FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

Análisis de la Efectividad y el Costo de 5 Técnicas de Verificación

Proyecto de Grado

María Stephanie De León
sdeleon@fing.edu.uy

Rosana Robaina Rode
rrobaina@fing.edu.uy

Tutor
Diego Vallespir
dvallesp@fing.edu.uy

Montevideo, Uruguay
Diciembre, 2009

Resumen

En el presente trabajo se aplica un experimento formal para comparar cinco técnicas de verificación: la técnica estática Inspecciones; las técnicas dinámicas de caja negra Partición de Equivalencia con Análisis de Valores Límite y Tablas de Decisión; y las técnicas estáticas de caja blanca Criterio de Cubrimiento de Condición Múltiple y Trayectorias Linealmente Independientes.

En el experimento se comparan las técnicas de verificación respecto a la efectividad y el costo en la detección de defectos. Catorce estudiantes avanzados de la carrera de Ingeniería en Computación de la Facultad de Ingeniería de la Universidad de la República, aplicaron las cinco técnicas de verificación a cuatro programas, según el diseño del experimento.

El resultado principal del estudio es que las técnicas de verificación Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite son más efectivas que la técnica Trayectorias Linealmente Independientes.

Agradecimientos

Agradecemos a todas las personas que de una u otra manera estuvieron relacionadas con el proceso de llevar a cabo este proyecto de grado.

A Diego Vallespir, nuestro Tutor, ya que sin su guía este proyecto de grado no hubiera sido posible.

A Celina Gutiérrez por toda la ayuda brindada para que podamos realizar el análisis de datos. Sin ella la tarea de analizar los datos hubiera sido mucho más ardua, y quizás errónea.

A todos los que leyeron la tesis en busca de mejoras y errores, incluso no conociendo de informática: gracias Damián y Manuel.

A Cecilia Apa, por la buena disposición y garra para sacar nuestros proyectos de grado adelante.

A toda nuestra familia por el apoyo.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La verificación de software es una disciplina crítica en el desarrollo de software. Un software con fallas o no performante puede ser muy costoso.

A la vez que es necesaria, la verificación de software también es costosa. Para planificar y ejecutar las pruebas, los verificadores deben considerar el software y la función que realiza, las entradas y cómo pueden ser combinadas, y el ambiente en el que el software eventualmente va a funcionar. Este difícil proceso consume tiempo y requiere sofisticación técnica y una adecuada planificación.

La selección de las técnicas de verificación a ser utilizadas es uno de los aspectos de la planificación para la cual es esencial un conocimiento objetivo y actualizado de las mismas. Existen técnicas que se basan en la revisión del código fuente, presentando distintas metodologías para llevarlo a cabo. Otras técnicas se basan en la ejecución del software, determinando diferentes criterios para la selección de casos de prueba que serán utilizados como entrada al sistema bajo estudio. La verificación será tan exitosa como efectivos y eficientes sean los casos de prueba seleccionados, o la metodología de revisión de código utilizada.

1.1. Contexto y Motivación

Conocer cuál técnica es la que mejor se ajusta en determinado contexto es un resultado sumamente útil, ya que permite a las organizaciones planificar la estrategia de verificación optimizando los recursos y por lo tanto reduciendo los costos. Además, si se conoce los tipos de defecto que presenta el software, puede utilizarse una técnica adecuada a dichos tipos de defecto.

Este estudio se enmarca en una propuesta para definir una arquitectura de procesos para la verificación *unitaria*. El fin es sentar las bases para poder conocer la efectividad y el costo de algunas técnicas de verificación con relación a ciertos tipos de defectos, para lograr establecer una estrategia óptima de remoción de defectos.

1.2. Problema y Objetivos

Actualmente no se cuenta con una herramienta o método que permita determinar cuál técnica de verificación utilizar ante determinada situación, como

ser un tipo de software, ciertos tipos de defectos, el nivel de experiencia en verificación que poseen los sujetos que le darán uso, etc. Esto trae aparejado una perdida de tiempo considerable si se selecciona en primera instancia una técnica que luego no resulta de utilidad, o si la técnica seleccionada no logra detectar defectos que resultan en fallas reportadas por los usuarios una vez que el sistema está en producción.

El objetivo general de este proyecto de grado es conocer, mediante la realización de un experimento formal, el costo y la efectividad de determinadas *técnicas de verificación* respecto a determinados *tipos de defectos*; y generar productos, herramientas y mecanismos para llevar a cabo experimentos formales que permitan reforzar los resultados obtenidos en esta investigación y otros de la índole.

1.3. Experimento

El experimento llevado a cabo en el marco de este proyecto de grado evalúa la efectividad y la eficiencia de cinco técnicas de verificación respecto a determinados tipos de defecto. Las técnicas bajo estudio son Inspecciones, Partición de Equivalencia con Análisis de Valores Límite, Tablas de Decisión, Cubrimiento de Condición Múltiple y Trayectorias Linealmente Independientes. Las clasificaciones de defectos utilizadas son la Clasificación Ortogonal de Defectos de IBM [Cla] y la Taxonomía de Defectos de Beizer [Bei90].

Inspecciones es una técnica estática o de lectura de código. Partición de Equivalencia con Análisis de Valores Límite y Tablas de Decisión son técnicas dinámicas de caja negra. Cubrimiento de Condición Múltiple y Trayectorias Linealmente Independientes son técnicas dinámicas de caja blanca.

El proceso experimental utilizado para realizar el experimento se basa en el descrito por Wohlin en el libro *Experimentation in Software Engineering: an Introduction* [WRH+00]. El mismo consta de 5 etapas: Definición, Planificación, Operación, Análisis e Interpretación, y Presentación y Empaquetado. La Presentación y Empaquetado del experimento se compone del presente informe y de los documentos entregados junto con el mismo.

En la etapa de Definición se definió en términos formales el objetivo del experimento. En la etapa de Planificación se formularon las hipótesis y el diseño del experimento, y se seleccionaron los sujetos que participarían del mismo. Se plantearon dos hipótesis sobre la efectividad de las técnicas de verificación, una que contempla la efectividad por tipo de defecto y otra que no realiza la distinción por tipo de defecto.

El diseño experimental resultó en un total de 40 verificaciones, constituidas de la siguiente manera: 13 sujetos aplicarían 3 técnicas de verificación distintas sobre 3 programas distintos (de un total de cuatro programas), y 1 sujeto aplicaría una técnica de verificación sobre uno de los programas. De esta forma, todas las técnicas serían ejecutadas 8 veces.

Los sujetos son estudiantes avanzados de la carrera Ingeniería en Computación de la Facultad de Ingeniería, de la Universidad de la República, que se encuentran realizando un curso-taller denominado Módulo de Taller. La selección de los estudiantes dentro de este curso fue aleatoria. Todos han cursado la materia Introducción a la Ingeniería de Software, por lo que se considera que poseen un conocimiento básico sobre verificación de software.

La Operación del experimento se dividió en dos fases: una de entrenamiento y otra de ejecución del experimento. Durante la etapa de entrenamiento los sujetos recibieron capacitación sobre las técnicas de verificación a utilizar y las taxonomías donde clasificarían los defectos. Luego de la capacitación se realizó una instancia de verificación previa al experimento, en la que cada sujeto aplicó las tres técnicas que utilizaría en el experimento sobre un mismo programa. Durante esta instancia se realizaron devoluciones a los sujetos sobre cómo aplicaron las técnicas y las taxonomías de defectos.

La Operación del experimento se subdividió en tres instancias individuales por sujeto, una para cada técnica de verificación a utilizar; y en una instancia para el sujeto que debía ejecutar una sola técnica. Durante la ejecución del experimento 2 sujetos abandonaron, lo que resultó en la no ejecución de 4 verificaciones.

A partir de las 36 verificaciones realizadas durante el experimento se obtuvieron los siguientes resultados: las técnicas Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite (ambas técnicas dinámicas de caja negra) son más efectivas que Trayectorias Linealmente Independientes (técnica dinámica de caja blanca).

En comparación con estudios anteriores, nuestros resultados son respaldados por Basili y Selby [BS87], pero contrariados por los obtenidos por Kamsties y Lott [KL95], y por Juristo y Vegas [JV03]. Basili y Selby obtuvieron que las técnicas de verificación de caja negra y de lectura de código se comportan igual respecto a la efectividad, pero mejor que las técnicas de caja blanca. En cambio, Kamsties y Lott obtuvieron que no hay diferencia entre la efectividad de las técnicas, y Juristo y Vegas hallaron que las técnicas de verificación de caja blanca y caja negra presentan la misma efectividad.

El análisis de efectividad diferenciado por tipo de defecto no pudo llevarse a cabo. Los defectos fueron clasificados por los sujetos a medida que eran detectados. Las clasificaciones resultaron de una diversidad tal que resultó inviable determinar una única clasificación por defecto. Tampoco pudimos realizar la clasificación de defectos por nuestra parte, dada la cantidad de defectos que totalizaban todos los programas y el tiempo con el que contábamos para ello.

No se obtuvieron resultados respecto al costo de las técnicas de verificación. Los programas sobre los que fueron aplicadas poseen tamaños, complejidad y cantidad de defectos distintos, lo que provoca que el análisis de costo deba realizarse respecto a cada uno de ellos. De cada técnica se cuenta con una o dos observaciones por programa. Este hecho, sumado a que las observaciones de una misma técnica por programa resultaron notoriamente distintas, impide realizar un análisis comparativo del costo de las mismas.

La técnica Inspecciones fue la única que se mantuvo en valores cercanos de costo por programa. Esto permitió observar que a medida que crece el tamaño del programa la técnica se vuelve más costosa. Este resultado era de esperar, dado que la técnica Inspecciones se basa en realizar revisiones del código fuente del programa; cuanto más grande sea la cantidad de código a revisar, es de suponer que más tiempo será necesario para ejecutar la técnica.

1.4. Productos

Para la realización del experimento se desarrollaron una serie de productos. Algunos de ellos para la preparación de los sujetos, previo a la ejecución del experimento. El resto para la ejecución del experimento en sí. Estos productos se encuentran a disposición y pueden ser utilizados para la realización de otros experimentos, o para la replicación del realizado en el marco de este proyecto de grado.

Entre los productos para la preparación de los sujetos se encuentran los *Micro-cursos* y la *Experiencia Cero*. Los Micro-cursos son cursos pequeños, cada uno de dos clases de 2 horas como máximo, que introducen las técnicas de verificación a utilizar en el experimento, las clasificaciones de defectos, y la herramienta de registro de datos.

La *Experiencia Cero* fue una instancia intermedia entre la capacitación de los sujetos y la ejecución del experimento. Su finalidad fue aplicar los conocimientos adquiridos en los Micro-cursos previo a la ejecución del experimento. Durante esta experiencia los sujetos aplicaron las técnicas de verificación que utilizarían en el experimento sobre un programa pequeño.

Para la ejecución del experimento se desarrollaron las llamadas *Guías* y la herramienta de registro de información, denominada *Grillo*.

Las *Guías* indican cómo llevar a cabo la verificación de un programa. Describen cuales son los pasos que componen la verificación, y los materiales necesarios para realizar cada uno de ellos. También indican qué datos se deben registrar durante cada paso y los entregables que deben enviarse a los docentes una vez finalizado. Se desarrollaron tres guías, una por cada tipo de técnica de verificación a utilizar: lectura de código, técnicas de caja blanca y técnicas de caja negra.

El registro de datos se realizó en la herramienta *Grillo*. Esta herramienta fue desarrollada expresamente para el experimento. En ella se ingresa información de los defectos detectados, como ser ubicación en el código fuente del programa (archivo, línea de código, etc.), tiempo empleado en aislarlo y su clasificación en las taxonomías de defectos. También se registra el tiempo empleado en el diseño de casos de prueba (en el caso de las técnicas dinámicas) y el tiempo empleado en la ejecución propiamente dicha de la técnica (ejecución de los casos de prueba en el caso de las técnicas dinámicas).

Los programas utilizados en el experimento completan el paquete de productos desarrollados. Conforman un total de cuatro, y se denominan Contabilidad, Matemático, MO-Latex y Parser. Fueron construidos en el lenguaje de programación Java y documentados con JavaDoc. Los programas poseen tamaños, complejidades y cantidad de defectos distintos. Los defectos que presentan son los introducidos durante su construcción. No poseen defectos detectables por un compilador.

1.5. Publicaciones

En el marco de este proyecto de grado se realizaron las siguientes publicaciones:

- D. Vallespir, S. De León C. Apa, R. Robaina, y J. Herbert. Efectiveness of Five Verification Techniques. In *Proceedings of the XXVIII International*

Conference of the Chilean Computer Society (2009), Santiago de Chile (Chile), 11 2009. [VCARH09]

- D. Vallespir y S. De León. Análisis y Ejemplos de la Taxonomía de Defectos de Beizer. Technical Report RT 08-19, PEDECIBA, Instituto de Computación – Facultad de Ingeniería, Universidad de la República, 12 2008. [VL08]

1.6. Estructura del documento

Lo que sigue del documento se divide en cuatro Partes. La Parte I, *Introducción a la Ingeniería de Software Empírica y Estado del Arte*, presenta los conceptos básicos de la Ingeniería de Software Empírica, profundizando en los experimentos formales. También incluye una descripción breve de las técnicas de verificación bajo estudio; las clasificaciones de defectos utilizadas; y de los experimentos formales realizados por otros autores sobre la efectividad y el costo de las técnicas de verificación.

En la Parte II, *Experimento*, se presenta el experimento realizado en el marco de este proyecto de grado. En esta Parte se describen las etapas de Definición, Planificación, Operación, Análisis e Interpretación del experimento.

La descripción de los productos desarrollados para la ejecución y posterior replicación del experimento se presentan en la Parte III, *Productos*. En ella se detallan los productos y herramientas utilizados para la ejecución del experimento y para la preparación previa de los sujetos, incluyendo las críticas y mejoras identificadas. También se describen los programas utilizados.

En la Parte IV, *Conclusiones y Trabajo a Futuro*, se presentan las conclusiones del estudio, y el trabajo a futuro.

Junto con que este informe se entregan los siguientes documentos:

- Descripción de los Programas.- Contiene información de los programas desarrollados en el marco de este proyecto de grado
- Documentación de la Herramienta Grillo.- Presenta el análisis de requerimientos, modelo de casos de uso, diagrama de clases, diagrama de base de datos y otra documentación de la herramienta de registro Grillo.
- Estándar de Codificación Java.- Contiene el estándar de codificación utilizado para desarrollar los programas verificados en este proyecto de grado.
- Guías.- Presenta las guías de verificación, para los tres tipos de técnicas de verificación: estática, dinámica de caja blanca y dinámica de caja negra.
- Lista de Comprobación.- *Check-list* utilizada para realizar las inspecciones de código en la verificación de los programas.
- Micro-cursos.- Contiene las diapositivas utilizadas en los micro-cursos de la Taxonomía de Defectos de Beizer y de las técnicas de verificación Tablas de Decisión, Partición de Equivalencia con Análisis de Valores Límite y CCCM.
- Test de Alternativas Ordenadas.- Cálculos realizados para llevar a cabo el Test de Alternativas Ordenadas en el Análisis de Datos del experimento.

Parte I

Introducción a la Ingeniería de Software Empírica y Estado del Arte

Capítulo 2

Ingeniería de Software Empírica

Este capítulo está basado en los libros *Software Metrics: A Rigorous and Practical Approach* [FP98], *Experimentation in Software Engineering: an Introduction* [WRH+00], *Basics of Software Engineering Experimentation* [JM03] y *Handbook of Parametric and Nonparametric Statistical Procedures* [She03].

La Ingeniería de Software Empírica son métodos, herramientas, técnicas y otras formas de trabajo que permiten mostrar, mediante la práctica, aquello que no se puede demostrar con la teoría. Se basa en la experimentación como método para corresponder ideas o teorías con la realidad. En la Ingeniería de Software existen muchas interrogantes que resolver, un ejemplo de estas es conocer cuál técnica de verificación es mejor para encontrar defectos en un programa con determinadas características. Para responder a estas interrogantes se utiliza la experimentación, la cual refiere a mostrar con hechos las especulaciones, suposiciones y creencias sobre la construcción de software.

Se pueden distinguir dos enfoques diferentes al realizar una investigación empírica, el enfoque cualitativo y el cuantitativo. El primero se basa en estudiar la naturaleza del objeto y en interpretar un fenómeno a partir de la concepción que las personas tienen de él. Los datos que se obtienen de estas investigaciones están principalmente compuestos por texto, gráficas, imágenes, u otros.

El enfoque cuantitativo se corresponde con encontrar una relación numérica entre dos o más grupos. Se basa en cuantificar una relación o comparar variables o alternativas bajo estudio. Los datos que se obtienen en éste tipo de estudios son siempre valores numéricos, lo que permite realizar comparaciones y análisis estadísticos.

Es posible utilizar los enfoques cualitativos y cuantitativos para investigar el mismo tema, pero cada enfoque responde a diferentes interrogantes. Se puede considerar que estos enfoques son complementarios más que competitivos, ya que el enfoque cualitativo puede ser usado como base para definir la hipótesis que luego puede ser correspondida cuantitativamente con la realidad. Cabe destacar que las investigaciones cuantitativas pueden obtener resultados más justificables y formales que los cualitativos.

Existen muchas estrategias empíricas, dependiendo del propósito de la investigación que se desea realizar. Las tres estrategias principales son: encuestas,

casos de estudio y experimentos formales. Cada una de estas se utiliza en un tipo de investigación distinto.

Una *encuesta* es un estudio retrospectivo de las relaciones y las salidas de una situación. Se puede realizar este tipo de investigación cuando una técnica o herramienta ya ha sido utilizada, o antes de que ésta sea introducida. Las encuestas son realizadas sobre una muestra representativa de la población, y luego los resultados son generalizados al resto de la población. El ámbito donde son mas comunes son las ciencias sociales, por ejemplo, aquellas para determinar cómo la población va a votar en la siguiente elección. En la Ingeniería de Software Empírica las encuestas se utilizan de forma similar, se obtiene un conjunto de datos de un evento que ha ocurrido para determinar cómo reacciona la población frente a una técnica, herramienta o método particular, o para determinar relaciones o tendencias. En un estudio es fundamental seleccionar correctamente las variables a estudiar, pues de ellas dependen los resultados que se pueden obtener. Si los resultados no permiten concluir sobre los objetivos del estudio, se han elegido mal las variables. Una de las características más relevantes de las encuestas es que proveen un gran número de variables para estudiar. Esto hace posible construir una variedad de modelos y luego seleccionar el que mejor se ajusta a los propósitos de la investigación, evitando tener que especular cuales son las variables más relevantes. Las encuestas no proveen control sobre las variables ni las medidas de la situación bajo estudio. Dependiendo del diseño de la investigación (cuestionario) las encuestas pueden ser clasificadas como cualitativas o cuantitativas.

Los *casos de estudio* son un método observacional, esto significa que están basados en la observación de una actividad o proyecto durante su curso. Son utilizados para monitorizar proyectos, o actividades y para investigar entidades o fenómenos en un periodo específico. En un caso de estudio se identifican los factores claves que pueden afectar la salida de una actividad, y se documentan las entradas, las limitaciones, los recursos y las salidas. El nivel de control de la ejecución es menor en los casos de estudio que en los experimentos. Esto se debe principalmente a que en los casos de estudio no se controla, sólo se observa, contrario a lo que ocurre en los experimentos. Los casos de estudio son muy útiles en Ingeniería de Software. Se usan en la evaluación industrial de métodos y herramientas porque evitan problemas a gran escala. Además, son fáciles de planificar aunque los resultados son difíciles de generalizar y comprender. Los casos de estudio no manipulan las variables, sino que éstas son determinadas por la situación que se está investigando. Al igual que las encuestas, los casos de estudio pueden ser clasificados como cualitativos o cuantitativos dependiendo de lo que se quiera investigar del proyecto en curso.

Los *experimentos formales* son métodos de investigación rigurosos y controlados. Normalmente son ejecutados en laboratorios para lograr obtener el grado de control necesario, ya que en una situación real existen algunos factores muy difíciles de controlar y otros imposibles. En un experimento formal los factores claves deben ser identificados para manipular su comportamiento de forma precisa y sistemática, con el fin de documentar sus efectos en los resultados que se obtienen. Los experimentos formales son usados para investigar diferentes aspectos, confirmar teorías, validar medidas, comparar métodos o herramientas entre otros. A diferencia de las encuestas y los casos de estudio, los experimentos formales son puramente cuantitativos, ya que se focalizan en realizar medidas sobre los factores, aplicarles cambios y volver a medir nuevamente. Durante este

proceso se recopila información cuantitativa para luego aplicar métodos estadísticos.

Una de las principales diferencias entre los experimentos formales, los casos de estudio y las encuestas es el control sobre la ejecución. Los experimentos son el único método de investigación donde el investigador tiene el control sobre la ejecución. Por ejemplo, en un caso de estudio si el gerente decide por alguna razón suspender o detener el proyecto sobre el que se está trabajando, el investigador no podrá continuar obteniendo datos. De forma similar, si en las encuestas las personas se niegan a proporcionar los datos que el investigador necesita, éste no podrá continuar su investigación.

Otra diferencia que se puede observar es la facilidad para poder replicar una investigación. En este caso tanto las encuestas como los experimentos formales son altamente replicables, ya que en un experimento se pueden volver a reproducir las mismas condiciones del experimento original, cambiando solamente la población a estudiar. De forma similar se replica una encuesta. En un caso de estudio la replicación es muy poco probable ya que no se tiene control sobre las variables, por lo que no se pueden reproducir las condiciones originales.

Al realizar un experimento formal puede usarse como guía un *proceso experimental* para la correcta implementación del experimento. Un proceso experimental consta de los conceptos básicos que se definen a continuación.

Unidad Experimental: es un objeto al que se le aplica un *tratamiento*. Ejemplos de unidades experimentales son programas, procesos y productos. En Ingeniería de Software, si se considera un experimento en el cual se comparen técnicas de verificación, la unidad experimental puede ser la pieza de código (software) sobre la que se aplican las técnicas.

Sujeto: persona que aplica el o los tratamientos. Por ejemplo, en un experimento de Ingeniería de Software los sujetos suelen ser estudiantes o profesionales en el área bajo estudio. Particularmente en el área de la verificación, los sujetos pueden ser verificadores (profesionales) o estudiantes.

Variable Independiente o Factor: son variables que caracterizan un proyecto e influyen en la evaluación de los resultados. Interesa estudiar el efecto que provocan en la salida al cambiar de valor. Al considerar un experimento donde se compara el desempeño de tres técnicas de verificación, la técnica de verificación es un factor o variable independiente.

Tratamiento o Alternativa: es un valor particular de un *factor* en un *experimento unitario*. Considerando el ejemplo anterior, las alternativas pueden ser Inspecciones, Tablas de Decisión y Trayectorias Independientes.

Variable Dependiente: son variables cuyo valor es afectado por los cambios en una o más *variables independientes*. Son la salida del experimento. Estas variables son medidas para verificar los efectos de las *variables independientes*. Si comparamos tres técnicas de verificación, aplicándolas sobre un mismo software, la cantidad de defectos que encuentra una técnica de verificación es un ejemplo de variable dependiente.

Parámetro: son aquellas características (cuantitativas o cualitativas) que son invariantes a lo largo del experimento. Son características que no influyen, o que no se quiere que influyan, en los resultados del experimento. Los parámetros no afectan las *variables dependientes*. En un experimento llevado a cabo con estudiantes como verificadores (sujetos), la experiencia de los mismos puede ser un parámetro.

Experimento Unitario o Elemental: es una ejecución de una combinación

de *alternativas* de *factores*, por un *sujeto* sobre una *unidad experimental*. Si se considera un experimento donde 3 técnicas son ejecutadas por 4 sujetos sobre 2 unidades experimentales, se pueden considerar hasta 24 experimentos unitarios. La cantidad de experimentos unitarios que se realicen depende del diseño del experimento, no necesariamente tienen que ejecutarse los 24.

Error Experimental: describe la diferencia entre los resultados obtenidos por dos *experimentos unitarios* que deben producir la misma salida. Es la variación producida por elementos distorsionales conocidos y no conocidos.

A continuación se presenta el proceso experimental descrito en *Experimentation in Software Engineering - An Introduction* [WRH+00]. Consta de 5 etapas principales: Definición, Planificación, Operación, Análisis e Interpretación, y Presentación y Empaquetado. El experimento realizado en el marco de este proyecto de grado, descrito en la Parte II, sigue este proceso experimental.

En la etapa de *Definición* se define el problema y los objetivos del experimento. Se deben establecer las hipótesis no formalmente, pero sí claramente. En *Planificación* se determina el contexto, las variables dependientes e independientes, se diseña el experimento, se considera la instrumentación y se evalúan las amenazas. En la etapa de *Operación* se recolectan los datos de las medidas. En la etapa de *Análisis e Interpretación* se analizan y evalúan los datos obtenidos. Finalmente, en *Presentación y Empaquetado*, los resultados son presentados y empaquetados. En las secciones que siguen se detallan cada una de estas etapas.

2.1. Definición

En esta fase se definen las bases que determinan al experimento. Para ello se debe definir el problema que se quiere resolver y la intención del experimento, así como también los objetivos.

Para el planteo del objetivo del experimento se debe definir el *objeto de estudio*, que es la entidad que va a ser estudiada en el experimento. Puede ser un producto, proceso, recurso u otro. También se debe establecer el *propósito*; la intención del experimento. Por ejemplo, evaluar diferentes técnicas de verificación. Se debe definir además el *foco de calidad*, que es el efecto primario que está bajo estudio, ejemplos son la efectividad y el costo de las técnicas de verificación. El *propósito* y el *foco de calidad* son las bases para las hipótesis del experimento.

Otro aspecto que se debe tener presente es la *perspectiva*; el punto de vista con que los resultados obtenidos son interpretados. Por ejemplo, los resultados de la comparación de técnicas de verificación pueden verse desde la perspectiva de un experimentador, de un investigador o de un profesional. Un experimentador verá el estudio como una demostración de cómo una técnica de verificación puede ser evaluada, o sea, examinará el trabajo como un ejemplo de uso de una metodología de experimentación particular que puede ser reutilizada en futuros estudios. Un investigador puede ver el estudio como una base empírica para refinar teorías sobre la verificación de software, enfocándose en los datos que apoyan o refutan estas teorías. Un profesional puede ver el estudio como una fuente de información sobre qué técnicas de verificación deberían aplicarse en la práctica.

Junto con los aspectos mencionados se debe definir el *contexto*, que es el ambiente en el que corre el experimento. En este punto se deben definir los sujetos que van a llevar a cabo el experimento y los artefactos que son utilizados

en la ejecución del mismo. Se puede caracterizar el contexto de un experimento en 4 tipos, según el número de sujetos y objetos definidos en él: un solo objeto y un solo sujeto, un solo sujeto a través de muchos objetos, un solo objeto a través de un conjunto de sujetos, o un conjunto de sujetos y un conjunto de objetos.

2.2. Planificación

La planificación es la etapa en la que se define cómo se va a llevar a cabo el experimento.

Esta etapa consta de las fases: selección del contexto, formulación de las hipótesis, elección de las variables, selección de los sujetos, diseño del experimento, instrumentación y evaluación de la validez.

La fase de *selección del contexto* es la fase inicial de la planificación, es decir, es lo primero que se debe hacer al comenzar a planificar un experimento. En esta fase se amplía el contexto definido en la etapa de *Definición*, especificando claramente las características del ambiente donde corre el experimento. Se define si el experimento se va a realizar en un proyecto real (en línea, *on-line*) o en un laboratorio (fuera de línea, *off-line*). Si los sujetos que realizan el experimento son estudiantes o profesionales. Si el problema es real, es decir, un problema existente en la industria o es un problema "de juguete". Y si el experimento es válido para un contexto específico o para un dominio general de la Ingeniería de Software.

Una vez que los objetivos están claramente definidos se pueden transformar en una hipótesis formal. La *formulación de la hipótesis* es una fase muy importante dentro de la etapa de *Planificación*, ya que la verificación de la misma es la base para el análisis estadístico. En esta fase se formaliza la definición del experimento en la hipótesis. Por ejemplo, si se tiene un experimento cuyo objetivo es descubrir cuál de tres técnicas de verificación es la mejor para encontrar defectos de un tipo particular, el investigador puede formular la siguiente hipótesis "la técnica B encuentra mas defectos del tipo 1 que las técnicas C y A". Esta hipótesis es cuantificable, se puede medir.

Usualmente se definen dos hipótesis, la hipótesis nula y la/s hipótesis alternativa/s. La hipótesis nula, denotada H_0, asume que no hay una diferencia significativa entre los tratamientos, con respecto a las variables dependientes que se están midiendo. Establece que si hay diferencias entre las observaciones realizadas, éstas son por casualidad, no producto del tratamiento aplicado. Esta hipótesis se asume verdadera hasta que los datos demuestren lo contrario, por lo que el foco del experimento está puesto en rechazarla. Un ejemplo de hipótesis nula es: "No hay diferencia en la cantidad de defectos encontrados por las técnicas de verificación".

En cambio la hipótesis alternativa, denotada H_1, afirma que existe una diferencia significativa entre los tratamientos con respecto a las variables dependientes. Establece que las diferencias encontradas son producto de la aplicación de los tratamientos. Ésta es la hipótesis a probar, para esto se debe determinar que los datos obtenidos son lo suficientemente convincentes para desechar la hipótesis nula y aceptar la hipótesis alternativa. Un ejemplo de hipótesis alternativa es, si se están comparando dos técnicas de verificación, decir que una encuentra más defectos que la otra. En caso de haber más de una hipótesis alternativa se denotan secuencialmente: H_1, H_2, H_3, ..., H_n.

Una vez definidas las hipótesis, se deben identificar cuales variables afectan el tratamiento. Luego de identificadas las variables se debe decidir el control que se ejercerá sobre las mismas. La *selección de las variables* dependientes como la de las independientes está relacionada, por lo que en muchos casos se realizan en simultáneo. Seleccionar estas variables es una tarea muy compleja, que en ocasiones implica conocer muy bien el dominio. Es importante definir las variables independientes y analizar sus características, para así investigar y controlar los efectos que ejercen sobre las variables dependientes. Se deben identificar las variables independientes que se pueden controlar y las que no.

También se deben identificar las variables dependientes. Con estas variables se mide el efecto de los tratamientos. Generalmente hay sólo una variable dependiente y se deriva de la hipótesis.

Otro aspecto importante al llevar a cabo un experimento es la *selección de los sujetos*. Esta selección influye principalmente en la generalización de los resultados del experimento. Para poder generalizar los resultados al resto de la población, la selección debe ser una muestra representativa de la misma. Cuanto mas grande es la muestra, menor es el error al generalizar los datos. Existen dos tipos de muestras que se pueden seleccionar: la probabilística, donde se conoce la probabilidad de seleccionar cada sujeto; y la no-probabilística, donde esta probabilidad es desconocida.

Luego de definir el contexto, formalizar las hipótesis, y seleccionar las variables y los sujetos, se debe *diseñar el experimento*. Es muy importante planear y diseñar cuidadosamente el experimento, para obtener datos que luego puedan ser interpretados mediante la aplicación de métodos de análisis estadísticos.

Para comenzar a diseñar un experimento se debe elegir el diseño adecuado. Se debe planificar y diseñar el conjunto de las combinaciones de tratamientos, sujetos y objetos, o sea, de los experimentos unitarios, con el fin de obtener los mejores resultados. Se describe cómo estos experimentos unitarios deben ser organizados y ejecutados.

El diseño del experimento debe ser consistente permitiendo la prueba de la hipótesis y debe ser correcto considerando las circunstancias que rodean al experimento.

La elección del diseño del experimento afecta el análisis estadístico y viceversa, por lo que al elegir el diseño del experimento se debe tener en cuenta cuál análisis estadístico es el mejor para rechazar la hipótesis nula y aceptar la alternativa.

Existen cuatro principios generales que se deben considerar en el diseño de un experimento, estos son replicación, ordenación aleatoria (*randomization*), bloqueo y balance.

La *replicación* es la repetición del experimento básico, esto significa repetir el experimento en idénticas condiciones. La replicación provee una estimación del error experimental que ayuda a conocer cuan confiables son los resultados del experimento. También permite estimar los efectos de algún factor que se desee. Es importante asegurar que la replicación no introduzca confusión. Cuando se introduce confusión es imposible separar los efectos de dos o mas variables, o sea, cuando se da un efecto, no se puede distinguir que variable lo causó. Por esta razón, el diseño del experimento debe identificar las condiciones sobre las cuales las replicaciones pueden realizarse, así como también al número y el tipo de las replicaciones, y las medidas que se realizan sobre ellas.

La *ordenación aleatoria* es uno de los principios de diseño mas importante.

Todos los métodos estadísticos que se utilizan para el análisis de los datos requieren que las observaciones provengan de variables independientes aleatorias. La ordenación aleatoria es usada en la selección de los sujetos, para que sea representativa; y para la asignación aleatoria de sujetos a tratamientos. Puede aplicarse también a objetos y al orden de ejecución de los experimentos unitarios. El uso principal de la ordenación aleatoria es para promediar el efecto de un factor que de otra manera estaría presente. Si se considera el diseño de un experimento llevado a cabo con el personal de una compañía (sujetos), en el cual se comparan una técnica de verificación estática y otra dinámica, se cumplirá el principio de ordenación aleatoria si:

- Se realiza una selección aleatoria de los verificadores disponibles en la compañía.
- La asignación a cada tratamiento (técnica estática o dinámica) se realiza aleatoriamente.

El *bloqueo* es utilizado para aumentar la precisión del experimento ya que permite eliminar efectos indeseados, producidos por factores, en la comparación entre tratamientos. Esta técnica se puede usar si el efecto producido por lo factores es conocido y controlable. Supongamos que en el experimento anterior los verificadores (sujetos) poseen diferentes niveles de experiencia. Algunos han utilizado las técnicas de verificación bajo estudio y otros no. Para minimizar el efecto de la experiencia, las personas se agrupan en dos grupos (bloques), uno con experiencia en las técnicas y otro sin experiencia. Dentro de cada bloque, el efecto indeseado es el mismo y se puede estudiar el efecto del tratamiento en ese bloque. Si se desea bloquear el efecto de la experiencia sin distinguir entre los niveles de experiencia de los sujetos (ya sea porque no es posible o porque no es de interés) se puede realizar el bloqueo de este efecto asignando cada tratamiento (técnica estática y dinámica) a todos los sujetos.

El *balance* es deseable porque simplifica el análisis estadístico de los datos, pero no es necesario. Para obtener un diseño balanceado se debe asignar el mismo número de sujetos a cada tratamiento. En el ejemplo anterior, el diseño es balanceado si cada grupo (bloque) está conformado por la misma cantidad de personas, y si dentro de los grupos cada técnica tiene asignado el mismo número de sujetos.

Es útil conocer y comprender los distintos tipos de diseño para poder elegir el más adecuado para un experimento. Existen muchos diseños de experimento, algunos más simples otros más complejos dependiendo de características como los factores o las alternativas. En nuestro estudio utilizamos un diseño llamado *diseño de un factor con k alternativas*. Este diseño es utilizado para comparar k tratamientos entre sí.

Existen dos variaciones de este diseño, el *diseño completamente aleatorio* y el *diseño completamente aleatorio bloqueado*. El *diseño completamente aleatorio* requiere que el experimento sea realizado en orden aleatorio, usa un sólo objeto para todos los tratamientos y los sujetos son asignados aleatoriamente. El *diseño completamente aleatorio bloqueado* es uno de los diseños más utilizado en el diseño de experimentos. En este diseño cada sujeto usa todos los tratamientos, bloqueando el efecto de los sujetos sobre el experimento. Se usa para minimizar el efecto de las diferencias entre los sujetos cuando estas variaciones son significativas. En este diseño se utiliza un solo objeto para todos los tratamientos y

el orden de ejecución de los tratamientos asignado a cada sujeto es aleatorio.

En un *diseño de un factor con k alternativas*, en general, la hipótesis nula es que la variable dependiente es igual para todos los tratamientos, y la hipótesis alternativa que la variable dependiente difiere para algún par de tratamientos. La formulación formal sería:

- $H_0 : \mu_1 = \mu_2 = \ldots = \mu_n$
- $H_1 : \mu_i \; ! = \mu_j$ para algún par de tratamientos (i, j)

donde μ_i es el significado de la variable dependiente para el tratamiento i, y n la cantidad de tratamientos.

El diseño utilizado en nuestro experimento es una adaptación de este último, donde la modificación es básicamente la utilización de más de un objeto para cada tratamiento.

Luego de diseñar el experimento y antes de la ejecución es necesario poner a punto todo lo necesario para la correcta ejecución del mismo. La *instrumentación* involucra, de ser necesario, capacitar a los sujetos, preparar los artefactos, guías, descripciones de los procesos, planillas y herramientas. También implica configurar el hardware, mecanismos de consultas y experiencias piloto entre otros. La finalidad de esta fase es proveer todo lo necesario para la realización y monitorización del experimento.

Una pregunta fundamental antes de pasar a ejecutar el experimento es cuán válidos son los resultados. Existen cuatro categorías de amenazas a la validez: validez de la conclusión, validez interna, validez del constructo y validez externa.

Las amenazas que afectan la **validez de la conclusión** refieren a la validez de las conclusiones estadísticas. Amenazas que afecten la capacidad de determinar si existe una relación entre el tratamiento y la salida, y si las conclusiones obtenidas al respecto son validas. Ejemplos de estas son la elección de los métodos estadísticos, y la elección del tamaño de la muestra, entre otros.

Las amenazas que influyen en la **validez interna** son aquellas referidas a observar relaciones entre el tratamiento y la salida que son producto de la casualidad y no el resultado de la aplicación de un factor. Esta "casualidad" es provocada por elementos desconocidos que influyen sobre los resultados sin el conocimiento de los investigadores. Es decir, la validez interna se basa en asegurar que el tratamiento en cuestión produce la salida observada.

Un ejemplo de **validez del constructo** se puede observar al momento de seleccionar los sujetos en un experimento. Si se utiliza como medida de la experiencia del sujeto el número de cursos que tiene hechos en la universidad, no se está utilizando una buena medida de la experiencia. En cambio, una buena medida puede ser utilizar el número de años de experiencia en la industria.

La **validez externa** está relacionada con la habilidad para generalizar los resultados. Se ve afectada por el diseño del experimento. Los tres riesgos principales que tiene la validez externa son: tener los participantes equivocados como sujetos, ejecutar el experimento en un ambiente erróneo y realizar el experimento en un momento que afecta los resultados.

2.3. Operación

Luego de diseñar y planificar el experimento, éste debe ser ejecutado para recolectar los datos que se quieren analizar. La operación del experimento consiste

en tres pasos: *preparación*, en el cual se seleccionan los sujetos y se preparan los artefactos; *ejecución*, donde los sujetos ejecutan el experimento y recolectan los datos; y la *validación de los datos* donde se validan los datos obtenidos. Antes de ejecutar el experimento se debe preparar todo lo necesario para hacer más fácil la ejecución.

Con respecto a la selección de los sujetos es muy importante que estos estén motivados y dispuestos a participar en el experimento. Los resultados obtenidos pueden volverse inválidos si los sujetos al momento que deciden participar no saben lo que tienen que hacer o tienen un concepto erróneo. Otro aspecto a considerar es la sensibilidad de los resultados que se obtienen de los sujetos, por ejemplo, es importante asegurar a los participantes que los resultados obtenidos sobre su rendimiento se mantienen en secreto y no se usarán para perjudicarlos en ningún sentido. Se debe tener en cuenta también los incentivos, ya que ayuda a motivar a los sujetos, pero se corre el riesgo de que participen sólo por el incentivo, lo que puede ser perjudicial para el experimento. Otro aspecto importante es el engaño, en caso de no tener otra alternativa que no sea engañar a los sujetos se debe procurar explicar y revelar el engaño lo más temprano posible. Si se aplica el engaño se debe tener especial cuidado en no comprometer información confidencial o sensible sobre los sujetos.

Como se vio en la *instrumentación*, para que los sujetos comiencen la ejecución es necesario tener prontos todos los instrumentos, formularios, herramientas, guías y otros. Uno de los aspectos más importantes para determinar los instrumentos necesarios es el diseño, así también como el o los métodos que se usan para recolectar los datos. Muchas veces se debe preparar un conjunto de instrumentos especial para cada sujeto y otras se utiliza el mismo conjunto de artefactos para todos los sujetos.

Existen muchas formas distintas de ejecutar los experimentos, pueden llevar días, meses e incluso años. Los datos pueden ser recolectados manualmente mediante el llenado de formularios por parte de los sujetos, manualmente soportado por herramientas, en entrevistas, o automáticamente por herramientas. La primera es la más común y no requiere mucho esfuerzo por parte del experimentador. Tanto en los formularios como en los métodos soportados por herramientas no es posible identificar inconsistencias o defectos hasta que no se recolecte la información, o hasta que los sujetos los descubran. En las entrevistas, el contacto con los sujetos es mucho mayor permitiendo una mejor comunicación con ellos durante la recolección de datos. Éste método es el que requiere mas esfuerzo por parte del investigador.

Un aspecto muy importante a la hora de ejecutar los experimentos es el ambiente de ejecución. Ya sea que el experimento se realiza dentro de un proyecto de desarrollo común o se crea un ambiente ficticio para su ejecución. En el primer caso el experimento no debería afectar el proyecto más de lo necesario, ya que la razón de realizar el experimento dentro de un proyecto es ver los efectos de los tratamientos en el ambiente del proyecto. Si el experimento cambia demasiado el ambiente del proyecto estos efectos se perderían.

Cuando se obtienen los datos, se debe chequear que fueron recolectados correctamente y que son razonables. Algunas fuentes de error son que los sujetos llenen mal sus planillas, o no recolecten los datos seriamente, lo que hace que se descarten datos. Es importante revisar que los sujetos hagan un trabajo serio y responsable y que apliquen los tratamientos en el orden correcto, o sea, que el experimento sea ejecutado en la forma en que fue planificado. De lo contrario

los resultados podrían ser inválidos.

2.4. Análisis e Interpretación

La escala de medida utilizada para recolectar los datos restringe el tipo de cálculos estadísticos que se pueden realizar. Una medida es un mapeo de un atributo de una entidad a un valor de medida, por lo general un valor numérico. Las entidades son objetos que se observan en la realidad, por ejemplo, una técnica de verificación. El propósito de mapear los atributos en un valor de medida es caracterizar y manipular los atributos formalmente. La medida seleccionada debe ser *válida*. Para ello, la medida no debe violar ninguna propiedad necesaria del atributo que mide, y debe ser una caracterización matemática adecuada del atributo.

El mapeo de un atributo a un valor de medida puede realizarse de varias formas. Cada mapeo posible de un atributo se conoce como *escala*. Si el atributo es el largo de un objeto, se puede medir en metros, centímetros o pulgadas, cada una de las cuales es una escala distinta de la medida de la longitud. Los tipos más comunes de escala son:

- Escala Nominal.- Es la menos poderosa de las escalas. Solo mapea el atributo de la entidad en un nombre o símbolo. El mapeo puede verse como una clasificación de las entidades acorde al atributo. Ejemplos de escala nominal son clasificaciones, etiquetados, etc.

- Escala Ordinal.- La escala ordinal categoriza las entidades según un criterio de ordenación. Es más poderosa que la escala nominal. Ejemplos de criterios de ordenación son “mayor que”, “mejor que” y “más complejo”. Ejemplos de escala nominal son grados, complejidad del software, etc.

- Escala de intervalo.- La escala de intervalo se utiliza cuando la diferencia entre dos medidas es significativa, pero no el valor en si mismo. Este tipo de escala ordena los valores de la misma forma que la escala ordinal, pero existe la noción de “distancia relativa” entre dos entidades. Esta escala es más poderosa que la ordinal. Ejemplos de escala de intervalo son la temperatura medida en Celsius o Fahrenheit.

- Escala ratio (cociente de dos números).- Si existe un valor cero significativo y la división entre dos medidas es significativa, se puede utilizar una escala ratio. Ejemplos de escala ratio son distancia, temperatura medida en Kelvin, etc.

Después de obtener los datos es necesario interpretarlos para llegar a conclusiones válidas. La interpretación se realiza en tres fases. En la primera se caracteriza el conjunto de datos usando *estadística descriptiva*, por ejemplo estudiando la tendencia central, dispersión, etc.; en la segunda se *reduce el conjunto de datos*, excluyendo los datos falsos o anormales; y en la tercera se analizan los datos en la *prueba de hipótesis*. Esta prueba consiste en evaluar estadísticamente las hipótesis, a un determinado nivel de significancia. El **nivel de significancia** es la probabilidad de rechazar la hipótesis nula cuando es verdadera.

La *estadística descriptiva* se utiliza antes de la prueba de hipótesis, para entender mejor la naturaleza de los datos y para identificar datos falsos o anormales. Los aspectos principales que se examinan son: la tendencia central, la dispersión y la dependencia.

A continuación se presentan las medidas más comunes de cada uno de estos aspectos. Para ello se asume que existen $x_1 \dots x_n$ muestras.

Las *medidas de tendencia central* indican "el medio" de un conjunto de datos. Entre las medidas más comunes se encuentran: la media aritmética, la mediana y la moda.

La **media aritmética** se conoce como el promedio, y se calcula sumando todas las muestras y dividiendo el total por el número de muestras:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.1}$$

La media, denotada $\bar{x}$, resume en un valor las características de una variable teniendo en cuenta a todos los casos. Es significativa para las escalas de intervalo y ratio.

La **mediana**, denotada $\tilde{x}$, representa el valor medio de un conjunto de datos, tal que el número de muestras que son mayores que la mediana es el mismo que el número de muestras que son menores que la mediana. Se calcula ordenando las muestras en orden ascendente o descendente, y seleccionando la observación del medio. Este cálculo está bien definido si n es impar. Si n es par, la mediana se define como la media aritmética de los dos valores medios. Esta medida es significativa para las escalas ordinal, de intervalo y ratio.

La **moda** representa la muestra más común. Se calcula contando el número de muestras para cada valor único y seleccionando el valor con más cantidad. La moda está bien definida si hay solo un valor más común que los otros. Si este no es el caso, se calcula como la mediana de las muestras más comunes. La moda es significativa para las escalas nominal, ordinal, de intervalo y ratio.

La media aritmética y la mediana son iguales si la distribución de las muestras es simétrica. Si la distribución es simétrica y tiene un único valor máximo, las tres medidas son iguales.

Las medidas de tendencia central no proveen información sobre la dispersión del conjunto de datos. Cuanto mayor es la dispersión, más variables son las muestras, cuanto menor es la dispersión, más homogéneas a la media son las muestras.

Las *medidas de dispersión* miden el nivel de desviación de la tendencia central, o sea, que tan diseminados o concentrados están los datos respecto al valor central. Entre las principales medidas de dispersión están: la varianza, la desviación estándar, el rango y el coeficiente de variación.

La **varianza** (s^2) que presenta una distribución respecto de su media se calcula como la media de las desviaciones de las muestras respecto a la media aritmética. Dado que la suma de las desviaciones es siempre cero, se toman las desviaciones al cuadrado:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{2.2}$$

Se divide por $n-1$ y no por n, porque dividir por $n-1$ provee a la varianza

Valor	Frecuencia	Frecuencia relativa
1	3	23 %
2	2	15 %
3	1	8 %
4	3	23 %
5	1	8 %
6	2	15 %
7	1	8 %

Cuadro 2.1: Ejemplo de una tabla de frecuencia

de propiedades convenientes. La varianza es significativa para las escalas de intervalo y ratio.

La **desviación estándar**, denotada s, se define como la raíz cuadrada de la varianza:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.3}$$

A menudo esta medida se prefiere sobre la varianza porque tiene las mismas dimensiones (unidad de medida) que los valores de las muestras. En cambio, la varianza se mide en unidades cuadráticas. La desviación estándar es significativa para las escalas de intervalo y ratio.

La dispersión también se puede expresar como un porcentaje de la media. Este valor se llama **coeficiente de variación**, y se calcula como:

$$100 \cdot \frac{s}{\bar{x}} \tag{2.4}$$

Esta medida no tiene dimensión y es significativa para la escala ratio. Permite comparar la dispersión o variabilidad de dos o más grupos.

El **rango** de un conjunto de datos es la distancia entre el valor máximo y el mínimo:

$$range = x_{max} - x_{min} \tag{2.5}$$

Es una medida significativa para las escalas de intervalo y ratio.

La **frecuencia** da a una visión general de dispersión de cada valor del conjunto de datos. Se construye una *tabla de frecuencia* tabulando cada valor único y contando las veces que sucede. La **frecuencia relativa** se calcula dividiendo la frecuencia por el número total de muestras. Para el conjunto de datos (1, 1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 7) con 13 muestras, se puede construir el cuadro de frecuencia 2.4. La frecuencia es significativa para todas las escalas.

Cuando el conjunto de datos consiste en muestras relacionadas de a pares (x_i, y_i) de dos variables, X e Y, puede ser interesante examinar la *dependencia* entre estas variables. Las principales medidas de dependencia son: regresión lineal, covarianza y el coeficiente de correlación lineal. No entraremos en detalle en estas medidas porque no se utilizan en nuestro experimento.

Para la prueba de hipótesis se utilizan métodos estadísticos. El resultado de aplicar estos métodos depende de la calidad de los datos. Si los datos no

representan lo que se cree, las conclusiones que se derivan de los resultados de los métodos son incorrectas. Errores en el conjunto de datos pueden ocurrir por un error sistemático, o por lo que se conoce en estadística con el nombre de *outlier*. Un *outlier* es un dato mucho más grande o mucho más chico de lo que se puede esperar observando el resto de los datos.

La *reducción de datos* a realizar en esta etapa trata de identificar *outliers*, no solo basándose en la ejecución del experimento, el cual sería el caso de determinar si los sujetos han participado seriamente, sino también observando los resultados de la ejecución aplicando estadística descriptiva.

Una vez identificado un *outlier* se debe identificar su origen para decidir que hacer con él. Si se debe a un evento raro o extraño que no volverá a ocurrir, el punto puede ser excluido. Si se debe a un evento extraño que puede volver a ocurrir, no es aconsejable excluir el valor del análisis, pues tiene información relevante. Si se debe a una variable que no fue considerada, debería ser considerado para basar los cálculos y modelos también en esta variable. En este caso es posible derivar dos modelos separados. Por ejemplo, si la variable no considerada es la experiencia de los sujetos, un modelo puede ser basado en sujetos "normales" (sin el *outlier*) y el otro basado en los sujetos inexperientes (o experientes).

El objetivo de la *prueba de hipótesis* es ver si es posible rechazar cierta hipótesis nula H_0. Si la hipótesis nula no es rechazada, no se puede decir nada sobre los resultados. En cambio, si es rechazada, se puede declarar que la hipótesis es falsa con una significancia dada (α). Este nivel de significancia también es denominado nivel de riesgo, ya que se corre el riesgo de rechazar la hipótesis nula, cuando en realidad es verdadera. Este nivel está bajo el control del experimentador.

Para probar H_0 se define una unidad de prueba t y un área crítica C, la cual es parte del área sobre la que varía t. A partir de estas definiciones se formula la prueba de significancia de la siguiente forma:

- Si $t \in C$, rechazar H_0
- Si $t \notin C$, no rechazar H_0

Por ejemplo, un experimentador observa la cantidad de defectos detectados por LOC de una técnica de verificación desconocida bajo determinadas condiciones, y quiere probar que no es la técnica B, de la cual sabe que en las mismas condiciones (programa, verificador, etc.) detecta 1 defecto cada 20 LOC. El experimentador sabe que también pueden haber otras técnicas que detecten 1 defecto cada 20 LOC. A partir de esto se define la hipótesis nula: "H_0: La técnica observada es la B". En este ejemplo, la unidad de prueba t es cada cuantos LOC se detecta un defecto y el área crítica es $C = \{1, 2, 3, \ldots, 19, 21, 22, \ldots\}$. La prueba de significancia es: si $t \leq 19$ o $t \geq 21$, rechazar H_0, de lo contrario no rechazar H_0.

Si se observa que $t = 20$, la hipótesis no puede ser rechazada ni se pueden derivar conclusiones, pues pueden haber otras técnicas que detecten un defecto cada 20 LOC.

El área crítica, C, puede tener distintas formas, lo más común es que tenga forma de intervalo, por ejemplo: $t \leq a$ o $t \geq b$. Si C consiste en uno de estos intervalos es unilateral. Si consiste de dos intervalos ($t \leq a$, $t \geq b$, donde $a < b$), es bilateral.

Hay varios métodos estadísticos, de aquí en adelante denotados *tests*, que pueden utilizarse para evaluar los resultados de un experimento, más específicamente para determinar si se rechaza la hipótesis nula. Cuando se lleva a cabo un *test* es posible calcular el menor valor de significancia posible (denotado *p*-valor) con el cual es posible rechazar la hipótesis nula. Se rechaza la hipótesis nula si el p-valor asociado al resultado observado es menor o igual que el nivel de significancia establecido.

Las siguientes son tres probabilidades importantes para la prueba de hipótesis:

- $\alpha = P(\text{cometer el error tipo I}) = P(\text{rechazar } H_0 | H_0 \text{ es verdadera})$. Es la probabilidad de rechazar H_0 cuando es verdadera.
- $\beta = P(\text{cometer el error tipo II}) = P(\text{no rechazar } H_0 | H_0 \text{ es falsa})$. Es la probabilidad de no rechazar H_0 cuando es falsa.
- $\text{Poder} = 1 - \beta = P(\text{rechazar } H_0 | H_0 \text{ es falsa})$. El poder de prueba es la probabilidad de rechazar H_0 cuando es falsa.

El experimentador debería elegir un test con un poder de prueba tan alto como sea posible. Hay varios factores que afectan el poder de un test. Primero, el test en si mismo puede ser más o menos efectivo. Segundo, la cantidad de muestras: mayor cantidad de muestras equivale a un poder de prueba más alto. Otro aspecto es la selección de una hipótesis alternativa unilateral o bilateral. Una hipótesis unilateral da un poder mayor que una bilateral.

La probabilidad de cometer un error tipo I se puede controlar y reducir. Si la probabilidad es muy pequeña, sólo se rechazará la hipótesis nula si se obtiene evidencia muy contundente en contra de esta hipótesis. La probabilidad máxima de cometer un error tipo I se conoce como la significancia de la prueba (α).

Los valores de uso más común para la significancia de una prueba son 0.01, 0.05 y 0.10. La significancia es en ocasiones presentada como un porcentaje, tal como 1 %, 5 % o 10 %. Esto quiere decir que el experimentador está dispuesto a permitir una probabilidad de 0.01, 0.05, o 0.10 de rechazar la hipótesis nula cuando es cierta, o sea, de cometer un error tipo I.

El valor de la significancia es seleccionado antes de comenzar a hacer el experimento en una de varias formas. El valor de α puede estar establecido en el área de investigación, por ejemplo, se puede obtener de artículos que se publican en revistas científicas. Otra forma de seleccionarlo es que sencillamente sea impuesto por la persona o compañía para la cual se trabaja. Finalmente, puede ser seleccionado tomando en cuenta el costo de cometer un error tipo I. Mientras más alto el costo, más pequeña debe ser la probabilidad α de cometer un error tipo I. El valor usual de α en las ciencias naturales y sociales es de 0.05.

La probabilidad de cometer un error tipo I no puede ser igual a cero ya que si se desea $\alpha = 0$, nunca se podría tomar la decisión de rechazar la hipótesis nula. Siempre que se tome la decisión de rechazar la hipótesis nula existe una probabilidad positiva de cometer un error tipo I, ya que la decisión se basa en una muestra y no en la población.

Los tests pueden ser paramétricos o no paramétricos. Los tests paramétricos están basados en un modelo que involucra una distribución específica. En la mayoría de los casos, se asume que algunos de los parámetros involucrados en un test paramétrico están normalmente distribuidos. Los tests paramétricos

también requieren que los parámetros puedan ser medidos al menos en una *escala de intervalo*. Si los parámetros no pueden medirse en al menos una escala de intervalo, generalmente no se puede utilizar un test paramétrico. En este caso hay un amplio rango de tests no paramétricos disponible.

Los tests no paramétricos no asumen lo mismo respecto a la distribución de los parámetros, son más generales que los paramétricos. Un test no paramétrico se puede utilizar en vez de un test paramétrico, pero el caso inverso no siempre puede darse.

En la elección entre un test paramétrico y un test no paramétrico hay dos aspectos a considerar:

- Aplicabilidad.- Es importante que las suposiciones en cuanto a las distribuciones de parámetros y las que conciernen a las escalas sean realistas.
- Poder.- El poder de los tests paramétricos es generalmente mayor que el de los tests no paramétricos. Por lo tanto, los test paramétricos requieren menos datos (experimentos más pequeños), que los tests no paramétricos, siempre que sean aplicables.

Aunque es un riesgo utilizar tests paramétricos cuando no se cuenta con las condiciones requeridas, en algunos casos vale la pena tomar el riesgo. Algunas simulaciones han mostrado que los tests paramétricos son bastante robustos a las desviaciones de las precondiciones (escala de intervalo), mientras las desviaciones no sean demasiado grandes.

ANOVA (*ANalysis Of VAriance*) es una familia de tests paramétricos que puede utilizarse para diseños de un factor con k alternativas. Sus hipótesis son:

- $H_0 : \mu_1 = \mu_2 = \ldots = \mu_k$, donde μ_a es la *media* de la muestra a.

  (las medias de todas las muestras son iguales, o sea, las muestras provienen de la misma población)

- $H_1 : \mu_i \neq \mu_j$ para al menos un par de muestras i, j, donde μ_a es la *media* de la muestra a.

  (existe una diferencia en la medias entre al menos dos de las k muestras, o sea, al menos dos muestras provienen de poblaciones distintas)

Los supuestos de ANOVA son los siguientes:

1. Distribución normal de los datos.- Se supone que los datos son de una o más poblaciones normalmente distribuidas.
2. Homogeneidad de varianzas.- Esta suposición significa que cada una de las muestras provienen de poblaciones con la misma varianza (la varianza no varía en los diferentes niveles del factor). Esta propiedad se conoce como *Homocedasticidad*. Cuando no se cumple, se dice que los datos son *heterocedásticos*.
3. Escala de intervalo.- Los datos deben medirse al menos a nivel de intervalo.
4. Independencia.- Los datos de los diferentes participantes deben ser independientes, lo que significa que el comportamiento de uno de los participantes no influye en el comportamiento de otro.

En términos de violaciones de la asunción de homogeneidad de varianza, ANOVA es bastante robusto cuando los tamaños de las muestras son iguales. Sin embargo, cuando las muestras son de tamaños desiguales, ANOVA puede producir un resultado no significativo, aunque exista una diferencia genuina entre las muestras.

Kruskal-Wallis es un test no paramétrico alternativo a ANOVA en el caso de un factor con k alternativas. Las hipótesis del test de Kruskal-Wallis son:

- $H_0 : \theta_1 = \theta_2 = \ldots = \theta_k$, donde θ_a es la *mediana* de la muestra a.

  (las medianas de todas las muestras son iguales, es decir, las muestras provienen de la misma población)

- $H_1 : \theta_i \neq \theta_j$ para al menos un par de muestras i, j, donde θ_a es la *mediana* de la muestra a.

  (existe una diferencia en la mediana entre al menos dos de las k muestras, es decir, al menos dos muestras provienen de poblaciones distintas)

Este test suele utilizarse cuando los datos violan alguna de las suposiciones necesarias para aplicar ANOVA. No tiene suposiciones sobre los datos, pero puede dar un resultado no significativo, aunque haya diferencia real entre las muestras, si los datos tienen un alto nivel de heterocedasticidad.

Para probar la homocedasticidad de los datos se suele utilizar el test de Levene. Las hipótesis del test de Levene son:

- $H_0 : \sigma_1 = \sigma_2 = \ldots = \sigma_k$, donde σ_a es la *varianza* de la muestra a.
- $H_1 : \sigma_i \neq \sigma_j$ para al menos un par de muestras i, j, donde σ_a es la *varianza* de la muestra a.

Para poder aplicar ANOVA, y en algunos casos Kruskal-Wallis, es necesario que el test de Levene no sea significativo (**no** se rechaza H_0), o sea, que las varianzas de las muestras sean similares o iguales. Esto prueba la homocedasticidad de los datos.

Una vez que se prueba que al menos dos de las k muestras provienen de poblaciones distintas se puede aplicar, entre otros, el test de Mann-Whitney para comparar las muestras dos a dos.

Las hipótesis nula del test de Mann-Whitney es:

- $H_0 : \theta_1 = \theta_2$, donde θ_a es la *mediana* de la muestra a.

Esto es, la mediana de la muestra 1 representa la misma población que la mediana de la muestra 2. Cuando ambas muestras tienen el mismo tamaño, esto se traduce en que la suma de los rangos de la muestra 1 sea igual a la suma de los rangos de la muestra 2 ($\sum R_1 = \sum R_2$). Una manera más general de expresar esto, que comprende también diseños con tamaños de muestra desigual, es que el promedio de los rangos de las dos muestras son iguales ($\bar{R}_1 = \bar{R}_2$).

Los *rangos* de una muestra se determinan ordenando todos los datos de la muestra de menor a mayor, y asignando al menor un rango de 1, al segundo un 2, y así hasta el n-ésimo. Si existen datos que se repiten, se asigna el rango promedio a cada uno de ellos (si existen cuatro datos idénticos que ocupan los rangos 11, 12, 13 y 14, se les asigna un rango de 12,5 a los cuatro).

Existen tres posibles hipótesis alternativas a la hipótesis nula:

- $H_1 : \theta_1 \neq \theta_2$, donde θ_a es la *mediana* de la muestra a.

Esto es, la mediana de la muestra 1 no representa la misma población que la mediana de la muestra 2. Cuando ambas muestras tienen el mismo tamaño, esto se traduce en que la suma de los rangos de la muestra 1 no es igual a la suma de los rangos de la muestra 2 ($\sum R_1 \neq \sum R_2$). Una manera más general de expresar esto, que comprende también diseños con tamaños de muestra desigual, es que el promedio de los rangos de las dos muestras no son iguales ($\bar{R}_1 \neq \bar{R}_2$). Esta es una **hipótesis alternativa bilateral**.

- $H_1 : \theta_1 > \theta_2$, donde θ_a es la *mediana* de la muestra a.

Esto es, la mediana de la población que representa la muestra 1 es mayor que la mediana de la población que representa la muestra 2 (la población 1 es mayor que la población 2). Cuando ambas muestras tienen el mismo tamaño, esto se traduce en que la suma de los rangos de la muestra 1 es mayor que la suma de los rangos de la muestra 2 ($\sum R_1 > \sum R_2$). Una manera más general de expresar esto, que comprende también diseños con tamaños de muestra desigual, es que el promedio de los rangos de la muestra 1 es mayor que el promedio de los rangos de la muestra 2 ($\bar{R}_1 > \bar{R}_2$). Esta es una **hipótesis alternativa unilateral**.

- $H_1 : \theta_1 < \theta_2$, donde θ_a es la *mediana* de la muestra a.

Esto es, la mediana de la población que representa la muestra 1 es menor que la mediana de la población que representa la muestra 2 (la población 1 es menor que la población 2). Cuando ambas muestras tienen el mismo tamaño, esto se traduce en que la suma de los rangos de la muestra 1 es menor que la suma de los rangos de la muestra 2 ($\sum R_1 < \sum R_2$). Una manera más general de expresar esto, que comprende también diseños con tamaños de muestra desigual, es que el promedio de los rangos de la muestra 1 es menor que el promedio de los rangos de la muestra 2 ($\bar{R}_1 < \bar{R}_2$). Esta es una **hipótesis alternativa unilateral**.

Sólo una de las hipótesis alternativas mencionadas puede ser empleada. Si la hipótesis alternativa que selecciona el investigador es respaldada, se rechaza la hipótesis nula.

2.5. Presentación y Empaquetado

En la presentación y el empaquetado de un experimento es esencial no olvidar aspectos e información necesaria para que otros puedan replicar o tomar ventaja del experimento y del conocimiento ganado durante su ejecución.

El esquema de reporte de un experimento generalmente cuenta con los siguientes títulos: Introducción, Definición del Problema, Planificación del Experimento, Operación del Experimento, Análisis de Datos, Interpretación de Resultados, Discusión y Conclusiones, y Apéndice.

En la *Introducción* se realiza una introducción al área y los objetivos de la investigación. En la *Definición del Problema* se describe en mayor profundidad el trasfondo de la investigación, incluyendo las razones para realizarla. En *Planificación del Experimento* se detalla el contexto del experimento incluyendo las hipótesis, que se derivan de la definición del problema; las variables que se deben medir, tanto independientes como dependientes; la estrategia de medida y

análisis de datos; los sujetos que participaran de la investigación; y las amenazas a la validez.

En la *Operación del Experimento* se describe cómo preparar la ejecución del mismo, incluyendo aspectos que permitan facilitar la replicación y descripciones que indiquen cómo se llevaron a cabo las actividades. Debe incluirse la preparación de los sujetos, cómo se recolectaron los datos y cómo se realizo la ejecución. En el *Análisis de Datos* se describen los cálculos y los modelos de análisis específicos utilizados. Se debe incluir información, como por ejemplo, tamaño de la muestra, niveles de significancia y métodos estadísticos utilizados, para que el lector conozca los prerrequisitos para el análisis. En la *Interpretación de los Resultados* se rechaza la hipótesis nula o se concluye que no puede ser rechazada. Aquí se resume cómo utilizar los datos obtenidos en el experimento. La interpretación debe realizarse haciendo referencia a la validez. También se deben describir los factores que puedan tener un impacto sobre los resultados.

Finalmente, en *Discusión y Conclusiones* se presentan las conclusiones y los hallazgos como un resumen de todo el experimento, junto con los resultados, problemas, desviaciones respecto al plan, etc. También se incluyen ideas sobre trabajos a futuro. Los resultados deberían ser comparados con los obtenidos por trabajos anteriores, de manera de identificar similitudes y diferencias. La información que no es vital para la presentación se incluye en el *Apéndice*. Esto puede ser, por ejemplo, los datos recavados y más información sobre sujetos y objetos. Si la intención es generar un paquete de laboratorio, el material utilizado en el experimento puede ser proveído en el apéndice.

Capítulo 3

Técnicas de Verificación

Una vez codificados los programas, es el momento de probarlos. Las técnicas de verificación de software se utilizan para el análisis y comprobación de los sistemas de software. Las hay de dos grandes tipos: técnicas estáticas y técnicas dinámicas.

Las **técnicas estáticas** buscan defectos sin ejecutar el código, se basan en un examen manual o automatizado del código fuente o de la documentación. Se analiza el programa para deducir su correcta operación. Dentro de este tipo de técnicas de verificación se encuentran las que son de: *análisis de código*, donde se revisa el código fuente buscando defectos; *análisis automatizado de código fuente*, donde se utiliza una herramienta de software para recorrer el código fuente y detectar posibles anomalías y defectos; y *verificación formal*, donde se parte de una especificación formal y se busca probar que el programa cumple con la misma.

Las **técnicas dinámicas** implican ejecutar una implementación del software con datos de prueba. Se experimenta con el comportamiento de un programa para ver si éste actúa como es esperado. Dentro de las técnicas dinámicas existe una subclasificación: técnicas de *caja blanca* y técnicas de *caja negra*.

Las técnicas de *caja negra* se llaman así porque su comportamiento simula la entrada de los casos de prueba en una caja negra de la que no se conoce el contenido (el código fuente del programa), sólo la salida (el resultado de la ejecución de los casos de prueba). Para diseñar los casos de prueba no se necesita el código del programa, se utilizan los requerimientos y/o la especificación.

Por el contrario, las técnicas de *caja blanca* se basan en el código. A partir de él se identifican los casos de prueba. Este tipo de técnicas dinámicas tiene en cuenta el comportamiento interno y la estructura del programa.

A continuación se describen brevemente la técnica de análisis de código *Inspecciones*; las técnicas de caja negra *Partición de Equivalencia con Análisis de Valores Límite* y *Tablas de Decisión*; y las técnicas de caja blanca *Trayectorias Linealmente Independientes* y *Criterio de Cubrimiento de Condición Múltiple*. Estas fueron las técnicas utilizadas en el experimento realizado en el marco de este proyecto de grado.

Por mayor información sobre estas técnicas de verificación pueden consultarse los libros *The Art of Software Testing* [Mye04], *Software Testing: A Craftsman's Approach* [Jor08] y *Software Error Detection through Testing and Analysis* [Hua09].

3.1. Inspecciones de software

Las Inspecciones de Software son un proceso de verificación estático en el que un sistema de software se *revisa* para encontrar errores, omisiones y anomalías. Generalmente las inspecciones se centran en el código fuente, pero puede inspeccionarse cualquier representación legible del software como los requerimientos o el modelo de diseño. Cuando se inspecciona un sistema se utiliza el conocimiento existente del mismo, su dominio de aplicación, el lenguaje de programación y/o el modelo de diseño para descubrir errores.

Para realizar la inspección se utiliza una lista de comprobación (*check-list*) que contiene los defectos más comunes, por ejemplo uso de variables no inicializadas o asignaciones de tipos no compatibles. Estas listas dependen del lenguaje de programación y de la organización. Aunque el proceso implica diferentes roles como lo son Moderador o Encargado de la Inspección, Secretario, Lector, Inspector y Autor; pueden ser realizadas por una sola persona como es el caso de nuestro experimento. Cuando la inspección es realizada por una sola persona suele ser llamada *prueba de escritorio*. La *check-list* utilizada en nuestro experimento puede consultarse en el documento *Lista de Comprobación*.

Como ventajas sobre las técnicas dinámicas podemos encontrar que *no* requiere la ejecución del código; puede aplicarse a cualquier representación del sistema (requerimientos, diseño, configuración, datos, pruebas de datos, etc.); y pueden descubrirse muchos defectos diferentes en una sola inspección. En cambio, en el uso de las técnicas dinámicas un defecto puede enmascarar a otro, por lo que se puede requerir varias ejecuciones. Una desventaja muy importante que tiene esta técnica es que no puede comprobar características no funcionales como rendimiento, usabilidad, etc.

3.2. Partición de Equivalencia y Análisis de Valores Límite

El objetivo de las Particiones de Equivalencia es definir casos de prueba que descubran clases de defectos, reduciendo así el número total de casos de prueba que hay que desarrollar para cubrir las condiciones de entrada. Para ello se divide el dominio de entrada de un programa en un número finito de clases de equivalencia (clases de datos) de los que se pueden derivar casos de prueba.

Se asume que realizar una prueba con un valor representativo de una clase es equivalente a realizar una prueba con cualquier otro valor de dicha clase de equivalencia. Es decir, si un caso de prueba correspondiente a un elemento de una clase provoca una falla, cualquier otro elemento de dicha clase debe provocarla. A su vez, si un caso de prueba correspondiente a un elemento de una clase es exitoso, cualquier otro elemento de dicha clase provocará el mismo resultado.

Las clases de equivalencia se identifican examinando cada condición de entrada y dividiéndola en dos o más grupos. Las clases de equivalencia pueden ser válidas (conjuntos de entradas validas para el programa) o no válidas, donde se agrupan valores erróneos para el programa.

Existen pautas para identificar las clases de equivalencia. Si la entrada especifica un conjunto de valores de entrada válidos, define una clase de equivalencia válida (dentro del conjunto) y otra no válida (fuera del conjunto). O si una

condición de entrada especifica una situación obligatoria, define una clase de equivalencia válida y otra inválida, entre otras.

Por ejemplo, supongamos que una especificación de un programa indica que el programa acepta de 5 a 10 entradas que son enteros de 3 dígitos mayores a 100. Aplicando la técnica Partición de Equivalencia para verificar este programa obtendríamos la tabla que se muestra en el cuadro 3.1:

Condición de Entrada	Clases	¿Válida?
Número de valores de entrada	Menor que 5	No
	Entre 5 y 10	Si
	Mayor que 10	No
Valores de entrada	Menor que 100	No
	Entre 100 y 999	Si
	Mayor que 999	No

Cuadro 3.1: Ejemplo de particiones de equivalencia

Al momento de definir los casos de prueba hay que tener en cuenta que el objetivo es minimizar la cantidad de casos. Para ello, se deben diseñar estratégicamente de modo que un caso de prueba considere tantas condiciones de entrada como sea posible intentando que no se oculten defectos. Además se debe tener en cuenta que se debe escribir cada caso de prueba para que cubra tantas clases de equivalencia válidas no cubiertas como sea posible. O que cubra una y solo una clase de equivalencia inválida no cubierta. En el ejemplo, de ser posible se diseñaría *un* caso de prueba que cumpla las dos clases válidas, y un caso de prueba adicional para cada clase inválida. Esto da un mínimo de 5 casos de prueba.

La técnica de Análisis de Valores Límite, como su nombre lo indica, se basa en el análisis de condiciones límite, que son aquellas que se hayan en los márgenes de las clases de equivalencia. Por lo tanto los casos de prueba se eligen de forma que ejerciten los valores límites. Esta técnica complementa la técnica de particiones de equivalencia, ya que en lugar de seleccionar cualquier caso de prueba de las clases válidas e inválidas, se eligen los casos de prueba en los límites. Siguiendo con el ejemplo, para probar los límites de la clase inválida “Menor que 5” habría que diseñar un caso de prueba que utilice como entrada 4 valores (caso límite). En el caso de la clase válida “Entre 5 y 10” habría que diseñar dos casos de prueba, uno para 5 valores de entrada y otro para 10 valores de entrada.

3.3. Tablas de Decisión

Una tabla de decisión es una representación matricial de la lógica de una decisión, que especifica las posibles condiciones para la decisión y las acciones resultantes. La técnica Tablas de Decisión consiste en diseñar casos de prueba basados en las condiciones sobre el dominio de entrada y las acciones a ejecutar o salidas resultantes. Se deben identificar las entradas válidas para determinada condición y las acciones que se disparan para cada conjunto de entrada, llamadas reglas.

Para realizar la tabla a partir de la cual se obtendrán los casos de prueba se deben determinar todas las condiciones que son relevantes al problema y determinar todos los valores que cada condición puede tomar, nombrar todas las posibles acciones que pueden ocurrir, listar todas las posibles reglas y definir las acciones para cada regla. Con estos datos se arma una tabla como se muestra en el cuadro 3.2.

Condiciones	Entradas
c1:	
c2:	
c3:	
Acciones	Reglas
a1:	
a2:	
a3:	

Cuadro 3.2: Tabla de Decisión

Esta técnica es muy útil para describir situaciones en las cuales las posibles combinaciones de resultados (acciones, salidas) dependen de combinaciones de varias condiciones de entrada. Las tablas de decisión se clasifican de acuerdo al tipo de entradas, si tiene entradas limitadas, es decir si las entradas son binarias (V/F, 0/1, S/N) o si tiene entradas extendidas, es decir si las entradas pueden asumir múltiples valores.

El cuadro 3.3 es un ejemplo de una tabla de decisión con entradas limitadas. En este ejemplo un local de venta de ropa envía catálogos a sus clientes. Los catálogos son de tres tipos: ropa masculina para adulto, ropa femenina para adulto y ropa para niño. Se envían los catálogos según las compras que realizó el cliente en el último mes.

Condiciones	Entradas							
c1: El cliente compró ropa masculina para adulto en el último mes	S	S	S	S	N	N	N	N
c1: El cliente compró ropa femenina para adulto en el último mes	S	S	N	N	S	S	N	N
c1: El cliente compró ropa de niño en el último mes	S	N	S	N	S	N	S	N
Acciones	Reglas							
a1: Envío del catalogo de ropa masculina para adulto	X	X	X	X				
a2: Envío del catalogo de ropa femenina para adulto	X	X			X	X		
a3: Envío del catalogo de ropa para niño	X		X		X		X	

Cuadro 3.3: Ejemplo de una Tabla de Decisión

3.4. Criterio de Cubrimiento de Condición Múltiple

Para aplicar la técnica de Cubrimiento de Condición Múltiple es necesario conocer dos conceptos fundamentales: *decisión* y *condición*. Una decisión indica por qué camino del código se va a seguir. Como ejemplos se pueden encontrar: *if*, *while*, *do-while*, y *case* entre otros. Una condición es una expresión lógica que compone una decisión, por ejemplo (edad < 25), (sexo = masculino), (estado civil = casado).

Esta técnica se basa en generar casos de prueba de tal forma que todas las combinaciones posibles de resultados de las condiciones en cada decisión se ejecuten al menos una vez. Por ejemplo, para una decisión (A y B) con A y B condiciones se deben ejecutar: A true y B true, A true y B false, A false y B true, y A false y B false.

En el ejemplo:

```
metodo(int a, int b) {
   If ((a > 1) and (b = 0))
      x = x / a;
   If ((a = 2) or (x > 1))
      x = x + 1;
}
```

Se tienen dos decisiones d_1 y d_2, compuestas por las condiciones c_1 y c_2, y c_3 y c_4 respectivamente:

- $d_1 = ((a > 1)\ and\ (b = 0))$
  - $c_1 = (a > 1)$
  - $c_2 = (b = 0)$
- $d_2 = ((a = 2)\ or\ (x > 1))$
  - $c_3 = (a = 2)$
  - $c_4 = (x > 1)$

Para satisfacer el criterio se deben ejecutar las siguientes 8 combinaciones:

1. $a > 1$, $b = 0$: cumple c_1 true, c_2 true, en d_1
2. $a > 1$, $b \neq 0$: cumple c_1 true, c_2 false, en d_1
3. $a \leq 1$, $b = 0$: cumple c_1 false, c_2 true, en d_1
4. $a \leq 1$, $b \neq 0$: cumple c_1 false, c_2 false, en d_1
5. $a = 2$, $x > 1$: cumple c_3 true, c_4 true, en d_2
6. $a = 2$, $x \leq 1$: cumple c_3 true, c_4 false, en d_2
7. $a \neq 2$, $x > 1$: cumple c_3 false, c_4 true, en d_2
8. $a \neq 2$, $x \leq 1$: cumple c_3 false, c_4 false, en d_2

Esta combinaciones no implican que necesariamente se deban generar 8 casos de prueba. Un caso de prueba puede ejecutar más de una combinación. La cantidad de casos de prueba dependerá de la habilidad de quien diseñe los mismos.

Para este ejemplo, la menor cantidad de casos de prueba (CP) que ejecuta las 8 combinaciones es 4. Una posibilidad es:

- CP_1: $a = 2$, $b = 0$, $x = 4$ cubre las combinaciones 1 y 5
- CP_2: $a = 2$, $b = 1$, $x = 1$ cubre las combinaciones 2 y 6
- CP_3: $a = 1$, $b = 0$, $x = 2$ cubre las combinaciones 3 y 7
- CP_4: $a = 1$, $b = 1$, $x = 1$ cubre las combinaciones 4 y 8

En estos casos de prueba se aprecia que la técnica puede no cubrir todos los caminos posibles, como es el caso del camino que resulta cuando d_1 es verdadera y d_2 es falsa, el cual nunca se ejecuta.

3.5. Trayectorias Linealmente Independientes

Esta técnica consiste en generar un conjunto de casos de prueba que ejecute cada trayectoria independiente en el código fuente una única vez. Una trayectoria t es independiente de un conjunto de trayectorias $T = \{t_1..t_n\}$ si *no* es una combinación lineal de éstas. t es combinación lineal del conjunto T si se puede escribir $t = \alpha t_1 + \beta t_2 + .. + \mu t_{n-1} + \omega t_n$. En otras palabras, una trayectoria es independiente de otra trayectoria si incluye una rama (mirando el grafo del flujo de control) que la otra no contiene. El número de trayectorias independientes se calcula usando la complejidad ciclomática. La formula para calcularlo es: $CC = Arcos - Nodos + 2$. La complejidad ciclomática da el número mínimo de casos de prueba necesarios para probar todas las trayectorias independientes.

Para generar los casos de prueba se debe construir el grafo de flujo de control para el código que se va a verificar, identificar las trayectorias independientes y verificar que la cantidad de trayectorias sea igual a la complejidad ciclomática. Por último, se genera un conjunto de casos de prueba que ejecute cada trayectoria independiente una única vez. A continuación se presenta un ejemplo de aplicación de esta técnica.

El código:

```
metodo(int a, int b) {
   If ((a > 1) and (b = 0))
      x = x / a;
   If ((a = 2) or (x > 1))
      x = x + 1;
}
```

presenta cuatro trayectorias, tres de las cuales son independientes, ya que $CC = 3$. La figura 3.1 muestra las cuatro trayectorias identificadas en el grafo de flujo de control del ejemplo.

En la figura 3.2 se muestra un subconjunto de trayectorias, del conjunto anterior, que son independientes.

La cuarta trayectoria puede obtenerse como una combinación lineal de las tres trayectorias independientes, como muestra la figura 3.3.

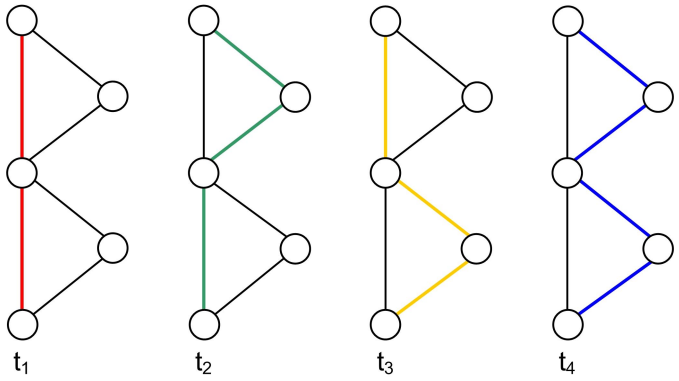

Figura 3.1: Trayectorias del Ejemplo

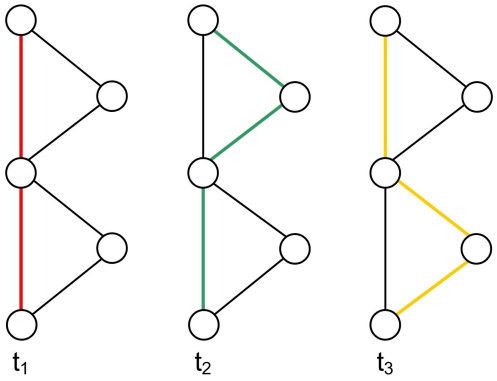

Figura 3.2: Trayectorias Independientes del Ejemplo

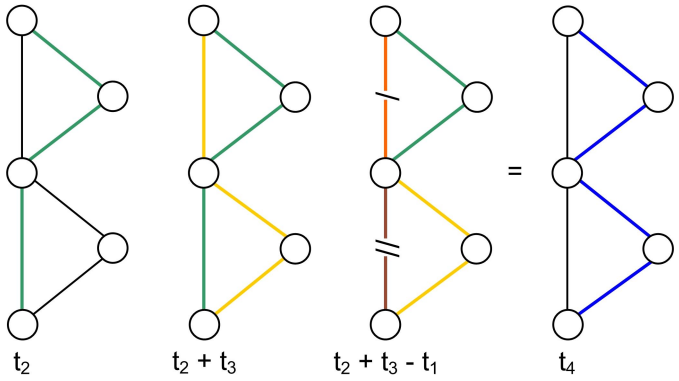

Figura 3.3: Combinación lineal de las Trayectorias Independientes

Capítulo 4

Clasificación de Defectos

Después de codificar un programa, por lo general se buscan los defectos para eliminarlos lo más pronto posible. Con este fin se prueban los programas para aislar la mayor cantidad de defectos, creando condiciones especiales donde se espera que el código no reaccione como está planeado. Por lo tanto, es importante saber qué clase de defectos se están buscando.

Resulta útil categorizar y rastrear los tipos de defectos que se encuentran. La información histórica puede ayudar a predecir qué tipos de defectos es más probable que tenga el código (ayudando a orientar las pruebas), así como grupos de determinados tipos de defectos pueden alertar sobre el funcionamiento de determinada característica o etapa del proceso de construcción de software. La comprensión de la cantidad y de la distribución de tipos de defecto puede ayudar a sugerir cuáles son los tipos de defecto que los verificadores deben buscar y a reducir el número de defectos inyectados en la construcción de los programas, dado que los desarrolladores pueden prestar atención en no cometer los defectos que ya conocen.

En nuestro experimento utilizamos dos clasificaciones de defectos: la Clasificación Ortogonal de Defectos de IBM [Cla] y la Taxonomía de Defectos de Beizer [Bei90]. Ambas clasificaciones se suponen ortogonales. Uno de los aspectos claves de una clasificación ortogonal de defectos es su ortogonalidad. Un esquema de clasificación es ortogonal cuando cada ítem que es clasificado pertenece exactamente a una única categoría. Esto es deseable porque permite clasificar los defectos en un programa de manera no ambigua, siendo significativa la información acerca del número de defectos de cada tipo en el programa. Si un defecto pudiera pertenecer a más de una categoría las mediciones perderían sentido. De la misma manera, la clasificación de defectos debe ser clara, de modo que sea probable que dos verificadores cualesquiera clasifiquen un defecto particular de la misma manera.

4.1. Clasificación Ortogonal de Defectos de IBM

Para esta clasificación, la información sobre un defecto se puede dividir en dos puntos específicos en el tiempo: la apertura del defecto y el cierre del defecto. Cuando un defecto está abierto, se conocen las circunstancias que condujeron a la exposición del defecto y el impacto que tiene para el usuario. Un defecto

pasa a estar cerrado cuando es corregido. En este punto se conocen la naturaleza exacta del defecto y el alcance de la corrección. Las categorías de la Clasificación de IBM capturan la semántica de un defecto desde estas dos perspectivas. Cada perspectiva se plasma en una sección: *Opener Section* y *Closer Section*.

Aunque en este estudio los defectos no son corregidos, el enfoque con el que se clasifican es el de la *Closer Section*. Se pueden observar sus categorías en el cuadro 4.1. La *Opener Section* no se utiliza en este estudio.

Target	Defect Type	Qualifer	Age	Source
Design/Code	Assignment/Initialization Checking Algorithm/Method Function/Class/Object Timing/Serialization Interface/O-O Messages Relationship	Missing Incorrect Extraneous	Base New Rewritten ReFixed	Developed In-House Reused From Library Outsourced Ported

Cuadro 4.1: Closer Section

Para clasificar en la *Closer Section* hay que dar valor a cinco atributos: *Target*, que representa la etapa de construcción de software en la que se cometió el defecto; *Defect Type*, que describe la corrección realizada, en nuestro caso, describe el defecto detectado; *Qualifer*, que indica si el defecto se debe a una omisión, un error o algo irrelevante; *Age*, que indica el momento en el que se introdujo el defecto; y *Source*, que define el defecto en función de su historia.

Para cada uno de estos atributos, tenemos los valores:

- Target

  1. Design/Code.- Único valor de este atributo. Coincide con la ubicación de los defectos que interesa detectar en este experimento: los ubicados en el código fuente del programa.

- Defect Type

  1. Assignment/Initialization.- Valor asignado incorrectamente o no asignado. Si se trata de múltiples asignaciones, pasa a ser de tipo *Algorithm*. Ejemplo: variable interna o variable dentro de un bloque de control no tiene valor (falto la inicialización) o tiene un valor incorrecto (se inicializó mal).
  2. Checking.- Errores causados por omisión o validación incorrecta de parámetros o datos en declaraciones condicionales. Ejemplos: (1) Valores mayores a 100 no son válidos, pero falta el chequeo de que los valores sean menores a 100. (2) El loop condicional debería haberse detenido en la novena iteración, pero continuó iterando mientras el contador era <= 10.
  3. Algorithm/Method.- Problemas de eficiencia o correctitud que afectan una tarea y pueden ser arreglados (re)implementando un algoritmo o estructura local de datos sin la necesidad de cambiar el diseño. Problema en el procedimiento, la plantilla, o función sobrecargada

que describe un servicio ofrecido por un objeto. Ejemplo: el diseño de bajo nivel llama a un algoritmo que mejora el rendimiento sobre un enlace retrasando la transmisión de algunos mensajes, pero la implementación transmite todos los mensajes tan pronto llegan. Falta el algoritmo que retrasa la transmisión.

4. Function/Class/Object.- El error requiere un cambio formal en el diseño, dado que afecta la capacidad, interfaces con el usuario final, interfaces de producto, interfaces con la arquitectura del hardware, o la/s estructura/s de datos global/es. Ejemplo: la base de datos no incluye el campo "domicilio" aunque los requerimientos lo especifican.
5. Timing/Serialization.- Falta la serialización de un recurso compartido, se serializa el recurso incorrecto, o se utiliza una técnica de serialización incorrecta. Ejemplo: Falta serialización al momento de actualizar un bloque de control compartido.
6. Interface/O-O Messages.- Problemas de comunicación entre módulos, componentes, *device drivers*, objetos y funciones; debido a macros, llamadas, bloques de control y listas de parámetros. Ejemplos: (1) Una base de datos implementa las funciones de inserción y eliminación, pero la interfaz para la eliminación no se encuentra accesible. (2) La interfaz especifica un puntero a un número, pero la implementación espera un puntero a un caracter.
7. Relationship.- Problemas relacionados con asociaciones entre procedimientos, estructuras de datos y objetos. Estas asociaciones pueden ser condicionales. Ejemplo: falta la relación de jerarquía entre dos clases o está especificada incorrectamente.

- Qualifer

1. Missing.- El defecto se debe a una omisión. Ejemplo: falta una declaración de asignación.
2. Incorrect.- El defecto se debe a un error. Ejemplo: un chequeo utiliza valores incorrectos.
3. Extraneous.- El defecto se debe a algo irrelevante o pertinente a la documentación. Ejemplo: hay una sección en el documento de diseño que no es pertinente con el producto actual y debe ser eliminada.

- Age

1. Base.- El defecto está en una parte del producto que no ha sido modificada por el proyecto actual, y no forma parte de una biblioteca. El defecto no fue inyectado por el proyecto actual, por lo tanto era un defecto latente.
2. New.- El defecto está en una función que fue creada por y para el proyecto actual.
3. Rewritten.- El defecto fue introducido como un resultado directo de re-diseño y/o re-implementación de funciones viejas en un intento de mejorar su diseño o calidad.
4. ReFixed.- El defecto fue introducido por la solución implementada para arreglar un defecto previo.

- Source
    1. Developed In-House.- El defecto se encontró en un área de código que fue desarrollada por el propio equipo de desarrollo de la organización.
    2. Reused From Library.- El defecto se encuentra usando parte de una librería. El problema puede ser porque la parte se utiliza incorrectamente, o porque hay un problema en ella.
    3. Outsourced.- El defecto está en una parte proveída por un agente externo a la organización.
    4. Ported.- El defecto tiene que ver con el uso de una parte que fue validada para un ambiente diferente.

4.2. Taxonomía de Defectos de Boris Beizer

A diferencia de la anterior, esta clasificación es jerárquica, y no hace distinción entre la información existente cuando se detecta el defecto y cuando se corrige, simplemente hace una descripción de los defectos que corresponden a cada categoría.

Beizer define 9 grandes categorías que se subdividen en varios niveles de subcategorías. Cada nivel de subcategorías afina la definición de su categoría padre.

Cada tipo de defecto se identifica con un número de 4 dígitos, uno por nivel de anidamiento. Algunas categorías tienen más de cuatro niveles de anidamiento, en este caso se utilizan subnúmeros separados por puntos para identificarlas: "1234.1.6". La letra "x" se utiliza como un marcador de posición, que se transforma en un número a medida que la taxonomía se expande.

A continuación se presenta un ejemplo de la jerarquización de la taxonomía:

- 3xxx representa la categoría *Structural Bugs*
- 32xx representa la categoría *Processing*, dentro de *Structural Bugs*
- 322x representa la categoría *Expression Evaluation*, dentro de *Processing*, dentro de *Structural Bugs*
- 3222 representa la categoría *Arithmetic Expressions*, dentro de *Expression Evaluation*, dentro de *Processing*, dentro de *Structural Bugs*
- 3222.1 representa la categoría *Wrong Operator* dentro de *Arithmetic Expressions*, dentro de *Expression Evaluation*, dentro de *Processing*, dentro de *Structural Bugs*.

Como puede observarse, en el ejemplo anterior la categoría *Structural Bugs* llega a tener 5 niveles de anidamiento de subcategorías, por lo que es necesario la utilización de un subnúmero.

El último dígito de un nivel de categorías es siempre el 9, por ejemplo: 9xxx, 39xx, 3229. Estas categorías (las identificadas con el número 9) se usan cuando no está disponible una descomposición más fina y el defecto encontrado no coincide exactamente con una de las categorías existentes. Por ejemplo, un defecto no clasificado es un defecto 9xxx, un defecto en *Structural Bugs* (3xxx) que no clasifica en ninguna subcategoría es 39xx, etc.

Las grandes categorías de la taxonomía son:

- (1xxx) Functional bugs: Requirements and Features.- Problemas en la definición de los requerimientos y/o especificación.
- (2xxx) Functionality as implemented.- Defectos en la implementación de funcionalidades del software.
- (3xxx) Structural bugs.- Defectos estructurales en el código. Por ejemplo: defectos de secuencia y control, defectos lógicos, defectos de procesamiento, defectos de inicialización, defectos y anomalías en el flujo de datos, etc.
- (4xxx) Data.- Defectos en la especificación de los objetos de datos, de sus formatos, del número de objetos, y de sus valores iniciales.
- (5xxx) Implementation.- Defectos detectados generalmente por los compiladores: datos sin declarar, rutinas sin declarar, problemas de inicialización, etc.; y defectos de documentación.
- (6xxx) Integration.- Defectos de integración, como ser defectos en interfaces externas, interfaces internas, defectos de control y secuencia, problemas en el manejo de recursos, etc.
- (7xxx) System and software architecture.- Defectos en la arquitectura del hardware, sistema operativo y arquitectura del software.
- (8xxx) Test definition or execution bugs.- Defectos en la implementación de las pruebas y/o en los criterios de prueba.
- (9xxx) Other bugs, unspecified.- Todo defecto que no corresponda a ninguna de las categorías anteriores.

En el experimento se utilizan las categorías: *(2xxx) Functionality as implemented*, *(3xxx) Structural bugs*, *(4xxx) Data*, *(6xxx) Integration* y *(9xxx) Other bugs, unspecified*. No se consideran las categorías que incluyen defectos fuera del diseño y/o del código, por ejemplo, defectos en la documentación. Por esta razón no se utilizan las categorías *(1xxx) Requirementes and Features* y *(5xxx) Implementation*. La categoría *Test definition or execution bugs (8xxx)* no se toma en cuenta porque el experimento no estudia los defectos que pudieran existir en las pruebas construidas por los verificadores ni en los criterios de prueba utilizados. Finalmente, los programas no interactúan con otros sistemas, y no se considera que existan defectos de interacción con el sistema operativo, o problemas con el hardware, etc. por lo que queda fuera la categoría *System and software architecture (7xxx)*.

A continuación se describe el primer nivel de subcategorías de cada una de las categorías utilizadas en el experimento. No se entrará en el detalle de los niveles más anidados porque no agrega al cometido de esta sección, basta con conocer que estos subniveles afinan las definiciones aquí presentadas.

Dentro de la categoría *(2xxx) Functionality as implemented* se encuentran las subcategorías:

- (21xx) Correctness.- Tiene que ver con la correctitud de la implementación. Ejemplo: una funcionalidad tiene un comportamiento incorrecto respecto a la especificación o la funcionalidad es correcta pero interactúa de forma incorrecta con otras funcionalidades.

- (22xx) Completeness, Features.- Defectos en la completitud con la que las funcionalidades están implementadas. Ejemplos: falta una funcionalidad o se implementó una funcionalidad que no estaba especificada o se duplicó o se superpuso con una ya existente.
- (23xx) Completeness, Cases.- Defectos en la completitud de los casos de las funcionalidades. Ejemplos: falta un caso de una funcionalidad, o se implementó un caso que no estaba especificado, o se duplico o superpuso un caso con uno ya existente, etc.
- (24xx) Domains.- El procesamiento del caso o la funcionalidad depende de una combinación de valores de entrada. Un defecto de dominio existe si un procesamiento inadecuado es ejecutado para una combinación de valores de entrada. El procesamiento se asume correcto. Ejemplos: dominio de entrada incorrecto, defectos en los límites entre los dominios de entrada, etc.
- (25xx) User Messages and Diagnostics.- Prompt's, listados u otra forma de comunicación con el usuario es incorrecta. Ejemplos: advertencias falsas, advertencias faltantes, mensajes incorrectos, errores en ortografía y formas, etc.
- (26xx) Exception Conditions Mishandled.- Condiciones de excepción ilógicas, problemas de recursos, modos de falla, etc., que requieren un manejo especial, y que no son manejadas correctamente o se utilizan mecanismos incorrectos de manejo de excepciones.

En la categoría *(3xxx) Structural bugs* se definen las subcategorías:

- (31xx) Control Flow and Sequencing.- Defectos específicamente relacionados con el flujo de control del programa. Ejemplos: caminos inaccesibles, código inalcanzable, código muerto, defectos en el control de loops, defectos en la inicialización del flujo de control y en cambios de estado que afectan el flujo de control, etc.
- (32xx) Processing.- Defectos relacionados con el procesamiento bajo la suposición de que el flujo de control es correcto. Ejemplos: se utiliza el algoritmo incorrecto; defectos en la evaluación de expresiones aritméticas, booleanas, etc.; defectos en la inicialización de variables, expresiones, funciones, etc. usadas en procesamiento (auxiliares); defectos en la precisión utilizada; tiempo de ejecución inadecuado; etc.

La categoría *(4xxx) Data* presenta las subcategorías:

- (41xx) Data definition, structure, declaration.- Defectos en la definición, estructura e inicialización de datos. Esta categoría se aplica si el objeto es declarado estáticamente en el código original o creado dinámicamente. Ejemplos: el tipo de datos con que se declara el objeto es incorrecto; defectos en la dimensión con que se declara un objeto; defectos en los valores iniciales asignados al objeto de datos, selección de valores por defecto incorrectos, o fallar al suministrar un valor por defecto en caso de ser necesario; defectos relacionados a la duplicación incorrecta o la falla al crear un objeto duplicado; el scope, partición o componente al cual aplica el objeto está especificado de forma incorrecta; etc.

- (42xx) Data access and handling.- Relacionado con el acceso y la manipulación de objetos de datos que se asumen correctamente definidos. Ejemplos: uso incorrecto del objeto debido al tipo de datos que presenta; defectos en la inicialización *dinámica* de objetos; defectos en duplicación y alias dinámico (en tiempo de ejecución) de objetos; defectos en el acceso a objetos, como ser leer, escribir, modificar, (y en algunos casos) crear y destruir.

La categoría *(6xxx) Integration* tiene dos subcategorías: *(61xx) Internal Interfaces* y *(62xx) External Interfaces and Timing*. No se describirá la segunda porque no hay defectos en el experimento que puedan clasificar en ella.

- (61xx) Internal Interfaces.- Defectos relacionados con las interfaces entre componentes que se comunican con el programa bajo test. Se asume que los componentes son correctos. Ejemplos: defectos relacionados con los componentes (subrutina, función, macro, programa, segmento de programa, etc.) de software que son invocados; defectos relacionados con los parámetros de la invocación, su número, orden, tipo, posición, valores, etc; defectos relacionados con la interpretación de los parámetros proporcionados por el componente invocado, en retorno al componente invocador, o sobre el pase del control a algún otro componente; el componente invocado no está inicializado, o está inicializado en un estado erróneo, o con datos incorrectos; el lugar o estado en el que se invoca a un componente en el componente invocador es incorrecto; el componente no debería haber sido invocado o ha sido invocado más a menudo que lo necesario.

La Taxonomía de Beizer puede consultarse en el libro de Boris Beizer, *Software Testing Techniques* [Bei90]. El reporte técnico *Análisis y Ejemplos de la Taxonomía de Defectos de Beizer* [VL08] realiza un análisis y brinda ejemplos de las categorías que la componen. En el reporte técnico *Marco teórico para evaluar taxonomías de defectos* [GV09] se discuten las dificultades de clasificar tanto en la Clasificación de IBM como en la Taxonomía de Beizer.

Capítulo 5

Efectividad y Costo de Técnicas de Verificación Respecto a Tipos de Defecto

El experimento presentado en la Parte II se basa en una línea de experimentos realizados por varios autores que han ido agregado conocimiento al ya generado por experimentos previos. El estudio original fue llevado a cabo por Basili y Selby en 1982, 1983 y 1984 [BS87]. Este experimento estudia la efectividad y eficiencia de diferentes técnicas de verificación respecto a varios tipos de defecto. Más adelante Kamsties y Lott [KL95] replicaron y extendieron este estudio en 1995. Esta replicación asume las mismas hipótesis que las del experimento de Basili y Selby, pero difiere en el lenguaje de programación utilizado y en el proceso de detección de defectos. Kamsties y Lott construyeron un paquete de laboratorio para facilitar la réplica externa del experimento. En 1997 el experimento fue replicado otra vez por Wood, Rober, Brooks y Miller [WRBM97], en busca de afianzar los resultados ya obtenidos. En este caso, el experimento siguió exactamente las mismas líneas que el realizado por Kamsties y Lott. Hicieron uso del paquete de laboratorio.

Uno de los trabajos más recientes en esta línea es el de Juristo y Vegas en 2003 [JV03], quienes replicaron el experimento de Kamsties y Lott. Juristo y Vegas alteraron algunas de las hipótesis originales, modificando el paquete de laboratorio. En su estudio comparan los resultados que obtienen con los obtenidos por los experimentos anteriores: el de Basili y Selby, el de Kamsties y Lott, y el de Wood *et al.* En el presente capítulo se describen los aspectos principales de estos 4 experimentos.

En el cuadro 5.1 se puede observar un resumen de las técnicas utilizadas, aspectos estudiados y resultados obtenidos en cada uno de ellos. En el cuadro se hace referencia a las técnicas estáticas como *lectura de código*, a las técnicas dinámicas de caja negra como *técnica funcional*, y a las técnicas dinámicas de caja blanca como *técnica estructural*.

Nota 1: Se entiende que se ***detecta*** *un defecto cuando se observa una falla*

en el software que evidencia su existencia, pero se desconoce la ubicación del mismo en el código fuente del programa. Se entiende que se ***aísla*** *un defecto cuando el mismo es ubicado en el código fuente del programa.*

Nota 2: ***Stepwise abstraction*** *es una técnica de lectura de código.*

Autor	Técnicas	Aspecto	Resultados
Basili & Selby'87	- Partición de Equivalencia y Análisis de Valores Límite. - 100 % de cubrimiento de sentencias. - *Stepwise abstraction.*	Efectividad en la detección de defectos	- Programadores experientes detectan más defectos con lectura de código, luego con la técnica funcional. - Programadores inexperientes: En un caso, no hay diferencia entre las técnicas. En el otro, la lectura de código y la técnica funcional se comportaron igual, y ambas mejor que la estructural. - Depende del tipo de programa. - La estimación personal sobre porcentaje de defectos detectados fue más precisa para lectura de código, y fue menos precisa para la técnica funcional.
		Costo de la detección de defectos	- Programadores experientes tienen mayor tasa de detección de defectos con lectura de codigo. - Las técnicas funcional y estructural presentan la misma tasa de detección. - Programadores inexperientes presentan la misma tasa de detección de defectos para todas las técnicas. - La tasa de detección de defectos depende del tipo de programa.
		Tipo de defecto	- La lectura de código detecta más defectos de interfaz que las otras técnicas. - La técnica funcional detecta más defectos de control que las otras técnicas.
Kamsties & Lott'95	- Partición de Equivalencia y Análisis de Valores Límite. - 100 % de cubrimiento de saltos, condiciones múltiples, bucles y operadores relacionales. - *Stepwise abstraction.*	Efectividad (detección)	Depende del programa, no de la técnica.
		Efectividad (ais.)	Depende del programa y del sujeto, no de la técnica.
		Eficiencia (detección)	- Sujetos inexperientes: La técnica funcional es la que emplea menos tiempo. El tiempo empleado en encontrar defectos también depende del sujeto. - La técnica funcional tiene la mayor tasa de detección de defectos.
		Eficiencia (ais.)	- Depende del sujeto. - Con sujetos inexperientes, la técnica funcional tiene en promedio la mayor tasa de detección de defectos.
		Eficiencia (total)	- Con sujetos inexperientes, la técnica funcional es la que emplea menos tiempo. - El tiempo también depende del sujeto.
		Tipo de defecto	Se observaron diferencias de efectividad entre las técnicas.
Wood *et al.*'97	- P.E. y A.V.L. - 100 % de cubrimiento de saltos. - *Stepwise abstraction.*	Efectividad (detección)	- Depende de la combinación de programa/técnica. - Depende de la naturaleza de los defectos.
		Técnicas combinadas	Mayor cantidad de defectos combinando técnicas.
Juristo & Vegas'02	- P.E. y A.V.L. - 100 % de cubrimiento de saltos. - *Stepwise abstraction.*	Efectividad	- Depende de la visibilidad de las fallas. - Depende del programa. - Las técnicas funcional y estructural se comportaron de la misma forma, y la lectura de código peor que ambas. - Depende del defecto en cuestión, y no del tipo de defecto.
		Tipo de defecto	- Los defectos cosméticos son los más difíciles de visualizar, independientemente de la técnica. - El tipo de defecto no impacta sobre la lectura de código.

Cuadro 5.1: Descripción de experimentos anteriores

5.1. Victor R. Basili y Richard W. Selby. 1987

En 1987 Basili y Selby presentan un estudio controlado [BS87], en el que aplican un experimento, buscando cómo verificar el software de una manera eficaz. Los aspectos sobre los que se centra el trabajo son: *efectividad en la detección de defectos*, definida como cantidad de defectos detectados; *costo de la detección de defectos*, entendido como defectos/esfuerzo; y *clases de defectos detectados*.

Basiliy y Selby plantean los objetivos del experimento en forma de preguntas, agrupadas bajo los aspectos mencionados:

Efectividad en la detección de defectos:

- ¿Cuál de las técnicas de verificación detecta mayor cantidad de defectos?
    1. ¿Cuál de las técnicas detecta el mayor porcentaje de defectos en los programas (los programas contienen cantidades diferentes de defectos)?
    2. ¿Cuál de las técnicas expone el mayor número (o porcentaje) de los defectos en los programas (defectos que son observables pero no necesariamente reportados)?
- ¿La cantidad de defectos observados es dependiente del tipo de software?
- ¿La cantidad de defectos observados es dependiente del nivel de experiencia del verificador?

Costo de la detección de defectos:

- ¿Cuál de las técnicas de verificación tiene la tasa más alta de detección de defectos (#defectos/esfuerzo)?
- ¿La tasa de detección de defectos es dependiente del tipo de software?
- ¿La tasa de detección de defectos es dependiente del nivel de experiencia del verificador?

Clases de defectos observados:

- ¿Las técnicas tienden a capturar diferentes clases de defectos?
- ¿Qué clases de defectos son observables pero no son registrados?

Se estudia la relación entre la eficacia de las pruebas mediante varios factores: técnica de verificación, tipo de software, tipo de defecto, experiencia del verificador, y cualquier interacción entre estos factores.

El estudio intenta reproducir una verificación real con la finalidad de evaluar la efectividad de las diferentes técnicas de verificación utilizadas lo más objetivamente posible, en contraste con otros estudios que crean la mejor situación posible. Para ello, los sujetos son elegidos tal que sean representativos de diferentes niveles de experiencia, los programas verificados corresponden a distintos tipos y reflejan estilos de programación comunes, y los defectos en los programas son representativos de aquellos que ocurren frecuentemente en el software.

Se analizan tres técnicas: partición en clases de equivalencia y valores límite (técnica dinámica de caja negra, también llamada *funcional*), cubrimiento de sentencias del 100 % (técnica dinámica de caja blanca, también llamada *estructural*) y un tipo de lectura de código, denominado en inglés *stepwise abstraction*, (técnica estática). Se compara la aplicación individual de las tres técnicas con la intención de identificar sus diferentes ventajas y desventajas.

En la lectura de código el verificador identifica subprogramas principales en el código, determina sus funciones y las compone para determinar una función única para el programa entero. Luego compara esta función derivada con la especificación del programa. En el caso de la técnica estructural, el verificador inspecciona el código fuente y luego elabora y ejecuta casos de prueba basados en el porcentaje de sentencias ejecutadas, en este caso, en el 100 % de las sentencias. Luego compara el comportamiento observado en la ejecución con la especificación del programa. En la técnica funcional, el verificador construye casos de prueba a partir de la especificación del programa utilizando el criterio propio de la técnica. Luego ejecuta el programa y contrasta su comportamiento con el indicado en la especificación.

El experimento consta de sujetos con un amplio rango de experiencia profesional (desde estudiantes hasta profesionales con años de experiencia) que aplican técnicas de verificación sobre diferentes programas. La experiencia de los sujetos se divide en tres franjas: principiante, avanzado e intermedio. Los sujetos provienen de dos grupos: unos son estudiantes avanzados o recibidos de la carrera Ciencias de la Computación, y se encuentran haciendo el curso "Software Design and Development" de la Universidad de Maryland. Todos tienen un buen nivel académico y algunos trabajan part-time. Los otros son programadores profesionales de la NASA y Computer Sciences Corporation. Estos individuos son matemáticos, físicos e ingenieros que desarrollan software de soporte para satélites.

En el caso de los estudiantes, en el curso reciben una presentación de las técnicas lectura de código, funcional y estructural. La asignación de estudiantes a niveles de experiencia se basa en la experiencia profesional y las calificaciones en cursos relevantes. En el caso de los profesionales, tienen noción de todas las técnicas, pero las que más utilizan son técnicas funcionales. De todas formas reciben una instrucción de 4 horas sobre las técnicas, dictada por Selby. La asignación de estos sujetos a niveles de experiencia fue en base a los años de experiencia profesional, el grado de profundidad, y la asignación sugerida por su gerente.

Los cuatro programas verificados fueron:

- P_1: es un procesador de texto.
- P_2: es una rutina de trazado matemático.
- P_3: es un tipo de dato abstracto que consiste en un conjunto de utilidades para procesamiento de listas.
- P_4: es un administrador para una base de datos de referencias bibliográficas.

Estos programas tienen tamaños y cantidad de defectos diferentes, y se consideran de distinto tipo. Fueron escritos en un lenguaje de alto nivel (FORTRAN

y Simpl-T) conocido por los verificadores. El código fuente no contiene comentarios. Esto crea un peor escenario para los verificadores que realizan lectura de código.

Los defectos en los programas representan una distribución razonable de defectos que ocurren comúnmente en el software. Todos los defectos en los programas P_3 y P_4 son naturales de su construcción. Los otros dos programas tienen una mixtura de defectos, introducidos por los desarrolladores de los programas durante su construcción (defectos "reales") y por siembra de defectos.

Los defectos se clasifican usando dos esquemas distintos. El primero clasifica los defectos en defectos de *omisión* o defectos de *comisión*. Defectos de comisión son el resultado de un segmento de código incorrecto, por ejemplo, sumar en vez de restar en una operación aritmética. Defectos de omisión son el resultado de un olvido o una omisión, valga la redundancia, por parte del programador, por ejemplo, falta una inicialización.

El segundo esquema clasifica los defectos en seis clases:

- inicialización.- inicialización incorrecta. Ej.: asignar a una variable un valor inicial incorrecto.
- cálculo.- cálculo erróneo. Ej.: mal posicionamiento de paréntesis en un cálculo aritmético.
- control.- se ejecuta el flujo de control incorrecto para determinada entrada. Ej.: predicado erróneo en una sentencia IF-THEN-ELSE.
- interfaz.- ocurre cuando un módulo tiene suposiciones incorrectas sobre entidades fuera del ambiente local del módulo. Ej.: pasar un argumento incorrecto a un procedimiento.
- datos.- uso incorrecto de una estructura de datos. Ej.: determinar el índice del último elemento de un arreglo de forma incorrecta.
- cosmético.- es un error administrativo cometido en la entrada del programa. Ej.: un error de ortografía en un mensaje de error sería un defecto cosmético.

El estudio se realiza en tres fases, intentando reproducir los resultados obtenidos en distintos ambientes. Entiéndase por ambiente el lugar donde se realizan las sesiones de verificación, el momento en el tiempo en el que se realizan, la cantidad de sujetos que componen los tres grupos de experiencia y su procedencia, y los tres programas que se verifican, de los cuatro existentes. Cada fase presenta un ambiente distinto.

En cada fase se aplicó un diseño factorial-fraccionario (*fractional factorial design*). Este diseño distingue las técnicas de verificación, y permite variar el programa sobre el que se aplican y el sujeto que las ejecuta, y con ello el nivel de experiencia. Cada sujeto verifica tres programas por fase, utilizándose los cuatro programas a través de las tres fases del estudio. El cuadro 5.2 muestra el diseño factorial fraccionario utilizado en la tercera fase del estudio. En él se observa que el sujeto S_1 está en el nivel de experiencia avanzado, y verifica con la técnica estructural el programa P_1, con la técnica funcional el programa P_3 y con lectura de código el programa P_4.

Las variables independientes son: la técnica de verificación, el nivel de experiencia de los verificadores y el tipo de software. En el diseño ocurren todas las

		Lectura de Código			Técnica Funcional			Técnica Estructural		
		P_1	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_4
	S_1	–	–	X	–	X	–	X	–	–
Sujetos	S_2	–	X	–	X	–	–	–	–	X
Avanzados	...					...				
	S_8	X	–	–	–	–	X	–	X	–
	S_9	–	X	–	X	–	–	–	–	X
Sujetos	S_{10}	–	–	X	–	X	–	X	–	–
Intermedios	...					...				
	S_{19}	X	–	–	–	–	X	–	X	–
	S_{20}	–	X	–	X	–	–	–	–	X
Sujetos	S_{21}	X	–	–	–	–	X	–	X	–
Principiantes	...					...				
	S_{32}	–	–	X	–	X	–	X	–	–

Cuadro 5.2: Diseño factorial fraccionario

posibles combinaciones de sus valores. Cada sujeto no utiliza dos veces la misma técnica, ni verifica dos veces el mismo programa. Utiliza tres técnicas distintas y verifica tres programas distintos. Las variables dependientes incluyen el número de defectos detectados, porcentaje de defectos detectados, tiempo total de detección de defectos y tasa de detección de defectos. Las técnicas dinámicas tienen variables dependientes adicionales, como ser tiempo de ejecución de casos, porcentaje de defectos que son observables a partir de los casos de prueba, etc.

El orden en el que se ejecutan las técnicas es aleatorio a través de los sujetos, en cada nivel de experiencia y en cada fase del estudio. Todos los sujetos en una fase verifican el mismo programa, en el mismo día. Con esto se evita que en el tiempo que transcurre entre una sesión de verificación y la otra, los sujetos puedan intercambiar información sobre los programas que afecte el desempeño de futuras sesiones.

Cada una de las tres fases está formada por tres partes: entrenamiento, tres sesiones de verificación y una sesión de seguimiento. Todos los grupos de sujetos son expuestos a un entrenamiento similar sobre las técnicas de verificación antes de comenzado el estudio.

Las medidas más relevantes utilizadas en el análisis de datos son:

- Para la efectividad en la detección de defectos.

    1. Número de defectos detectados: es el número de defectos detectados por un sujeto aplicando una técnica de verificación sobre un programa.
    2. Porcentaje de defectos detectados: es el porcentaje de defectos de un programa, detectado por un sujeto, aplicando una técnica de verificación.

3. Porcentaje percibido de defectos detectados: es la estimación de un sujeto sobre el porcentaje de defectos de un programa que cree ha detectado con su verificación.

- Para el costo de la detección de defectos.

  1. Tasa de detección de defectos (número de defectos/hora): es el número de defectos detectados por un sujeto aplicando una técnica dividido por el esfuerzo en horas empleado.
  2. Tiempo de detección: es el número total de horas que un sujeto emplea en verificar un programa utilizando una técnica.

Los resultados principales del estudio son los siguientes:

- Con los programadores profesionales, la lectura de código detecta más defectos, y tiene una mayor tasa de detección de defectos que la técnica funcional y la estructural. La técnica funcional detectó más defectos que la técnica estructural, pero no fueron diferentes en la tasa de detección de defectos.
- En un grupo de estudiantes de nivel avanzado, la lectura de código y la técnica funcional no fueron diferentes en los defectos encontrados, pero fueron ambas superiores respecto a la técnica estructural, mientras que en el otro grupo avanzado, no hubo diferencias entre las técnicas.
- Con los estudiantes en nivel avanzado, no hubo diferencia en la tasa de detección de defectos entre las tres técnicas.
- El número de defectos observados, tasa de detección de defectos, y esfuerzo total en la detección dependen del tipo de software verificado.
- La lectura de código detecta más defectos de interfaz que las otras técnicas.
- La técnica funcional detecta más defectos de control que los otros métodos.
- Cuando se pidió estimar el porcentaje de defectos detectados, los lectores de código dieron la estimación más precisa, mientras que los que aplicaron la técnica funcional dieron la estimación menos precisa.

5.2. Erik Kamsties y Christopher M. Lott. 1995

Kamsties y Lott [KL95] replicaron y extendieron el experimento llevado a cabo por Basili y Selby en los 80', para identificar condiciones bajo las cuales una técnica ayuda a un desarrollador a detectar la mayoría de los defectos (máxima efectividad) y a hacerlo más rápidamente (máxima eficiencia).

La efectividad se define como el porcentaje de fallas observadas y defectos aislados. La eficiencia se define como el número de fallas o defectos dividido por el tiempo en horas empleado en su detección (tasa de detección de fallas o defectos).

Se realizaron *dos* replicaciones del experimento extendido. La extensión consiste en el agregado del paso de aislar el defecto después de revelar y observar la falla.

Una falla es "revelada" si la salida del programa revela un comportamiento desviado de la especificación, y es "observada" si el sujeto observa el comportamiento desviado.

Entre las diferencias con el experimento de Basili y Selby se incluyen el paso adicional de aislar la falla con las hipótesis asociadas, el lenguaje de programación (C en vez de FORTRAN), los programas y defectos, y objetivos más específicos de cubrimiento para la técnica estructural (100 % de cubrimiento de bifurcaciones, condiciones múltiples, bucles, y operadores relacionales en vez de 100 % de cubrimiento de sentencias).

Las hipótesis del experimento son:

- (H1) Las técnicas difieren en su efectividad y eficiencia.
- (H2) Las técnicas difieren en la efectividad en el aislamiento de los defectos de los distintos tipos.
- (H3) Medidas de la motivación y habilidad de los sujetos predicen su efectividad y eficiencia.

Las técnicas que se estudian en el experimento son: el tipo de lectura de código utilizado por Basili y Selby, *stepwise abstraction*; partición en clases de equivalencia y análisis de valores límite (técnica funcional); y cubrimiento del 100 % de bifurcaciones, condiciones multiples, bucles, y operadores relacionales (técnica estructural).

En la lectura de código se provee a los sujetos del código fuente para que a partir del mismo construyan una especificación. Luego acceden a la especificación original del programa y la comparan con la que construyeron. En esta técnica, una inconsistencia (análogo a una falla en los otros tratamientos) es "revelada" si el sujeto captura el comportamiento desviado en su especificación, y es "observada" si el sujeto observa la inconsistencia entre las especificaciones. Múltiples fallas (inconsistencias) causadas por el mismo defecto se cuentan como una falla.

En la verificación estructural, los sujetos reciben el código fuente del programa impreso, pero no acceden a la especificación. A partir del código intentan construir casos de prueba que alcancen el 100 % de cubrimiento de saltos, condiciones múltiples, bucles y operadores relacionales. Utilizan una versión instrumentada del programa para ejecutar sus casos de prueba y ver los informes de los valores de cobertura alcanzados. Para la instrumentación del programa y los

informes se utiliza la herramienta GCT (*Generic Coverage Tool*). Los sujetos desarrollan casos de prueba adicionales hasta alcanzar el 100 % de cubrimiento, o hasta que creen que no alcanzarán un cubrimiento mejor. Una vez ejecutados los casos de prueba los sujetos imprimen los resultados obtenidos, apagaban la computadora, y reciben una copia de la especificación del programa. Utilizan la especificación para observar fallas en los resultados obtenidos.

En la verificación funcional los sujetos reciben la especificación del programa, pero no ven el código fuente. Identifican casos de prueba a partir de las clases de equivalencia, poniendo atención en los valores límite. Luego ejecutan sus casos de prueba en la computadora. Previo a la realización de la verificación funcional se indica a los sujetos que no deben generar casos de prueba adicionales una vez que han comenzado a ejecutar, pero los autores no pueden ni prevenir que lo hagan ni medir en que medida lo han hecho. La ejecución finaliza cuando los sujetos imprimen los resultados obtenidos y apagan la computadora. Llegado este punto, los sujetos utilizan la especificación para observar las fallas que fueron reveladas en los resultados obtenidos.

Al terminar la ejecución de cada técnica, los sujetos aislan los defectos que causan las fallas observadas. No se provee de ninguna técnica para el aislamiento de defectos. Finalmente, los sujetos generan una lista de fallas observadas y defectos encontrados.

Para todas las técnicas, un defecto es "aislado" si el sujeto describe el problema en el código con suficiente precisión. Los autores distinguen entre los defectos encontrados utilizando la técnica (falla revelada y observada) y los defectos encontrados por casualidad (no hay falla revelada u observada).

Se utilizan seis programas, tres para entrenamiento y tres para el experimento en sí. Los programas para el entrenamiento son: "count", cuenta palabras ingresadas desde la entrada estándar; "series", genera series de números; y "tokens", ordena *tokens* alfanuméricos. Su tamaño es aproximadamente la mitad del de los programas utilizados para el experimento. Para hacer el período de entrenamiento simple, todos los sujetos aplican lectura de código al programa "count", la técnica funcional al programa "series" y la técnica estructural al programa "tokens".

Los programas utilizados en el experimento real son:

- ntree.- implementa un tipo abstracto de datos, a saber, una especie de árbol.
- cmdline.- evalúa un número de opciones suministradas en línea de comandos.
- nametbl.- implementa otro tipo abstracto de datos, algo así como una tabla de símbolos simple.

Cada programa se compone de un único archivo que contiene un conjunto de funciones en C, y están documentados con un archivo *header* (.h) de aproximadamente 30 líneas. Las funciones tienen aproximadamente entre 10 y 30 líneas.

Los programas tienen algunos defectos originales de su construcción (identificados por los desarrolladores), pero la mayoría fueron sembrados a posteriori. Todos los defectos causan fallas observables y ningún defecto oculta a otro. Los defectos se clasifican según el doble esquema utilizado por Basili y Selby:

- Según la ausencia de código necesario o la presencia de código incorrecto: omisión o comisión.
- Según la clase de defecto: inicialización, cálculo, control, interfaz, datos o cosmético.

Los defectos difieren en cantidad y tipo a través de los programas. Hay al menos un representante de cada tipo de defecto en el experimento (no en cada programa).

Los sujetos son estudiantes del curso de laboratorio "Software Engineering I" de la Universidad de Kaiserslautern y han recibido clases sobre las técnicas de verificación. Todos se encuentran en su tercer o cuarto año de estudios y poseen un manejo pobre del lenguaje C. Dado que el experimento se ejecuta por sujetos inexperientes, no es segura la generalización de los resultados.

El diseño utilizado para el experimento es factorial-fraccionario (*fractional-factorial*), pues un sujeto no aplica todas las técnicas a todos los programas. El diseño se resume en el cuadro 5.3.

Programa y Día	Lectura de Código	Técnica Funcional	Técnica Estructural
pgm. 1 (día 1)	grupos 1, 2	grupos 3, 4	grupos 5, 6
pgm. 2 (día 2)	grupos 3, 5	grupos 1, 6	grupos 2, 4
pgm. 3 (día 3)	grupos 4, 6	grupos 2, 5	grupos 1, 3

Cuadro 5.3: Resumen del diseño experimental

Las variables independientes son la técnica de verificación, el programa y el orden en el que se aplican las técnicas. Los sujetos son una variable independiente no controlada. Las variables dependientes son el número de fallas y defectos detectados, así como el tiempo empleado en aplicar las técnicas. Los sujetos aplican tres técnicas de verificación sobre tres programas en seis ordenes diferentes. Todos los sujetos ven el mismo programa en el mismo día para evitar trampas. Son asignados aleatoriamente a uno de seis grupos. La pertenencia a un grupo asigna las combinaciones de programa-técnica a ser ejecutadas, así como el orden en el que se aplican las técnicas.

La permutación de programas y técnicas a través de los grupos asegura que cada sujeto usa cada técnica, que ocurren todas las combinaciones de programa-técnica, y que se ejecutan todos los ordenes posibles en la aplicación de las técnicas.

Respecto al análisis, las hipótesis se evalúan estudiando los factores: efectividad, tiempo, eficiencia, efectividad relacionada con los tipos de defecto, y motivación y habilidad de los sujetos.

Entre los resultados principales del estudio se observa que, en general, los verificadores que utilizaron la técnica funcional *observaron* más *eficientemente* las fallas, emplearon mucho más tiempo adicional en *aislar* los defectos, y aún así, en general, fueron más *eficientes* en *aislar* defectos. Los sujetos que aplicaron lectura de código emplearon mucho más tiempo en identificar inconsistencias pero *aislaron* los defectos en poco tiempo adicional.

Las técnicas no difirieron significativamente respecto a la *efectividad*. La *efectividad en observar* defectos se vio afectada por el programa, así como por

los sujetos.

La *efectividad en aislar* los defectos, así como el *tiempo* empleado en la *detección* y en el *aislamiento* de defectos dependen del sujeto.

Las técnicas difirieron respecto a la *efectividad por tipo de defecto*: en la replicación uno se observó que la lectura de código fue significativamente peor en ayudar a los sujetos a *observar* fallas causadas por defectos de omisión o de control, así como que los sujetos que la aplicaron fueron los más efectivos al *aislar* defectos de interfaz y de datos, posiblemente porque examinan las estructuras de datos y las interfaces directamente en el código. Este resultado no se repitió en la segunda replicación. De la misma forma, en la segunda replicación la técnica funcional fue superior en *aislar* defectos cosméticos, pero esto no ocurrió en la primer replicación.

Los resultados de las dos replicaciones confirman fuertemente la hipótesis H1 y débilmente la hipótesis H2, pero no aportan información para confirmar la hipótesis H3: las medidas realizadas por los propios sujetos sobre su motivación y habilidad no permiten predecir valores de efectividad y eficiencia confiables.

Los resultados del experimento sugieren que para sujetos sin experiencia en el lenguaje de programación utilizado y las técnicas de verificación empleadas, cualquiera de las técnicas tendrá la misma *efectividad* al detectar defectos. Respecto a la *eficiencia*, los resultados sugieren que se aplique la técnica de verificación funcional.

Las diferencias observadas *por tipo de defecto* respecto a la *efectividad* a través de las técnicas sugiere que una combinación de las mismas tendrá mayor desempeño que cualquier técnica aislada.

5.3. M. Wood, M. Roper, A. Brooks y J. Miller. 1997

Afirmaciones como la siguiente: "la detección de los defectos debe basarse en técnicas dinámicas y estáticas, así como diferentes técnicas deberían usarse en combinación pues cada una encontrará diferentes defectos", han sido reforzadas por experimentos como el de Basili y Selby en 1987 [BS87] y el de Kamsties y Lott en 1995 [KL95]. En 1997 Wood, Roper, Brooks y Miller [WRBM97] realizan un experimento con la intención de investigar estas conclusiones.

El experimento es una replicación del llevado a cabo por Kamsties y Lott en 1995. Los programas utilizados en el entrenamiento de los sujetos y en el experimento, así como los defectos, son los proveídos por el paquete de laboratorio producido por Kamsties y Lott.

Las técnicas de verificación son las mismas usadas por Kamsties y Lott, salvo el criterio de la técnica estructural, que pasa de cubrimiento del 100 % de saltos, condiciones múltiples, bucles y operadores relacionales, a sólo cubrimiento del 100 % de bifurcaciones. Wood *et al.* consideran poco aplicable el criterio de cubrimiento utilizado por Kamsties y Lott. El procedimiento para aplicar las técnicas también es el mismo: primero se observan las fallas y luego se aislan en el código los defectos que las provocan. En el caso de la técnica estructural, se utiliza la misma herramienta para reportar los valores de cubrimiento alcanzados (CGT).

Los sujetos son estudiantes con altas calificaciones de un curso de Ingeniería de Software práctico, en la Universidad de Strathclyde. Todos ellos han completado 2 años de estudio, incluyendo cursos de programación en C. El experimento fue presentado como parte del curso y los estudiantes eran conscientes de que su trabajo iba a ser usado para propósitos experimentales. Los estudiantes se dividieron en seis grupos, cada grupo fue balanceado en términos de la habilidad de los estudiantes, donde la habilidad fue medida por el desempeño en cursos de programación anteriores. Previo al experimento, los estudiantes recibieron clases sobre cada técnica de verificación junto con tres sesiones de 2 horas de entrenamiento supervisado, para practicar la ejecución de cada técnica. El experimento fue organizado en tres sesiones de tres horas (un programa por sesión), realizándose una sesión por semana durante tres semanas sucesivas. Los sujetos fueron organizados bajo condiciones de examen, prohibiendo toda cooperación entre ellos.

Todo el trabajo realizado por los sujetos fue registrado en fichas de datos especialmente diseñadas para ello. También se registró toda interacción con la computadora utilizando un entorno de guías y se usaron cuestionarios para conocer los puntos de vista de los sujetos respecto a todo el proceso experimental.

El experimento combina tres técnicas, tres programas y seis grupos de sujetos. Sujetos en el mismo grupo prueban las mismas combinaciones de técnica-programa el mismo día. Cada sujeto realiza tres verificaciones, sin repetir programa ni técnica. El cuadro 5.4 muestra la asignación de los grupos a las combinaciones técnica-programa.

Las variables independientes del experimento son la técnica de verificación y el tipo de software. Las variables dependientes examinadas son el número de fallas observadas, el número de defectos detectados, el tiempo empleado en observar fallas y el tiempo empleado en aislar los defectos.

	Lectura de Código			Técnica Funcional			Técnica Estructural		
	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3
Grupo 1	X	–	–	–	–	X	–	X	–
Grupo 2	–	X	–	X	–	–	–	–	X
Grupo 3	–	–	X	–	X	–	X	–	–
Grupo 4	X	–	–	–	X	–	–	–	X
Grupo 5	–	X	–	–	–	X	X	–	–
Grupo 6	–	–	X	X	–	–	–	X	–

Cuadro 5.4: Asignación de grupos a pares técnica-programa

Muchos sujetos fallaron al completar la parte de aislamiento de defectos, por lo que el análisis de estos datos se vio mermado. Los únicos datos considerados son los relacionados con la observación de fallas.

Entre otras cosas se observó que el desempeño de las técnicas varía de defecto a defecto y de programa a programa, o sea, que el programa (y por lo tanto los defectos) impactan sobre la efectividad de las técnicas. El desempeño se mide en términos de cuantos sujetos, usando cada técnica, observan cada falla.

Se observó también un gran sensibilidad de las técnicas a los distintos defectos en los programas. De esto, parece ser que prescindir de una de las técnica de verificación puede tener efectos adversos. Los autores proponen para trabajos futuros la creación de una clasificación de defectos basada en la sensibilidad de las técnicas a los distintos defectos.

Como las técnicas parecen encontrar distintos defectos, se explora la efectividad relativa de equipos hipotéticos de verificadores usando combinaciones de cada técnica. Las permutaciones investigadas son:

- Pares de lectores de código
- Pares de verificadores funcionales
- Pares de verificadores estructurales
- Un lector de código y un verificador funcional
- Un lector de código y un verificador estructural
- Un verificador estructural y un verificador funcional
- Tripletas de lectores de código
- Tripletas de verificadores funcionales
- Tripletas de verificadores estructurales
- Un lector de código, un verificador funcional y un verificador estructural

De este estudio se observa que a través de los tres programas, hay una tendencia general de mejora sustancial al moverse de individuos a pares, y de pares a tripletas. El desempeño de las mismas combinaciones de técnicas varía sustancialmente de programa a programa. Aunque la lectura de código es relativamente

pobre como técnica individual, es generalmente efectiva cuando se combina con otros lectores de código o con otras técnicas. Consistentemente con lo anterior, los mejores resultados son obtenidos combinando diferentes técnicas (independientemente del programa), p. ej. combinando lectura de código y la técnica funcional, combinando lectura de código y la técnica estructural, combinando la técnica funcional y la estructural, y lo mejor de todo, el empleo de las tres técnicas en conjunto. Esto a su vez es consistente con que la efectividad de las técnicas varíe dependiendo de la naturaleza de los defectos.

No se logró comprobar con la fuerza suficiente la percepción de que las técnicas descubren ciertas clases de defectos. Si este fuera el caso, entonces combinar dos verificadores de una misma técnica no mostraría ninguna mejora. En cambio, esta combinación muestra una mejora sustancial en todos los casos. Esto demuestra que la sensibilidad de cada técnica ante determinados defectos no es fidedigna. Es decir, hay una variabilidad en el uso de las técnicas que determina su efectividad hacia un defecto particular. Sujetos aplicando la misma técnica generalmente no encontraron los mismos defectos. Aunque parece que las técnicas son sensibles a la naturaleza de los defectos, existe el problema fundamental del componente humano, o sea, la variabilidad de los sujetos en la aplicación de las técnicas.

En conclusión, las diferentes técnicas tienen diferentes fortalezas y debilidades en términos de los defectos que ayudan a encontrar. Su efectividad absoluta así como su efectividad relativa dependen de la naturaleza de los programas y más específicamente de la naturaleza de los defectos en los programas.

5.4. Natalia Juristo y Sira Vegas. 2003

Bajo el supuesto de que la efectividad relativa de las técnicas de verificación depende del programa y del tipo de defecto, Juristo y Vegas realizan en 2003 un estudio [JV03] para comparar la efectividad relativa de distintas técnicas de verificación y para intentar relacionar las técnicas de verificación con los tipos de defecto detectados.

Las hipótesis generales del estudio son:

- H_0 : La efectividad de las técnicas es independiente del tipo de defecto.
- H_1 : La efectividad de las técnicas es dependiente del tipo de defecto.

H_1 se descompone en $i \times j$ subhipótesis diferentes de la forma: *la técnica* t_i *es la más efectiva para defectos del tipo* f_j.

El estudio se compone de dos fases, o experimentos, llamados Experimento I y Experimento II, diseñados diferente para estudiar distintos factores. Ambos se realizaron en la Universidad Politécnica de Madrid, el primero en 2001 y el segundo en 2002. El primer experimento permitió refinar las hipótesis para la ejecución del segundo experimento, mediante la discusión de los resultados obtenidos.

El primer experimento se basa en los hallazgos del experimento realizado por Wood *et al.* [WRBM97], quienes concluyeron que la relación entre técnicas de verificación y tipos de defecto debía ser investigada. El diseño del primer experimento incluye cuatro programas, cuatro tipos de defecto por programa (contienen dos defectos de tres de los tipos, y un defecto del cuarto tipo, totalizando 9 defectos por programa) y cada sujeto aplica sólo una técnica.

La variable dependiente es la efectividad, la cual es medida en términos del número de sujetos que detectan un defecto determinado, para cada defecto en un programa. Las técnicas, los tipos de defecto y los programas son los factores del estudio (variables independientes). Se quiere determinar que impacto tienen sobre la efectividad.

Las hipótesis para el Experimento I son:

- H_{01} : La técnica de verificación no tiene impacto sobre el número de defectos detectados.
- H_{11} : La técnica de verificación impacta sobre el número de defectos detectados.
- H_{02} : El tipo de defecto no tiene impacto sobre el número de defectos detectados.
- H_{12} : El tipo de defecto impacta sobre el número de defectos detectados.
- H_{03} : El uso de diferentes técnicas de verificación sobre tipos de defecto distintos no tiene impacto sobre el número de defectos detectados.
- H_{13} : El uso de diferentes técnicas de verificación sobre tipos de defecto distintos impacta sobre el número de defectos detectados.
- H_{04} : El uso de diferentes técnicas de verificación sobre programas distintos no tiene impacto sobre el número de defectos detectados.

- H_{14} : El uso de diferentes técnicas de verificación sobre programas distintos impacta sobre el número de defectos detectados.

- H_{05} : Tipos de defecto diferentes en programas distintos no tienen impacto sobre el número de defectos detectados.

- H_{15} : Tipos de defecto diferentes en programas distintos impactan sobre el número de defectos detectados.

Las técnicas utilizadas son las mismas que en el experimento de Wood *et al.*: para la verificación funcional, clases de equivalencia y valores límite (TF); para la verificación estructural, criterio de cubrimiento de decisión/condición (TE); y para la verificación estática, el tipo de lectura de código *stepwise abstraction* (LC). El procedimiento para ejecutarlas es básicamente el mismo salvo algunos aspectos. En la técnica de verificación estructural los sujetos no utilizan ninguna herramienta para asegurar el cubrimiento de los saltos, pues las autoras quieren comparar las técnicas bajo las mismas condiciones. Esto afecta el tiempo que les lleva a los sujetos generar los casos de prueba, no la calidad de la tarea de verificación, dado que los programas son lo suficientemente simples como para que los sujetos sean capaces de generar los casos de prueba sin necesidad de una herramienta. De todas formas, no se tiene en cuenta el tiempo de generación de casos de prueba por parte de los sujetos, pues el experimento no examina esta variable dependiente. Las autoras no consideran que el aislamiento del defecto sea necesario para la comparación de las técnicas, dado que las técnicas no proveen ninguna ayuda para esta tarea.

Se utilizan cuatro programas distintos, programados en C, que se clasifican en dos tipos: programas que implementan tipos de datos y programas que implementan funciones. Tres de estos programas están incluidos en el paquete de laboratorio de Kamsties y Lott, y el cuarto fue desarrollado por las autoras:

- Nametbl (datos): Implementa la estructura de datos de una tabla simbólica y sus operaciones.

- Ntree (datos): Implementa la estructura de datos de un árbol n-ario y sus operaciones.

- Cmdline (funcional): Lee una línea de entrada y retorna un resumen.

- Trade (funcional): Lee un archivo de transacciones de negocios y retorna estadísticas acerca de las transacciones que contiene.

El factor *programa* tiene cuatro niveles, uno por cada programa usado. El tamaño promedio de los programas es aproximadamente de 200 líneas de código, excluyendo lineas en blanco y comentarios.

Respecto a los defectos, se considera que todos los defectos causan fallas observables y, por lo tanto, ninguno cubre a otro. Cada programa contiene la misma cantidad de defectos de cada tipo. La clasificación de defectos es un subconjunto de la utilizada por Basili:

- Inicialización (comisión/omisión): es una inicialización incorrecta o faltante de una estructura de datos.

- Control (comisión/omisión): el programa sigue un camino incorrecto del flujo de control, ya sea porque hay un predicado erróneo, o porque falta uno.
- Cosmético (comisión/omisión): por ejemplo, un error al escribir un mensaje de error o un mensaje de error que no aparece y debería hacerlo.

Todos los tipos de defecto son utilizados en el experimento. El factor tipo de defecto tiene 9 niveles: un defecto cosmético por omisión, un defecto cosmético por comisión, un defecto de inicialización por omisión, dos defectos de inicialización por comisión, dos defectos de control por omisión, y dos defectos de control por comisión. Se duplican tres tipos de defecto (se usan dos defectos de esos tipos) con el objetivo de aumentar la fiabilidad de los resultados del experimento.

Los sujetos son estudiantes de quinto año de la Escuela de Informática, Universidad Politécnica de Madrid, programa 1983. Están familiarizados con las técnicas, porque tomaron un curso relacionado en su cuarto año, pero su conocimiento práctico es muy escaso. Durante el experimento se les pidió que llenaran una hoja de auto-evaluación con respecto a su conocimiento del lenguaje de programación, etc.

Las verificaciones se realizan de a grupos de sujetos. Cada grupo representa un conjunto de personas que realizan el experimento individualmente, al mismo tiempo, sobre el mismo programa, aplicando la misma técnica. Cada combinación de programa-técnica se ejecuta tantas veces como sujetos hay en el grupo al que corresponde, y tantas veces como la cantidad de experimentos que la utilicen (en este caso dos: todas las combinaciones se utilizan en los dos experimentos). La asignación de los grupos a los días, programas y técnicas se realizó de forma aleatoria.

El diseño resultante es un diseño de tres factores con replicación, visible en el cuadro 5.5. Los 8 grupos que ejecutan las técnicas dinámicas se componen de 12 sujetos. Los 4 grupos que ejecutan lectura de código se componen de 25 sujetos.

	Funcional						Datos					
	Programa 1			Programa 2			Programa 3			Programa 4		
	LC	TE	TF	LC	TE	TF	LC	TE	TF	LC	TE	TF
Grupo 1	–	–	–	–	–	–	–	–	X	–	–	–
Grupo 2	–	–	–	–	–	X	–	–	–	–	–	–
Grupo 3	–	–	–	–	–	–	X	–	–	–	–	–
Grupo 4	–	–	–	–	X	–	–	–	–	–	–	–
Grupo 5	–	–	–	–	–	–	–	–	–	–	–	X
Grupo 6	–	–	–	–	–	–	–	–	–	X	–	–
Grupo 7	–	–	X	–	–	–	–	–	–	–	–	–
Grupo 8	–	–	–	X	–	–	–	–	–	–	–	–
Grupo 9	–	–	–	–	–	–	–	–	–	–	X	–
Grupo 10	–	–	–	–	–	–	–	X	–	–	–	–
Grupo 11	–	X	–	–	–	–	–	–	–	–	–	–
Grupo 12	X	–	–	–	–	–	–	–	–	–	–	–

Cuadro 5.5: Diseño del experimento

El experimento consta de cinco sesiones diferentes, realizadas cada una en

un día distinto. En la primer sesión se explicaron las razones de la ejecución del experimento, y se entregó la documentación correspondiente. En cada una de las sesiones siguientes se verifica un programa distinto. Los estudiantes son consientes desde el principio de que están participando en un experimento y que sus resultados serán utilizados para calificarlos. Se califican según dos parámetros: forma de ejecución de la técnica y cantidad de defectos detectados. En la primer sesión se pide a los estudiantes que estudien la documentación y que entreguen un ejercicio completo que incluye la aplicación de las tres técnicas, obligándolos a asimilar los conceptos explicados. Los estudiantes no tienen conocimiento sobre que técnica les fue asignada o que programa verificarán hasta que comienza el experimento real.

Los defectos y tipos de programa se seleccionaron con la finalidad de que fueran representativos de la realidad, o sea, que simulen los defectos que los programadores cometen al codificar.

Del análisis de datos del primer experimento se obtiene que el número de personas que descubre un defecto depende del programa que se está verificando, la técnica que se está usando y el defecto en cuestión. Hay defectos que se descubren más fácilmente en ciertos programas y defectos que se descubren más fácilmente usando ciertas técnicas. Por lo tanto, la detección de un defecto está influenciada por: la combinación defecto/técnica y la combinación defecto/programa. Los principales hallazgos respecto a la interacción defecto/técnica son:

1. Los defectos cosméticos son los más difíciles de visualizar, no influye sobre ellos la técnica.

2. El tipo de defecto no impacta sobre la lectura de código (se detectan todos los defectos por igual).

3. La técnica funcional se comporta mejor que la técnica estructural para defectos de omisión, y la técnica estructural se comporta igual o mejor que la técnica funcional para defectos de comisión, aunque, en general, la técnica funcional tiende a comportarse mejor. Sin embargo, no se logra distinguir que tipo de defecto detecta mejor cada una de las dos técnicas.

Los principales hallazgos respecto a la interacción defecto/programa son:

1. Los defectos cosméticos son los más difíciles de visualizar, no influye sobre ellos el programa (al igual que la técnica).

2. La clasificación de los programas en *funcionales* y de *datos* no parece ser significativa, pues no se puede establecer ningún patrón de comportamiento para los defectos.

Los resultados del Experimento I respecto a la combinación defecto/técnica sentaron las bases para el Experimento II, los resultados respecto a la combinación defecto/programa se dejan para futuras investigaciones.

Bases para el Experimento II:

- Sólo en casos excepcionales defectos del mismo tipo se comportaron igual (tanto para programas como para técnicas), lo que hace sospechar a las autoras que quizás el esquema de clasificación de defectos no es apropiado, pues no permite detectar los tipos de defecto para los que las técnicas

funcional y estructural son más adecuadas. Teniendo en cuenta que no todos los defectos se replicaron, se crean dos versiones de cada programa insertándoles diferentes defectos, aunque del mismo tipo.

- A partir del Experimento I no es posible saber si un defecto no fue detectado porque la técnica no generó un caso de prueba que lo evidenciara o porque el sujeto no fue capaz de ver la falla que provoca el defecto. En el Experimento II se examina la capacidad de detección de la técnica y la visibilidad de la falla. Para este propósito se generaron casos de prueba que detectan todos los defectos. Los sujetos utilizan estos casos en vez de los propios. Los generados por ellos se utilizan después para evaluar la capacidad de detección de la técnica.

- Como parece que los tipos de defecto no tienen relación con la técnica estática, en el Experimento II se investiga si la visualización de los defectos depende de su ubicación en el código.

- También surge la duda de si la aleatoriedad de los grupos funciona bien. Si no lo hiciera, los sujetos más capaces podrían estar nucleados en un grupo, por lo que en el Experimento II todos los sujetos aplican todas las técnicas en vez de sólo una.

El objetivo del segundo experimento es investigar:

- Influencia de la visibilidad del defecto: para saber que tan visibles son las fallas causadas por un defecto.

- Influencia de la técnica y del tipo de defecto: para reforzar los resultados obtenidos en el primer experimento sobre la influencia de la técnica de verificación en la detección de defectos.

- Influencia de la ubicación del defecto: en un intento de encontrar alguna característica de los defectos que influencie la efectividad de la lectura de código.

Las hipótesis del Experimento II son las del Experimento I más las siguientes:

- H_{06} : La visibilidad de las fallas generadas por los defectos no tiene impacto sobre la efectividad.

- H_{16} : La visibilidad de las fallas generadas por los defectos impacta sobre la efectividad.

- H_{07} : La posición de los defectos no tiene impacto sobre la efectividad.

- H_{17} : La posición de los defectos impacta sobre la efectividad.

Las variables dependientes son las mismas que para el Experimento I.

Los factores de este experimento son los mismos que para el experimento anterior: técnica, tipo de defecto y programa, más uno nuevo, la versión del programa.

Para este experimento cambia el procedimiento para aplicar las técnicas: se separa la generación de casos de prueba y la detección de defectos. Primero, los sujetos aplican la técnica para generar los casos de prueba. Estos casos se

utilizan luego en el análisis para determinar que defectos detecta cada técnica. Segundo, los sujetos ejecutan los casos de prueba generados por las autoras, que detectan todos los defectos.

Respecto a los defectos, se utilizan los mismos tipos que para el experimento anterior, más el tipo *Cálculo (comisión)*. Cada programa incluye 7 defectos, uno de cada tipo de defecto, a diferencia del Experimento I, en el que había más de un defecto de un mismo tipo por programa. La incorporación de las versiones permite replicar los defectos de todos los tipos. El tamaño de los programas no es suficiente para replicar todos los defectos sin que se enmascaren entre si. Dos versiones del mismo programa sólo difieren en los defectos que contienen. Los defectos fueron introducidos con la finalidad de que defectos del mismo tipo provocaran las mismas fallas.

Para balancear el diseño, se descarta el programa introducido por las autoras, utilizándose los tres programas genuinos del paquete de laboratorio del experimento original. La variabilidad en la habilidad de los sujetos se bloquea haciendo que todos apliquen todas las técnicas, conduciendo al diseño experimental mostrado en el cuadro 5.6. Cada grupo se compone de entre 7 y 8 sujetos. Los sujetos que participan en el Experimento II no han participado ni tenido conocimiento del Experimento I.

	Cmdline			Ntree			Nametbl		
	LC	TE	TF	LC	TE	TF	LC	TE	TF
Grupo 1	X	–	–	–	X	–	–	–	X
Grupo 2	X	–	–	–	–	X	–	X	–
Grupo 3	–	X	–	–	–	X	X	–	–
Grupo 4	–	X	–	X	–	–	–	–	X
Grupo 5	–	–	X	X	–	–	–	X	–
Grupo 6	–	–	X	–	X	–	X	–	–

Cuadro 5.6: Diseño del experimento

En este experimento, la efectividad se define como el número de estudiantes que generan un caso de prueba capaz de detectar el defecto.

Como resultado general se obtiene que el número de personas que generaron un caso de prueba que detecta un defecto depende de la versión, la técnica usada y el defecto en cuestión. Hay defectos que se detectan más fácilmente para ciertos programas, defectos que se detectan mejor usando ciertas técnicas y programas que se comportan mejor para ciertas técnicas.

Los hallazgos del Experimento II corroboraron algunas de las sospechas del Experimento I.

1. La visibilidad de las fallas influye en la efectividad, y depende del defecto que la provoca, el programa y la versión en la que ocurre. A partir de este resultado se logró establecer una taxonomía de fallas, donde los mensajes de error que no aparecen y resultados incorrectos son los más visibles.

2. Las técnicas funcional y estructural se comportaron de la misma forma, y la lectura de código peor que ambas. Esto refuta el hallazgo en el Experimento I, sobre que la técnica funcional se comporta mejor que la técnica estructural para algunos tipos de defecto. Esto se debe a que el tipo de

falla fue un efecto escondido en el Experimento I. Este efecto impactaba en la detección de las fallas, y por lo tanto, en la de los defectos. Queda la interrogante de si, por su modus operandi, los sujetos que ejecutan la técnica funcional son más sensibles a la detección de fallas, y por eso la diferencia en el Experimento I.

3. La interacción programa/técnica impacta sobre el número de defectos descubiertos, aunque las técnicas funcionales y estructurales se comportan igual para el mismo programa. Por lo tanto, independientemente de la técnica (estructural o funcional) que se aplique, siempre será menos efectiva para un tipo de programa que para otro.

4. La posición de un defecto en el código no tiene ninguna influencia sobre el número de sujetos que lo detectan usando lectura de código.

5. La versión influye sobre el número de sujetos que generan un caso de prueba que detecta un defecto. De esto se entiende que es el defecto en cuestión, y no el tipo de defecto, más que el tipo o forma del programa, que determina los defectos que se descubren. Esto reafirma la necesidad de una clasificación de defectos sensible a las técnicas, que parece no ser la utilizada.

Parte II

Experimento

Capítulo 6

Definición, Planificación y Operación

El objetivo de nuestro experimento es estudiar distintas técnicas de verificación, con el propósito de evaluar la efectividad y costo que presentan ante distintos tipos de defectos.

El experimento es llevado a cabo por los estudiantes que realizan un Módulo de Taller, esto es un curso-taller de la carrera de Ingeniería en Computación de la Facultad de Ingeniería de la Universidad de la República. De aquí se puede observar que el experimento se realiza a nivel académico y no industrial. En general los estudiantes se encuentran cursando entre el cuarto y quinto año de la carrera. Este experimento tiene como trasfondo un problema real: qué técnicas utilizar para atacar determinados tipos de defectos. El hecho de contextualizar el experimento en un curso de grado permite la replicación del mismo a posteriori.

Las técnicas estudiadas son las presentadas en el Capítulo 3 (Técnicas de Verificación) de la Parte I:

- Inspecciones
- Criterio de Cubrimiento de Condición Múltiple
- Trayectorias Linealmente Independientes
- Partición de Equivalencia con Análisis de Valores Límite
- Tablas de Decisión

6.1. Hipótesis

Una de las variables bajo estudio es la efectividad de las técnicas de verificación. La *efectividad* se define en función de la cantidad de defectos detectados, cuanto mayor sea la cantidad de defectos que se detecte, más efectiva será la técnica.

Las hipótesis a continuación se plantean en dos marcos, uno distinguiendo la efectividad por tipo de defecto (hipótesis 2), y otro sin realizar esta distinción (hipótesis 1):

- H_{01} : Las técnicas tienen la misma *efectividad*.
- H_{11} : Las técnicas tienen distinta *efectividad*.

  H_{11} se descompone en $C(n,2)$[1] hipótesis de la forma: la técnica t_i es más *efectiva* que la técnica t_j, donde n es igual a la cantidad de técnicas de verificación utilizadas en el estudio y $1 \leq i < j \leq n$.
- H_{02} : Las técnicas tienen la misma *efectividad* respecto a los tipos de defecto.
- H_{12} : Las técnicas tienen distinta *efectividad* respecto a los tipos de defecto.

  H_{12} se descompone en $m \times C(n,2)$ hipótesis de la forma: la técnica t_i es más *efectiva* que la técnica t_j para el tipo de defecto d_k, donde n y m son la cantidad de técnicas de verificación y de tipos de defecto respectivamente, $1 \leq i < j \leq n$, y $1 \leq k \leq m$.

La efectividad se mide como sigue, según se distinga o no por tipo de defecto:

1. Sin distinguir por tipo de defecto:
   - *Efectividad*.- se calcula como el porcentaje de defectos detectados (cantidad de defectos detectados dividido la cantidad total de defectos).
2. Distinguiendo por tipo de defecto:
   - *Efectividad*.- se calcula como el porcentaje de defectos detectados de ese tipo de defecto.

Se utiliza como medida de efectividad el *porcentaje* de defectos y no la *cantidad* porque los programas sobre los que se aplican las técnicas poseen distinta cantidad de defectos. Medir la efectividad mediante porcentaje permite la comparación de los datos.

6.2. Selección de sujetos, programas y defectos

Los sujetos que participan del experimento son un total de 14 estudiantes de cuarto y quinto año de la carrera de Ingeniería en Computación de la Facultad de Ingeniería de la Universidad de la República, que se encuentran cursando un Módulo de Taller cuya finalidad es la realización del experimento. Todos han realizado el curso Introducción a la Ingeniería de Software, en el que vieron una introducción a la verificación de software. Esta introducción incluye una presentación breve de las técnicas y de los procedimientos de verificación utilizados en este experimento. Se considera que poseen niveles similares de experiencia respecto a la verificación de software.

En el experimento se utilizan cuatro programas, considerados representativos de distintos tipos de programas reales. Los programas fueron construidos exclusivamente para el experimento, por estudiantes de otro Módulo de Taller, y fueron suministrados a los sujetos para su verificación. Los programas debían

[1] Número de combinaciones de n elementos tomados de a 2. Se calcula como $\frac{n!}{2!(n-2!)}$.

ser desarrollados sin ejecutar ningún tipo de prueba, tanto durante como una vez finalizada su construcción, siguiendo un estándar de codificación, que puede consultarse en el documento *Estándar de Codificación Java*. El único requisito era que compilaran, por lo tanto, los programas no tienen defectos sintácticos, pues estos son detectados por el compilador. Todos los programas fueron construidos en el lenguaje de alto nivel Java, con el que los sujetos están familiarizados, ya que lo han usado en varios cursos de la carrera, y documentados con JavaDoc. Los cuatro programas son:

- Contabilidad.- Sistema de liquidación de sueldos. Permite registrar en base de datos información de empleados y realizar operaciones con los sueldos, como ser aumento y liquidación.
- Matemático.- Realiza cálculos matemáticos de relativa complejidad, relacionados con la correlación entre dos conjuntos de valores.
- MO-Latex.- Genera exámenes múltiple opción a partir de preguntas que extrae de una base de datos de forma aleatoria. Las posibles respuestas a una pregunta, así como cuál es la correcta, también se extraen de la base de datos.
- Parser.- Es un procesador de texto. Recibe un archivo con código Pascal y realiza chequeos estructurales sobre el mismo, generando un documento xml que muestra las estructuras reconocidas.

Las características generales de los programas se presentan en el Capítulo 9 de la Parte II.

Todos los defectos en el experimento fueron introducidos en la construcción de los programas por sus desarrolladores. Por lo tanto, no se puede asegurar que todas las fallas sean observables, y menos aún, que todos los defectos puedan ser detectados. Tampoco es posible conocer que defectos ocultan a otros defectos. Dada la falta de información, se asume que los defectos totales de cada programa serán iguales a la totalidad de defectos detectados por los sujetos en el experimento, más los detectados por este y otro proyecto de grado.

Para cubrir el hecho de que un defecto oculte a otro es que se permite que cada verificador solicite la modificación del programa que está verificando. La modificación tiene como objetivo remover este defecto y poder descubrir los que oculta. Debe solicitar la modificación y no hacerla él porque es posible que introduzca nuevos defectos en su corrección. La corrección realizada por nuestra parte no está 100 % libre de defectos, pero es más controlada y medida, por lo que hay menos probabilidades de que se introduzcan defectos.

Los tipos de defecto considerados provienen de la Clasificación Ortogonal de Defectos de IBM [Cla] y de la Taxonomía de Defectos de Beizer [Bei90].

De la Clasificación de IBM se utiliza la categoría *Closer Section* y, dentro de esta, las subcategorías *Defect Type* y *Qualifier*. La subcategoría *Tarjet* tiene el valor *Design/Code* para todos los defectos del experimento, y las subcategorías *Age* y *Source* no aplican, por la forma de construcción de los programas. Se considera que la combinación de un valor de la subcategoría *Defect Type* y un valor de la subcategoría *Qualifier* es un tipo de defecto. Como la clasificación presenta 7 posibles valores para la subcategoría *Defect Type* y 3 posibles valores para la subcategoría *Qualifier*, se tienen un total de 21 tipos de defecto.

En el caso de la Taxonomía de Beizer se utilizan las categorías: *Functionality As Implemented (2XXX)*, *Structural Bugs (3XXX)*, *Data (4XXX)*, *Integration (6XXX)* y *Other Bugs, Unspecified (9XXX)*. El criterio de selección de estas categorías se centra en los defectos que clasifican en ellas y las características de los programas.

Dada la diferencia de profundidad en las distintas categorías de la Taxonomía de Beizer, llegando algunas hasta el nivel 2 y otras pasando el nivel 6, los defectos del experimento se clasifican sólo hasta el nivel 3, en caso de que exista este nivel para la categoría.

6.3. Diseño

El diseño experimental utilizado es de *un factor con más de dos tratamientos*. Este diseño distingue a través de las técnicas de verificación, pero no distingue los programas sobre los que se aplican, ni la experiencia u otra característica de los sujetos que las utilizan.

El diseño tiene una única variable independiente (*factor*): la técnica de verificación. Los tratamientos de la variable son: Inspecciones, Criterio de Cubrimiento de Condición Múltiple, Trayectorias Linealmente Independientes, Partición de Equivalencia con Análisis de Valores Límite y Tablas de Decisión.

Las variables dependientes examinadas son: número de defectos detectados, porcentaje de defectos detectados, tiempo total de detección de defectos y tasa de detección de defectos. El tiempo total de detección de defectos **no** incluye el tiempo empleado en aislar los defectos en el código. Está conformado por el tiempo empleado en el diseño de los casos de prueba (para las técnicas dinámicas) y el tiempo empleado en la ejecución de la técnica. Este último tiempo es la ejecución de los casos de prueba en caso de ser una técnica dinámica, o la ejecución de las revisiones en caso de ser la técnica de lectura de código (Inspecciones). En el caso de Inspecciones, el tiempo de diseño de casos de prueba es cero.

El cuadro 6.1 muestra el diseño del experimento. En el se observa que el Sujeto 1 verifica primero el programa Contabilidad con la técnica Inspecciones, segundo el programa Matemático con la técnica Criterio de Cubrimiento de Condición Múltiple y tercero el programa MO-Latex con la técnica Trayectorias Linealmente Independientes. El orden en el que cada sujeto ejecuta las combinaciones técnica-programa asignadas se representa con el número correspondiente en la tabla. Todos los sujetos verifican tres programas distintos y utilizan tres técnicas distintas.

Cada celda de la tabla representa un experimento unitario. Cada columna de la tabla representa un par técnica-programa. Todas las combinaciones posibles de técnica y programa se ejecutan en el experimento. Como se puede observar, cada combinación se ejecuta 2 veces. De aqui que cada técnica se utiliza 8 veces y que cada programa es verificado 10 veces, 2 veces por cada técnica. Balancear la cantidad de veces que se ejecuta cada técnica sobre cada programa permite bloquear el impacto de las diferencias entre los programas sobre los resultados.

La asignación de los sujetos a los pares técnica-programa se realizó de forma aleatoria (conformando los experimentos unitarios).

Primero se identificaron todas las posibles combinaciones de técnica y programa (representadas por las 20 columnas del cuadro 6.1).

	Contabilidad					Matemático					MO-Latex					Parser				
	Ins	CCCM	TLI	PE$_y$AVL	TD	Ins	CCCM	TLI	PE$_y$AVL	TD	Ins	CCCM	TLI	PE$_y$AVL	TD	Ins	CCCM	TLI	PE$_y$AVL	TD
Sujeto 1	1						2						3							
Sujeto 2					2	3													1	
Sujeto 3				3								1						2		
Sujeto 4										1	2						3			
Sujeto 5			1						2						3					
Sujeto 6		2						3								1				
Sujeto 7	3													1						2
Sujeto 8							1						2						3	
Sujeto 9					1	2						3								
Sujeto 10				2						3								1		
Sujeto 11			3								1						2			
Sujeto 12									1						2	3				
Sujeto 13		1						2						3						
Sujeto 14																				1

Cuadro 6.1: Diseño del Experimento

Luego se calculó cuantos experimentos unitarios eran necesarios para ejecutar dichas combinaciones dos veces, lo que resultó en 40 experimentos. Para asignar los experimentos unitarios a los sujetos se decidió agrupar los experimentos de a tres, tal que en un mismo grupo no se repitieran ni el programa ni la técnica. Un sujeto no puede verificar dos veces el mismo programa ni utilizar dos veces la misma técnica. Si lo hiciera ganaría conocimiento del programa y/o la técnica, generando un mejor escenario para la segunda utilización del programa y/o técnica, lo que anularía la validez de los datos.

A continuación se fijó el orden de ejecución de los experimentos unitarios dentro de cada grupo, de forma tal de balancear lo máximo posible el efecto de aprendizaje de los verificadores, que recaería sobre la *primer* técnica a ser ejecutada. Si se observa la tabla, todas las técnicas se ejecutan en primer lugar 3 veces (celdas con el número 1), salvo la técnica Trayectorias Linealmente Independientes, que sufre este efecto sólo 2 veces. Cada grupo de experimentos unitarios está representado en la tabla por una fila, y el orden de ejecución por el número correspondiente.

Una vez armados los grupos de experimentos unitarios y definido su orden de ejecución, se asignó mediante sorteo un sujeto a cada grupo. El grupo correspondiente al Sujeto 14 en el cuadro 6.1 tiene sólo una experimento unitario asignado, esto se debe a que agregar los dos experimentos faltantes desbalancearía la cantidad de veces que se ejecuta cada técnica, así como la cantidad de veces que se ejecuta sobre cada programa.

De esta forma se obtuvo un experimento balanceado y aleatorio.

6.4. Evaluación de la Validez

A continuación se estudian las amenazas a los cuatro aspectos de la validez: conclusión, interna, constructo y externa. En general no se encontraron amenazas significativas, salvo por aquellas que impiden que los resultados sean generalizados a nivel industrial.

6.4.1. Validez de la conclusión

Una de las posibles amenazas a la *validez de la conclusión* es la robustez de los test a aplicar para analizar los datos. Como fue mencionado en el Capítulo 2 (Ingeniería de Software Empírica), en caso de que los datos no cumplan con las condiciones para aplicar el test paramétrico ANOVA, se utilizará su homólogo no paramétrico Kruskal-Wallis. En caso de no poder aplicar este test, se buscaran otras alternativas no paramétricas, de manera tal de que las asunciones del test elegido no se vean afectadas por los datos.

Otra posible amenaza a la validez de la conclusión es la influencia de los sujetos en el cumplimiento del objetivo del experimento. Si bien pueden existir otros motivos que la alimenten, esta amenaza se ve disminuida por el hecho de que los sujetos no conocen cuál es el objetivo del experimento. Es más, desconocen la existencia del experimento que se encuentra por detrás del Módulo de Taller.

La confiabilidad en la aplicación de los tratamientos y la heterogeneidad de los sujetos también amenazan la validez de conclusión. La forma en que los

sujetos aplican las técnicas afecta directamente los resultados obtenidos. El experimento cuenta con una fase de entrenamiento que permite chequear que los sujetos apliquen en forma correcta las técnicas, y nivelar el conocimiento de los mismos. Esto, sumado a que los sujetos proseen un nivel mínimo común de conocimiento de verificación, parece cubrir la posible influencia de estas amenazas.

6.4.2. Validez interna

La difusión o la imitación de los tratamientos es una amenaza social a la validez interna. Esta amenaza refiere a los sujetos que obtienen información sobre los resultados obtenidos por otros sujetos y utilizan este conocimiento en sus resultados. También refiere a aquellos sujetos que imitan el comportamiento de otros sujetos.

Para combatir esta amenaza se solicitó a los sujetos que no compartieran información de ningún tipo. Se aclaró que la ejecución de las verificaciones del Módulo de Taller debían ser individuales, y que no se calificaría la cantidad de defectos detectados sino la calidad del proceso de verificación. A pesar de las medidas tomadas, no tenemos la certeza de que no hayan compartido información, dado que las verificaciones se realizaron de forma domiciliaria.

Otra posible amenaza reside en que los sujetos alteren o sesguen los resultados obtenidos en razón de sus intereses o gustos. Por ejemplo, la técnica de verificación que mas agrade a un sujeto será seguramente en la que emplee mayor esfuerzo y dedicación. Este efecto puede anularse si ocurre que la técnica que más agrada a un sujeto es la que menos le agrada a otro, o puede no anularse y afectar los resultados del experimento.

Si esta amenaza afectara nuestro experimento, como la muestra de los sujetos se considera representativa de la población, este comportamiento formaría parte de los resultados. Es decir, se podría generalizar, ya que la población se comporta de esta manera.

Por lo tanto, esta amenaza no afecta nuestro experimento, ya sea porque los resultados se compensan o porque son parte de la realidad, siendo de relevancia marcar la tendencia.

6.4.3. Validez del constructo

Entre las amenazas a la validez del constructo se encuentran las amenazas al diseño del experimento.

Una posible amenaza a esta validez se da cuando se toma sólo una medida en el experimento, ya que con la información de una sola medida se pueden obtener conclusiones incorrectas.

Por ejemplo, se consideran los sujetos, A, B, C y D, quienes aplican la técnica inspecciones sobre un mismo software. Si el sujeto A encuentra 23 defectos, el sujeto B encuentra 20, el C encuentra 22 y el sujeto D encuentra 3 defectos, no contamos con elementos para evaluar por qué el sujeto ha encontrado tan pocos defectos respecto al resto.

En cambio, si sumamos la información proporcionada por otra medida, como por ejemplo el tiempo de ejecución, se puede obtener una idea mas precisa de la realidad. Considerando el tiempo de ejecución en horas, se tiene que A realizo la inspección en 2 horas, B lo hizo en 1,5 y C en 1,75 horas, mientras que la inspección realizada por D duró 3 minutos, dada esta información se puede

comprender mejor el resultado obtenido con la medida *cantidad de defectos*, y de esa forma decidir si será considerado como un dato válido o no.

Este ejemplo muestra por qué cruzando (*cross-checking*) los resultados de distintas medidas se obtienen conclusiones mas representativas de la realidad. En este experimento se tienen en cuenta varias medidas, como lo son el tiempo de detección de defectos, la cantidad de defectos encontrados, etc., por lo que podemos considerar que esta amenaza está cubierta.

La interacción entre los tratamientos que aplica un sujeto también puede afectar la validez de constructo. En el experimento este no es un riesgo, porque la aplicación de los tratamientos es independiente.

6.4.4. Validez externa

Dentro de las amenazas a la *validez externa* encontramos la relación entre la selección de los sujetos y el tratamiento. Este es el efecto de tener una población que no es una muestra representativa de la población en la que queremos generalizar los resultados. En este experimento los sujetos fueron voluntarios dentro de un grupo de sujetos con similares características. Son estudiantes avanzados de la carrera Ingeniería en Computación, con interés en la verificación de software.

El hecho de que sean voluntarios, y que dentro de ese grupo de voluntarios se hayan seleccionado 14 sujetos al azar hace que sean representativos de los estudiantes de la carrera, pero no de los profesionales en la industria.

Una amenaza ligada con la anterior es la relación entre las características del ambiente en el que se lleva a cabo el experimento y los tratamientos. Amenazas del estilo son que el ambiente en el que se ejecuta el experimento, en nuestro caso conformado por los programas, sujetos, y guías de verificación, no sea representativo del ambiente sobre el que se quiere generalizar los datos. En nuestro caso, complementando la amenaza anterior, los programas y posiblemente las guías, no son representativos de los utilizados a nivel industrial. Por ejemplo, los programas son casos simplificados de programas de mayor envergadura utilizados en la industria.

Dado el efecto de las amenazas mencionadas, los datos no son generalizables a nivel industrial.

6.5. Operación

La operación se dividió en dos etapas: una de entrenamiento y otra de ejecución de los experimentos unitarios. El material utilizado en la etapa de entrenamiento fue preparado con antelación. En esta etapa los sujetos concurrieron al dictado de micro-cursos sobre las técnicas de verificación y las clasificaciones de defectos que serían utilizadas en el experimento.

Estos micro-cursos se dictaron como parte del Módulo de Taller en el que se lleva a cabo el experimento. Fueron clases de 2 horas aproximadamente, teóricas y prácticas. La práctica consistió en la resolución en clase, generalmente por parte del docente, de un ejercicio que trata la aplicación de la técnica, o la clasificación de un defecto en la taxonomía en cuestión. Una vez finalizados, se le suministró a los estudiantes el material utilizado para su dictado, como fuente de consulta.

Adicionalmente se dictó un micro-curso para presentar la herramienta en la que registrarían la información de los experimentos unitarios (p. ej.: defectos detectados). Esta herramienta, llamada Grillo, es una aplicación web, desarrollada para la ejecución del experimento.

El dictado de los micro-cursos se realizó durante 4 semanas. La distribución de los mismos en dicho período puede observarse en el cuadro 6.2.

Semana	**Micro-curso**
Sem. 1	Criterio de Cubrimiento de Cond. Múltiple Trayectorias Lin. Independientes Part. de Equiv. y Análisis de Val. Límite Tablas de Decisión
Sem. 2	Inspecciones Clasificación de IBM Taxonomía de Beizer (primera parte)
Sem. 3	Herramienta Grillo
Sem. 4	Taxonomía de Beizer (segunda parte)

Cuadro 6.2: Dictado de los micro-cursos

Posteriormente, también como parte del entrenamiento, se realizó lo que fue denominado como *Experiencia Cero*. En esta experiencia cada sujeto debía aplicar en un mismo programa las tres técnicas que utilizaría en el experimento. Los resultados de sus verificaciones fueron evaluados y recibieron las correcciones o apreciaciones correspondientes.

Para la ejecución de la *Experiencia Cero* se entregó a los sujetos una serie de guías con los lineamientos que debían seguir para aplicar cada tipo de técnica. Las guías totalizan tres: una para Inspecciones, otra para técnicas de caja blanca y otra para técnicas de caja negra. Además de la metodología a seguir, las guías detallan las entregas intermedias y finales que se deben realizar durante la aplicación de la técnica, así como el material que deben esperar por parte de los docentes, como ser código del programa, versión ejecutable, especificación, etc. Estas guías son las utilizadas luego para la ejecución de los experimentos unitarios.

En el Capítulo 8 de la Parte III se describen en detalle los productos mencionados (micro-cursos, guías, *Experiencia Cero*, etc.).

Una vez finalizada la etapa de entrenamiento se procedió con la realización del experimento. El experimento consta de tres fases individuales para cada verificador, denominadas *Experiencia 1*, *Experiencia 2* y *Experiencia 3*. En cada experiencia, un sujeto realiza el experimento unitario correspondiente que le fue asignado en el diseño.

La ejecución de cada experiencia no tiene plazo establecido, pero se estima que pueden durar entre uno y dos meses. La duración finalmente depende de cada sujeto. Dado esto, cada experiencia se comienza una vez que finaliza la anterior. Así, la *Experiencia 1* fue ejecutada por cada sujeto una vez que hubo terminado la *Experiencia Cero*, la *Experiencia 2* luego de la 1, y así sucesivamente. Dadas ciertas complicaciones de los estudiantes durante la ejecución del experimento, como ser la presentación a exámenes y entregas de trabajos obligatorios a otros cursos de la carrera, algunas experiencias se vieron demoradas, hasta llegar

algunas a durar 3 meses.

En total el experimento duró 10 meses, este es el tiempo comprendido entre el envío de la *Experiencia Cero* (31/10/2008) a los estudiantes, y la entrega de la última *Experiencia 3* pendiente (19/07/2009).

El procedimiento seguido para ejecutar cada tipo de técnica se detalla a continuación.

Técnicas de Caja Negra.- La figura 6.1 ilustra el procedimiento para aplicar una técnica de caja negra, mostrando cuáles son las entradas y las salidas de cada paso. Las salidas presentadas dentro de un recuadro son los registros que debe realizar el sujeto al finalizar cada paso. Para aplicar este tipo de técnica primero se provee a los sujetos de la especificación y una versión ejecutable del programa. Con este material generan los casos de prueba, aplicando la técnica que corresponda. Una vez finalizados, los casos son ejecutados, anotando las fallas observadas. Luego de la ejecución, se hace entrega a los sujetos del código fuente del programa, para que aíslen los defectos que provocaron las fallas observadas.

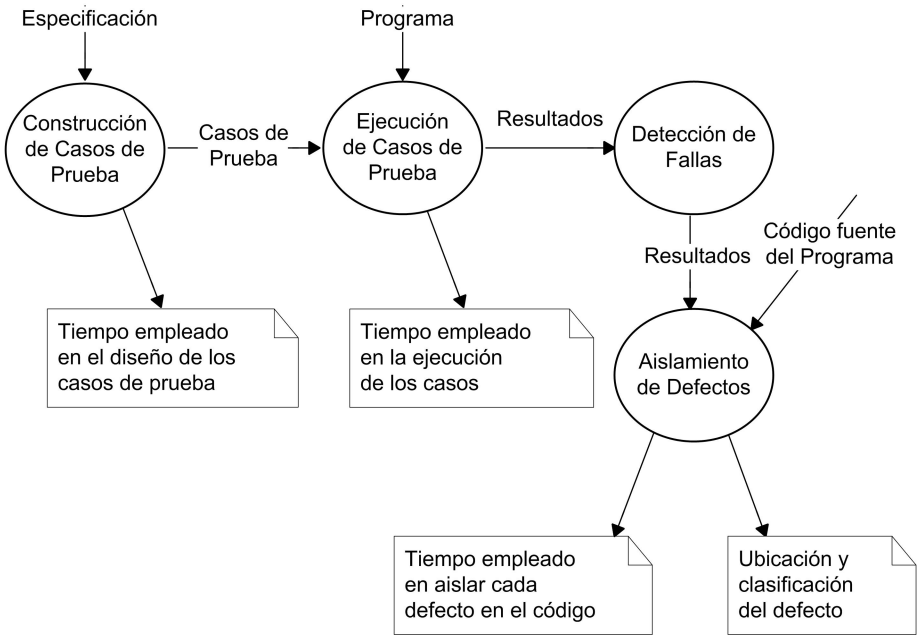

Figura 6.1: Pasos para aplicar una técnica de caja negra

Técnicas de Caja Blanca.- La figura 6.2 ilustra el procedimiento para aplicar una técnica de caja blanca. Para las técnicas de caja blanca en primera instancia se provee a los sujetos del código fuente y de la especificación del programa. Con estos dos elementos deben generar los casos de prueba, cumpliendo con el criterio de la técnica correspondiente. Una vez finalizados, los casos son ejecutados, anotando las fallas observadas. Posteriormente se busca en el código los defectos que provocaron las fallas.

Inspecciones.- La figura 6.3 ilustra el procedimiento para aplicar la técnica de verificación Inspecciones. Para aplicar esta técnica se provee a los sujetos de la especificación del programa y el código fuente, sobre el cual se realizan sucesivas

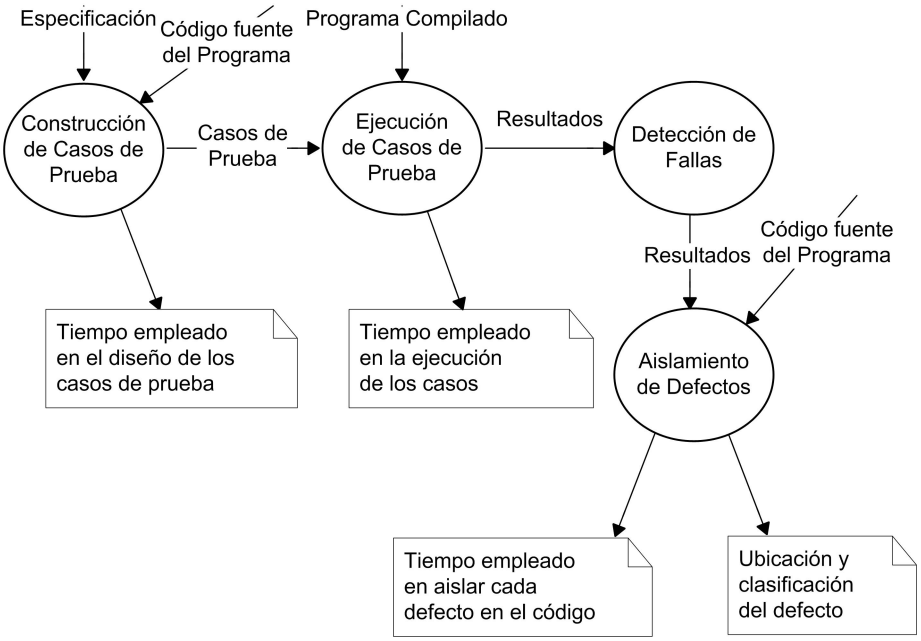

Figura 6.2: Pasos para aplicar una técnica de caja blanca

lecturas de código en busca de items de una lista de comprobación (*check-list*). Por cada ítem de la lista se debe realizar una lectura completa del código. Se les recomienda no realizar revisiones de más de 2 horas de continúo. Con esta técnica los defectos se encuentran *durante* la ejecución de la verificación.

Cada defecto detectado se registra y clasifica en las taxonomías de Beizer e IBM.

Todos los datos de cada experiencia son registrados en la herramienta Grillo. Esto incluye registrar el tiempo empleado en el diseño de casos de prueba (para las técnicas dinámicas), el tiempo empleado en la ejecución de las técnicas y la fecha de finalización de la experiencia. Además, por cada defecto se registra información sobre su ubicación (archivo, línea, etc.), tiempo empleado en aislarlo en el código, y clasificación en ambas taxonomías. En el caso de Inspecciones, el tiempo empleado en la detección de cada defecto es cero.

Cabe aclarar que la utilización de la herramienta para el registro de las experiencias fue a partir de la *Experiencia 1*. En la *Experiencia Cero* los datos se recavaron en una planilla electrónica. Paralelamente al uso de la herramienta, se solicitó a los estudiantes que continuaran registrando los datos en planilla electrónica como en la *Experiencia Cero*, como método de respaldo de la información.

Respecto a la ejecución del diseño, de los 14 sujetos 2 no completaron las experiencias indicadas. El Sujeto 10 ejecutó la *Experiencia Cero* pero abandonó antes de comenzar las experiencias del experimento. El Sujeto 11 ejecutó hasta la *Experiencia 2* inclusive, pero esta se extendió más de lo previsto, y luego de terminarla abandonó, dejando sin ejecutar la *Experiencia 3*. De estos hechos resulta que Trayectorias Linealmente Independientes cuente con dos observaciones menos, y Partición de Equivalencia con Análisis de Valores Límite y Tablas

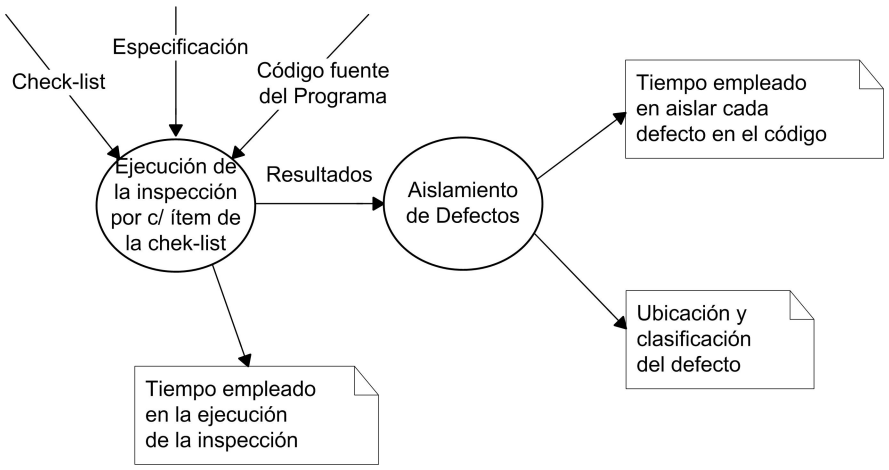

Figura 6.3: Pasos para aplicar la técnica Inspecciones

de Decisión con una observación menos.

Esto deja un total de 36 experimentos unitarios, de 40 iniciales, a ser utilizados para el análisis estadístico y la interpretación de los datos.

Sobre la confiabilidad de la información registrada, sólo se notaron anomalías a nivel de la clasificación de los defectos. La clasificación de un mismo defecto, dentro de cada taxonomía, fue muy variable a través de los sujetos que lo detectaron. Este hecho se vio intensificado en la taxonomía de Beizer.

Capítulo 7

Análisis e Interpretación

Aquí se presenta el análisis de los datos recolectados en todas las experiencias del experimento, según el objetivo y las hipótesis planteadas anteriormente.

Como primer paso, para visualizar los datos recolectados se utilizará estadística descriptiva. Para la prueba de hipótesis se considerará el nivel de significancia $\alpha = 0,1$.

7.1. Efectividad en la detección de defectos

La primer hipótesis planteada refiere a la efectividad en la detección de defectos de cada técnica de verificación. Para estudiar la distribución de los datos para cada técnica utilizaremos histogramas. En cada histograma se grafica la cantidad de sujetos que obtuvieron determinando porcentaje de efectividad en la detección de defectos. La figura 7.1 muestra los cinco histogramas resultantes.

En ellos se observa que las distribuciones de PE y AVL, y de Tablas de Decisión se asemejan a una normal, aunque ambas están sesgadas positivamente (hacia la derecha). Sin embargo, las distribuciones para CCCM, Inspecciones y TLI no son tan simétricas. De hecho, las dos primeras poseen un sesgo positivo y la última un sesgo negativo (hacia la izquierda). La falta de normalidad de las distribuciones nos impide utilizar ANOVA.

Para facilitar su comparación, las distribuciones se presentan en conjunto en el histograma de la figura 7.2. El mismo muestra la efectividad para las cinco técnicas, dividiendo la población en tres clases de efectividad. La primera clase incluye aquellos sujetos con efectividad de entre un 0 % y 20 %. La segunda clase incluye aquellos sujetos con efectividad de entre 20 % y 40 %. La tercer clase incluye aquellos con una efectividad de entre 40 % y 60 % (ningún sujeto superó el 60 % de efectividad).

En general, esto parece sugerir que los sujetos que aplicaron Tablas de Decisión tienen mayor efectividad que el resto de los sujetos, seguido por CCCM e Inspecciones; así como Trayectorias Linealmente Independientes parece ser la técnica menos efectiva, dado que ninguno de los sujetos que la utilizó sobrepasa el 20 % de efectividad. Debemos tener en cuenta que Trayectorias Linealmente Independientes fue ejecutada por dos sujetos menos que CCCM e Inspecciones, y por uno menos que PE y AVL, y Tablas de Decisión.

En total, CCCM e Inspecciones fueron ejecutadas por 8 sujetos, Tablas de

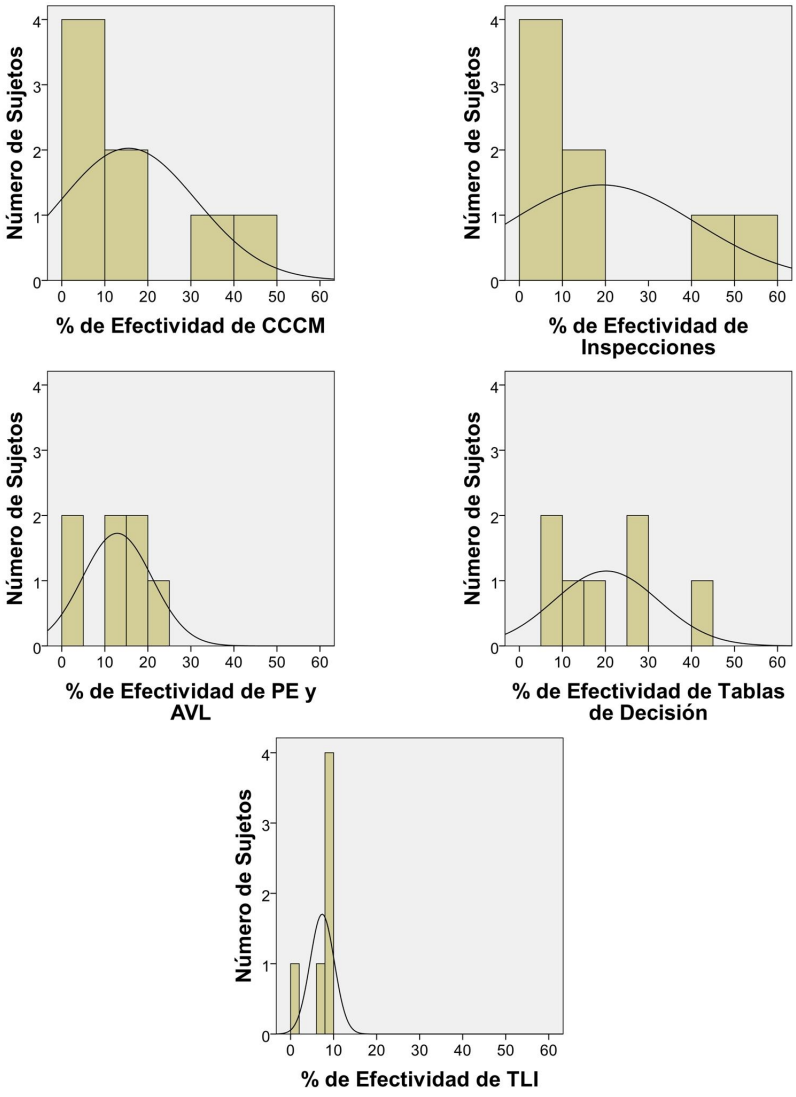

Figura 7.1: Histogramas de la efectividad de las técnicas de verificación

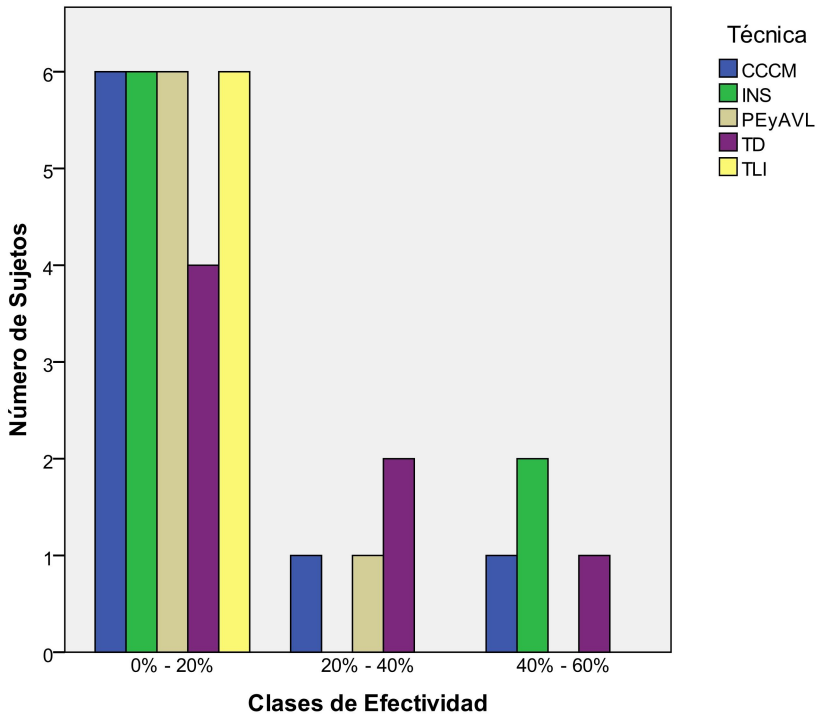

Figura 7.2: Distribución de frecuencia para la efectividad (en clases)

Técnica	Número de sujetos	Mediana	Media	Desviación Estándar
CCCM	8	8,3	15,5	15,7
Inspecciones	8	11,3	19,1	21,8
PE y AVL	7	14	12,9	8,1
TD	7	18,8	20,2	12,2
TLI	6	8	7,3	2,8

Cuadro 7.1: Efectividad de las distintas técnicas de verificación

Decisión y PE y AVL por 7, y Trayectorias Linealmente Independientes por 6. Los valores de mediana, media y desviación estándar de la efectividad de cada técnica se encuentran en el cuadro 7.1.

Para obtener un mejor entendimiento de los datos se realiza el diagrama de caja (*box plot*) de la figura 7.3. Este diagrama es útil para visualizar la simetría (normalidad), el sesgo y los puntos extremos de la distribución de los datos, permitiendo contrastar conjuntos de datos diferentes de una misma variable, en este caso, de la *efectividad*.

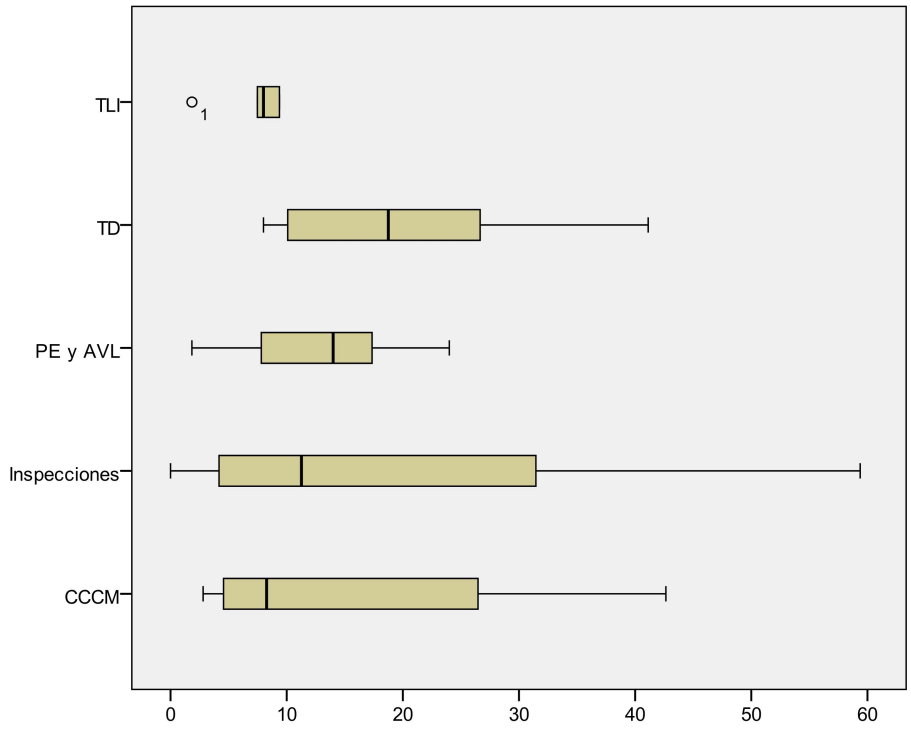

Figura 7.3: *Box plot* de la efectividad de las técnicas de verificación

La figura 7.4 explica las medidas utilizadas en un diagrama de caja. Los segmentos que se salen de la "caja" se llaman "bigotes" (*whiskers*). Una gráfica de caja divide los datos en cuatro partes iguales. El bigote izquierdo, la parte izquierda de la caja, la parte derecha de la caja, y el bigote derecho representan

cada uno un cuarto de los datos (25 %).

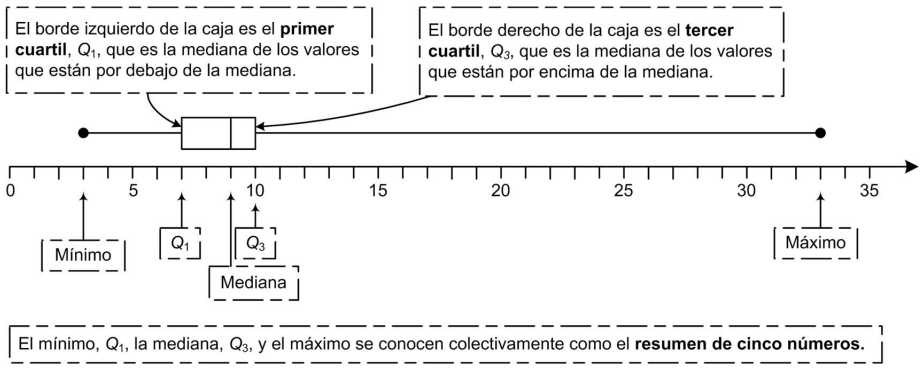

Figura 7.4: Ejemplo de Gráfico de Caja (*Box plot*)

El largo de la caja se conoce con el nombre de *Rango Intercuartílico* (IQR), y se calcula como $Q3-Q1$. Este rango da una noción de la "difusión" del 50 % de los datos centrales. Esta interpretación es la más robusta, porque el 50 % central no está afectado por los *outliers* o valores extremos, y ofrece una visualización más imparcial de la difusión (o dispersión) de los datos.

Los bigotes de la caja representan el límite teórico dentro del cual se encuentran todos los datos. El bigote superior, $Maximo$, es $Q3 + 1{,}5 * IQR$ y el bigote inferior, $Minimo$, es $Q1 - 1{,}5 * IQR$. Los valores de los bigotes son truncados al dato real más cercano, para evitar valores sin significado (como valores negativos de efectividad). Los valores por fuera de los bigotes inferior y superior son datos *outliers*, y se muestran explícitamente con un círculo.

De la figura 7.3, se puede observar una cierta superioridad en efectividad por parte de la técnica Tablas de Decisión respecto al resto de las técnicas, esta superioridad es casi absoluta respecto a la técnica Trayectorias Linealmente Independientes.

En los diagramas de caja de las distribuciones de CCCM e Inspecciones se observa un desplazamiento de la caja hacia el bigote izquierdo, haciendo que el extremo derecho sea más largo, y la mediana se ubica más cerca de la línea izquierda de la caja que de la derecha. Esto significa que las distribuciones están sesgadas hacia la derecha, como ya lo habíamos observado en los histogramas correspondientes, por lo que la mayoría de los valores son "pequeños", y hay unos pocos excepcionalmente grandes.

Como ocurrió con los histogramas, los diagramas de caja de PE y AVL, y de Tablas de Decisión no muestran un sesgo marcado. De todas formas, PE y AVL parecen presentar un leve sesgo hacia la derecha. En estas distribuciones (sesgadas hacia la derecha) la mayoría de los valores son "grandes", y hay unos pocos excepcionalmente pequeños.

El diagrama de caja de TLI muestra la existencia de un *outlier* a la izquierda de la distribución, marcado con un círculo. En el cuadro 7.2 se muestra la efectividad alcanzada por el sujeto que obtuvo el *outlier* a través de sus experiencias.

Durante la ejecución del experimento no se detectaron anomalías en la realización de estas experiencias. Como puede observarse, en las dos primeras el sujeto obtuvo una efectividad similar, mientras que la tercera se dispara respecto a las anteriores. Dado que las dos primeras utilizan técnicas de caja blanca, y la

Experiencia	Técnica	Efectividad
Primera	CCCM	3,1 %
Segunda	TLI	1,8 %
Tercera	PE y AVL	19,6 %

Cuadro 7.2: Experiencias realizadas por el Sujeto 3

tercera de caja negra, se podría atribuir la diferencia a un mayor facilidad para aplicar técnicas de caja negra por parte del sujeto. Otro aspecto a considerar puede ser la experiencia ganada por el sujeto al llegar a la tercer experiencia. El dato parece no ser incorrecto, por lo tanto no será excluido del análisis.

Según lo que se observa en los estadísticos descriptivos, la tendencia parece ser que Tablas de Decisión es la que posee mayor efectividad, seguida de PE y AVL, Inspecciones, TLI y CCCM.

Como se mencionó anteriormente, la falta de normalidad en las distribuciones de las muestras impide aplicar ANOVA para la prueba de hipótesis. El resultado de la alternativa no paramétrica a ANOVA, Kruskal Wallis, resultó ser no significativo (todas las muestras provienen de la misma población).

Ante esta situación ejecutamos el test de Levene para comprobar la homocedasticidad de los datos, y resultó significativo, con un p-valor de 0,017. Esto indica que los datos son heterocedásticos con un 98,3 % de confianza. Como se mencionó anteriormente, ante datos con un alto nivel de heterocedasticidad, el test de Kruskal Wallis puede dar un resultado no signifcativo cuando existe una diferencia real entre las muestras.

Finalmente utilizamos un test no paramétrico, denominado Test de Alternativas Ordenadas, el cual no se ve afectado por la heterocedasticidad de los datos. Este test puede consultarse en [dAO]

El estadístico resultante de aplicar el Test de Alternativas Ordenadas es $L = 4,08$. El detalle de los cálculos se muestra en el documento *Test de Alternativas Ordenadas*. Este estadístico establece que se rechaza H_{01} a nivel α si $L \geq Z_\alpha \sqrt{Var\,(L)}$.

El cuadro 7.3 muestra los valores de $Z_\alpha \sqrt{Var\,(L)}$ para los distintos valores de α.

α	Z_α	$Z_\alpha \sqrt{Var\,(L)}$
0,01	2,33	5,01
0,05	1,65	3,54
0,10	1,28	2,76

Cuadro 7.3: Valores a comparar con el estadístico L

Como puede observarse, para $\alpha = 0,05$ $L = 4,08 \geq 3,54 = Z_\alpha \sqrt{Var\,(L)}$, por lo tanto se rechaza la hipótesis nula H_{01} con un 95 % de confianza.

Dado que existe un ordenamiento entre la efectividad de las técnicas de verificación, procedemos a aplicar el test de Mann-Whitney para estudiar los casos 2 a 2.

Del test de Mann-Whitney se obtuvo que Tablas de Decisión es más efectiva que Trayectorias Linealmente Independientes con un p-valor de 0,0105 (99 %

de confianza), y que Partición de Equivalencia con Análisis de Valores Límite es más efectiva que Trayectorias Linealmente Independientes con un p-valor de 0,066 (93,4 % de confianza).

Cabe observar que los tamaños de las muestras de las que se han obtenido nuestros resultados son reducidos (entre 6 y 8 observaciones por muestra). Es necesario realizar posteriores experimentos con una mayor cantidad de observaciones por muestra para reforzar estos hallazgos.

Estos resultados aplican al contexto en el que fueron obtenidos. Esto es, a un nivel académico, con sujetos con nivel principiante de experiencia en la verificación de software, y con conocimientos básicos de las técnicas de verificación utilizadas. No es posible generalizar los datos a otros contextos, como ser la industria. Para ello debería repetirse el experimento con sujetos representativos del ambiente industrial, aplicando las técnicas sobre programas con características (tamaño, complejidad, cantidad de defectos, etc.) del tipo de software desarrollado en la industria.

Nuestros resultados son respaldados por algunos de los hallazgos obtenidos en los estudios anteriores, presentados en el Capítulo 5 de la Parte I, y contrariados por otros.

En el estudio de Basili y Selby [BS87] las técnicas de verificación fueron ejecutadas por tres grupos de sujetos, diferenciados por el nivel de experiencia de los individuos que componían cada grupo: avanzado, intermedio y principiante. Nuestros resultados son sólo comparables con los resultados obtenidos por los grupos intermedio y principiante (estudiantes con mediana y mínima experiencia en verificación respectivamente).

Los sujetos que participaron en los otros estudios poseen características similares a los que participaron de nuestro experimento: estudiantes avanzados de la carrera Ingeniería en Computación, con un conocimiento básico de las técnicas a utilizar.

En el experimento de Basili y Selby, los sujetos de los grupos con nivel intermedio y principiante de experiencia no obtuvieron diferencias en efectividad al utilizar las técnicas de verificación de lectura de código y dinámica de caja negra. Pero con ambas obtuvieron resultados superiores a la técnica dinámica de caja blanca. Este hallazgo respalda el obtenido en nuestro estudio. Esto es, que las técnicas de verificación Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite son más efectivas que la técnica Trayectorias Linealmente Independientes.

Los hallazgos de Kamsties y Lott [KL95] en su estudio contrastan con el nuestro. Ellos obtuvieron que la efectividad depende del programa, no de la técnica (las técnicas presentan la misma efectividad).

Nuestros resultados también fueron contrariados por los obtenidos por Juristo y Vegas [JV03]. En su estudio, las técnicas de caja negra y caja blanca se comportaron de la misma forma, y la técnica de lectura de código peor que ambas.

No es posible evaluar la hipótesis de efectividad por tipo de defecto, debido a que no se cuenta con una clasificación única por defecto. Las clasificaciones de los defectos fueron realizadas por los sujetos a medida que aislaban los defectos en el código fuente de los programas. Esta clasificación resultó en una gran variedad, de la cual fue inviable obtener una única clasificación por defecto. Dado el escaso tiempo con el que contábamos para realizar esta tarea, y la cantidad de defectos que totaliza los existentes en los programas, la clasificación de los defectos quedó

fuera del alcance de este proyecto de grado.

7.2. Costo de la detección de defectos

A continuación se realiza un breve estudio del *costo* de la detección de defectos para cada técnica por programa. La figura 7.5 muestra los costos en minutos obtenidos en el experimento. Cada medida de costo es la suma del tiempo empleado en el diseño de casos de prueba más el tiempo empleado en la ejecución de la técnica. En el caso de Inspecciones el tiempo de diseño de casos es cero. Si el diseño del experimento se hubiera ejecutado en su totalidad se observarían 2 puntos de cada técnica por programa.

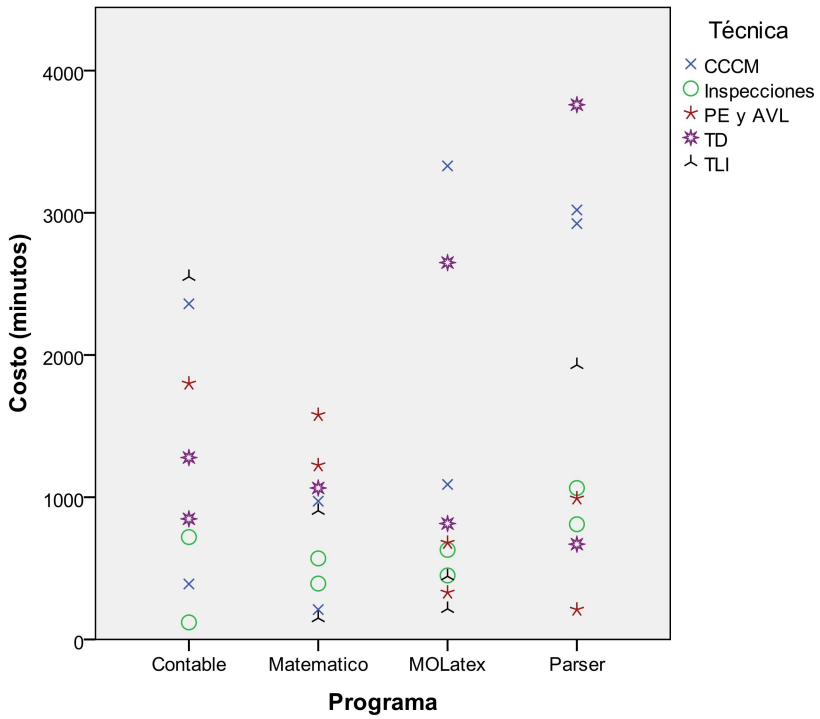

Figura 7.5: Costos de cada Técnica por Programa

Como puede observarse, el costo de las técnicas varia enormemente de programa a programa, e incluso dentro de un mismo programa. Este es el caso de Tablas de Decisión en Parser, y de CCCM en los programas Contabilidad y MO-Latex. CCCM tiende a encontrarse entre los valores más altos, aunque presenta valores bajos.

La técnica Inspecciones es la que presenta valores más cercanos por programa, lo que permite observar que fue claramente más costosa al ser aplicada en Parser que en los programas MO-Latex y Matemático. Los costos de esta técnica en el programa Parser prácticamente doblan los del programa Matemático. Esta variación parece ser acorde a la diferencia de tamaño de los programas mostrada en el cuadro 7.4.

Programa	Líneas de Codigo			
	Total s/com.	**Total c/com.**	**Total por método**	**Prom. por método**
Contabilidad	1979	3109	1497	9,073
Matemático	468	729	375	13,393
MO-Latex	566	1124	362	5,934
Parser	828	1617	634	9,906

Cuadro 7.4: Tamaños de los programas

Además, Partición de Equivalencia con Análisis de Valores Límite parece haber sido más costosa en Matemático que en MO-Latex.

Los puntos alejados de una misma técnica en un programa no permiten observar tendencias, y menos realizar comparaciones de la técnica entre los programas. Para poder analizar cada caso, y lograr identificar si alguno de ellos es un *outlier* es necesario contar con una mayor cantidad de muestras de cada técnica por programa.

Parte III

Productos

Capítulo 8

Elementos para la preparación del experimento

Es necesario preparar a los sujetos previo a la realización del experimento. Deben ser instruidos en los elementos a utilizar y procesos a seguir. Para ello se desarrolló material para ocho *micro-cursos*, que presentan los diferentes temas que deben conocer los sujetos para ejecutar el experimento, y una pequeña experiencia piloto (denominada *Experiencia Cero*), para aplicar los conceptos adquiridos en los micro-cursos. A su vez, los experimentadores deben controlar la ejecución del experimento, por lo que deben proporcionar a los sujetos elementos que los guíen en la ejecución y en la obtención y registro de los datos. Para ello se construyeron tres *guías*, que muestran el proceso a seguir por los sujetos para la ejecución del experimento; y una *herramienta* que permite el registro de los datos recolectados.

En las siguientes secciones se presentan cada uno de los elementos mencionados, analizando las mejoras identificadas para cada uno de ellos. Una mejora que aplica a todos es realizar encuestas que evalúen la usabilidad y calidad de los productos posteriormente a la utilización de los mismos, para su mejora continúa.

El cuadro 8.1 indica quién ha estado a cargo de la construcción de cada producto. El signo "+" indica que ha sido desarrollado por el presente proyecto de grado, el símbolo "-" indica que ha sido construido por otro proyecto de grado, y el símbolo "O" que fue desarrollado en conjunto. Las abreviaciones MC y Prg significan Micro-Curso y Programa respectivamente.

8.1. Guías

Para unificar el proceso de verificación seguido por los sujetos se cuenta con tres guías, una para cada tipo de técnica: estáticas, dinámicas de caja blanca y dinámicas de caja negra. Este proyecto de grado participó en el diseño de las tres guías.

Las guías indican las actividades que deben realizarse al llevar a cabo una

Producto	**Desarrollador**	**Tareas**
Guías	O	Diseño de las guías.
Herramienta	O	Análisis y especificación de requerimientos. Monitorización de su desarrollo. Verificación.
MC Inspecciones	–	
MC CCCM	O	Armado del micro-curso.
MC TLI	–	
MC TD	+	Armado y dictado del micro-curso.
MC PE y AVL	+	Armado y dictado del micro-curso.
MC Beizer	+	Armado y dictado del micro-curso.
MC IBM	–	
MC Herramienta	–	
Experiencia Cero	–	
Prg Contabilidad	–	
Prg Matemático	+	Armado y redacción de la especificación del programa. Monitorización de su desarrollo. Verificación.
Prg MO-Latex	–	
Prg Parser	+	Armado y redacción de la especificación del programa. Monitorización de su desarrollo. Verificación.

Cuadro 8.1: Desarrollo de los Productos

experiencia. Muestran lo que se debe hacer y el orden en el que se debe hacer.

Cabe destacar que las guías no indican cómo aplicar cada técnica, sino los pasos a seguir en la ejecución de una experiencia con determinado tipo de técnica de verificación.

Las guias se componen de tres partes: propósito, criterios de entrada y fases del proceso de verificación. El *propósito* describe el objetivo de la guía. Los *criterios de entrada* son los elementos necesarios para poder llevar a cabo las fases del proceso. Las *fases* se componen de las actividades que se deben realizar para ejecutar la verificación. Cada fase tiene su propia guía, que indica el *propósito* de la fase, los *criterios de entrada* de la fase y las *actividades* que comprende.

Las tres guías se componen de las fases: Preparación, Ejecución y Finalización. Adicionalmente, las guías de *caja blanca* y *caja negra* contienen la fase Diseño. En la figura 8.1 se observa el orden en el que se ejecutan las fases.

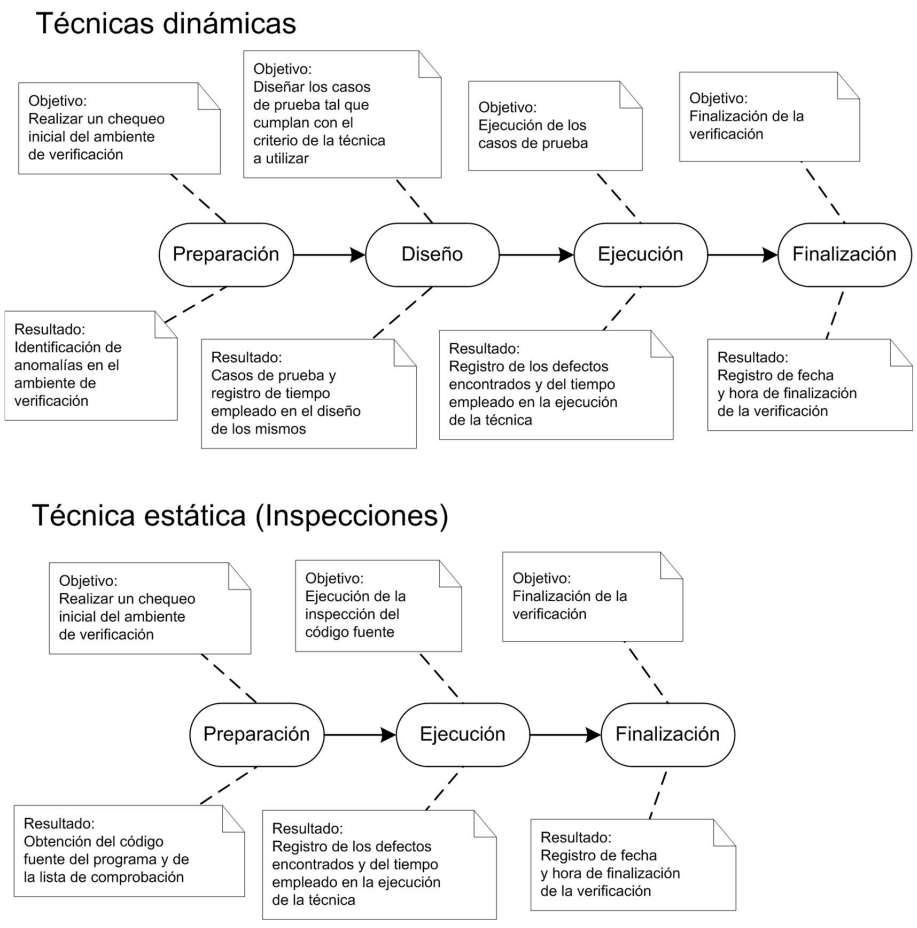

Figura 8.1: Fases que componen las Guías

La fase de *Preparación* consiste en descargar los archivos necesarios para realizar la experiencia. La especificación del programa y los .class para verificar con una técnica de caja negra, o los .java para verificar con una técnica de caja blanca o estática. En el caso de caja blanca y caja negra es necesario configurar

el ambiente para las pruebas.

En la fase de *Diseño* se diseñan los casos de prueba para cumplir con la técnica en cuestión. Luego se eliminan los casos de prueba que no puedan ser ejecutados y los restantes se codifican en JUnit. Luego de codificar los casos de prueba estos se deben enviar al personal encargado de monitorear las experiencias de los sujetos, y se deben registrar los tiempos empleados.

La fase de *Ejecución* es particular para cada guía. En la guía para las técnicas estáticas se indica que para cada ítem de la lista de comprobación (*check-list*) se debe realizar la inspección del código fuente del programa y registrar los defectos detectados. Luego de terminado este ciclo se deben registrar los tiempos empleados.

Para las guías de caja negra y caja blanca, la fase de *Ejecución* consiste en, por cada clase, ejecutar los casos de prueba y analizar la salida esperada. Luego, para cada caso de prueba que haya fallado, se deben encontrar y registrar los defectos. Una vez finalizada la ejecución de los casos de prueba y la detección de los defectos, se registran los tiempos empleados.

En la fase de *Finalización* se registran la fecha y hora de finalización de la experiencia.

Críticas y mejoras luego de la aplicación

Al comienzo de la ejecución del experimento se notó confusión por parte de los sujetos en cómo realizar el proceso de verificación, más allá de cómo ejecutar las técnicas. A su vez ocurrieron irregularidades en la entrega y el registro de información. Esto fue solucionando en el transcurso del experimento.

Estos hechos parecen ser producto, entre otras cosas, de una mala o nula utilización de las guías, dado que muchas de las consultas que se realizaron estaban explícitas en las mismas. Para mejorar la comprensión y motivar el uso de las guías por parte de los sujetos podría ser conveniente realizar una clase, dictada en conjunto con las ya existentes, que presente y explique las guías, en lo posible mediante ejemplos prácticos. Esto permitiría evacuar en forma temprana las dudas sobre el proceso de verificación y la utilización de las guías, así como tener un conocimiento más certero de cuál es el nivel de usabilidad que tienen para los sujetos, identificando posibles mejoras a realizar sobre las mismas.

8.2. Herramienta

La herramienta *Grillo*, es un aplicativo web que permite registrar información sobre experiencias de verificación unitaria. Fue diseñada y construida especialmente para el registro de los datos del experimento. Su construcción estuvo a cargo de una pasantía, codirigida con otro proyecto de grado y con el director del nuestro. En el marco de este proyecto de grado se realizaron tareas de análisis y especificación de los requerimientos, verificación de la herramienta y monitorización de su desarrollo.

Entre sus características no funcionales se encuentra que es una aplicación multiusuario. Es soportada por los navegadores Internet Explorer (versión 6.0 o posterior) y Mozilla Firefox (1.4 o posterior).

La herramienta cuenta con un catálogo de técnicas de verificación y un catálogo de taxonomías de defectos. El primero está compuesto por las técnicas Inspecciones, Partición de Equivalencia y Análisis de Valores Límite, Tablas

de Decisión, Criterio de Cubrimiento de Condición Múltiple, y Trayectorias Linealmente Independientes. El segundo está compuesto por las taxonomías de defectos de IBM y de Beizer.

La aplicación maneja dos perfiles de usuario: administrador y verificador. Los usuarios con perfil *administrador* pueden operar con todas las funcionalidades del sistema. Los usuarios con perfil *verificador* pueden acceder a un conjunto limitado de funcionalidades.

Las funcionalidades del aplicativo pueden dividirse en funcionalidades *administrativas* y funcionalidades para el *registro de información de la ejecución de las experiencias*.

Entre las funcionalidades *administrativas* están el mantenimiento de usuarios, archivos, programas y experiencias. Estas funcionalidades están reservadas para los usuarios con perfil administrador.

En el mantenimiento de usuarios se crean, modifican y eliminan los usuarios del sistema. En el de archivos se permite ingresar en el sistema los archivos que componen los programas verificados por los usuarios. En el de programas se ingresan los nombres de los programas que se utilizarán en las experiencias. Al ingresar un programa se deben indicar cuáles son los archivos que lo componen. Estos archivos deben haber sido ingresados previamente en el sistema. El mantenimiento de experiencias permite ingresar los datos necesarios de cada experiencia para que puedan ser realizadas por un usuario. Entre estos datos están la técnica a aplicar, el programa a verificar y el usuario que tendrá asignada la experiencia.

Entre los datos que se permite *registrar de la ejecución de las experiencias* están el tiempo total de diseño de casos de prueba (sólo para las técnicas dinámicas), el tiempo total de ejecución (no incluye el tiempo de casos de prueba ni el de encontrar los defectos en el código), y la fecha de finalización de la experiencia. Los usuarios con perfil verificador sólo pueden modificar y consultar los datos de las experiencias que tienen asignadas. Los usuarios con perfil administrador pueden modificar los datos de ejecución de todas las experiencias.

Además del registro de tiempos, los usuarios pueden registrar los defectos detectados durante la ejecución de una experiencia. Aquellos con perfil verificador sólo pueden registrar defectos en las experiencias que tienen asignadas, mientras que los usuarios con perfil administrador pueden registrar defectos en cualquier experiencia.

El registro de un defecto comprende ingresar el nombre del archivo en el que se encontró, la ubicación dentro del archivo, y el tiempo empleado en encontrarlo y clasificarlo en las taxonomías de IBM y de Beizer.

Toda la documentación de la herramienta *Grillo* puede consultarse en el documento *Documentación de la Herramienta Grillo*. Esta documentación incluye la especificación de requerimientos, el documento de casos de uso y documentación sobre el esquema de base de datos.

Críticas y mejoras luego de la aplicación

Por cuestiones de tiempo no se pudieron implementar todas las funcionalidades de la herramienta.

Entre las funcionalidades que quedaron pendientes están el mantenimiento del catálogo de técnicas de verificación y el mantenimiento del catálogo de taxonomías de defectos. Actualmente el catálogo de técnicas de verificación es

cargado directamente en la base de datos que utiliza el sistema, y el catálogo de taxonomías de defectos forma parte del código de la aplicación.

Además, la aplicación carece de consultas. Entre las que se identifican como prioritarias están:

- Cantidad de defectos detectados en la ejecución de una experiencia.
- Porcentaje de defectos de un programa detectados en la ejecución de una experiencia.
- Efectividad de la técnica de verificación utilizada en una experiencia.
- Tiempo de diseño de los casos de prueba en la ejecución de una experiencia.
- Tiempo de ejecución de los casos de prueba en la ejecución de una experiencia.
- Tiempo total de detección de los defectos encontrados en la ejecución de una experiencia. Este tiempo es la suma de los dos anteriores.
- Cantidad de defectos detectados por hora (tasa de detección) en la ejecución de una experiencia. El tiempo utilizado para este cálculo es el de detección de los defectos (tiempo de diseño de casos de prueba más el tiempo de ejecución de los mismos).
- Todas las consultas anteriores diferenciadas por tipo de defecto para la taxonomía de IBM y de Beizer.
- Todas las consultas anteriores resumidas en un comparativo entre experiencias que tienen la misma técnica.
- Todas las consultas anteriores resumidas en un comparativo entre experiencias que tienen el mismo programa.
- Todas las consultas anteriores resumidas en un comparativo entre experiencias que tienen el mismo verificador.

Para poder satisfacer las consultas anteriores es necesario, entre otras cosas, que la herramienta pueda distinguir qué registros de defecto se refieren al mismo defecto real. Por ejemplo, un verificador puede reportar un defecto ingresando sólo en que estructura se encuentra (IF, WHILE, etc.), mientras que otro que detecta el mismo defecto puede registrarlo asociándole una línea de código exacta. Ambos registros refieren al mismo defecto, por lo que sería erróneo considerarlos a ambos en los cálculos de las consultas. Deberían contarse como uno solo.

Una vez que la herramienta provea la posibilidad de identificar los registros de defecto que hacen referencia al mismo defecto, se podría generar un "registro unificado". Este registro sería, según la herramienta, la forma correcta de registrar el defecto. De esta forma se podría agregar una funcionalidad que presentara todos los defectos (diferenciar de registro de defecto) detectados en una experiencia. La descripción de cada defecto sería el "registro unificado" correspondiente a cada uno. Se debería permitir al usuario modificar los datos de este registro, como cualquier otro registro de defecto. Los cambios deberían ser almacenados por la herramienta.

8.3. Micro-cursos

Para preparar a los sujetos, así como también para nivelar su conocimiento, se cuenta con 8 micro-cursos. Los micro-cursos tratan sobre las técnicas, las taxonomías y la herramienta en la cual se registran los datos de las experiencias. Este proyecto de grado participó en la preparación del micro-curso de la técnica CCCM, y en la preparación y dictado de los micro-cursos de las técnicas Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite y de la taxonomía de defectos de Beizer.

Se cuenta con 5 micro-cursos sobre las técnicas, una por cada técnica a utilizar: Inspecciones, Particiones de Equivalencia y Análisis de Valores Límite, Tablas de Decisión, Criterio de Cubrimiento de Condición Múltiple y Trayectorias Linealmente Independientes. Estos micro-cursos duran aproximadamente 2 horas y son básicamente expositivas. Incluyen ejercicios pequeños y simples que ayudan a una mejor comprensión de los conceptos.

Se cuenta con 2 micro-cursos sobre taxonomías de defectos. Una sobre la Taxonomía de Defectos de Beizer y otra sobre la Clasificación Ortogonal de Defectos de IBM. La clase sobre la taxonomía de IBM dura aproximadamente una hora, y la clase sobre la taxonomía de Beizer aproximadamente cuatro.

En la clase de la taxonomía de IBM se explica qué es y se describen las categorías que la componen. Luego se profundiza en aquellas categorías utilizadas en el experimento, dando ejemplos de defectos que clasifican en ellas.

La estructura de la clase de la taxonomía de Beizer es bastante similar a la de IBM. Primero se hace una introducción, luego se describen todas las categorías de la taxonomía, y finalmente se profundiza en aquellas que se utilizan en el experimento.

La clase de la herramienta es la más pequeña. Básicamente indica cuáles son los datos que se deben registrar de cada experiencia y cómo hacerlo. Esta clase comienza con una breve descripción del proceso de verificación. Luego se identifican y explican los tiempos que se deben registrar y muestra dónde se debe realizar el registro. También se indica cómo registrar las fechas y los defectos encontrados con sus respectivas clasificaciones. Para mayor claridad en las explicaciones se utilizan capturas de pantalla de la herramienta.

Las diapositivas de los micro-cursos desarrolladas en el marco de este proyecto de grado pueden consultarse en el documento *Micro-cursos*.

Respecto al orden en el que se dicten los micro-cursos, se recomienda comenzar por las que describen las técnicas de verificación, seguidos por los micro-cursos de las taxonomías, comenzando con la taxonomía de IBM. Comenzar con la taxonomía de IBM permite introducir los conceptos de taxonomía y clasificación de defectos con mayor facilidad. Es conveniente dictar el micro-curso sobre la herramienta en último lugar, dado que utiliza los conceptos introducidos por el resto de los micro-cursos.

Críticas y mejoras luego de la aplicación

Luego del dictado de los micro-cursos, durante la ejecución del experimento, surgieron dudas por parte de los sujetos acerca de conceptos básicos que fueron expuestos en las clases.

Una manera de mejorar la comprensión de los conceptos por parte de los estudiantes podría ser aumentar su participación en las clases, por ejemplo, in-

cluyendo ejercicios más reales para que resuelvan en clase. Esto permite adelantar la aplicación de la técnica por parte de los sujetos, propiciando la aparición de dudas tempranas sobre la ejecución de las técnicas y la clasificación de los defectos. Esto implicaría extender el tiempo de las clases, ya que se debe dar a los participantes un tiempo razonable para que puedan resolver los ejercicios planteados.

8.4. Experiencia Cero

Además del dictado de las clases, se cuenta con una experiencia previa al experimento, denominada Experiencia Cero, como método de preparación de los sujetos. En esta experiencia cada sujeto ejecuta las tres técnicas de verificación que utilizará en el experimento real sobre el mismo programa. El programa es un pequeño desarrollo en el lenguaje de programación Java. Su funcionalidad es ordenar un arreglo eliminando los elementos repetidos. Al igual que en las experiencias del experimento, en esta experiencia deben aplicarse las técnicas al programa utilizando las guías correspondientes, registrar y clasificar los defectos detectados, y registrar los tiempos empleados.

Una de las diferencias con las experiencias del experimento es que los sujetos tienen devolución del trabajo que han realizado por parte de los docentes, de esta manera aprenden de los errores cometidos. Esta última es la característica más valiosa de la realización de esta experiencia.

Entre otros, se espera que su ejecución permita a los sujetos adquirir práctica en la aplicación de las técnicas, el seguimiento de los procedimientos definidos en las guías, la clasificación en las taxonomías de defectos y el registro de información en la herramienta.

Los detalles de esta experiencia, así como los resultados obtenidos luego de su aplicación, pueden consultarse en [VH09].

Críticas y mejoras luego de la aplicación

A pesar de la realización de la *Experiencia Cero* los sujetos presentaron dificultades varias durante la ejecución del experimento real, principalmente en el uso de las técnicas que les correspondían, siendo el objetivo principal de realizar esta experiencia minimizar los problemas en la aplicación de las técnicas durante el experimento.

Una de las mejoras identificadas para implementar en la experiencia es realizar un documento final con los errores que se cometieron durante su ejecución en todos los niveles (aplicación de las técnicas, clasificación de los defectos, uso de las guías, registro en la herramienta), para que los sujetos aprendan no solo de sus errores, sino también de los cometidos por otros sujetos. De esta manera se minimiza la posibilidad de que durante la ejecución del experimento los sujetos cometan errores ya cometidos por otros en la *Experiencia Cero*. Como es una experiencia con fines de aprendizaje, y todos los sujetos verifican el mismo programa, y varios utilizan las mismas técnicas, no sería incorrecto que conozcan los errores cometidos por otros sujetos, por el contrario, aportaría a la finalidad de la experiencia.

Capítulo 9

Programas

En el experimento se utilizan en total cinco programas: uno para entrenamiento (*Experiencia Cero*) y cuatro para el experimento en sí.

El programa utilizado en la *Experiencia Cero* consta de dos clases. Una de ellas posee 18 líneas de código, y la otra 19, sin contar las líneas de comentarios. Cada clase consta de un único método público y su especificación en JavaDoc. El programa recibe como parámetro un arreglo de enteros y devuelve un arreglo ordenado y sin elementos repetidos. La descripción completa del programa puede encontrarse en [VH09].

Los cuatro programas utilizados en el experimento son:

- Contabilidad (Sistema con uso de base de datos).- Implementa una versión simplificada de un sistema de sueldos. Cuenta con manejo de base de datos.
- Matemático (Sistema de cálculo).- Realiza cálculos matemáticos y probabilísticos.
- MO-Latex (Sistema con uso de base de datos).- Genera exámenes múltiple opción a partir de preguntas seleccionadas por el usuario. Las preguntas, y sus respectivas respuestas, se encuentran almacenadas en una base de datos.
- Parser (Procesamiento de texto).- Procesa texto y analiza su sintaxis. Es un compilador simplificado.

Los programas fueron construidos por estudiantes de cuarto y quinto año de la carrera Ingeniería en Computación de la Facultad de Ingeniería, Universidad de la República, en el lenguaje de alto nivel Java y documentados mediante la generación de JavaDoc para cada uno de sus métodos.

Las especificaciones se idearon de manera independiente, por cuatro personas distintas, por lo que los programas resultaron ser de diferentes tamaños, acorde al tipo y complejidad del problema que propone la especificación. Este proyecto participó en la construcción de la especificación, posterior monitorización del desarrollo, y verificación de los programas Matemático y Parser.

Los cuatro programas se consideran "reales" por el modo en el que se construyeron y el problema que resuelven. Estos problemas son versiones simplificadas de problemas reales de mayor envergadura o más generales. Fueron construidos por estudiantes a los que se les solicitó desarrollar un programa a partir de

una especificación, como ocurre en la industria y en algunos cursos de grado de la carrera. Aunque se consideran de distintos tipos, no se puede asegurar que sean representativos de los mismos, dado que los problemas que resuelven son bastante específicos. Los defectos que presentan son los introducidos por los desarrolladores en su construcción. Los programas están compilados, por lo que no poseen defectos detectables por un compilador.

En las secciones que siguen se presentan más en detalle los programas Matemático, MO-Latex y Parser. En la última sección se presentan métricas de todos los programas.

Las especificaciones de los programas Matemático, MO-Latex y Parser, así como la documentación de análisis y diseño, manuales de usuario y configuración, documentación JavaDoc, y otros, pueden consultarse en el documento *Descripción de los Programas*. El programa Contabilidad puede consultarse en el proyecto de grado *Diseño y Ejecución de un Experimento con 5 Técnicas de Verificación Unitaria* de Cecilia Apa [Apa09].

9.1. Matemático

El programa Matemático recibe como parámetros dos conjuntos de números del mismo tamaño, x e y, y una estimación x_k, a partir de los cuales calcula:

1. la correlación entre los dos conjuntos de números x e y.- La correlación determina la relación entre dos conjuntos de datos numéricos. Toma valores entre $+1$ y -1. Resultados cercanos a $+1$ implican una fuerte relación positiva, cuando x crece también crece y. Resultados cercanos a -1 implican una fuerte relación negativa, cuando x decrece también decrece y. Resultados cercanos a 0 implican que no hay relación.

2. la significancia de la correlación.- El test de significancia determina la probabilidad de que una correlación fuerte sea al azar, y sea por lo tanto de ninguna significancia práctica. Por ejemplo, un conjunto con solamente dos puntos tendrá siempre $r^2 = 1$, pero esta correlación no es significativa.

3. los parámetros de regresión lineal β_0 y β_1 para un conjunto de n pares de datos de la forma $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, siendo n el tamaño de los conjuntos x e y.- La regresión lineal es una manera óptima de aproximar una línea a un conjunto de datos. La línea de regresión lineal cumple que la distancia de todos los puntos a esa línea se reduce al mínimo. La ecuación de una línea se puede escribir como $y_k = \beta_0 + \beta_1 x_k$.

4. la proyección y_k a partir de la estimación x_k, donde $y_k = \beta_0 + \beta_1 x_k$.

5. el intervalo de predicción del 70 % para la estimación x_k.

Este programa tiene una fuerte carga de cálculos matemáticos, por lo que su mayor complejidad reside en que los cálculos se realicen en forma correcta.

El programa consta de 13 clases. El diagrama de clases de la figura 9.1 muestra la dependencia entre ellas. No están incluidas las clases de Excepción y la clase que contiene el método *main* (**EjecucionTarea**). Tampoco incluye los métodos privados de cada clase, sólo los públicos.

La clase **EjecucionTarea** (Main) llama los siguientes métodos para resolver los cálculos:

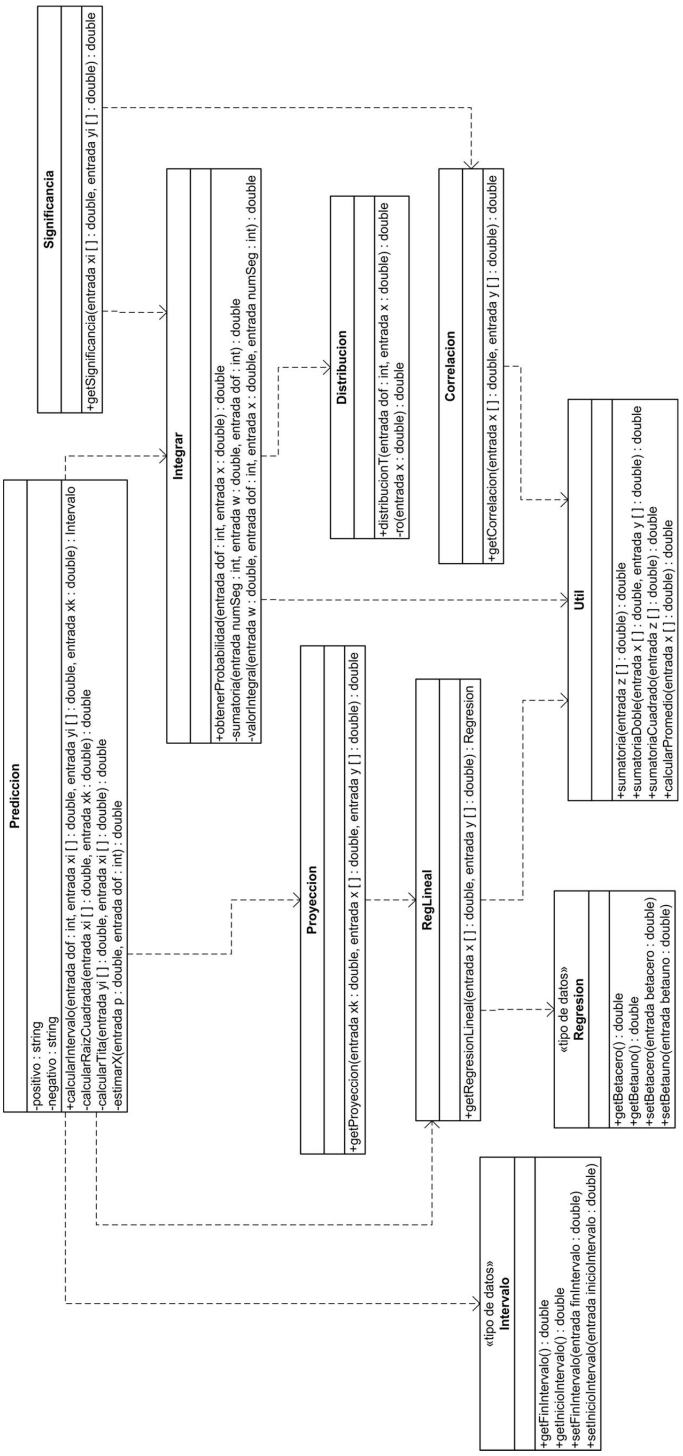

Figura 9.1: Diagrama de clases del programa Matemático

- **getCorrelacion(double[ ], double[ ])** de la clase **Correlacion** para resolver el punto 1.
- **getSignificancia(double[ ], double[ ])** de la clase **Significancia** para resolver el punto 2.
- **getRegresionLineal(double[ ], double[ ])** de la clase **RegLineal**, que retorna el resultado en forma del tipo de dato **Regresion**, para resolver el punto 3. Para obtener los parámetros por separado se utilizan los métodos **getBetacero()** y **getBetauno()** de la clase **Regresion**.
- **getProyeccion(double, double[ ], double[ ])** de la clase **Proyeccion** para resolver el punto 4.
- **calcularIntervalo(int, double[ ], double[ ], double)** de la clase **Prediccion**, que retorna el resultado en forma del tipo de dato **Intervalo**, para resolver el punto 5. Para obtener el comienzo y el fin del intervalo se utilizan los métodos **getInicioIntervalo()** y **getFinIntervalo()** respectivamente, de la clase **Intervalo**.

El siguiente fragmento es un ejemplo de documentación JavaDoc de uno de los métodos del programa:

```
/**
 * Este método retorna la correlacion entre dos conjuntos de
 * valores x e y.
 *
 * @param x - conjunto de valores de tipo double.
 * @param y - conjunto de valores de tipo double.
 * @return retorna la correlación entre dos conjuntos de
 * valores x e y
 * @throws NoDatosException lanza una excepción cuando
 * el conjunto de valores es vacio.
 * @version 1.0
 */
```

En el documento *Descripción de los Programas* se exponen en detalle los defectos que contienen las clases **Correlacion** e **Integrar**, y el resumen de los defectos de todas las clases.

9.2. MO-Latex

MO-Latex es un programa que genera exámenes múltiple opción a partir de una lista de preguntas seleccionadas por el usuario. Las preguntas y sus respectivas respuestas se encuentran almacenadas en una base de datos.

Este programa genera dos archivos LaTeX, uno de ellos contiene el examen de múltiple opción y el otro el mismo examen con las respuestas correctas de cada pregunta en color rojo.

Para generar los exámenes se utiliza el paquete *eqExam* de Latex. Las preguntas y respuestas se almacenan en una base de datos HSQLDB.

HSQLDB (*Hyperthreaded Structured Query Language Database*) es un sistema gestor de bases de datos libre escrito en Java. Por más información consultar la página web http://www.hsqldb.org/.

Al ejecutar el programa se presenta el siguiente menú:

```
-----------------------------------------------------
--------------------MENU PRINCIPAL-------------------
-----------------------------------------------------

Opcion 1 - Generar Examen
Opcion 2 - Listado de Preguntas
Opcion 0 - Salir

-----------------------------------------------------
Esperando entrada >>>
```

La opción 1 genera los archivos LaTeX. Para ello solicita al usuario que ingrese el nombre de la Facultad, el nombre de la asignatura, el nombre de la prueba, el día de la prueba, el mes de la prueba, el año de la prueba, el autor de la prueba, la ruta del directorio donde se guardarán los archivos y los números de pregunta a incluir en el examen separados por un espacio. La opción 2 del menú permite conocer las preguntas existentes en la base de datos.

Las preguntas solo tienen una respuesta correcta. El usuario no puede elegir el conjunto de respuestas a mostrar en el examen para cierta pregunta. La cantidad de respuestas desplegada por pregunta es igual a las existentes en la base de datos.

Existen dos tipos de respuesta, las simples y las encadenadas. Ejemplos de respuesta encadenada son las opciones (b) y (c) en la siguiente pregunta múltiple opción:

```
Un diseño debe ser
  a) verificable
  b) (a) y completo respecto a los requerimientos del producto
  c) (b) y lo más simple posible
  d) Ninguna de las anteriores.
```

Para este tipo de respuestas se guarda en la base de datos la referencia a la respuesta predecesora y el texto de la respuesta. En el caso del ejemplo anterior, para la respuesta (b) se guarda que (a) es su predecesora y se almacena el texto "y completo respecto a los requerimientos del producto".

La base de datos que da soporte al programa consta de tres tablas: *Preguntas*, donde se almacena el identificador (clave de la tabla) y el texto de cada pregunta; *Respuestas*, cuya clave se compone de su identificador y del identificador de la pregunta de la que es respuesta, almacena el texto de la respuesta y si es correcta o no; y *RespuestaDependencias*, que mantiene las dependencias entre las respuestas de una pregunta, tiene como campos el identificador de la pregunta, el identificador de la respuesta predecesora y el identificador de la respuesta misma.

La estructura de la base de datos se muestra en la figura 9.2. La clave primaria de cada tabla está precedida por la abreviación PK (*Primary Key*) y las claves foráneas por FK (*Foreign key*).

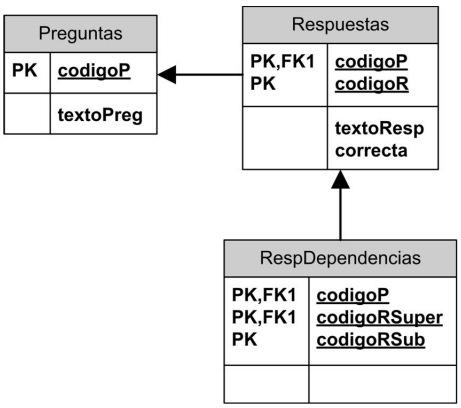

Figura 9.2: Esquema de base de datos del programa MO-Latex

El programa consta de 9 clases y 1 interfaz. El diagrama de clases de la figura 9.3 muestra la dependencia entre ellas. No está incluida la clase que contiene el método *main* (**Principal**). Tampoco incluye los métodos privados de cada clase, sólo los públicos.

Las clases **Respuesta**, **Pregunta**, **DataRespuesta** y **DataPregunta** son tipos de datos que se utilizan para modelar las preguntas y las respuestas del examen.

La clase **Examen** modela el examen y la solución del mismo. Provee el método **agregarPregunta(Pregunta)**, que permite el agregado de las preguntas seleccionadas por el usuario al examen; y el método **generarExamen()** que genera los archivos LaTeXpara el examen y la solución.

La clase **Principal** (Main) maneja la interacción con el usuario mediante línea de comandos, invocando los métodos correspondientes de la interfaz **ICtrlPreguntas**, según la opción que seleccione el usuario, y ejecuta el generador de archivos LaTeX.

El siguiente fragmento es un ejemplo de documentación JavaDoc de uno de los métodos del programa:

```
/**
 * Genera el cabezal de los archivos "preguntas.tex" o
 * "solucion.tex" en el fujo de stream correspondiente, segun
 * indique la variable solucion (false o true respectivamente).
 *
 * @param flujoTex  Hacia donde se dirije la salida.
 * @param solucion solucion indica que tipo de salida se desea
 *  generar (de preguntas.tex o de solucion.tex).
 * @version 1.0
 * @since  26 Ago 2008
 */
```

En el documento *Descripción de los Programas* se exponen en detalle los defectos que contienen los métodos *generarCabezal(PrintWriter, boolean)* y *generarCuerpo(PrintWriter, boolean)* de la clase **Examen**. Ambos métodos son privados, por lo que no figuran en el diagrama de clases, y son utilizados por el

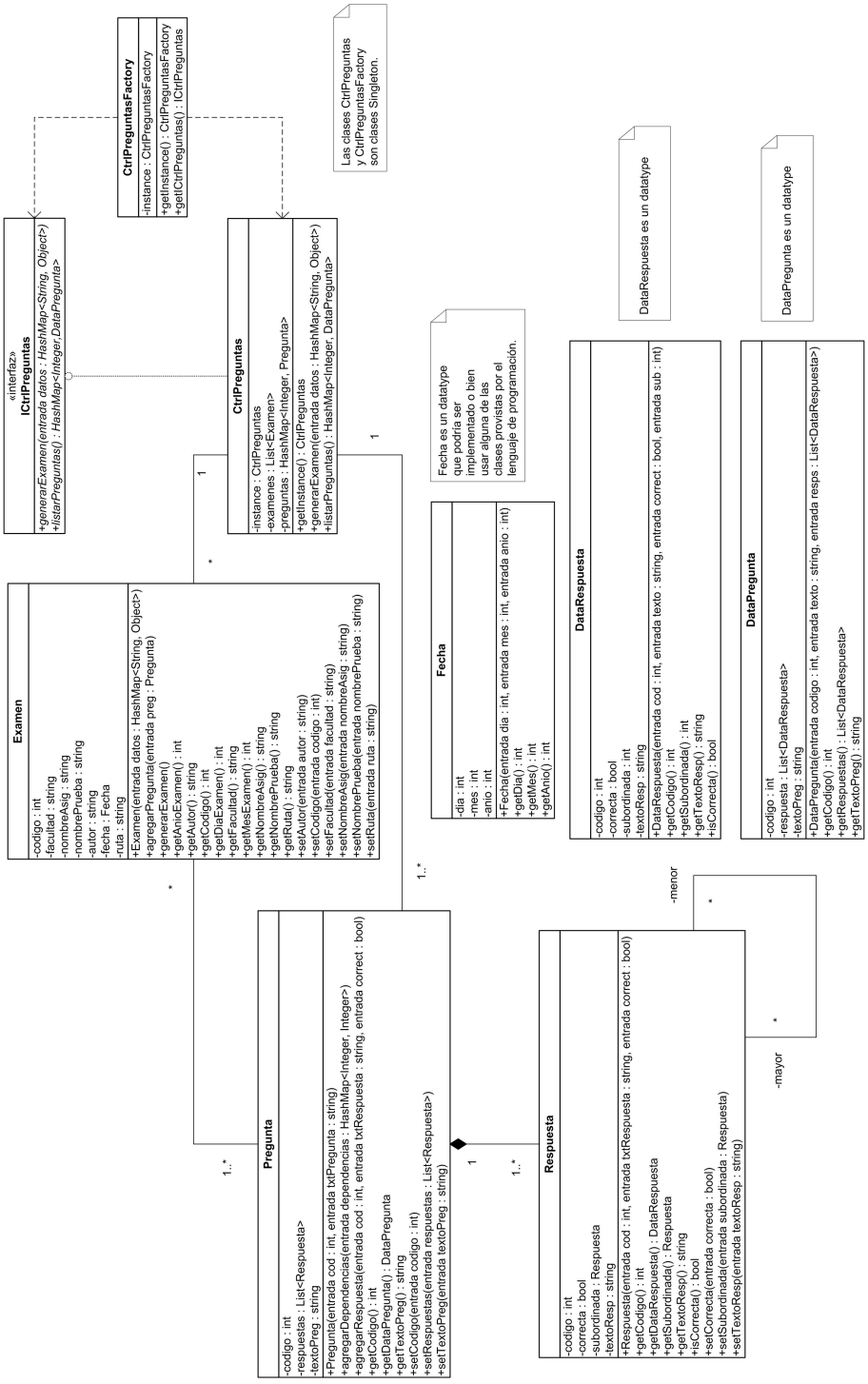

Figura 9.3: Diagrama de clases del programa MO-Latex

método *generarExamen()*. También se muestran los defectos del método público *listarPreguntas()* de la clase **CtrlPreguntas**, y el resumen de los defectos de todas las clases.

9.3. Parser

El programa Parser recibe el código fuente de un programa escrito en una variante del lenguaje de programación Pascal y genera dos archivos: un archivo xml que contiene las declaraciones, estructuras y operadores reconocidos en el código del programa y un archivo de texto que contiene los errores sintácticos detectados en el mismo.

Identifica como sentencias reconocibles: la sentencia de declaración del programa (PROGRAM), el comienzo y la finalización del programa (BEGIN y END), declaraciones de variables y constantes (CONST y VAR), sentencias de asignación (:=), estructuras (FOR, WHILE, REPEAT, IF), funciones y procedimientos (FUNCTION, PROCEDURE) y comentarios (texto entre "(*" y "*)").

Por cada sentencia reconocida el programa escribe una etiqueta en el archivo xml. En el caso de que la sentencia incluya el uso de una variable, se agrega el nombre de la variable dentro de la etiqueta. Por ejemplo, la etiqueta correspondiente a una asignación (:=) es `<ASIGN var=nombre_de_la_variable_asignada>...</ASIGN>`.

A continuación se presenta el código fuente de un programa y el archivo xml generado a partir del mismo:

Programa:

```
PROGRAM ejemplo;
   CONST ancho = 4;
   VAR altura: Integer;
   VAR area: Integer;
   VAR total: Integer;
   VAR iter: Integer;
   VAR texto: String;
   VAR alto: Integer;

   FUNCTION calcArea(base: Integer): Integer
   BEGIN
      altura := ancho/2;
      calcArea := base*altura/2;
   END

BEGIN
   alto := 4;
   area := calcArea(alto);
   IF (area < 3) THEN
      area := area*3;
   ELSE
   BEGIN
      area := area -2;
      IF (area < 3) THEN
         area := area*3;
   END
   texto := "Ej. de calculo de area y estructuras de datos"
   total := 0;
   FOR iter := 1 TO 40 DO
   BEGIN
      total := total + iter*2;
   END
END.
```

Archivo xml:

```
<PROGRAM name=ejemplo>
   <CONST name=ancho></CONST>
   <VAR name=area></VAR>
   <VAR name=total></VAR>
   <VAR name=iter></VAR>
   <VAR name=alto></VAR>
   <FUNCTION name=calcArea type=Integer>
   <PARAM name=base type=Integer></PARAM>
   <BEGIN>
      <ASIG var=area><BARRA></BARRA></ASIG>
      <ASIG var=calcArea><MULT></MULT><BARRA></BARRA></ASIG>
   </BEGIN>
   </FUNCTION>
<BEGIN>
   <ASIG var=alto></ASIG>
   <ASIG var=area>
      <CALL name=calcArea>
         <ARG var=alto></ARG>
      </CALL>
   </ASIG>
   <IF>
   <COND><MENOR></MENOR></COND>
   <THEN>
      <ASIG var=area><MULT></MULT></ASIG>
   </THEN>
   </IF>
   <ELSE>
      <ASIG var=area><RESTA></RESTA></ASIG>
      <IF>
      <COND><MENOR></MENOR></COND>
      <THEN>
         <ASIG var=area><MULT></MULT></ASIG>
      </THEN>
      </IF>
      </ELSE>
         <ASIG name=total></ASIG>
   <FOR>
   <ASIG var=iter></ASIG>
   <TO></TO>
   <DO>
      <ASIG var=total><SUMA><MULT></MULT></SUMA></ASIG>
   </DO>
   </FOR>
</BEGIN>
</PROGRAM>
```

Los chequeos sintácticos que realiza Parser son un pequeño subconjunto de los que realiza un compilador del lenguaje Pascal. Para pasar estos chequeos el código fuente del programa debe cumplir que todas las sentencias terminen en punto y coma, que cada paréntesis abierto sea cerrado y que todo BEGIN tenga un END asociado. En caso de que el programa no cumpla con estas condiciones Parser informa de ello en el archivo de texto.

Al finalizar el reconocimiento y chequeo del código de entrada Parser genera los archivos correspondientes y emite un mensaje indicando la cantidad de errores detectados, o en su defecto informa que no han habido errores.

El programa consta de 9 clases y 1 enumerado. Las clases cuentan con una gran cantidad de métodos por lo que resulta imposible incluirlas en el mismo diagrama de clases. Por esta razón se presentan tres diagramas. El de la figura 9.5 muestra la clase **LectorTokens** y sus dependencias. El de la figura 9.4 muestra la clase **Parser** y sus dependencias. Se presentan en color gris las clases que ya fueron introducidas en el diagrama anterior. El diagrama de la figura 9.6 muestra la clase **Funcion** y sus dependencias. Esta clase fue implementada pero no se utiliza, lo cual es un defecto del programa. Los diagramas no incluyen la clase del tipo Main (**Principal**). Tampoco incluyen los métodos privados de cada clase, sólo los públicos.

Las clases **Const**, **Declaracion**, **Param** y **Token** modelan las constantes, las declaraciones de procedimientos y funciones, sus parámetros, y las estructuras reconocidas en el código (*tokens*) respectivamente.

La clase **ManejoArchivos** provee métodos que imprimen las etiquetas en el archivo xml y los errores detectados en el archivo de texto. Cuenta con un método por tipo de etiqueta. La clase **LectorTokens** se encarga de identificar los componentes atómicos (caracter, número, palabra, símbolo), llamados *tokens*, de las sentencias del programa. El reconocimiento de estructuras y el chequeo sintáctico del programa se realiza en la clase **Parser**.

La clase **Principal** (Main) maneja la interacción con el usuario mediante línea de comandos, invocando a los métodos de la clase **LectorTokens** para procesar el archivo con el código fuente del programa, y el método *armarGrafo(ArrayList<Token>, BufferedWriter, BufferedWriter)* de la clase **Parser** para realizar el reconocimiento de estructuras y los cheques sintácticos. Este método es el encargado de generar el archivo xml y el archivo de texto con los errores encontrados.

El siguiente fragmento es un ejemplo de documentación JavaDoc de uno de los métodos del programa:

```
/**
 *  Lee de la lista de tokens los tokens correspondientes
 *  a la declaracion de una variable y la guarda en la lista
 * de variables (atributo de la clase)
 *  En caso de encontrar un error se imprime el mismo en el
 *  archivo de errores y se pasa a la siguiente linea
 *
 * @param tokens array con los tokens leidos
 * @param index indice del token actual
 * @param salida archivo en el que se va escribiendo el .xml
 * @param errorOut archivo en el que se van escribiendo los
 *                  errores encontrados
```

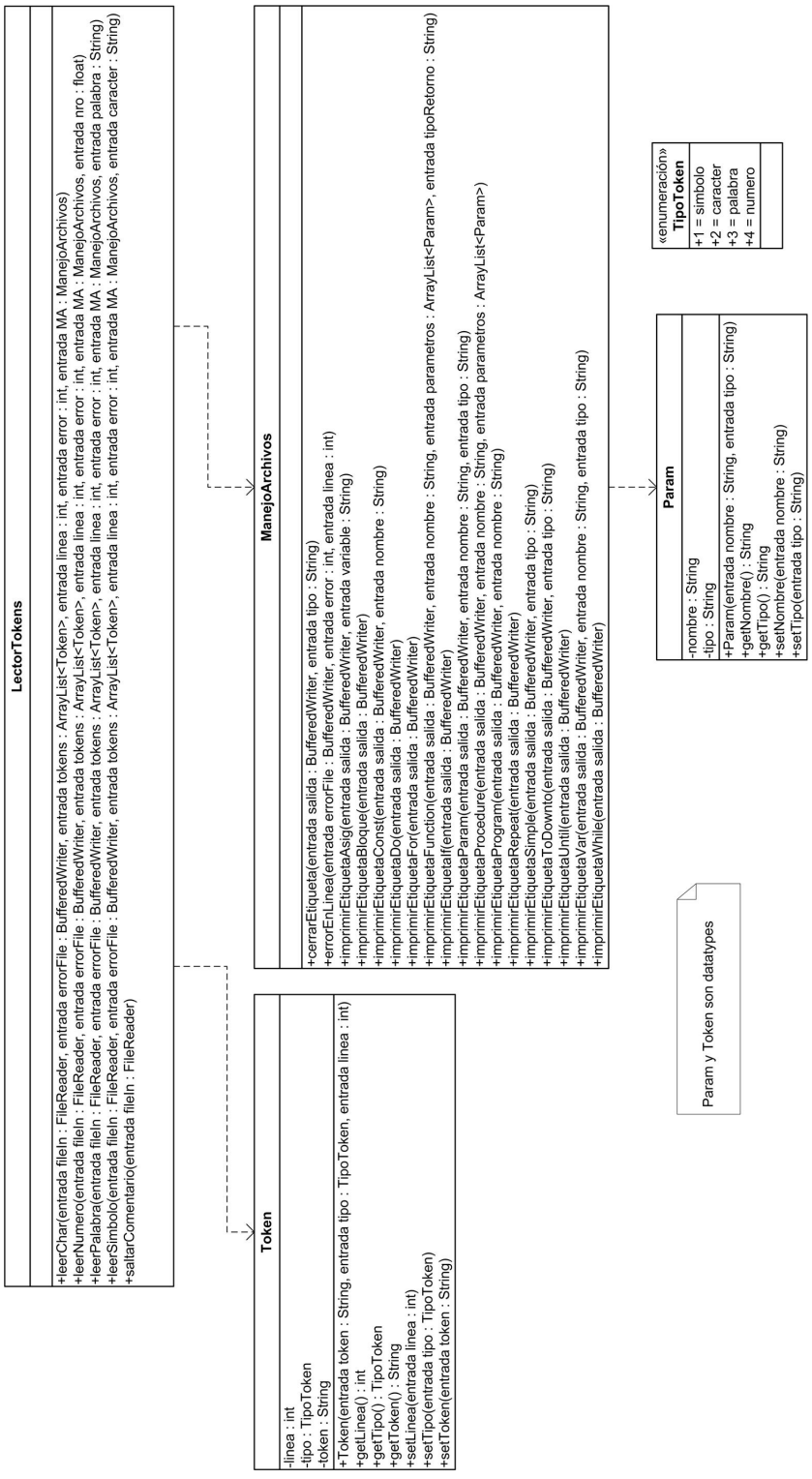

Figura 9.4: Diagrama de Clases del Programa Parser

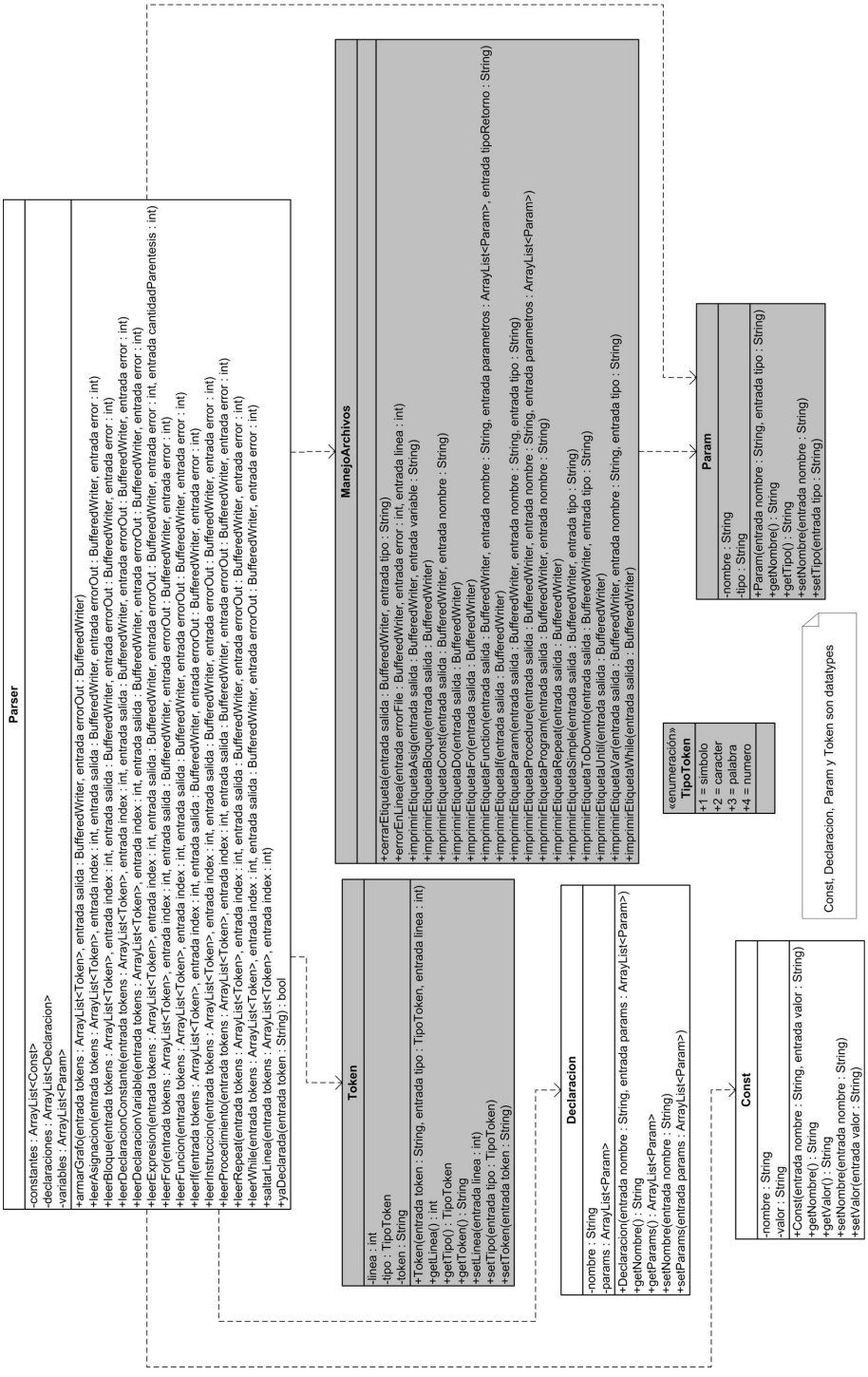

Figura 9.5: Diagrama de Clases del Programa Parser

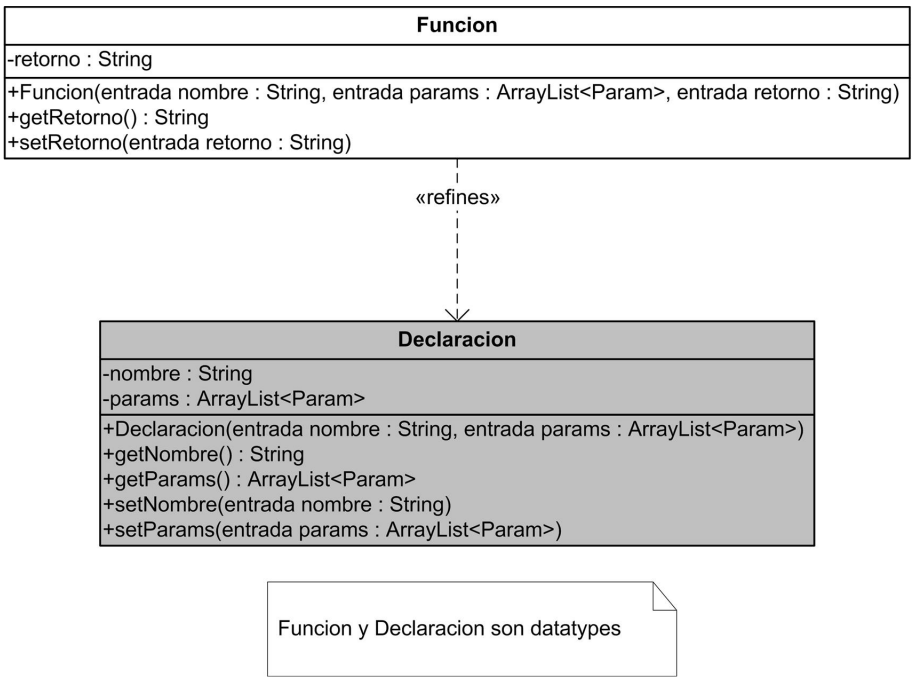

Figura 9.6: Diagrama de Clases del Programa Parser

```
* @param error contador con la cantidad de errores encontrados
*                hasta el momento
*
* @throws IOException
*  Se produce si no se puede escribir en alguno de los
*      archivos de salida
*
* @version 1.0
*/
```

En el documento *Descripción de los Programas* se exponen en detalle los defectos que contiene el método *leerDeclaracionVariable(ArrayList< Token>, int, BufferedWriter, BufferedWriter, int)* de la clase **Parser**. Este método procesa las declaraciones de variables en el código fuente del programa. En caso de encontrar un error se realiza la correspondiente impresión en el archivo de texto que contiene los errores del programa. El resumen de los defectos de todas las clases no se incluye en este caso porque realizar el análisis de todos (272 defectos en total) excede el alcance de este proyecto de grado.

9.4. Métricas de los programas

Las métricas permiten conocer características de los programas, como ser complejidad, tamaño, calidad, etc. Estas características pueden utilizarse, entre otras cosas, para la comparación entre programas. A través de ellas se puede conocer cuál tiene mayor tamaño, complejidad, calidad, etc., y de esa forma

seleccionar el más indicado según las finalidades de quién realiza el estudio.

A continuación se muestran algunas métricas de los programas presentados. La definición de cada una de ellas se encuentra en el cuadro 9.1. Las métricas a nivel de las clases de cada programa pueden consultarse en el documento *Descripción de los Programas*.

Métrica	**Descripción**
Complejidad ciclomática promedio	Se calcula como la suma de la *Complejidad ciclomática* de cada método del programa, sobre #Métodos. La *Complejidad ciclomática* de un método cuenta el número de flujos a través del mismo. Cada vez que ocurre una bifurcación esta métrica se incrementa en uno.
#Clases	Total de *clases* del programa.
#Interfaces	Total de *interfaces* del programa.
#Enumerados	Total de *enumerados* del programa.
Total de líneas de código s/com.	Total de líneas de código del programa, excluyendo las líneas en blanco y los comentarios.
Total de líneas de código c/com.	Total de líneas de código del programa, excluyendo las líneas en blanco (incluye las líneas de comentarios).
Total de líneas de código por método	Total de líneas de código incluidas en el cuerpo de los métodos del programa, excluyendo las líneas en blanco y los comentarios.
Líneas de código prom. por mét.	Se calcula como el *Total de líneas de código por método* sobre *#Métodos*.
#Metodos	Total de métodos del *programa*, incluyendo los estáticos.
#Defectos	Total de defectos del *programa*
#Defectos por línea de código	Se calcula como *#Defectos* sobre *Total de líneas de código s/com.*

Cuadro 9.1: Métricas de Programa

En el cuadro 9.2 se observan la complejidad ciclomática promedio, cantidad de clases, cantidad de interfaces y cantidad de enumerados de cada uno de los programas. En el cuadro 9.3 se observan la cantidad total de líneas de código sin comentarios, con comentarios, dentro de los métodos y promedio por método, la cantidad de métodos, cantidad de defectos y cantidad de defectos por línea de código de cada programa.

Programa	**Complejidad ciclomática promedio**	**#Clases**	**#Interfaces**	**#Enumerados**
Contab.	2,164	12	2	0
Matemático	3,357	13	0	0
MOLatex	1,59	9	1	0
Parser	2,781	9	0	1

Cuadro 9.2: Métricas de los programas

Programa	Líneas de Código				#Métodos	#Defectos	#Defectos por línea de código
	Total s/com.	Total c/com.	Total por método	Prom. por método			
Contab.	1979	3109	1497	9,073	165	107	0,054
Matemático	468	729	375	13,393	28	50	0,107
MOLatex	566	1124	362	5,934	61	32	0,057
Parser	828	1617	634	9,906	64	272	0,329

Cuadro 9.3: Métricas de los programas

Si comparamos las métricas de estos programas con los utilizados en los experimentos de Basili y Selby, Kamsties y Lott, Wood *et al.*, y Juristo y Vegas, presentados en el Capítulo 5 de la Parte I, podemos apreciar que son sensiblemente distintos.

B&S utilizan cuatro programas que presentan diferentes tamaños y cantidad de defectos. Fueron construidos en los lenguajes FORTRAN y Simpl-T. El código fuente no contiene comentarios. Las métricas conocidas de los programas de B&S se presentan en el cuadro 9.4

Programa	Líneas de código	Complejidad ciclomática	#Rutinas	#Defectos
P_1 - text formatter	159	18	3	1
P_2 - mathematical plotting	145	32	8	6
P_3 - numeric data abstraction	147	18	9	7
P_4 - database maintainer	335	57	7	12

Cuadro 9.4: Métricas de los programas de B&S

K&L utilizan tres programas que, a diferencia de B&S, presentan tamaños y cantidad de defectos similares. Fueron construidos en el lenguaje de programación C. Cada programa es un conjunto de funciones de aproximadamente 10-30 líneas, documentadas con un archivo *header* (.h) de aproximadamente 30 líneas. Las métricas conocidas de los programas de K&L se presentan en el cuadro 9.5

Programa	Total de líneas de código	Líneas en blanco	Líneas con comentarios	Líneas sin blancos ni com.	#Defectos
ntree.c	235	38	4	193	6
ntree.h	25	7	1	18	
cmdline.c	245	26	0	219	9
cmdline.h	34	5	1	29	
nametbl.c	251	40	5	207	7
nametbl.h	31	8	6	23	

Cuadro 9.5: Métricas de los programas de K&L

Los programas de los experimentos de Wood *et al.*, y Juristo y Vegas son los mismos utilizados por Kamsties y Lott. Juristo y Vegas utilizaron un programa adicional de su autoría, denominado *Trade*, pero no contamos con información sobre las métricas del mismo.

Como puede observase el tamaño de los programas es medianamente similar entre B&S y K&L, pero sensiblemente distinto a los nuestros. Así lo es también la cantidad de defectos y cantidad de rutinas (métodos en nuestro caso). La

complejidad ciclomática no es comparable debido a que se desconoce si es o no promedio y qué parte del programa se tuvo en cuenta para su cálculo.

Como se describe en el Capitulo 5 de la Parte I, los programas utilizados en los experimentos de B&S y K&L resuelven problemas simples, como la implementación de un tipo de datos, procesamiento de texto, etc. Los programas utilizados en este proyecto de grado resuelven problemas más complejos, de ahí que posean mayor tamaño, cantidad de defectos y complejidad ciclomática.

B&S expresan que los programas utilizados por ellos corresponden a diferentes tipos y reflejan los estilos comunes de programación. Para realizar estas aseveraciones suponemos que debieron tener determinado control sobre el desarrollo de los mismos. También aseveran que los programas contienen una distribución de defectos razonable, como las que ocurren frecuentemente en los programas. Para ello realizaron siembra de defectos en algunos programas. Estos hechos restan "realidad" a la construcción y a los defectos de los programas.

Los programas utilizados por K&L fueron pensados para que su complejidad no dificultara la tarea de verificación. Por ejemplo, se consideran lo suficientemente simples para que el uso de técnicas de verificación de caja blanca sin herramientas de apoyo no se vea perjudicado respecto al uso de las otras técnicas. Además, la mayoría de los defectos han sido sembrados con objetivos específicos, como ser que todos causen fallas observables, que un defecto no oculte a otro, y que las fallas sean provocadas por ciertos datos de entrada.

Nuestros programas poseen los defectos originales de su construcción, no poseen defectos adicionales. Esto impide asegurar que un defecto oculte a otro, y conocer que datos de entrada causarán las fallas. Su construcción no fue controlada, ni dirigida para obtener determinada estructura de código. El modo de construcción de nuestros programas y el origen de sus defectos hace que sean más "reales" que los utilizados en los estudios anteriores, y por lo tanto más representativos de los desarrollados en la industria.

Un análisis ampliado de las diferencias entre los programas utilizados en los estudios realizados anteriormente puede consultarse en [VMBH09].

Parte IV

Conclusiones y Trabajo a Futuro

Capítulo 10

Conclusiones y Trabajo a Futuro

En este capítulo se presentan las conclusiones del estudio realizado y el trabajo a futuro identificado en esta línea de investigación.

10.1. Conclusiones

En el marco de este proyecto de grado se realizó un experimento formal para comparar las técnicas de verificación Inspecciones, Partición de Equivalencia con Análisis de Valores Límite, Tablas de Decisión, Criterio de Cubrimiento de Condición Múltiple y Trayectorias Linealmente Independientes. En el experimento se evalúan la efectividad y el costo de las técnicas respecto a determinados tipos de defectos. Se utilizan dos esquemas para clasificar los defectos: la Clasificación Ortogonal de Defectos de IBM [Cla] y la Taxonomía de Defectos de Beizer [Bei90]

Para la realización del experimento se desarrollaron un conjunto de productos. Algunos de ellos fueron utilizados durante la ejecución del mismo y otros para la preparación previa de los sujetos que participarían en él.

Los productos fueron desarrollados con la intención adicional de permitir la replicación del experimento, o la realización de experimentos similares.

Para la preparación de los sujetos se dictaron una serie de Micro-cursos que introducían y ejemplificaban el uso de las técnicas de verificación a ser utilizadas y de las taxonomías en las que se clasificarían los defectos. Los cursos resultaron efectivos, pero se notó la falta de ejercicios para la realización en clase por parte de los sujetos.

Los conceptos adquiridos en los micro-cursos fueron aplicados en una instancia previa al experimento, denominada *Experiencia Cero*. En esta experiencia ejecutaron las técnicas que utilizarían luego en el experimento. Durante la Experiencia Cero se realizaron devoluciones a los sujetos sobre la forma en la que aplicaron las técnicas y en la que clasificaron los defectos detectados. Esta experiencia fue de gran utilidad para mejorar el manejo de las técnicas y de las clasificaciones con el que los sujetos llegaron al experimento. Una mejora identificada para realizar a esta experiencia es realizar un documento final luego de su aplicación que resuma los errores cometidos por todos los sujetos durante

la aplicación de las técnicas y la clasificación de los defectos. De esta forma se equilibra aún más y se mejora el manejo de las técnicas y las clasificaciones con el que los sujetos enfrentan el experimento.

Para la ejecución del experimento se construyeron tres guías, una para cada tipo de técnica de verificación: lectura de código, dinámica de caja blanca y dinámica de caja negra. Las guías indican cómo debe llevarse a cabo una verificación en la que se aplica determinado tipo de técnica de verificación. Aunque comenzaron a utilizarse en la Experiencia Cero, la adaptación de los sujetos a las guías fue un proceso que duró toda la ejecución del experimento. Para acortar el tiempo de adaptación se identifica como posible mejora el introducir las guías con anterioridad, agregando un micro-curso a los ya existentes.

Para el registro de datos del experimento se utilizó una herramienta web, denominada Grillo. Esta herramienta fue desarrollada expresamente para el experimento. A diferencia de las guías, comenzó a utilizarse con el experimento. Durante la Experiencia Cero los datos se registraron en planillas electrónicas.

La herramienta permite el ingreso de los tiempos empleados en la verificación y los defectos detectados. De cada defecto se registra su ubicación en el código fuente (archivo, programa, etc.), una descripción y su clasificación en las dos taxonomías.

Además de los productos para la realización del experimento, se desarrollaron cuatro programas. Estos programas son los verificados por los sujetos. Fueron implementados en el lenguaje de programación Java y documentados con JavaDoc. Presentan tamaños, complejidad y cantidad de defectos distintos. Los defectos que presentan son los originales de su construcción. No poseen defectos detectables por un compilador. Se consideran programas "reales", porque no poseen defectos ficticios, resuelven problemas de relativa complejidad y su proceso de construcción no fue manipulado. En contraste con los programas utilizados en otros estudios que siguen esta línea de investigación, que poseen aspectos "artificiales", como ser inyección de defectos adicionales en el código, resuelven problemas simples que no eleven la complejidad de la verificación, y/o controlan su construcción para que la estructura del código no perjudique la aplicación de técnicas de caja blanca.

El experimento fue ejecutado por 14 estudiantes que aplicaron las cinco técnicas de verificación sobre los cuatro programas.

Existen muchas perspectivas desde las cuales observar los estudios empíricos de las técnicas de verificación. Un aspecto clave del trabajo presentado, especialmente desde la perspectiva de un *experimentador*, es el uso de una metodología de experimentación y un diseño formal. Los resultados del experimento, que se resumen a continuación, pueden ser utilizados para refinar las teorías de un *investigador* acerca de la verificación de software, o para orientar la aplicación de las mismas por parte de un *profesional*.

A partir del análisis de datos se obtuvo que:

- Tablas de Decisión es más efectiva que Trayectorias Linealmente Independientes con un 99 % de confianza
- Partición de Equivalencia y Análisis de Valores Límite es más efectiva que Trayectorias Linealmente Independientes con un 93,4 % de confianza

Los resultados sugieren que las técnicas Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite (técnicas dinámicas de caja negra)

son más efectivas en la detección de defectos que la técnica Trayectorias Linealmente Independientes (técnica dinámica de caja blanca).

Trayectorias Linealmente Independientes fue ejecutada por 6 sujetos, mientras que Tablas de Decisión y Partición de Equivalencia con Análisis de Valores Límite fue ejecutada por 7 sujetos. El tamaño reducido de observaciones por técnica provoca que el agregado de una observación a una de las muestras pueda variar los resultados obtenidos.

La replicación del experimento realizado, con un número mayor de observaciones por técnica, es necesaria para reforzar y validar estos resultados.

Respecto al costo de las técnicas, dada la variabilidad en tamaño, complejidad y cantidad de defectos de los programas, el costo de cada técnica fue estudiado individualmente para cada programa.

Según el diseño del experimento, cada técnica sería ejecutada dos veces sobre cada programa. Lo que da 2 observaciones para cada técnica por programa. Algunas de las verificaciones dictadas por el diseño no fueron ejecutadas, provocando que algunas técnicas fueran aplicadas sólo una vez sobre algunos programas.

Además que las muestras son pocas, en la mayoría de los casos ocurrió que las dos observaciones de las técnicas para un mismo programa son considerablemente distintas. Este escenario nos impide siquiera aventurar una tendencia en el costo de las técnicas de verificación.

La técnica Inspecciones fue la única que se comportó de manera similar por programa. Esto permitió observar que se vuelve más costosa a medida que crece el tamaño del programa.

Fuera de los resultados obtenidos cabe realizar algunas observaciones sobre la comparación de la efectividad-costo de estas técnicas de verificación. Cuando se estudia el esfuerzo asociado con una técnica, se debe diferenciar lo que es el costo de la detección y el costo del aislamiento del defecto. Los sujetos que aplican lectura de código detectan un defecto al mismo tiempo que lo aislan. Las técnicas de verificación dinámicas sólo permiten detectar la existencia de fallas; por lo que los sujetos que las aplican deben buscar en el código fuente y dedicar un esfuerzo adicional para aislar los defectos que las provocan.

El trabajo presentado en este documento difiere de los estudios anteriores en varios aspectos:

- Es el primer experimento que estudia la efectividad de las técnicas de verificación Inspecciones, Tablas de Decisión y Trayectorias Linealmente Independientes. Este es un aporte fundamental ya que sienta las bases para posteriores investigaciones centradas en estas técnicas.
- Es el primer estudio que utiliza cinco técnicas de verificación. Duplicando la cantidad de técnicas de caja blanca y de caja negra de los estudios anteriores.
- Los programas utilizados poseen tamaños y complejidades mayores, y resuelven problemas más cercanos a los existentes en la industria.
- Los defectos existentes en los programas son los remanentes de su construcción, y no provienen de medios artificiales como la siembra de defectos.

El estudio empírico presentado establece una línea base de productos que pueden ser utilizados para la realización y replicación de experimentos vincula-

dos con esta línea de investigación. Los resultados presentados fueron calculados a partir de las verificaciones de un conjunto de individuos con un nivel de experiencia principiante que aplicaron cinco técnicas de verificación a programas de tamaño y complejidad medianos y variables. La extrapolación directa de los resultados a otros entornos de prueba no está implícito.

10.2. Trabajo a Futuro

De la investigación realizada se desprenden, entre otras, las siguientes líneas de trabajo:

- Realizar replicaciones del experimento para reforzar los resultados, en el mismo contexto y en otros.
- Realizar experimentos relacionados con la efectividad y el costo de otras técnicas de verificación, teniendo en cuenta también otros aspectos como lo son el tipo de programa, nivel de experiencia de los sujetos, etc.
- Estudiar si existe una relación entre los tipos de defecto que tiene un programa y las métricas que presenta el programa.
- Estudiar si las clasificaciones de defectos utilizadas fueron adecuadas.
- Realizar la clasificación de los defectos de los programas en las taxonomías de IBM y Beizer.

La replicación del experimento es fundamental para dar respaldo empírico a los resultados obtenidos. Para ello se cuenta con los productos utilizados y la descripción que se realiza del experimento en el presente informe. Para poder realizar el análisis de costos sería conveniente utilizar sólo uno de los programas desarrollados. La diferenciación de la efectividad de las técnicas de verificación por tipo de defecto requiere que los defectos sean clasificados de antemano por los experimentadores. Dejar esta tarea a los sujetos puede desembocar en el resultado obtenido en este experimento: múltiples clasificaciones para el mismo defecto.

Con respecto a los productos desarrollados, se distingue un aspecto prioritario a atacar: el desarrollo de funcionalidades de importancia en la herramienta de registro Grillo. Estas funcionalidades incluyen la implementación de consultas y la distinción de qué registros de defectos representan el mismo defecto.

Bibliografía

[Apa09] C. Apa. *Diseño y Ejecución de un Experimento con 5 Técnicas de Verificación Unitaria*. PhD thesis, Facultad de Ingeniería. Universidad de la República., 11 2009. 9

[Bei90] Boris Beizer. *Software Testing Techniques 2nd Edition*. International Thomson Computer Press, 2 sub edition, 6 1990. 1.3, 4, 4.2, 6.2, 10.1

[BS87] V.R. Basili and R.W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, 1987. 1.3, 5, 5.1, 5.3, 7.1

[Cla] IBM Research: Orthogonal Defect Classification. `http://www.research.ibm.com/softeng/ODC/ODC.HTM`. 1.3, 4, 6.2, 10.1

[dAO] Test de Alternativas Ordenadas. `http://www.dm.uba.ar/materias/optativas/metodos_no_parametricos/2004/2/NoparI09L.pdf`. 7.1

[FP98] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. PWS Publishing Co., 1998. 2

[GV09] F. Grazioli and D. Vallespir. Marco teórico para evaluar taxonomías de defectos. Technical Report RT 09-17, PEDECIBA, Instituto de Computación – Facultad de Ingeniería, Universidad de la República, 11 2009. 4.2

[Hua09] J. C. Huang. *Software Error Detection through Testing and Analysis*. Wiley, 5 2009. 3

[JM03] N. Juristo and A.M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic, 2003. 2

[Jor08] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition*. AUERBACH, 3 edition, 2 2008. 3

[JV03] Natalia Juristo and Sira Vegas. Functional Testing, Structural Testing, and Code Reading: What Fault Type Do They Each Detect? In Reidar Conradi and Alf Inge Wang, editors, *ESERNET*, volume 2765 of *Lecture Notes in Computer Science*, pages 208–232. Springer, 2003. 1.3, 5, 5.4, 7.1

[KL95] Erik Kamsties and Christopher M. Lott. An empirical evaluation of three defect-detection techniques. In *Proceedings of the Fifth European Software Engineering Conference*, pages 362–383, 1995. 1.3, 5, 5.2, 5.3, 7.1

[Mye04] Glenford J. Myers. *The Art of Software Testing, Second Edition.* Wiley, 2 edition, 6 2004. 3

[She03] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures, Third Edition.* Chapman & Hall/CRC, 3 edition, 8 2003. 2

[VCARH09] D. Vallespir, S. De León C. Apa, R. Robaina, and J. Herbert. Effectiveness of Five Verification Techniques. In *Proceedings of the XXVIII International Conference of the Chilean Computer Society (2009)*, Santiago de Chile (Chile), 11 2009. 1.5

[VH09] D. Vallespir and J. Herbert. Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. In *Proceedings of the Mexican International Conference in Computer Science (2009)*, Ciudad de México (México), 9 2009. 8.4, 9

[VL08] D. Vallespir and S. De León. Análisis y Ejemplos de la Taxonomía de Defectos de Beizer. Technical Report RT 08-19, PEDECIBA, Instituto de Computación – Facultad de Ingeniería, Universidad de la República, 12 2008. 1.5, 4.2

[VMBH09] D. Vallespir, S. Moreno, C. Bogado, and J. Herbert. Towards a Framework to Compare Formal Experiments that Evaluate Testing Techniques. *Research in Computing Science*, 2009. 9.4

[WRBM97] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and Combining Software Defect Detection Techniques: a Replicated Empirical Study. In M. Jazayeri and H. Schauer, editors, *Proceedings of The Sixth European Software Engineering Conference / Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 262–277. Lecture Notes in Computer Science, 1997. 5, 5.3, 5.4

[WRH+00] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction.* Kluver Academic Publishers, 2000. 1.3, 2