (4) M a r t i n M a c k a y and J o h n M c C a l l. The Technology Revolution – Is it Payback Time?. Business Briefing: Pharmatech 2004.
- (5) International Federation For Information Processing. Distinctive characteristics of Scientific Applications. Ottawa Oct. 2-4 2000.
http://www.nsc.liu.se/~boein/ifip/woco8_dc.html
- (6) Luke Nguyen-Hoan, Shayne Flint, Ramesh Sankaranarayana. A Survey of Scientific Software Development. ACM-IEEE International Symposium on Empirical Software Engineering and Measurement 2010.
- (7) Roger S. Pressman. Ingeniería de Software. Un Enfoque Práctico. Sexta Edición. Mexico: McGraw-Hill Interamericana; 2007. p.796-823.
- (8) Segal, Judith (2008). In: *Proceedings of the Psychology of Programming Interest Group, PPIG 08*, 10-12- September 2008, University of Lancaster, UK.
- (9) Luke Nguyen-Hoan, Shayne Flint, Ramesh Sankaranarayana. A Survey of Scientific Software Development. ACM-IEEE International Symposium on Empirical Software Engineering and Measurement 2010.
- (10) Morris, Chris and Segal, Judith. Some challenges facing scientific software developers: The case of molecular biology. IEEE e-Science 2009, 9 -11.
- (11) Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffry K. Hollingsworth, Forrest Shull, Marvin V. Zelkowitz. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. Journal IEEE Software. Volume 25 Issue 4, July 2008.

-
- (12) Medha Umarji, Carolyn Seaman, A. Gunes Coru, Hongfang Liu. Software Engineering Education for Bioinformatics. 22nd Conference on Software Engineering Education and Training 2009.
- (13) Steve Berczuk, Brad Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Ed. Addison – Wesley, Boston. 2002.
- (14) Welcome to Brad Appleton's Documents. CM Crossroads; 2000 [acceso 13 de Abril del 2011]. Patterns and Software: Essential Concepts and Terminology. Disponible en: <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>.
- (15) Steve Berczuk . End-of-release Branching Strategies . StickyMinds.com. 11/8/2010 [acceso 19 de Abril del 2011]. Disponible en: <http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=16454&tth=DYN&tt=siteemail&iDyn=2>
- (16) Stephen P Berczuk . Reliable Codelines . Pattern Languages of Programs Conference 2001.
- (17) Lucas Cordeiro, Cassiano Becker, Raimundo Barreto. Agile Patterns for Multi-site Software Development . Departamento de Ciência da Computação - Universidade Federal do Amazonas (UFAM), Brazil .
- (18) <http://www.qengroup.com/home/images/stories/gestion%20de%20configuracion.jpg>
- (19) Wide Thoughts. Gašper Kozak. 15 de Junio de 2009 [Acceso el 18 de Julio de 2011]. Thoughts. <http://gasper.kozak.si/blog/2009/06/15/the-steep-learning-curve-misunderstanding/>.
- (20)Git Reference. Acceso el 18 de Julio de 2011. Reference. <http://gitref.org/>
- (21)Martin Fowler. Martin Fowler, 1 de Mayo del 2006 [acceso el 18 de Julio de 2011]. Continuous Integration. <http://martinfowler.com/articles/continuousIntegration.html>
- (22)David Matthews, Greg Wilson, Steve Easterbrook. Configuration Management for Large-Scale Scientific Computing at the UK Met Office . IEEE 2008.

8. Bibliografía

1. Creative Commons. Acceso 1 de Agosto de 2011. Choose a License. <http://creativecommons.org/>
2. Diane Kelly y Rebecca Sanders. Assessing the Quality of Scientific Software. In First International Workshop on Software Engineering for Computational Science & Engineering (2008).
3. David Woollard, Chris A. Mattmann, Daniel Popescu y Nenad Medvidovic. KADRE: Domain-Specific Architectural Recovery For Scientific Software Systems. ACM 2010.
4. Diane F. Kelly. A Software Chasm: Software Engineering and Scientific Computing. IEEE Computer Society 2007.
5. Martin Mackay and John McCall. The Technology Revolution – Is it Payback Time?. Business Briefing: Pharmatech 2004.
6. International Federation For Information Processing. Distinctive characteristics of Scientific Applications. Ottawa Oct. 2-4 2000. http://www.nsc.liu.se/~boein/ifip/woco8_dc.html
7. Luke Nguyen-Hoan, Shayne Flint, Ramesh Sankaranarayana. A Survey of Scientific Software Development. ACM-IEEE International Symposium on Empirical Software Engineering and Measurement 2010.
8. Roger S. Pressman. Ingeniería de Software. Un Enfoque Práctico. Sexta Edición. Mexico: McGraw-Hill Interamericana; 2007. p.796-823.
9. Segal, Judith (2008). In: *Proceedings of the Psychology of Programming Interest Group, PPIG 08*, 10-12- September 2008, University of Lancaster, UK.
10. Morris, Chris and Segal, Judith. Some challenges facing scientific software developers: The case of molecular biology. IEEE e-Science 2009, 9 -11.
11. Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffry K. Hollingsworth, Forrest Shull, Marvin V. Zelkowitz. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. Journal IEEE Software. Volume 25 Issue 4, July 2008.
12. Medha Umarji, Carolyn Seaman, A. Gunes Coru, Hongfang Liu. Software

- Engineering Education for Bioinformatics. 22nd Conference on Software Engineering Education and Training 2009.
13. Steve Berczuk, Brad Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Ed. Addison – Wesley, Boston. 2002.
 14. Welcome to Brad Appleton's Documents. CM Crossroads; 2000 [acceso 13 de Abril del 2011]. Patterns and Software: Essential Concepts and Terminology. Disponible en:
<http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>.
 15. Steve Berczuk . End-of-release Branching Strategies . StickyMinds.com. 11/8/2010 [acceso 19 de Abril del 2011]. Disponible en:
<http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=16454&tth=DYN&tt=siteemail&iDyn=2>
 16. Stephen P Berczuk . Reliable Codelines . Pattern Languages of Programs Conference 2001.
 17. Lucas Cordeiro, Cassiano Becker, Raimundo Barreto. Agile Patterns for Multi-site Software Development . Departamento de Ciência da Computação - Universidade Federal do Amazonas (UFAM), Brazil .
 18. <http://www.qengroup.com/home/images/stories/gestion%20de%20configuracion.jpg>
 19. Wide Thoughts. Gašper Kozak. 15 de Junio de 2009 [Acceso el 18 de Julio de 2011]. Thoughts.
<http://gasper.kozak.si/blog/2009/06/15/the-steep-learning-curve-misunderstanding/>.
 20. Git Reference. Acceso el 18 de Julio de 2011. Reference. <http://gitref.org/>
 21. Martin Fowler. Martin Fowler, 1 de Mayo del 2006 [acceso el 18 de Julio de 2011]. Continuous Integration.
<http://martinfowler.com/articles/continuousIntegration.html>
 22. Susan M. Baxter, Steven W. Day, Jacquelyn S. Fetrow, Stephanie J. Reisinger. Scientific Software Development Is Not an Oxymoron . 8 de Septiembre de 2006.
 23. David Sloan, Catriona Macaulay, Paula Forbes, Scott Loynton, Peter Gregor . User

-
- research in a scientific software development project .BCS HCI 09 Proceedings of the 2009 British Computer Society Conference on HumanComputer Interaction (2009).
24. Robert C. Cannon, Marc-Oliver Gewaltig, Padraig Gleeson, Upinder S. Bhalla, Hugo Cornelis, Michael L. Hines et al. Interoperability of Neuroscience Modeling Software: Current Status and Future Directions . (2007).
 25. David Matthews, Greg Wilson, Steve Easterbrook. Configuration Management for Large-Scale Scientific Computing at the UK Met Office . IEEE 2008.
 26. Ustun Yildiz, Adnene Guabtni, Anne H.H. Ngu . Business versus Scientific Workflow: A Comparative Study . (2009).
 27. Steve M. Easterbrook , Timothy C. Johns . Engineering the Software for Understanding Climate Change . IEEE 2009.
 28. Veit Hoffmann, Horst Lichter, Alexander Nyßen . Processes and Practices for Quality Scientific Software Projects . (2010)
 29. Jo Erskine Hannay, Carolyn Macleod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, Greg Wilson. How Do Scientists Develop and Use Scientific Software? . ICSE Workshop on Software Engineering for Computational Science and Engineering (2009).
 30. James F. Bowring . Building Cyber Infrastructure for Geochronology: A Case Study in Collaborative Software Engineering Research . (2008).
 31. Samuel Z. Guyer, Calvin Lin. BROADWAY: A SOFTWARE ARCHITECTURE FOR SCIENTIFIC COMPUTING. (2000).
 32. Judith Segal . Models of scientific software development .*First International Workshop on Software Engineering in Computational Science and Engineering.* (2008)
 33. Rasmus H. Fogh, Wayne Boucher, Wim F. Vranken, Anne Pajon, Tim J. Stevens, T. N. Bhat, et al. A framework for scientific data modeling and automated software development . (2004).
 34. Magnus Sletholt, Hans Petter Langtangen , Dietmar Pfahl , Jo Erskine Hannay. Making Modern Scientific Software Development Explicitly Agile . (2010).

-
35. Peter B. Andersen, Florian Prange, Søren Serritzlew. Software engineering as a part of scientific practice.
 36. Catherine Letondal ,Uwe Zdun. Anticipating Scientific Software Evolution as a Combined Technological and Design Approach . Second International Workshop on Unanticipated Software Evolution 2003.
 37. Susan Dart . Concepts in Configuration Management Systems . 1990.
 38. Lucas de Oliveira Arantes, Ricardo de Almeida Falbo, Giancarlo Guizzardi. Evolving a Software Configuration Management Ontology . 2007.
 39. Sun Services White Paper . BEST-PRACTICE RECOMMENDATIONS CONFIGURATION MANAGEMENT . 2007.
 40. Aberdeen Group. The Configuration Management Benchmark Report . 2007.
 41. Introducing Continuous Integration. 2007. http://www.google.com/url?sa=t&source=web&cd=5&ved=0CEEQFjAE&url=http%3A%2F%2Fptgmedia.pearsoncmg.com%2Fimages%2F9780321336385%2Fsamplechapter%2F0321336380_CH02.pdf&rct=j&q=Introducing%20Continuous%20Integration&ei=ERE4TtfpMJK2tgel87yZAw&usg=AFQjCNEy2KIuUkeQvSeD5yx4ZEmUNAKoaQ&sig2=8ykNwru3p6zmRolDVBGc9A&cad=rja.
 42. Steve Berczuk, Brad Appleton and Steve Konieczka . Agile SCM – Patterns and Software Configuration Management . CM Crossroads 2004.
 43. R. Owen Rogers . Scaling Continuous Integration . 2008.
 44. Dario André Louzado, Lucas Carvalho Cordeiro . Aplicando Padrões de Gerência de Configuração de Software em Projetos Geograficamente Distribuídos . 5th Latin American Conference on Pattern Languages of Programming 2005.
 45. Hans Cristian Muller Santa Cruz . Programando en Fortran . 2007.
 46. Ing. Arturo J. López García . Guía de Programación en Fortran 90/95 . 2004.
 47. J. San Fabian . INTRODUCCION A LA PROGRAMACION FORTRAN . 2008.
 48. Un Breve Curso de FORTRAN. Acceso 2 de Agosto de 2011. http://www.uam.es/personal_pdi/ciencias/ruben/master/CNC/FORTRAN/FORTRA N90.html
 49. FORTRAN Tutorial. Acceso 2 de Agosto de 2011.

-
- <http://folk.uio.no/steikr/doc/f77/tutorial/>
50. Git Wiki Page. Acceso el 2 de Agosto de 2011.
https://git.wiki.kernel.org/index.php/Main_Page
51. Pro Git. Acceso el 2 de Agosto de 2011. <http://progit.org/>
52. gittutorial(7) Manual Page. Acceso el 2 de Agosto de 2011.
<http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html>
53. Everyday GIT With 20 Commands Or So. Acceso el 2 de Agosto de 2011.
<http://www.kernel.org/pub/software/scm/git/docs/everyday.html>
54. Git - SVN Crash Course. Acceso el 2 de Agosto de 2011.
<http://git-scm.com/course/svn.html>

9. Glosario

Active Development Line: Línea activa de desarrollo. Es la rama donde se realizan las modificaciones al código.

Branch: Rama.

Commit: Operación para guardar los cambios en una rama.

Desarrollador de Software Científico: Referencia a un programador que no cuenta con los conocimientos formales de Ingeniería de Software.

Línea estable: Es una rama cuyos elementos guardados no producen errores y para que se agreguen nuevos componentes, estos deben pasar una suite de tests. Tampoco se realizan modificaciones de ninguna clase en estas líneas de desarrollo.

Main Line: Línea principal de desarrollo. Es una línea estable donde no se realizan modificaciones de ninguna clase.

Merge: Operación que unifica los elementos de una rama con los de otra rama.

Patrón de Configuración: Una solución estandarizada y genérica.

Resiliente: Capacidad de sobreponerse, resistir o adaptarse a los cambios.

Rollback: Operación de retorno. Retornar a un estado previo.

SCM: Software Configuration Management/Gestión de Configuración de Software.

Snapshot: Fotos.

Software Científico: Software que normalmente no ha sido desarrollado bajo los estándares de Ingeniería de Software.

Software Comercial: Software desarrollado por empresas siguiendo los estándares de Ingeniería de Software.

Software Tradicional: Software que sigue los estándares de desarrollo de Ingeniería de Software.

Suite de Test: Es un conjunto de test orientados a garantizar la calidad de un elemento.

Tags: Etiquetas. Se utilizan para etiquetar las modificaciones que se agregan en una rama determinada.

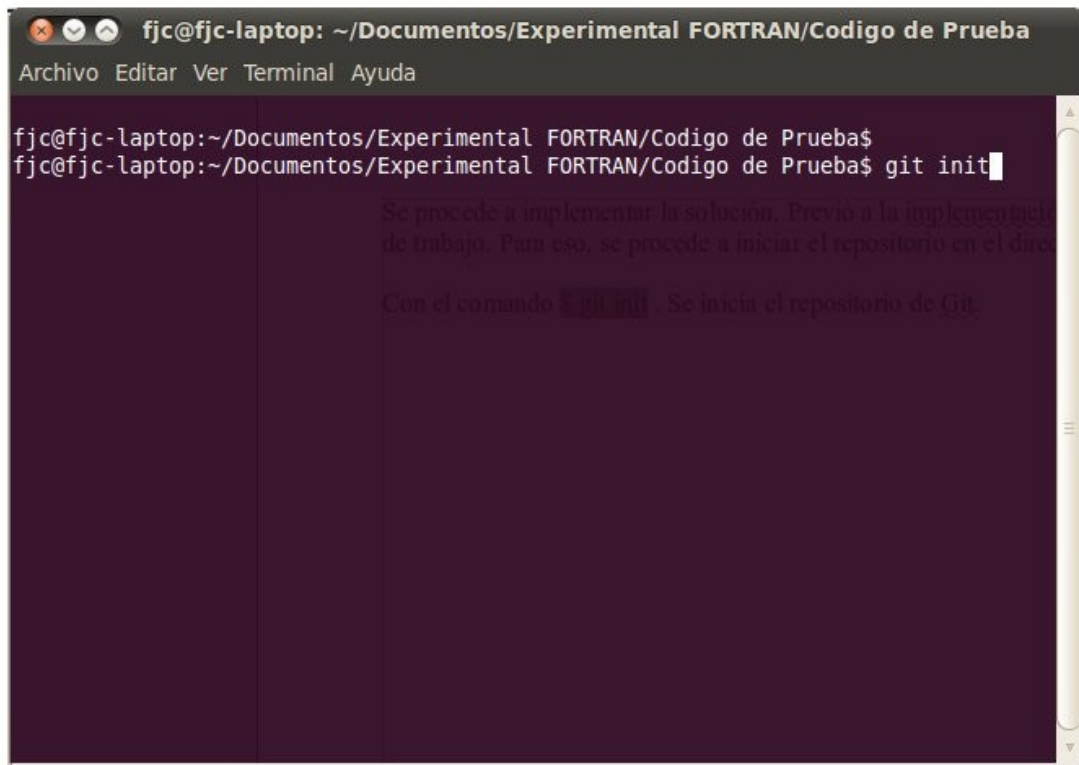
Workspace: Es el espacio de trabajo del desarrollador de software.

10. Anexo

10.1. Soluciones.

10.1.1. Inicialización del repositorio.

Para comenzar con la implementación de la solución, es necesario preparar el espacio de trabajo. Para poder trabajar con el active development line, se va a utilizar Git. Previamente, es necesario tener listo el repositorio, en el directorio de trabajo del proyecto. Primero, se debe inicializar el repositorio de datos. Entonces, abrimos la consola de linux y nos posicionamos en el directorio donde se encuentra el proyecto. Una vez allí, se va a teclear el comando `$ git init`. Con este comando se inicializa el repositorio de Git.

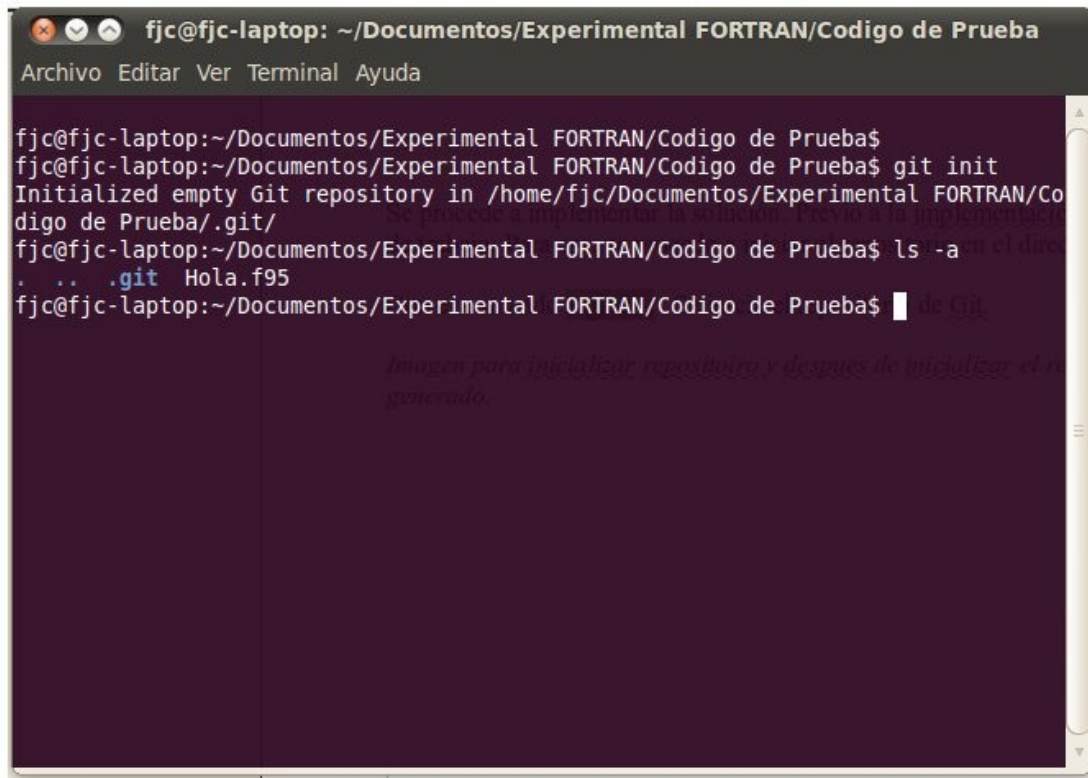


The image shows a terminal window titled "fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba". The window has a menu bar with "Archivo", "Editar", "Ver", "Terminal", and "Ayuda". The terminal content shows the user at the prompt "fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba\$" typing the command "git init". The prompt changes to "fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba\$ git init". Faint, semi-transparent text in the background of the terminal reads: "Se procede a implementar la solución. Previo a la implementación de trabajo. Para eso, se procede a iniciar el repositorio en el directorio. Con el comando `$ git init`. Se inicia el repositorio de Git."

```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo  Editar  Ver  Terminal  Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git init
```

Ilustración 50 - Se inicializa el repositorio.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git init
Initialized empty Git repository in /home/fjc/Documentos/Experimental FORTRAN/Codigo de Prueba/.git/
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ ls -la
.  ..  .git  Hola.f95
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustracion 51 - Mensaje al finalizar la inicialización del repositorio y comprobación de la creación de éste.

Una vez que se ha finalizado con la inicialización del repositorio, se debe proceder a crear la rama de trabajo generando un snapshot y commit inicial de todos los archivos que vamos a grabar en el repositorio. Con esta operación se va a establecer la rama master, que es la rama principal. Por lo tanto, primero se van a agregar los archivos con el comando `$ git add <nombre>`. Este comando nos permite añadir un archivo o varios archivos de ser preciso, escribiendo los nombres de los mismos, uno al lado de otro, separados por un espacio. También se puede utilizar el comando `$ git add .`, el cual actúa de forma recursiva agregando todos los archivos modificados del directorio de trabajo.

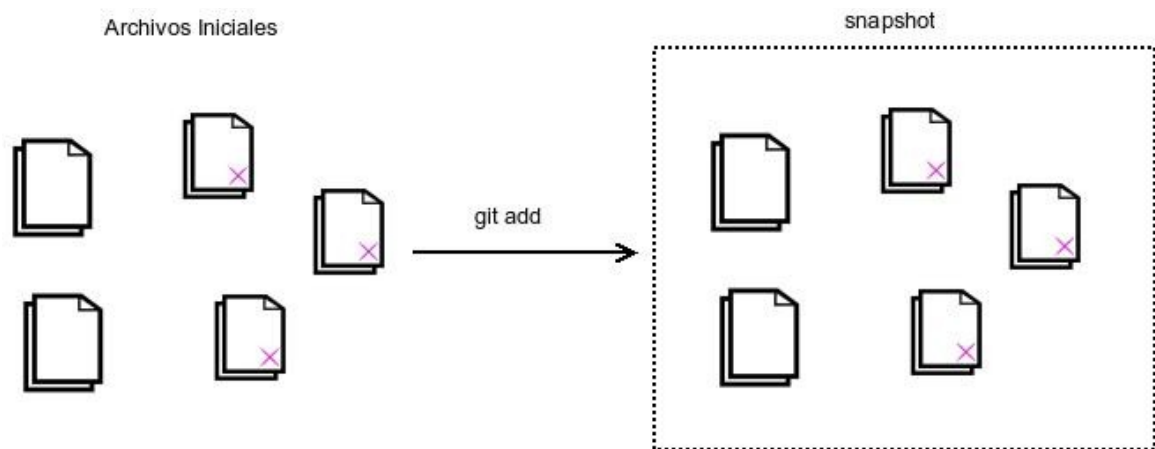


Ilustración 52 - Inicialización. Comando git add.

```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add Hola.
f95
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add .
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Con el comando `$ git init` Se inicia el repositorio de Git.

Imagen para inicializar repositorio y después de inicializar el repositorio se genera.

Luego se procede a generar la rama de trabajo. Previo a realizar snapshot de los archivos que dan lugar a la rama `master`. Para agregar los archivos con el comando `git add "nombres archivo"`, varios archivos o `git add .`, que agrega todos los archivos y directorios.

Para ello vamos a utilizar el comando `$ git branch "nombre"`. Donde `"nombre"` es el nombre de la rama de trabajo que se va a generar.

Imagen generando la rama de trabajo.

Ilustración 53 - Snapshot inicial del proyecto.

Después de que se ha tomado el snapshot inicial, se procede a realizar el commit, para agregar los archivos al repositorio. Para ello, se va a utilizar el comando `$ git commit -m 'mensaje'`, donde el mensaje es un texto corto que explica qué es lo que se está haciendo con ese commit. En caso de querer agregar un texto más detallado, se debe utilizar el comando `$ git commit`, el cual abre un editor de texto.

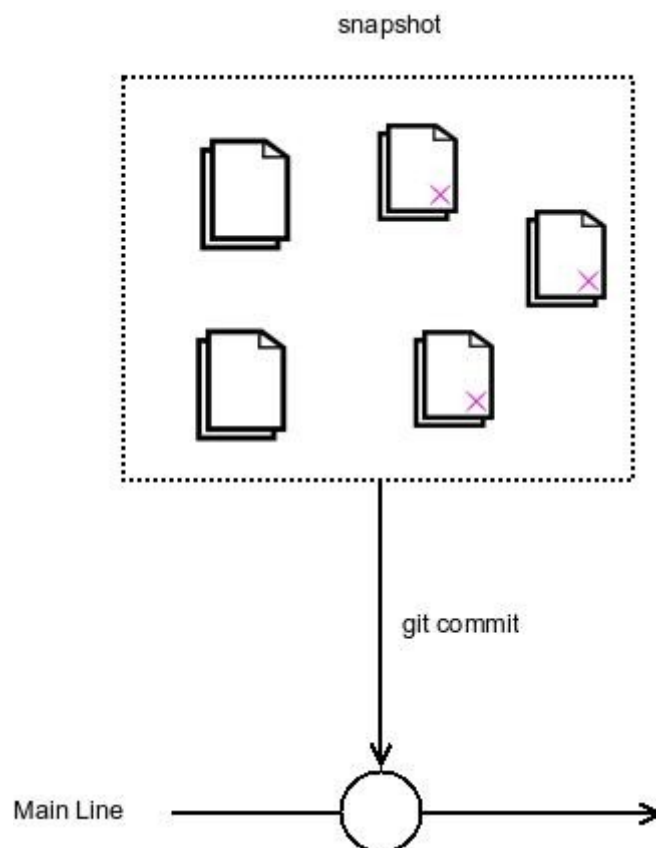


Ilustración 54 - Inicialización. Comando git commit.

```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git commit -m
'Imagen inicial del proyecto'
[master (root-commit) bf7407e] Imagen inicial del proyecto
Committer: FJC <fjc@fjc-laptop.(none)>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

1 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 Hola.f95
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 55 - Commit del snapshot inicial del proyecto.

Una vez que se ha finalizado con el proceso del snapshot y el commit inicial, se va a proceder a crear la rama trabajo con el comando `$ git branch <nombre>`.

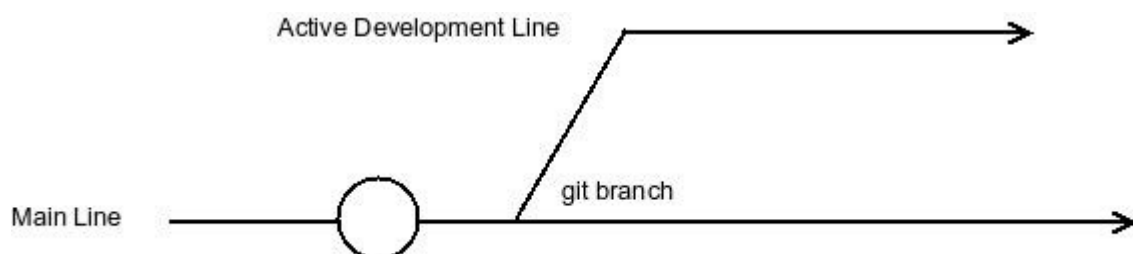


Ilustración 56 - Inicialización. Comando branch.

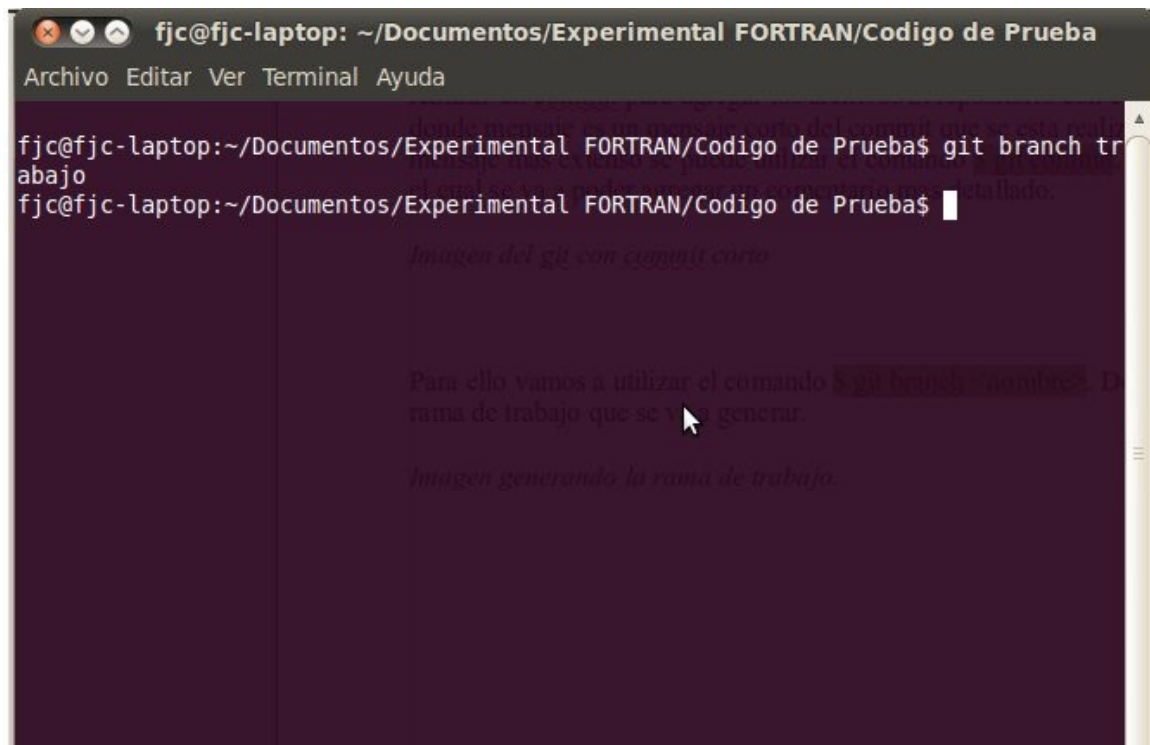


Ilustración 57 - Creación de la rama trabajo

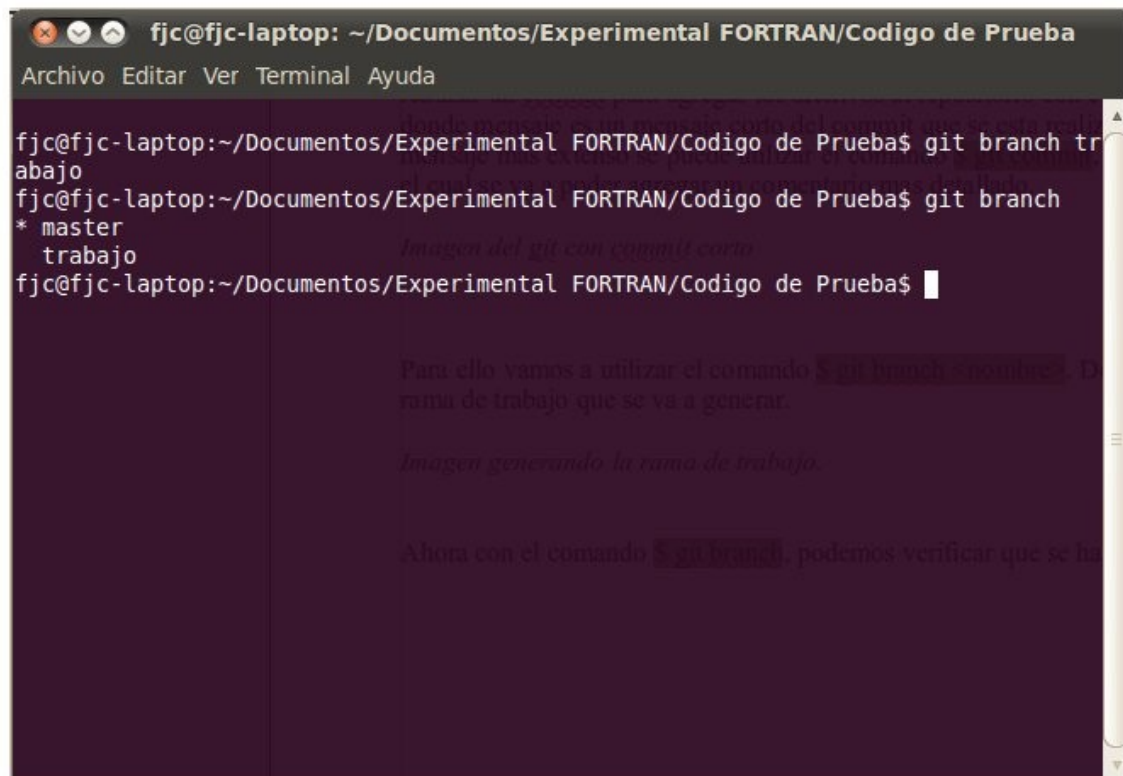
Para verificar si se ha creado la rama, utilizamos el comando `$ git branch`. Este comando nos muestra en pantalla las ramas que existen en el repositorio. El asterisco nos indica en qué rama nos encontramos posicionados, en este caso es la rama master.

Ejemplo de script que prepara el espacio de trabajo.

```
#!/bin/bash

# Nos posicionamos en el directorio donde se va a inicializar
# el repositorio
cd /home/fjc/Documentos/Experimentos/Codificacion/

# Se procede a ejecutar los comandos para inicializar el repositorio
# y preparar el entorno de trabajo.
git init
git add .
git commit -m'Inicialización'
git branch Trabajo
git checkout Trabajo
```



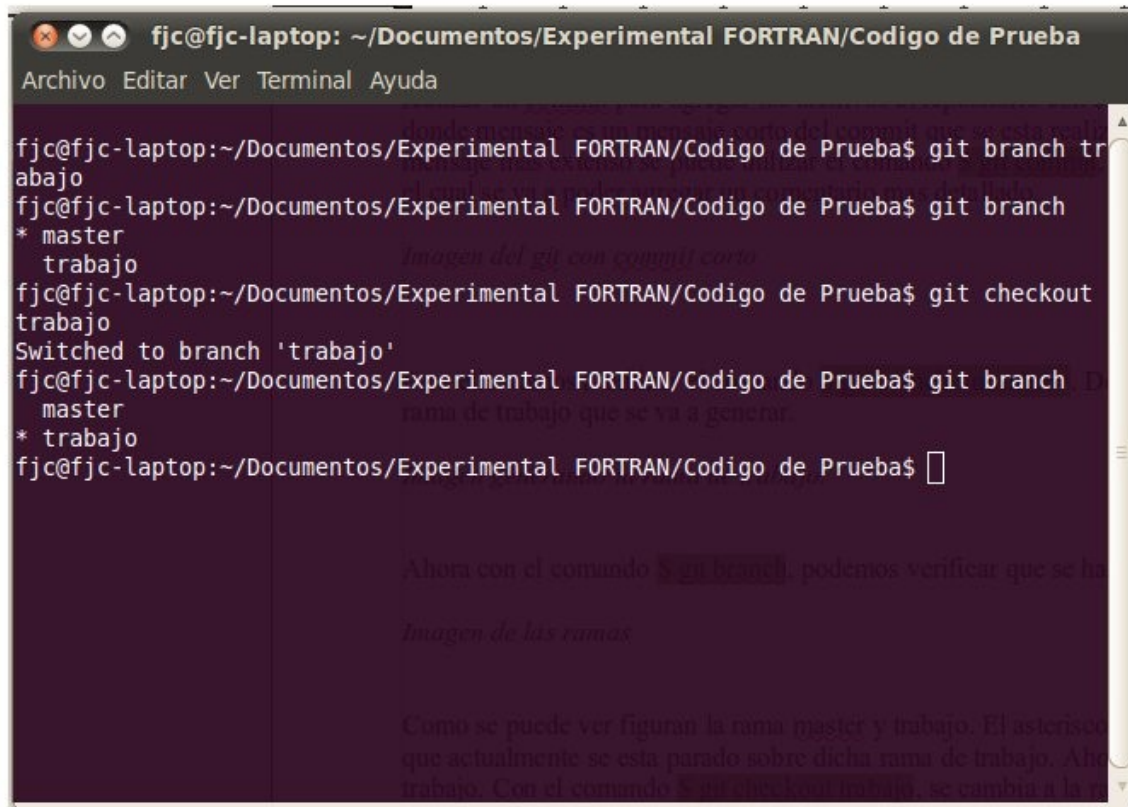
The image shows a terminal window titled "fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba". The window has a menu bar with "Archivo", "Editar", "Ver", "Terminal", and "Ayuda". The terminal output shows the following commands and results:

```
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git branch trabajo
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git branch
* master
  trabajo
```

Below the terminal output, there is a faint, semi-transparent text overlay that reads: "Imagen del git con commit corto", "Para ello vamos a utilizar el comando \$ git branch <nombre>. D", "rama de trabajo que se va a generar.", "Imagen generando la rama de trabajo.", and "Ahora con el comando \$ git branch, podemos verificar que se ha".

Ilustración 58 - Corroboración de que se creó la rama y sobre qué rama se está posicionado.

Por último nos debemos posicionar en la rama trabajo. Para esto, vamos a tipear el comando `$ git checkout trabajo`.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git branch trabajo
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git branch
* master
  trabajo
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git checkout trabajo
Switched to branch 'trabajo'
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git branch
* master
  trabajo
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

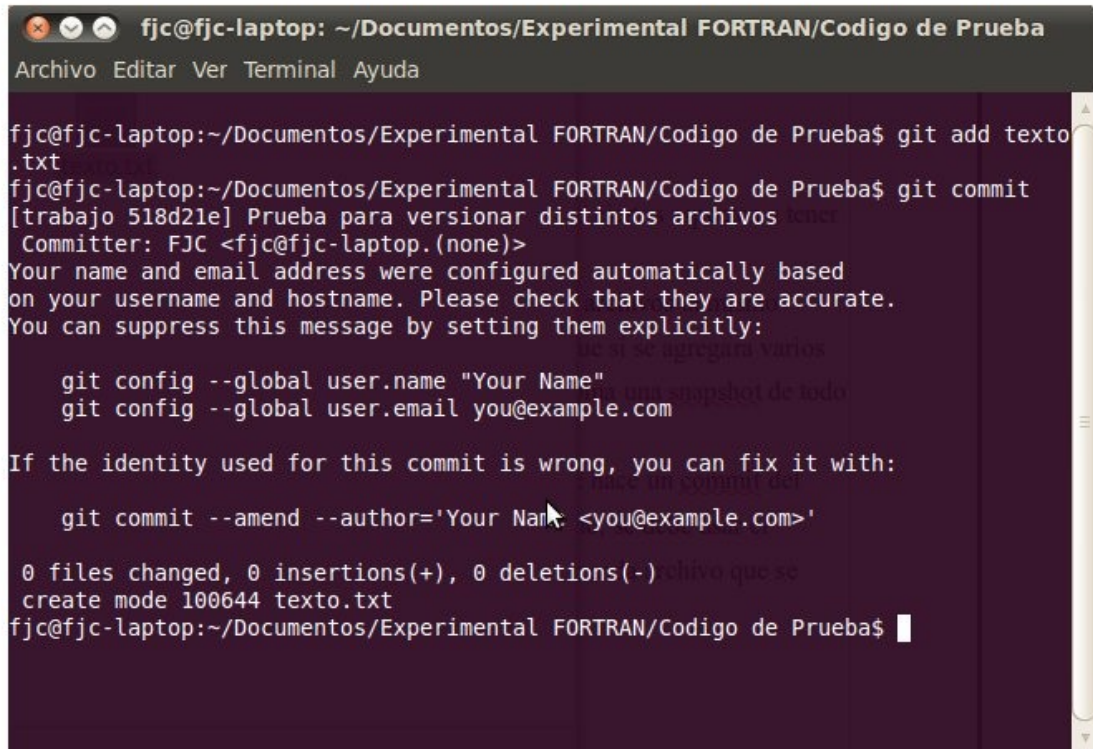
Ilustración 59 - Cambiar de rama y comprobación de que se cambió de rama.

Con los pasos que se han llevado a cabo hasta acá, se ha finalizado la inicialización del repositorio de Git, para ser utilizado en el proyecto.

10.1.2. Recomendaciones para utilizar Git.

En el siguiente apartado, se procede a brindar recomendaciones sobre el manejo de los comandos de Git que se han utilizado en este proyecto.

- El comando add puede usarse para tomar una snapshot de uno o más archivos al mismo tiempo. Si a dicho comando, se le agrega un punto (`add .`), funciona de la misma manera que si se agregaran varios archivos al mismo tiempo con el comando `git add`, con la diferencia de que toma una snapshot de todo el directorio donde se encuentra (archivos y subdirectorios).
- La operación commit, hay que tener en cuenta que hace un commit del snapshot. Por lo tanto, en caso de querer realizar un commit individual para cada archivo que ha sido modificado, se debe usar el comando `$ git commit <Archivo>` o `$ git commit <Archivo> -m 'Texto'`. Esto se hace por cada archivo que se quiera commitear por separado.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda

fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add texto.txt
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git commit
[trabajo 518d21e] Prueba para versionar distintos archivos
Committer: FJC <fjc@fjc-laptop.(none)>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

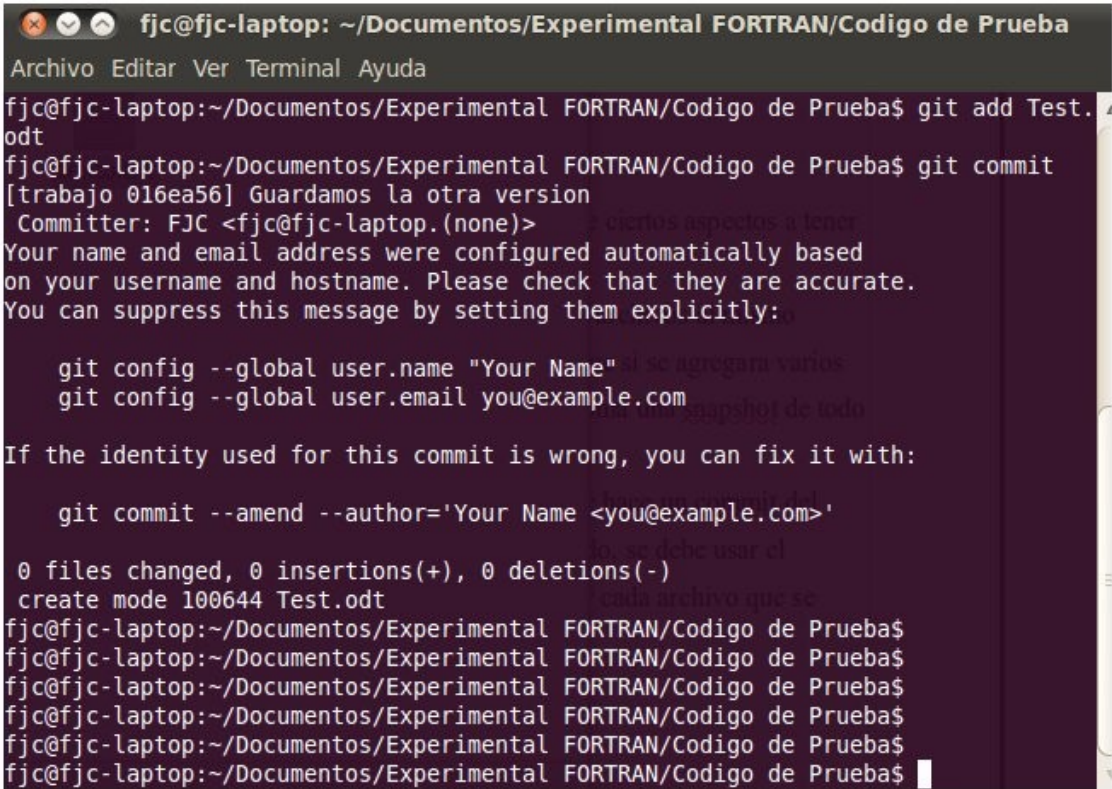
    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

0 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 texto.txt
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 60 - Agregando archivos modificados por separados. Imagen 1.



```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git add Test.odt
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git commit
[trabajo 016ea56] Guardamos la otra version
Committer: FJC <fjc@fjc-laptop.(none)>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'

0 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Test.odt
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$
```

Ilustración 61 - Agregando archivos modificados por separados. Imagen 2.

10.1.3. Rollback con Git.

En caso de ser necesario realizar un rollback a la versión anterior o una versión más antigua, se debe utilizar la opción checkout. Para esto, primero deberemos escribir el comando `$ git log`, con el fin de poder determinar a qué versión es a la que queremos retornar y el valor del commit que posee dicha versión. Una vez que se han obtenido los datos necesarios, escribimos el comando `$ git reset - -hard <valor>`, donde valor, va a ser el número de commit que obtuvimos al usar la opción log.

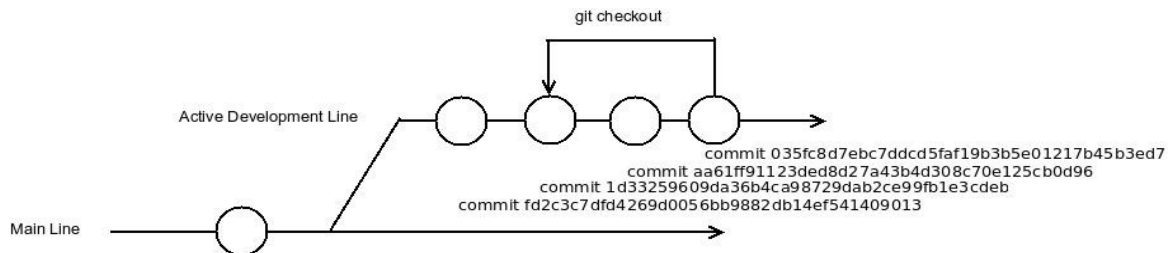


Ilustración 62 - Ejecución del rollback de una modificación.

Se debe copiar por completo el valor del commit. Otra opción es a través del comando `$ git log --oneline`, y copiar el valor que aparece al lado izquierdo del commit que necesitamos.

```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda
fjc@fjc-laptop:~$ cd Documentos/Experimental\ FORTRAN/Codigo\ de\ Prueba/
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git log
commit 016ea56cf285a8db362d39a782db11f92f1f0d47
Author: FJC <fjc@fjc-laptop.(none)>
Date: Sun Jun 5 20:44:58 2011 -0300

Guardamos la otra version anterior o una version más antigua, se
log, para poder

commit 518d21eb03d6134731d3049440e4530fc7c38d81
Author: FJC <fjc@fjc-laptop.(none)>
Date: Sun Jun 5 20:43:47 2011 -0300

Prueba para versionar distintos archivos

Esto es un archivo de texto.

commit e39f19c3f1eb01d266cc07dbef2ea47fa69565d8
Author: FJC <fjc@fjc-laptop.(none)>
Date: Sun Jun 5 20:38:59 2011 -0300

Version 0.0 del Texto

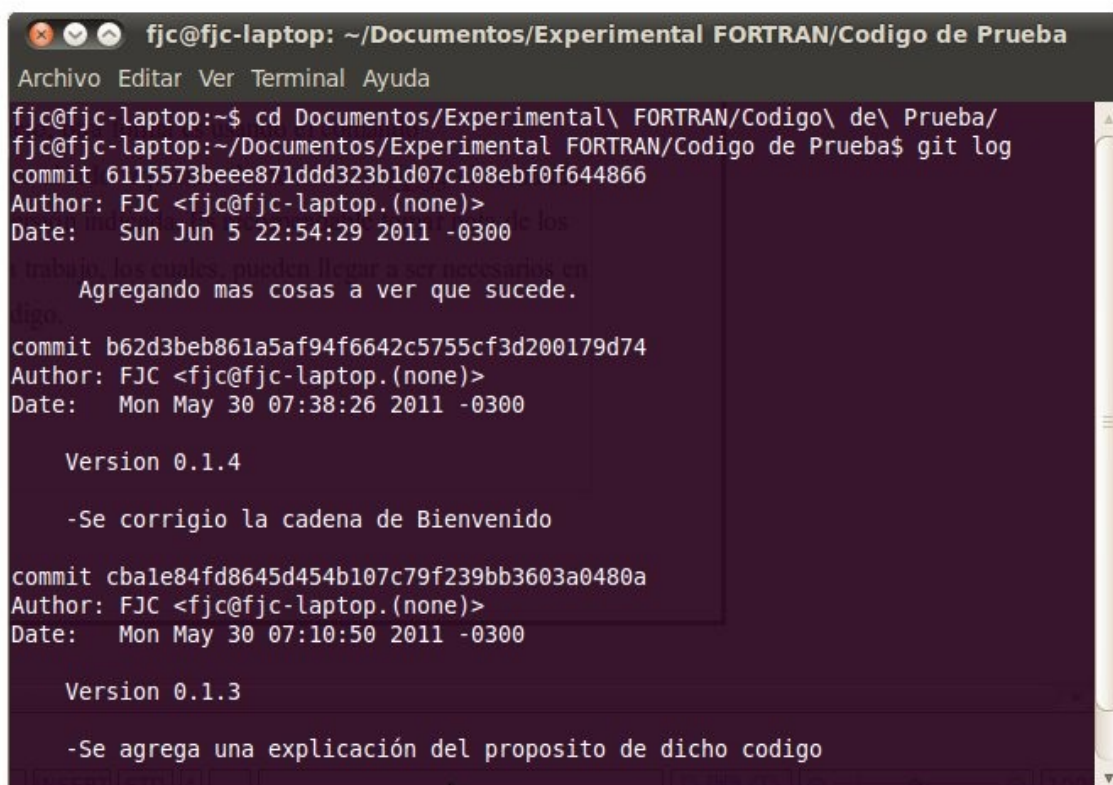
commit 74fa45b3d30e40a16bc7a94a2045b59e3e697696
Author: FJC <fjc@fjc-laptop.(none)>
```

Ilustración 63 - Usando comando log para determinar la versión y valor del commit.

```
Version 0.1.4
fjc@fjc-laptop:~/Documentos/0/Demostracion/Codificacion$ git reset --hard 32cb3c
2ff7be68318b9fb553998daaf019e23fe1
HEAD is now at 32cb3c2 Se agrego espacio al principio del codigo
```

Ilustración 64 - Ejecutando el rollback.

Con escribir el comando log se puede corroborar que se ha vuelto a la versión indicada. Es recomendable tomar nota de los valores del último commit realizado en la rama trabajo. Estos valores pueden llegar a ser necesarios, en caso de se quiera volver a dicha versión del código.

A screenshot of a terminal window titled "fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba". The terminal shows the execution of "git log" which displays three commit entries. The first entry is for commit 6115573beee871ddd323b1d07c108ebf0f644866, dated Sun Jun 5 22:54:29 2011, with the message "Agregando mas cosas a ver que sucede." and "Version 0.1.4". The second entry is for commit b62d3beb861a5af94f6642c5755cf3d200179d74, dated Mon May 30 07:38:26 2011, with the message "-Se corrigio la cadena de Bienvenido" and "Version 0.1.4". The third entry is for commit cbale84fd8645d454b107c79f239bb3603a0480a, dated Mon May 30 07:10:50 2011, with the message "-Se agrega una explicación del proposito de dicho codigo" and "Version 0.1.3".

```
fjc@fjc-laptop: ~/Documentos/Experimental FORTRAN/Codigo de Prueba
Archivo Editar Ver Terminal Ayuda
fjc@fjc-laptop:~$ cd Documentos/Experimental\ FORTRAN/Codigo\ de\ Prueba/
fjc@fjc-laptop:~/Documentos/Experimental FORTRAN/Codigo de Prueba$ git log
commit 6115573beee871ddd323b1d07c108ebf0f644866
Author: FJC <fjc@fjc-laptop.(none)>
Date:   Sun Jun 5 22:54:29 2011 -0300

    Agregando mas cosas a ver que sucede.

    Version 0.1.4

    -Se corrigio la cadena de Bienvenido

commit b62d3beb861a5af94f6642c5755cf3d200179d74
Author: FJC <fjc@fjc-laptop.(none)>
Date:   Mon May 30 07:38:26 2011 -0300

    Version 0.1.4

    -Se corrigio la cadena de Bienvenido

commit cbale84fd8645d454b107c79f239bb3603a0480a
Author: FJC <fjc@fjc-laptop.(none)>
Date:   Mon May 30 07:10:50 2011 -0300

    Version 0.1.3

    -Se agrega una explicación del proposito de dicho codigo
```

Ilustración 65 - Corroborando que se retornó a la versión anterior.

10.1.4. Requisitos para implementar Integration Build

Para poder implementar la Integration Build, es necesario haber instalado algún sistema control de versión y haber preparado el ambiente de trabajo del proyecto.

10.1.5. Recomendaciones al trabajar con Integration Build.

Para poder realizar el proceso de integración, tanto en nuestro propio espacio de trabajo como en el main line, es necesario contar con todos los componentes necesarios y actualizados a la última versión estable, para poder llevar a cabo una correcta integración (sub módulos, base de datos, apis, dll, scripts, etc).

Se recomienda atomizar los cambios, a fin de reducir el tiempo de integración y poder obtener una mayor cantidad de ellas, para que el proceso de integración sea más exitoso.

Los test que se realicen deben ser ágiles y rápidos, para evitar demoras innecesarias en el proceso de integración.

10.2. Presentación

10.2.1. Speech para el Workshop.

Tenemos este programa escrito en FORTRAN que no esta convergiendo y pensamos que puede ser un problema de precisión, por lo que decidimos cambiar todas las variables de nuestro código a doble precisión. A medida que vamos llevando a cabo dicha modificación, van surgiendo distintas versiones del código, hasta obtener la versión definitiva de este.

Pero resulta que el programa sigue sin converger, por lo que debemos retornar a la versión original. Sin embargo, tenemos el problema de que no tenemos una copia del código original, por lo que el rollback debe ser llevado manualmente, con lo que esto implica, ya que hay que ir deshaciendo cada una de las modificaciones que realizamos en el código, evitando obviamente que este se estropee en el proceso.

Este problema ya ha sido resuelto dentro del ámbito de la Ingeniería de Software, pero las

soluciones están orientadas al software comercial. Lo que se ha hecho, es tomar algunas de estas soluciones y adaptarlas al desarrollo de software científico. Lo que se ha escogido, son dos patrones de SCM, que son soluciones genéricas y se encuentran estandarizadas.

Por ejemplo, el active development line nos permite realizar modificaciones al código preservando tanto la versión original como las distintas versiones que se vayan generando a raíz de esto. Para esto se toman snapshots, estas son fotos de las distintas versiones que se van generando producto de los cambios que vamos realizando y que las podemos ir guardando. El lugar donde las guardamos son ramas o branches, que nos permiten separar la foto original que le tomamos al código de las fotos que fuimos sacando a las distintas versiones que fuimos obteniendo. De esta forma en una rama voy guardando y preservando las distintas versiones que obtuve y en otra tengo salvaguardada la versión original. A su vez podemos ejecutar rollbacks, ya sea a una foto anterior o a la foto original. Todas estas fotos se puede agregar un tag, es decir, una etiqueta que nos permita identificar cada una de las fotos que hemos ido guardando en las distintas ramas, e inclusive, podemos agregarle a cada una de las fotos una descripción de lo que se le ha hecho. Por último, podemos agregar algunas fotos o todas ellas a la rama principal, dependiendo de las necesidades de ese momento.

La integración nos permite ir agregando estas fotos que fuimos sacando al main line paulatinamente. De esta forma, se evita que se produzcan errores al momento de ser agregadas. Para esto, se cuenta con un workspace, un espacio de trabajo especial, que va a contener todos los elementos necesarios para que nosotros armemos temporalmente una build, es decir que vamos a realizar un foto-montaje con todos los componentes que tenemos almacenados. Esta foto-montada va a ser sometida a testings con el fin de determinar si puede ser agregada al main line. Si este foto-montaje local tiene éxito, entonces vamos a importar a nuestro espacio de trabajo la foto original para repetir el proceso nuevamente, con la finalidad de corroborar que no se valla a estropear la foto original cuando intentemos agregar nuestras fotos. Si se pasa exitosamente esta etapa, entonces podemos agregar nuestras fotos al main line.

Vamos a proceder con una pequeña demostración de lo explicado hasta aquí.

Con esto, se busca realizar un intercambio de conocimientos entre nosotros y ustedes,

ofreciéndoles estas soluciones genéricas adaptadas, que permiten lidiar con los problemas que se suscitan a diario.

Hay que tomar en cuenta que el desarrollo de software científico debe ser reconocida como una disciplina especial, y que por ende se debe tomar en cuenta de que no se pueden aplicar las soluciones que existen actualmente en el mercado de desarrollo de software directamente sin haber realizado una adaptación de estas.

10.2.2. Pasos de la Demostración del Workshop.

1. Posicionarse en la rama trabajo (la rama nueva).
2. Modificar el código.
3. Grabar las modificaciones.
4. Mostrar el log.
5. Compilar el código.
6. Ejecutar el programa que se creó.
7. Realizar la operación merge.
8. Mostrar el log.
9. Ejecutar un rollback.
10. Mostrar el código.

10.2.3. Filminas de la presentación del Workshop.

¿ Que pasa si se quiere realizar una modificación ?

```
SUBROUTINE secante(f,x0,x1,n,tol,raiz,clave)
! -----
! ALGORITMO DE LA SECANTE para encontrar una solución
! de f(x)=0, siendo f una función continua, dada las
! aproximaciones iniciales x0 y x1.
! -----
! Bloque de declaración de argumentos
! -----
USE precision, WP => DP
IMPLICIT NONE
INTERFACE
FUNCTION f(x)
! Función que define la ecuación
USE precision, WP => DP
IMPLICIT NONE
REAL(WP) :: f
REAL(WP), INTENT(IN) :: x
END FUNCTION f
END INTERFACE
REAL(WP), INTENT(IN)
```

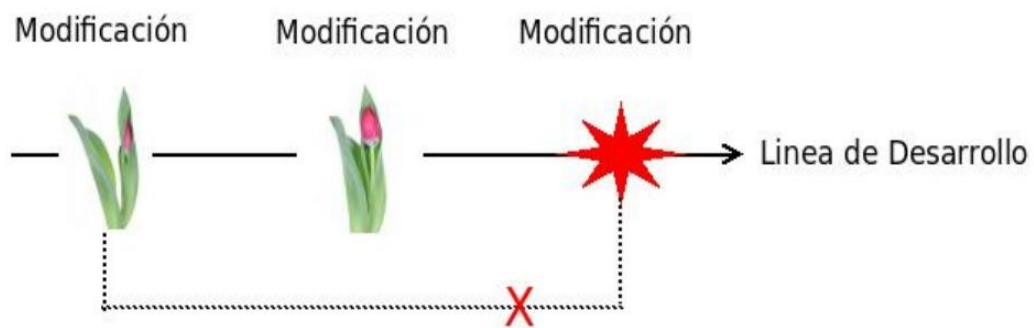
El programa no esta produciendo los resultados esperados y sospechamos que puede ser un problema de convergencia.

¿ Que pasa si se quiere realizar una modificación ?

```
REAL(WP), INTENT(IN)
:: tol
! Tolerancia para el error relativo
REAL(WP), INTENT(OUT)
:: raiz ! Aproximación a la raiz
INTEGER, INTENT(OUT)
:: clave ! Clave de éxito:
!
0 : éxito
! >0 : iteraciones excedidas
! -----
! Bloque de declaración de variables locales
! -----
INTEGER :: i
REAL(WP):: xx0, xx1, fx0, fx1
! -----
! Bloque de procesamiento
! -----
xx0 = x0
xx1 = x1
```

Se toma la decisión de modificar todas las variables a doble precisión.

¿ Qué pasa si falla la modificación ?



El problema persiste y no se puede retornar directamente a la versión original.

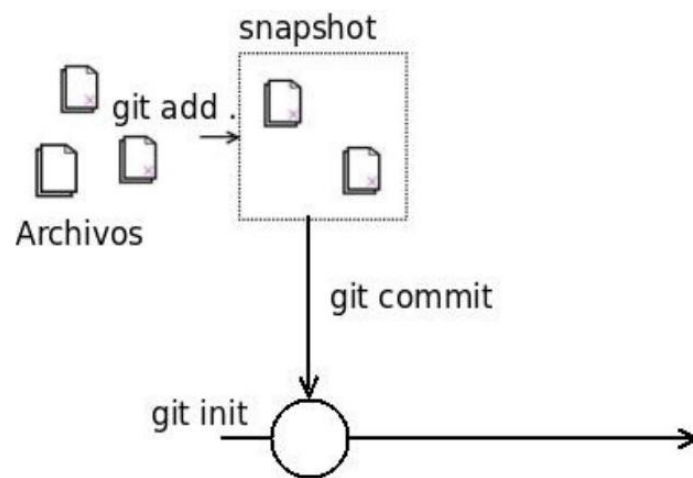
Solución

Active Development Line

Integration Build

Se propone la utilización de algunos patrones de SCM para atacar esta clase de problema.

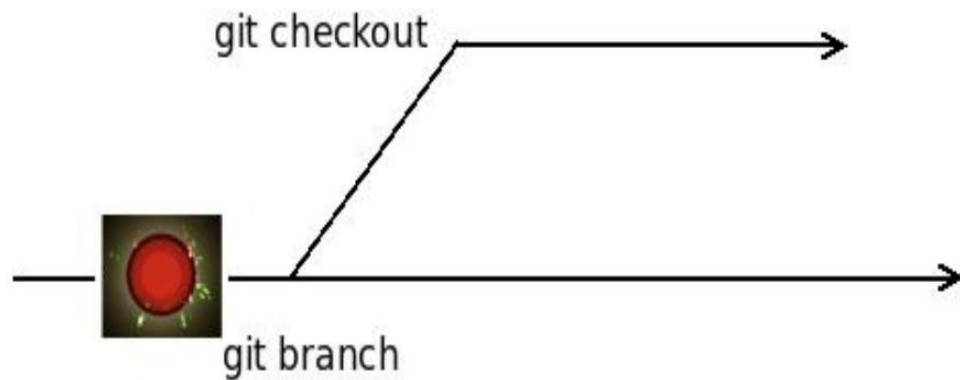
Active Development Line Inicialización



Antes de comenzar, es necesario preparar el espacio de trabajo. Para eso, hay que tomar una snapshot inicial.

Active Development Line

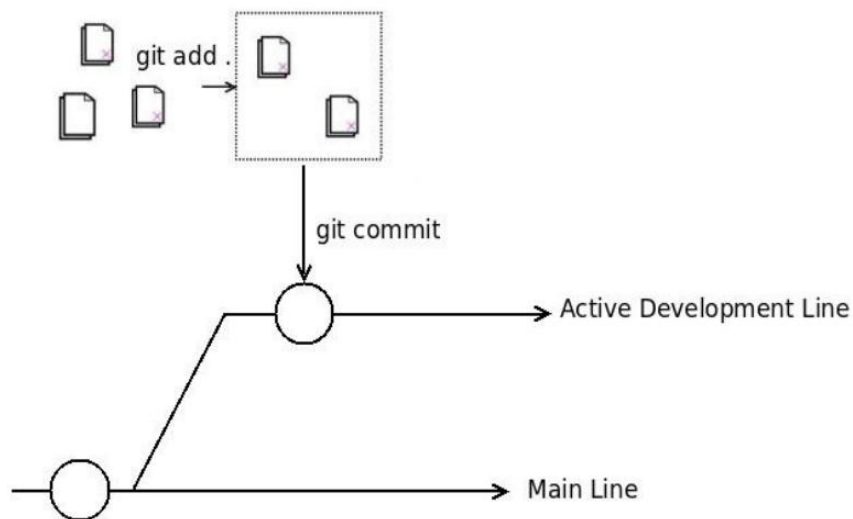
Inicialización



Se crea una branch de trabajo, para separar las distintas snapshot de las modificaciones de la snapshot original.

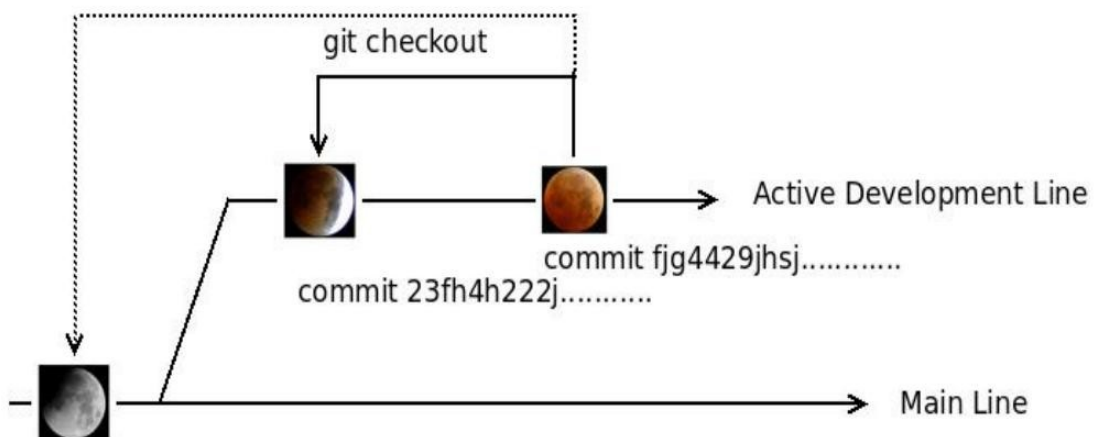
Active Development Line

Trabajar con Modificaciones



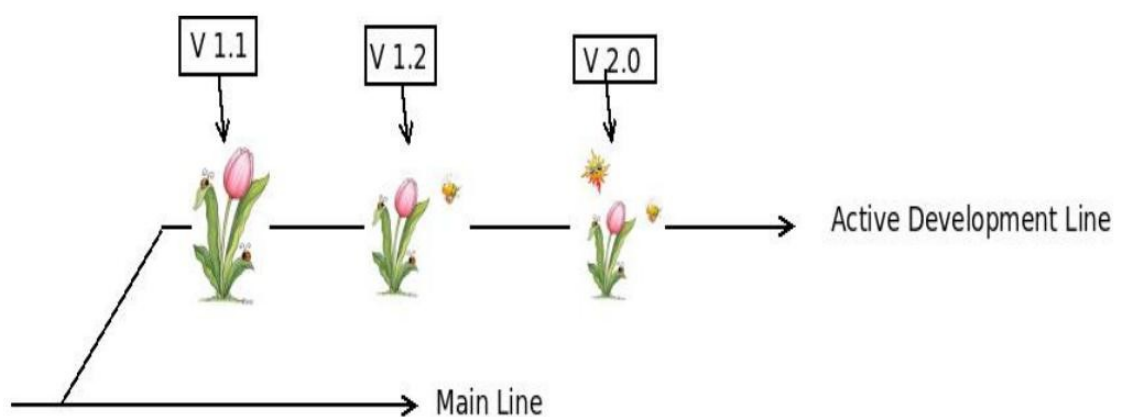
Se toma una snapshot de la modificación y se la commitea a la línea de desarrollo para guardarla.

Active Development Line Rollback



Se pueden ejecutar rollbacks a versiones anteriores o a la versión original.

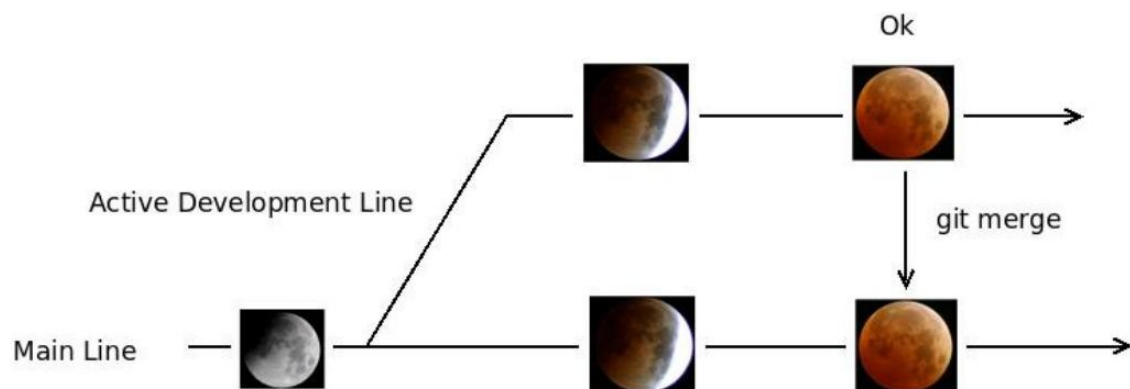
Active Development Line Tags



Se puede agregar tags y descripción a las snapshot de las modificaciones, para poder identificarlas.

Active Development Line

Merge con el Main Line



Se puede agregar las distintas snapshot a la rama principal o solamente algunas de ellas.

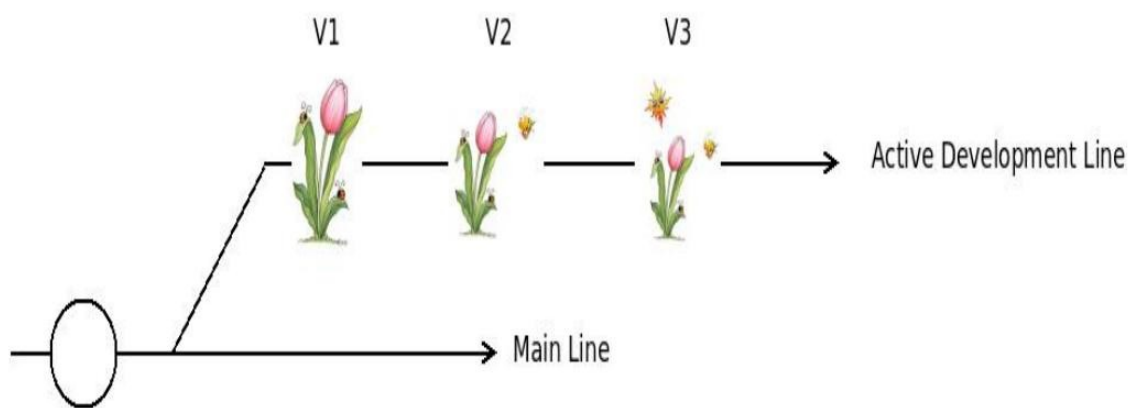
Realizar una modificación

```
! -----  
INTEGER :: i  
REAL(WP):: xx0, xx1, fx0, fx1  
! -----  
! Bloque de procesamiento  
! -----  
xx0 = x0  
xx1 = x1  
fx0 = f(x0)  
fx1 = f(x1)  
DO i= 2,n  
  raiz = xx1 - fx1*((xx1-xx0)/(fx1-fx0))  
  IF (ABS((raiz-xx1)/raiz) < tol) THEN  
    clave = 0  
    n  
    = i  
    RETURN  
  ENDIF  
7  
  xx0 = xx1
```

Retomamos nuestro problema. El programa no produce los resultados esperados.

Integration Build

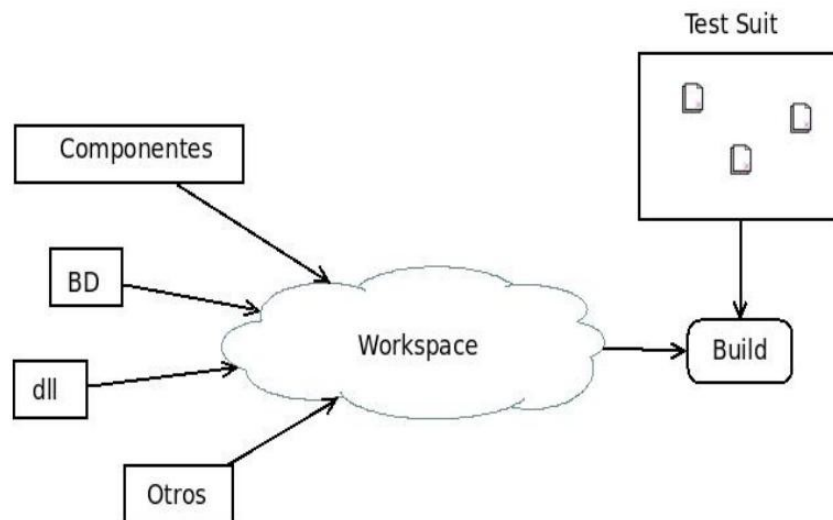
¿Como funciona?



Tenemos los distintos commits que guardamos y queremos agregar algunos o todos a la linea principal (main line).

Integration Build

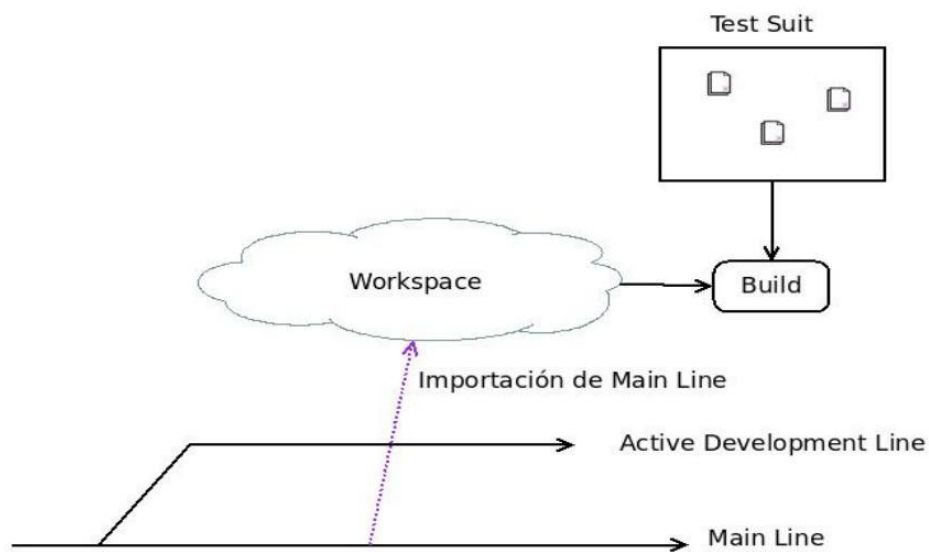
Integración en el Espacio de Trabajo



Se utiliza un espacio de trabajo especial que contiene todo lo necesario para llevar a cabo la integración, generar una build y someterla a una suit de testing.

Integration Build

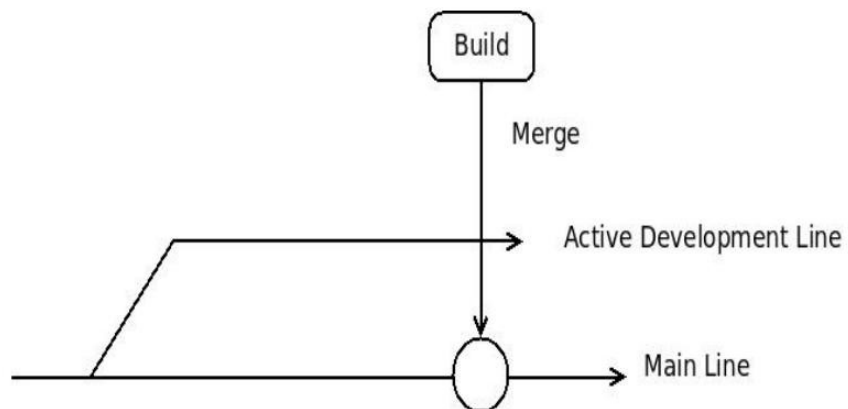
Integración con los Componentes del Main Line



Si la integración local (rama de desarrollo) pasa los test satisfactoriamente, se puede proceder a realizar el mismo proceso con el commit original de la rama principal.

Integration Build

Ejecución del Merge



Si la build que se genero del proceso de integración original pasa el testing, se puede agregar las modificaciones realizadas al main line.

DEMOSTRACIÓN

10.3. Validación de la Propuesta.

Para llevar a cabo la validación de la propuesta de este proyecto, se analizaron diversas alternativas, siendo una demostración directa de la implementación el mejor mecanismo que proveería un feedback. Esta demostración se realizó en el “Primer WorkShop: El Desarrollo de Software en ambientes Científicos y Técnicos”, efectuado en el Instituto Universitario Aeronáutico de Córdoba – Argentina, en el mes de Octubre del 2011. Esta, debía introducir el contexto y marco teórico presentando de una manera práctica la solución propuesta en este trabajo, a través de la gestión de configuración, que ha sido el

objeto de estudio de este trabajo. Los resultados obtenidos de esta presentación, sirvieron para ajustar distintos aspectos de la implementación y delinear futuros estudios dentro de esta área.

A través de una demostración, se llevo a cabo la implementación de las soluciones expuestas en esta publicación. A tal fin, se utilizó la herramienta GIT, la cual es un repositorio, en conjunto con scripts. Como lenguaje de programación se utilizó FORTRAN 95.

Para el active development line, se creó la rama trabajo que permitió mejorar el manejo de las modificaciones, preservando el código original y las distintas versiones que se originaron de este. En una rama (vendría a ser el main line) se guardó la versión original del código, mientras que las distintas modificaciones realizadas se almacenaron en la rama trabajo. De esta manera, se preservó la versión original del código al momento de realizar las modificaciones. Al contar con todas las versiones almacenadas en un repositorio, se pudo ejecutar fácilmente los rollbacks a cualquiera de las versiones, eliminando la complejidad y los riesgos que posee la ejecución de esta operación manualmente. Cada una de las versiones contó con una nota de liberación, que incluía información sobre que se había realizado en esa modificación, permitiendo determinar su utilidad para los desarrolladores. A su vez, los tags brindaron una identificación de cada una de las versiones, lo que facilitó la ejecución de operaciones como el rollback y merge.

La integration build permitió mejorar la construcción de las builds al centralizar todos los elementos necesarios para llevar a cabo dicho proceso. Al definirse el tiempo que debía demorar un ciclo de integración, permitió agilizar este proceso evitando así que se lo realizara innecesariamente. Además, permitió corroborar que las modificaciones que se habían realizado, se integraban correctamente con los distintos elementos de nuestro propio workspace, así como del main line. Para ejecutar la demostración de la integración, se utilizó los tiempos definidos en el proyecto de grado y se simplificó el proceso, ya que los pasos y modificaciones que se utilizaron no eran extensas.

Como agregado a esta implementación, se utilizaron scripts que permitieron mejorar el rendimiento de las operaciones. Esto evitó un uso extenso de la consola de linux, ya que permitió brindar cierta automatización de las soluciones propuestas, evitando que se

tuviera que escribir varios de los comandos para llevar a cabo las distintas operaciones, e inclusive, el tener que posicionarse en el directorio de trabajo.

Del feedback obtenido de la demostración, se pudo corroborar que el tiempo es un factor crucial en el desarrollo de software científico y una de las mayores preocupación de los responsables de los grupos de investigación. Además, se detecto la necesidad de gestionar un desarrollo distribuido, así como la necesidad de que las soluciones y herramientas puedan soportar esta situación. También manejan distintas plataformas (SO) y diferentes tipos de archivos, lo que hace que necesiten un sistema de gestión robusto capaz de soportar estas condiciones de trabajo. Asimismo, demostraron interés por la solución, específicamente en aquellos apartados que son críticos para ellos. Si esta solución en conjunto con las herramientas propuestas eran capaces de manejar un desarrollo distribuido, múltiples formatos de archivos y si es libre o comercial. Por otra parte, la idea de mantener una línea estable a la que puedan retornar les fue interesante ya que les permite mejorar la gestión de sus proyectos de desarrollo, así como el proceso de integración les brindaba cierto nivel de seguridad y calidad al desarrollo. Con toda la información recabada de esta demostración, se pudo constatar la utilidad de la solución propuesta, a la vez que se obtuvo datos relevantes respecto al desarrollo del software científico, que permite dejar la puerta abierta para seguir realizando trabajos futuros sobre este ámbito.