USO DE MÉTODOS ESTADÍSTICOS PARA VALIDACIÓN DE SOFTWARE

SONIA VICTORIA VIVAS PRIETO

BOGOTA D.C.
UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
ENERO DE 2007

USO DE MÉTODOS ESTADÍSTICOS PARA VALIDACIÓN DE SOFTWARE

SONIA VICTORIA VIVAS PRIETO

Trabajo de grado presentado como requisito
para optar al titulo de
Maestro en Ingeniería

Director: RODRIGO CARDOSO
Profesor Asociado

BOGOTA D.C.
UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
ENERO DE 2007

CÓMO LEER ESTE DOCUMENTO **3**

INTRODUCCIÓN **5**

1.1 OBJETIVOS 5
1.2 ORGANIZACIÓN DEL DOCUMENTO 6

ANTECEDENTES Y TEMAS RELACIONADOS **7**

2.1 ASEGURAMIENTO DE CORRECCIÓN 8
2.1.1 Verificación *8*
2.1.1.1 Verificación formal 9
2.1.1.2 Pruebas (*Testing*) 10
2.1.2 Corrección normal *21*
2.2 HERRAMIENTAS 21
2.2.1 JUnit: Pruebas unitarias *21*
2.2.2 JML: Lenguaje de especificación y prueba *22*
2.3 ELEMENTOS DE ESTADÍSTICA 23
2.3.1 Algunos conceptos *24*
2.3.1.1 Muestreo 24
2.3.1.2 Intervalos de Confianza 25
2.3.1.3 Distribuciones probabilísticas 26
2.3.1.4 Datos para aplicaciones de informática 29

VALIDACIÓN ESTADÍSTICA **31**

3.1 LA IDEA FUNDAMENTAL 32
3.2 UNA METODOLOGÍA DE VALIDACIÓN 34
3.2.1 Validación estadística de métodos *36*
3.2.1.1 Modalidades de prueba 36
3.2.1.2 Determinación del ambiente de prueba 37
3.2.1.3 Generación de datos de entrada 38
3.2.1.4 Distribuciones de probabilidad para datos 39
3.2.1.5 Generación de datos aleatorios con una distribución dada 41
3.2.1.6 Número de datos de la muestra 47
3.2.2 Validación estadística de clases *49*
3.2.2.1 Invariante de clase 49
3.2.2.2 Planes de prueba: Integración 50
3.2.3 Validación estadística de productos de software *50*
3.2.3.1 Planes de prueba: Sistema 51
3.2.4 Reutilización de pruebas *52*
3.2.4.1 Almacenamiento de pruebas 52
3.2.4.2 Regresión 54

JSVT: UNA HERRAMIENTA PARA VALIDACIÓN ESTADÍSTICA **57**

4.1 OBJETIVOS Y ALCANCE 58
4.2 REQUERIMIENTOS 58
4.3 ORGANIZACIÓN DEL SOFTWARE 59
4.3.1 Anotación de especificaciones *59*
4.4 ARQUITECTURA DE JSVT 60
4.4.1 Librería de distribuciones *60*
4.4.2 Kernel *62*
4.4.3 Interfaz *63*
4.5 VALIDACIÓN DE MÉTODOS CON JSVT 64
4.6 CÓMO SE IMPLANTÓ LA VALIDACIÓN ESTADÍSTICA DE MÉTODOS 65
4.6.1 Generación de datos *65*
4.6.2 Definición del ambiente de prueba *66*
4.6.3 Ejecución de los casos de prueba *66*
4.6.3.1 Uso de jmlrac 66

4.6.3.2 Uso de JVM para oráculos 66
4.6.4 Reutilización de las pruebas 67
4.6.5 Pruebas para JSvt 67

CONCLUSIONES **68**

BIBLIOGRAFÍA **71**

ANEXO A **74**

JML **74**

INSTALACIÓN 74
LO NECESARIO DE JML PARA ESTE TRABAJO 74
HERRAMIENTAS 76
DESCRIPCIÓN GRAMATICAL 76

ANEXO B **78**

MANUAL DE USUARIO PARA JSVT **78**

COMO USAR JSVT 78
Instalación 79
Generación de una nueva prueba 79
Modificación del classpath 80
Definición de las distribuciones para los parámetros 81
Correspondencia de parámetros 86
Definición de la confianza 86
Definición del ambiente de prueba 87
En espera de los resultados 88
En la ventana de resultados 89
Guardar una prueba 89
Pruebas JUnit-JML 89
Regresión 90

ANEXO C **91**

EJEMPLOS DE USO DE JSVT **91**

EJEMPLO 1. EL BANCO. 91
EJEMPLO 2. EL BARCO PIRATA 94
EJEMPLO3. EL BOSQUE 98

Cómo leer este documento

Este documento está planeado de manera que se pueda leer en diferentes niveles de profundidad según el interés del lector.

Hay cuatro niveles de profundidad principales. El primero considera la lectura de la introducción y las conclusiones, lo que ofrece al lector una visión global del documento, exponiendo los objetivos, alcances y posibilidades de trabajo futuro de la investigación. Un segundo nivel de lectura involucra los resúmenes de cada capítulo, donde se explica brevemente la temática del mismo. Un tercer nivel de detalle esta en la lectura de las introducciones a los capítulos, que ofrecen una descripción de los temas que se van a considerar dentro de ellos.

El último nivel considera la lectura completa de los capítulos, según el interés del lector. El documento está organizado en tres capítulos principales (además de la introducción y conclusiones), el primer capítulo considera los conceptos necesarios para entender los capítulos siguientes, el segundo capítulo explica la propuesta de validación que se hace en esta tesis y el tercero muestra la forma en la que dicha propuesta se implementó.

Capítulo 1

Introducción

Ante la presencia generalizada de software en casi todas las tareas cotidianas, es necesario pensar en la competitividad y por tanto en la calidad de los productos entregados. Las pruebas de software son útiles para detectar defectos en la etapa de producción, pero dado que con alta certeza debe garantizarse la corrección del producto finalizado, es necesario utilizar otros métodos como la verificación formal o chequeos a profundidad, buscando posibles problemas en el software, ambas muy costosas. Este es un proceso largo, y a veces es imposible revisar todos los posibles escenarios.

Puede parecer redundante o innecesario hacer pruebas exhaustivas de un programa de computador, cuando estos programas pueden ser simplemente probados por usuarios y los defectos corregidos antes de comercializar el producto. De hecho, es usual ver errores en los programas de computador que se utilizan diariamente. Sin embargo, es claro que en muchos casos un sistema inestable puede causar daños irreparables. Pero revisar todos los posibles estados puede llegar a ser imposible. Entonces, la pregunta crucial es qué método de verificación nos puede dar una alta certeza con bajos costos. El primer paso es, por supuesto, revisar el estado del arte de la verificación y validación de software y utilizar ese conocimiento para obtener el resultado deseado.

Este problema es el centro de este proyecto de tesis, que en general busca desarrollar software de apoyo para el proceso de verificación de software, utilizando pruebas y métodos estadísticos para garantizar, en alguna medida, la calidad, de modo que sea posible aproximarse a la corrección.

1.1 Objetivos

El objetivo principal de esta investigación es proponer una metodología de validación de software que permita asegurar la corrección de un producto a través de pruebas. La metodología que se proponga debe resolver el problema de la inseguridad de las pruebas, sin alcanzar los costos en tiempo y recursos que tiene la verificación formal.

La propuesta considera la estadística como centro del aseguramiento de corrección, observando distribuciones e intervalos de confianza como base de dicho aseguramiento, mejorando la seguridad del proceso de pruebas.

El objetivo principal se llevará a cabo a través de este documento y un producto de software que exploran e implementan dicha metodología.
Este objetivo puede observarse como la combinación de los siguientes objetivos específicos:

- Considerar un proceso de verificación que debe permitir:
  - Diferentes modalidades de prueba.
  - Reutilización de pruebas.
  - Revisión de la precondición, la poscondición e invariantes de métodos y clases.
  - Definición de la confianza con la que se quiere realizar la prueba, y con ello la cantidad de casos de prueba necesarios.
  - Definición de los elementos necesarios para generar casos de prueba.
  - Ejecución de los casos de prueba y guardar su resultado.
  - Prueba de métodos, clases y productos de software.
  - Generación de pruebas en JUnit como apoyo al proceso de pruebas realizado.

1.2 Organización del documento

Este documento empieza por describir brevemente los antecedentes y conceptos de verificación y validación de productos de software y de estadística, que serán necesarios a lo largo de este documento. En el capítulo 2 se hace un recuento de las ideas de ingeniería de software que se utilizan para estructurar la metodología que se va a proponer.
El capítulo 3 hace referencia a la metodología propuesta. Se define el término validación estadística y se expone la idea de realizar pruebas utilizando la generación de datos aleatorios. Se discute la posibilidad de generar transformaciones sobre las funciones de distribución, y su impacto en la definición de dichas funciones sobre los parámetros.

El capítulo 4 describe JSvt, una herramienta de validación estadística basada en la propuesta del capítulo anterior. En este capítulo se describen la forma en la que se implementó la metodología, los objetivos y alcances de la herramienta, junto a su arquitectura y su diseño.

Finalmente, se analizan los logros conseguidos, exponiendo los avances alcanzados. También se discuten las posibilidades de trabajo futuro, y se consideran las posibilidades de aplicación de este trabajo a la cotidianidad de los procesos de desarrollo de software.

Capítulo 2

Antecedentes y temas relacionados

Resumen. Los procesos de pruebas como parte del aseguramiento de calidad son definidos en este capítulo, donde se discuten los puntos importantes en el proceso de generación de pruebas, como lo son los tipos de pruebas posibles, la forma en la que se generan los datos de entrada, las métricas mas utilizadas, la forma en la que se definen los planes de prueba, y específicamente, se discute como las características de la programación orientada a objetos afecta estos planes. La segunda parte de este capítulo discute brevemente la herramientas existentes para la realización de pruebas unitarias, específicamente se explora JUnit, ya que esta herramienta será utilizada a lo largo del proyecto. Se habla también de JML, un lenguaje de modelado, y la forma en la que este lenguaje se puede utilizar para definir especificaciones, facilitando la generación de pruebas de funcionamiento. Finalmente se discuten conceptos de estadística, como el muestreo aleatorio, los intervalos de confianza y las distribuciones de probabilidad. Estos conceptos se exponen brevemente con el objetivo de familiarizar al lector con ideas que serán utilizadas mas adelante.

El *aseguramiento de la calidad* consiste en un conjunto de actividades que aseguran que los procesos de software (es decir todas las actividades relacionadas con el diseño, desarrollo, expansión y mantenimiento del software) y los productos (es decir software, datos asociados, documentación) cumplen los requerimientos, estándares y procedimientos. [NAS2007]

Todo proceso de software incluye actividades de aseguramiento de calidad. La forma en la que dicho aseguramiento se lleva a cabo, se basa en una evaluación completa del proyecto, centrada en los riesgos que se puedan generar durante su desarrollo y en el ambiente en que se va a usar. El aseguramiento de calidad hace parte de todo desarrollo de software, aun si no se planea como tal, ya que cada programador debe probar su trabajo de acuerdo a los estándares con los que se planteó el proyecto.

Un proceso de aseguramiento de calidad bien planeado y ejecutado minimizará los errores y los costos derivados de los mismos. La formalidad de dicho proceso dependerá del presupuesto del proyecto y la relación de costo de un error contra el beneficio de no tenerlos. [NAS2007]

Este documento se centra en el aseguramiento de calidad restringido a la corrección, es decir que la calidad del producto se medirá únicamente en función a la cantidad de errores que tenga (aunque es claro que solo un error ya implica que el producto no es correcto). A este proceso se le denominará *aseguramiento de la corrección*. Este proyecto considerará este proceso a través de herramientas de corrección de software y herramientas estadísticas.

En este capítulo se describirán brevemente los conceptos que son relevantes en el desarrollo de este proyecto, divididos en tres partes principales: corrección de software, herramientas y estadística.

2.1 Aseguramiento de corrección

Como se definió anteriormente, el término *aseguramiento de corrección* será utilizado para referirse al proceso de aseguramiento de calidad restringido únicamente a la corrección del producto.
Los procesos de corrección aseguran que el software que está siendo desarrollado o modificado va a satisfacer los requerimientos, y que cada paso del proceso de construcción genera los productos correctos. Estos procesos están planeados como una evaluación sistemática y técnica del software y los productos asociados al desarrollo. [NAS2007]

La ingeniería de software sugiere que los procesos de corrección deben realizarse al final de cada fase del proceso de desarrollo, de manera que se asegure que los requerimientos están bien determinados (correctos, suficientes, consistentes, claros y realistas), y que el producto resultado de la fase satisface estos requerimientos. [BRU2002]

Al proceso de corrección se le da el nombre de proceso de *validación y verificación* (V & V). Aunque hay diversas definiciones para estos términos, se puede englobar la variedad en las definiciones dadas por los estándares:
La IEEE define *verificación* como la revisión de que el resultado de un a fase de desarrollo satisface los requerimientos de esa fase [IEE1990].
La ESA considera la *verificación* como el acto de revisar, inspeccionar, probar, chequear, auditar o establecer y documentar si los ítems, procesos, servicios y documentos satisfacen los requerimientos especificados [ESA1991].

Según la ANSI/IEEE, la definición de *validación* es el proceso de evaluar un sistema o componente durante o al final del proceso de desarrollo para determinar si este satisface los requerimientos especificados [IEE1990]. Puede decirse que la validación es verificación 'de fin a fin'. [ESA1995]

La *confiabilidad* del software se define como la probabilidad de que un sistema de software no causará la falla del sistema durante un tiempo especificado bajo condiciones dadas [IEE1989].

2.1.1 Verificación

Las actividades de verificación incluyen [ESA1995]:

- Revisiones técnicas e inspecciones al software
- Revisar que los requerimientos del software representan los requerimientos del usuario.
- Revisar que los componentes de diseño representan los requerimientos del software.

- Pruebas unitarias.
- Pruebas de integración.
- Pruebas de sistema.
- Pruebas de aceptación.
- Auditoria.

Las actividades de verificación suelen incluir llevar a cabo pruebas formales [ESA1995]. Las actividades que deben ser conducidas en un proyecto se describen en el plan de verificación y validación de software (SVVP, por sus siglas en inglés) [IEE1998].

2.1.1.1 Verificación formal

La verificación formal consiste en probar matemática y lógicamente que el código representa la especificación. La verificación formal es un proceso que incrementa la calidad, pero tiene un alto costo. Solo es eficiente respecto a costos si los costos de una falla en el software son muy altos [HOL2001].

Las pruebas formales demuestran que todas las entradas que cumplan un estado inicial específico (precondición) resultarán en un estado final definido (poscondición). Es decir que una fase de desarrollo es correcta, cuando en su resultado no se da la posibilidad de que una entrada que cumpla la precondición, genere un comportamiento inesperado en la poscondición. [PER2000] denomina estos comportamientos como *defectos*, y así se hará referencia a ellos en adelante.

De este modo, es necesario especificar los requerimientos de forma matemática para poder hacer verificación formal. Una de las formas más conocidas para realizar especificaciones formales son las triplas de Hoare[CAR1991].
Una tripla de Hoare es una fórmula lógica $\{Q\}S\{R\}$ donde Q es una aserción que representa la precondición, S es el programa escrito en el lenguaje de comandos guardados de Dijkstra y R es la aserción que representa la poscondición. Esta es la notación que se utilizará en este documento para definir las especificaciones de los requerimientos.

Con la especificación generada de esta manera, se puede definir el término corrección formalmente [CAR1991]. Se considera la definición de reglas de inferencia que describen la ejecución de cada una de las instrucciones básicas del lenguaje de programación utilizado, y se dice que un programa es correcto respecto a su especificación cuando se demuestra, utilizando las reglas de inferencia, que cuando sea ejecutado el programa, producirá al final un estado que cumple la poscondición para cualquier entrada que cumpla la precondición.

Otro enfoque de la verificación formal es la exploración exhaustiva: se hace uso de una buena especificación para probar el producto a través de la ejecución de pruebas que consideren todas las entradas posibles, de modo que será correcto si en todos los casos se produce la salida esperada. Aunque el número de casos posibles es finito, es muy grande, por lo que esta solución es excesivamente costosa. Una de las soluciones a este último problema es el chequeo de modelos, que se describirá más adelante.

El principal problema de la verificación formal está en la necesidad de expresar los requerimientos y diseños del software en una forma matemática, y la dificultad que esto representa. Otra de las dificultades de la verificación formal es la necesidad de destrezas lógicas que la mayoría de los programadores no domina, y aun si lo hiciesen, la verificación es una práctica complicada, que está sujeta a errores que arruinan la confianza de corrección. En el caso de la programación orientada a objetos, la verificación formal presenta como problema la necesidad de representar conceptos complejos como herencia, excepciones, interfaces, que son un reto tanto para representarlos formalmente como para verificarlos [QUI2006]
Algunas de las áreas en las que los métodos formales han sido exitosos son los protocolos y los sistemas seguros. [ESA1995]

2.1.1.1.1 Chequeo de modelos (*Model Checking*)

Este es un método para verificar algorítmicamente sistemas formales. El primer paso consiste en generar un modelo formal del software basado en su diseño y el proceso de chequeo consiste en verificar que este modelo satisface una especificación formal. La especificación es descrita utilizando formulas de lógica temporal. [CLA1999]
El modelo es expresado como un sistema de transición que consta de nodos y arcos, un conjunto de proposiciones atómicas se asocia a cada nodo y éstos representan los estados del sistema. Los arcos representan las posibles ejecuciones que alteran el estado, mientras que las proposiciones atómicas representan las propiedades básicas que se mantienen en un momento dado de la ejecución. [CLA1999]
Formalmente, el problema puede ser definido como: dada una propiedad, expresada con la formula de lógica temporal `p`, y un modelo `M` con estado inicial `s`, decida si `M,s |= p.` [CLA1999]
Las herramientas de *Model Checking* tienen el problema de la explosión combinatoria del espacio de estados, que debe ser resuelta si se quiere resolver la mayoría de los problemas de la vida real. [CLA1999]

La generación del modelo formal es un proceso complicado. En el caso especifico de este proyecto, se presentan problemas con las características de la programación orientada a objetos y problemas de costos en materia de recursos, por lo que el chequeo de modelos no será utilizado.

2.1.1.2 Pruebas (*Testing*)

Debido a los problemas que tiene la verificación formal, y al presupuesto limitado de la mayoría de los proyectos, la corrección suele afirmarse seleccionando un conjunto de casos de prueba, con la esperanza de que este conjunto represente el total de las posibilidades, de modo que se puedan observar todos los posibles comportamientos de todos los posibles datos en el conjunto elegido. Esto, sin embargo, es solo una aproximación a la corrección,

ya que al no verificar todos los posibles casos, no se puede asegurar la corrección del producto.
El ideal es tener un mínimo de casos de prueba para obtener un nivel de confianza aceptable.
Es claro que las pruebas no son tan confiables como la verificación formal, pero en muchos casos las pruebas son la única forma de corrección que se ajusta a los presupuestos de dinero, tiempo y recursos.
Una prueba es un intento sistemático de localizar errores en forma planeada en el software implementado [BRU2002]. Por lo tanto, probar es operar el software con entradas reales o simuladas, bajo condiciones específicas, para demostrar que un producto satisface sus requerimientos, y si no lo hace, para identificar dónde hay problemas. Cuando alguna parte de la especificación no pueda ser verificada con una prueba, se debe considerar otra técnica en el plan de pruebas, como validación de requerimientos, validación de especificaciones, inspecciones de diseño e inspecciones de código [DUI2004] [NAS2007][ESA1995].
De aquí se deriva la más general de las clasificaciones de pruebas: estáticas y dinámicas. La diferencia principal reside en que las pruebas estáticas no requieren la ejecución del software mientras que las dinámicas si [BRU2002].

En este documento se utiliza la palabra *prueba* para referirse a una prueba dinámica, es decir a la ejecución del programa con una entrada válida, cuyo resultado se valida con respecto a unos resultados esperados. En este punto es importante resaltar que el resultado de una ejecución de prueba debe ser comparable contra el de un *oráculo*, de modo que sea posible validarlo. [1]
La forma en la que se evaluará el software debe ser definida durante el diseño, no cuando la codificación esté completa. El diseño debe ser cambiado hasta que sea posible probarlo. [ESA1995]

El principal problema de las pruebas radica en el centro de su definición: las pruebas se construyen para probar que el sistema tiene errores, no para demostrar que no los tiene.
"Probar puede mostrar la presencia de errores, pero no su ausencia"(Dijkstra) [KOU2000]
Otro problema está en la necesidad de planear y diseñar las pruebas de modo que sea posible asegurar la corrección del programa. Ya que las pruebas se hacen justamente para no tener que explorar todos los caminos, el diseño del plan de pruebas debe considerar las necesidades del programa y el problema radica en ese plan.
"Los probadores no son mejores en diseño de las pruebas que los programadores en el diseño del código". [KOU2000]

A pesar de los problemas que presentan las pruebas, siguen siendo la forma con mayor viabilidad para aproximarse a demostrar la corrección de un programa, por lo que este trabajo busca determinar una metodología de prueba que mejore la confiabilidad del proceso de pruebas.

---

[1] Un *oráculo* es una salida de una ejecución, que se considera correcta. Puede ser obtenido con especificaciones formales, versiones anteriores del programa, programas que hacen lo mismo en forma diferente, ejecuciones anteriores de la prueba, etc.

Debido a que se hará uso de las pruebas como centro del aseguramiento de la corrección, se considera importante exponer brevemente algunos conceptos significativos sobre pruebas.

Pruebas de caja negra. Las *pruebas de caja negra* son un método en el que los datos de la prueba se derivan de los requerimientos funcionales, sin considerar la estructura final del programa [PER1990].

El nombre del método se refiere a la forma en la que se trata al programa en el momento de la prueba: se considera que es una caja negra, de la que solo es posible conocer su especificación, sus entradas y sus salidas. La funcionalidad se infiere al observar cada salida respecto a su respectiva entrada.
La validación en este caso, suele estar sujeta a la especificación, de modo que las salidas se comparan con la especificación y de este modo se verifica la corrección. Los detalles de implementación no son de interés cuando se trabaja con este tipo de pruebas.

El principal problema de las pruebas de caja negra esta en la necesidad de una *buena* especificación, es decir una especificación que no de pie a ambigüedades y considere todos los detalles necesarios. La mejor forma de asegurar que no haya ambigüedades es evitar usar el lenguaje natural, y en cambio utilizar lenguaje formal; esto, sin embargo, no asegura que la especificación sea completa. Los problemas de especificación son aproximadamente el 30 % de todos los defectos del software [BEI1995]

La selección de datos de entrada juega un papel importante en la calidad de la prueba, pues como no es viable revisar todos los datos posibles, el conjunto de datos que se seleccione debe ser lo suficientemente bueno como para representar a la totalidad de los datos.

Pruebas de caja blanca. En el método de *pruebas de caja blanca* se usa una perspectiva interna del sistema para diseñar los casos de prueba. Contrario a las pruebas de caja negra, la estructura, el flujo y la implementación del software son parte importante de la prueba. El plan de prueba se genera alrededor de los detalles de implementación, como el lenguaje de programación y la lógica. Los casos de prueba se derivan de la estructura del programa.

Una idea fuerte al usar pruebas de caja blanca es la de tratar exhaustivamente un aspecto del software, logrando diferentes niveles de *cubrimiento* (líneas de código, arcos, caminos, condiciones).

Una ventaja de este tipo de prueba sobre las pruebas de caja negra es que es posible determinar el flujo de los datos durante la ejecución, haciendo posible la localización de código innecesario.

El problema de este tipo de prueba es que es necesario que el probador tenga total conocimiento del diseño y la codificación del software, lo que puede implicar una falta de objetividad en los casos de prueba.

La selección de datos de entrada depende de la especificación y los caminos que se quiera ejercitar, pero usualmente se utiliza la generación aleatoria de datos [PAN1999].

Pruebas en el proceso de desarrollo de software

Una forma de probar es haciendo uso de la organización del proceso de desarrollo de software. De este modo se hacen pruebas para cada componente desarrollado, para la integración de esos componentes y para el sistema completo.

Pruebas unitarias. Estas pruebas tratan de encontrar defectos en los objetos participantes o en los subsistemas, o en ambos, con respecto a la especificación de los requerimientos. [BRU2002]

Pruebas de Integración. Estas pruebas están relacionadas con la localización de defectos cuando se prueban juntos componentes que se han probado de manera individual.

Pruebas de estructura de sistema. Estas pruebas son la culminación de las pruebas de integración, en las que se prueban todos los componentes del sistema integrados.

Pruebas de Sistema. Estas pruebas consideran todos los componentes juntos, vistos como un sistema, para identificar errores con respecto a los objetivos del sistema.

Pruebas funcionales. Prueban los requerimientos funcionales del sistema, y si esta disponible el manual de usuario.

Pruebas del desempeño. Revisan los requerimientos no funcionales y los objetivos de diseño. De estas se hablará en la siguiente sección

Pruebas de aceptación y de instalación. Revisan los requerimientos contra el acuerdo del proyecto y deben ser realizadas por el cliente, con apoyo de los desarrolladores si es necesario.

Otras Pruebas. Aunque no es el centro de esta investigación, es bueno conocer las pruebas, diferentes a las de funcionalidad, que hacen parte de los planes de pruebas de software. Todas las pruebas que se mencionaran a continuación suelen realizarse, al menos en parte, utilizando pruebas de funcionalidad, y guardando los datos necesarios para el tipo de prueba dado.

Pruebas de desempeño. Además de la funcionalidad es posible probar el desempeño del software. No todos los sistemas tienen especificaciones de desempeño explícitamente, pero siempre hay indicaciones implícitas en este respecto.

Las evaluaciones de desempeño usualmente incluyen: uso de recursos, tiempo de respuesta a estímulos, tiempo de procesamiento y longitudes de cola. Usualmente se consideran recursos como el ancho de banda, CPU, espacio en disco, operaciones de acceso al disco, uso de memoria [PAN1999].

Las pruebas de desempeño suelen ser utilizadas para identificar cuellos de botella y abrazos mortales.

Pruebas de Fiabilidad. La fiabilidad del software se refiere a la probabilidad de que haya una falla en una operación del sistema. Es claro que un software fiable es aquel que no falla de manera inesperada, por lo que es necesario probar la robustez y la tolerancia a fallas.
La robustez se refiere al grado en el que el software puede funcionar bien en la presencia de entradas excepcionales o fallas de entorno. [IEE1990] En las pruebas de robustez no importa si la funcionalidad del software es correcta.
En las pruebas de tolerancia a fallas el software se ejercita buscando colmar los recursos, fallas en las actividades y cargas altas sostenidas.

Pruebas de Seguridad. Cualquier falla en el software puede implicar una falla de seguridad. Las pruebas de seguridad son comunes en sistemas que requieren acceso a Internet, pues este modo es en el que mas se presentan violaciones a la seguridad. Para probar la seguridad se suelen simular ataques para buscar vulnerabilidades.

2.1.1.2.1 Generación de datos

Tan importante como el diseño de la prueba es la decisión de que datos de entrada se van a utilizar para ejecutar las mismas. Los datos que se utilicen en la prueba deberían ser representativos de la totalidad de los datos posibles, de modo que la fiabilidad de la prueba sea la mejor. Además, los datos deberían conllevar la ejecución de toda parte del código. En general, hay que evitar la subjetividad en la selección de los datos de entrada.

Hay varias técnicas para la generación de los datos de entrada, a continuación se enumeran algunas:

Generación Aleatoria. Es la técnica más sencilla en términos de concepto [DUI2004]. Se generan los datos al azar en el espacio de datos posible (este espacio es determinado por la especificación). Se hace uso de las distribuciones de probabilidad para la generación de datos. La técnica tiene varios problemas:

- El primero es la forma en la que implementa la generación de datos distribuidos probabilística mente, ya que puede que el espacio de entrada no se pueda describir fácilmente.
- El segundo es la falta de conocimiento sobre la distribución de datos de entrada, pues suponer una distribución específica puede llevar a la generación de datos que nunca se verían en la realidad, y aun más grave, la falta de generación de datos que si se podrían ver, que son fallas en potencia.
- Un tercer problema es la necesidad de una buena especificación, si la especificación no es buena, el espacio de datos posibles no estaría bien definido y por lo tanto los datos que se generen tampoco.

Generación guiada por el código. Esta forma de generación considera el código fuente de la solución para buscar los datos que ejercitan la mayor cantidad de líneas. Es un proceso iterativo que revisa los elementos del programa para encontrar datos que los ejerciten, de forma que se maximice la posibilidad de encontrar una falla en el

código [DUI2004]. Esta es una técnica de caja blanca, ya que es necesario conocer el código.

Generación guiada por fallas. En este caso se busca generar datos que tengan alto riesgo de generar error. Los casos de borde son un ejemplo de esta técnica. [DUI2004]

2.1.1.2.2 Métricas

La generación de pruebas y su posterior ejecución no son suficientes para asegurar la corrección de un producto. Es necesario medir qué tan buena es la prueba co n relación a lo que se espera de ella. Ya que las pruebas se hacen para no revisar todos los posibles casos, las métricas son las que permiten definir la fiabilidad de la prueba.

Cubrimiento. Las métricas de cubrimiento pretenden revisar que tanto de los subproductos del sistema se esta ejercitando en las pruebas. A cada subproducto se hace corresponder una forma de cubrimiento a medir. Por ejemplo, cubrimiento de requerimientos o de interfaces. Las formas más comunes de medir el cubrimiento son: de línea, de arco, de camino y de condición [BRU2002].
Confiabilidad. La confiabilidad es la probabilidad de que el software opere sin fallas por un periodo determinado de tiempo. En este proyecto se enfoca la confiabilidad en la detección de defectos, independientemente del diseño del software y del diseño de las pruebas [BRU2002].
Esta es la métrica en la que se basará este proyecto.
Otras.

Complejidad Ciclomática. Mide la cantidad de lógica de decisión en un modulo de software [DUI2004].

Existe una gran variedad de métricas, según lo que se quiera evaluar y nivel de detalle que se quiera aplicar. [KAN2001] presenta una selección de métricas interesante.

2.1.1.2.3 Planes de pruebas de software

Independientemente de la actividad de prueba, toda prueba debe considerar un plan específico. En general deben considerarse las siguientes tareas [ESA1995]:

- Una actividad de especificación de pruebas, que produce un *diseño de las pruebas* para cada requerimiento o componente. Cada diseño implica una familia de casos de prueba.

- Una actividad que toma la especificación de cada caso de prueba y produce el *código de la prueba*, los *datos de entrada* y los *procedimientos* necesarios para correr las pruebas.
- Una actividad que tome el código de las pruebas y lo conecta con el software que se va a probar, produciendo un *software de prueba* ejecutable.
- Una actividad que ejecuta las pruebas de acuerdo con el diseño de las mismas, utilizando los datos de entrada. Los datos de salida producidos pueden incluir información de cubrimiento, de rendimiento o datos producidos por las pruebas.
- Una actividad que revise que las pruebas ejecutadas han ejercitado las partes del software que debían probar.
- Una actividad que estudie el consumo de recursos del software.
- Una actividad que compara las salidas de las pruebas con los datos esperados o con los datos de salida de pruebas anteriores, y decide si las pruebas pasaron o fallaron.
- Una actividad que guarda los datos de las pruebas para volverlas a correr.

2.1.1.2.4 El caso OO

En este trabajo se utiliza la programación orientada a objetos, por lo que se dará una breve explicación de este paradigma y su efecto en los casos de prueba.

El paradigma de programación orientado a objetos establece la creación de modelos basados en la abstracción del mundo real. La programación orientada a objetos puede verse como un conjunto de objetos que cooperan entre ellos, de manera diferente a la forma tradicional en la que se ve el programa: como un conjunto de funciones. En este estilo de programación cada objeto puede ser visto como una parte independiente con responsabilidades específicas, y con ellas la posibilidad de comunicarse con otros objetos y procesar datos.

La programación orientada a objetos está pensada para mejorar la flexibilidad y la mantenibilidad en la programación. Gracias a su condición modular, el código orientado a objetos tiende a ser mucho más simple de desarrollar y más fácil de revisar, de modo que los sistemas complejos son mas fáciles de representar, por lo que este paradigma es bastante popular en la ingeniería de software de gran escala [JAV2007].

Los elementos principales de la programación orientada a objetos son [JAV2007]:

- *Objeto.* Un objeto es una parte del software con estado y comportamiento relacionado. Los objetos en el software suelen ser utilizados para modelar objetos del mundo real. Cada objeto tiene estados y comportamientos, los primeros modelados como variables y los segundos como métodos.
- *Clase.* Una clase es un prototipo sobre el que se crean los objetos. De este modo define en abstracto las características y comportamientos de un objeto.
- *Herencia.* La herencia es la forma en la que se puede organizar el código para indicar versiones mas especializadas de una clase. De este modo una clase tendrá

"subclases" que tienen todas las características y comportamientos de la primera y algunos mas.

- *Encapsulamiento.* La comunicación entre objetos está definida por las responsabilidades de cada uno de ellos, por lo que se utilizan *interfaces* para exponer la forma enviar mensajes entre objetos.
- *Abstracción.* Consiste en simplificar la complejidad de la realidad mediante el modelado de clases apropiadas para el problema y hacer uso adecuado de la herencia.
- *Polimorfismo.* Este es comportamiento que varía dependiendo de la clase en la cual es invocado, de modo que dos clases diferentes pueden reaccionar diferente al mismo mensaje.

2.1.1.2.4.1 Arquitectura de software OO

La arquitectura de software es una herramienta que permite diseñar software al abstraer un modelo basado en la realidad, considerando los elementos, estructuras, conceptos e interacciones que este pueda tener. Esta permite facilitar el entendimiento, predicción, manejo y optimización del software. Al no mantener una buena arquitectura, el riesgo de falla aumenta. [FUS1996]

La arquitectura de software tiene como objetivo principal entender lo que se está construyendo, de modo que se pueda observar el panorama completo, definir responsabilidades y establecer los puntos en los que puede haber problemas. La mantenibilidad del software está altamente ligada a la arquitectura del mismo, pues es mediante la arquitectura que es posible establecer los elementos responsables de cada requerimiento, y por lo tanto es posible definir que tanto cambio hay que hacer para cambiar un requerimiento [MAL2003].

En el caso de la arquitectura de software orientada a objetos hay dos enfoques principales: el de diseño y codificación y el de sistema.
La *arquitectura del diseño y la codificación* esta marcada por la esencia de la programación OO: clases y métodos. Los métodos de una clase se dividen en los métodos constructores (construyen el objeto), los métodos *modificadores* (*setters,* modifican los campos privados del objeto), los *observadores* (*getters*, retornan el valor de un campo privado del objeto) y los de *responsabilidad* (implementan las responsabilidades del objeto, diferentes a mantener sus características).
La *arquitectura del sistema* esta definida por diferentes conceptos [FUS1996]:

- *Subsistemas.* Un subsistema es un paquete de clases altamente relacionadas con una interfaz bien definida. Permiten separar la lógica de la comunicación.
- *Particiones.* Divide la aplicación para separar áreas no relacionadas de la funcionalidad de la misma. Simplifica la interacción entre subsistemas, evitando fallas de integración.
- *Librerías.* Son particiones grandes de funcionalidad común. Permite a los desarrolladores enfocarse en problemas separados.

- *Capas.* Cada capa representa una abstracción particular del sistema y tiene responsabilidades específicas, por ejemplo aísla la lógica de la interfaz.

La combinación de estos conceptos se usa para definir sistemas complejos [MAL2003].

2.1.1.2.4.2 Plan de Pruebas OO

Con esto en mente, el plan de pruebas dentro del paradigma orientado a objetos está orientado a clases y métodos.
De este modo, las pruebas unitarias empiezan por probar cada objeto desarrollado y luego prueba los subsistemas. Las pruebas de integración prueban la comunicación entre subsistemas.
Gracias a la organización de la arquitectura OO, los planes de pruebas se definen dentro de esa misma idea básica, considerando primero un objeto: sus responsabilidades y características, luego un paquete de objetos, luego una capa, etc.

El problema de la programación OO a la hora de manejar los planes de pruebas es que no es posible probar una clase, sino una instancia de la misma (un objeto), con lo que puede que las pruebas queden incompletas.
Esta investigación se centra en pruebas OO.

2.1.1.2.4.3 Plan de pruebas como un DAG

Es posible organizar el plan de pruebas como un grafo. Teniendo en cuenta el interés de esta investigación en el paradigma OO, se considerará la organización de las pruebas en un *grafo acíclico dirigido* (DAG) para este caso.

El grafo de pruebas del sistema será la unión de varios grafos. Cada uno de estos grafos representará un subsistema, siendo a su vez cada subsistema una unión de grafos, cada uno representando un paquete y éste a su vez será una unión de grafos que representan clases.

Considérese el grafo de la clase. Este grafo tendrá como nodos los métodos y como raíz los invariantes de la clase. Los nodos terminales del grafo serán los métodos modificadores y observadores, ya que estos métodos no hacen llamados a ningún otro método. Los constructores deberían ser terminales en el grafo de la clase (pues no deberían llamar a ningún método de su propia clase), pero su calidad de terminal dependerá de que no llame a métodos de otras clases. Los demás métodos se vinculan al grafo desde la raíz, considerando las llamadas que hagan a otros métodos.
Teniendo en cuenta que las llamadas pueden ser entre clases, empieza a formarse el grafo del paquete. Y si se piensa en las llamadas entre paquetes, se generará el grafo del subsistema.

Se trata de un grafo dirigido pues los arcos representan llamados entre métodos, que van en una sola dirección. Se habla de un grafo acíclico pues no se consideran corrutinas, *multi-threads* ni excepciones. De este modo, no se presentan ciclos en los llamados, permitiendo

generar un orden topológico para las pruebas, con lo que es posible plantear el plan de pruebas siguiendo el orden del grafo.

Para aclarar la idea, considérese como ejemplo un sistema bancario. Este sistema tiene varios subsistemas: sistema de cuentas, sistema de préstamos, sistema de pago domiciliado de servicios, etc. Cada uno de esos subsistemas está comunicado con los otros a través del cliente: la historia crediticia del cliente afecta su cuenta, y su cuenta afecta el pago de servicios, para dar un ejemplo. Dentro del sistema de cuentas se pueden diferenciar los paquetes de interfaz y de *kernel*, donde el primero maneja la comunicación del subsistema con el usuario y el segundo maneja la lógica del subsistema. Dentro del *kernel* se encuentran las clases cuentas, cliente, cuenta de ahorros y cuenta corriente. Cada cuenta es responsable de manejar la cantidad de dinero y los extractos. El sistema es muy simple, pero servirá para ejemplificar el grafo.
El diagrama de clases del subsistema de cuentas sería el siguiente:

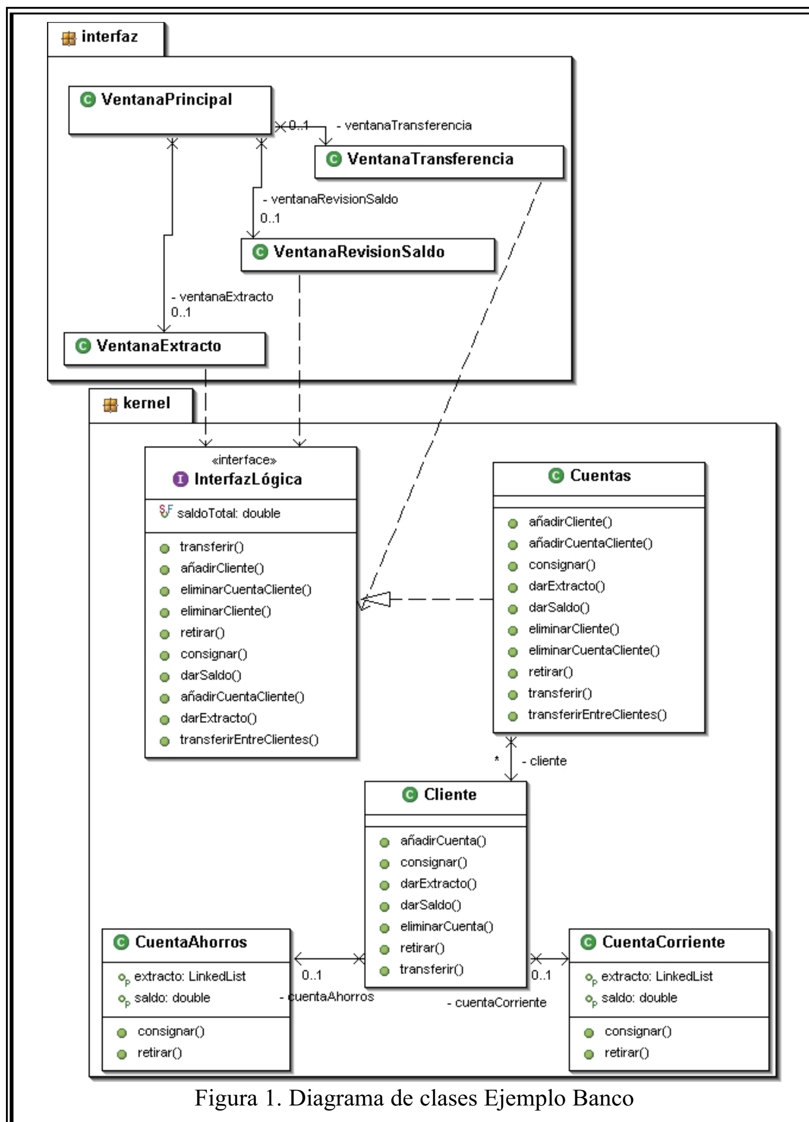


Figura 1. Diagrama de clases Ejemplo Banco

Para efectos el ejemplo, el enfoque se hace sobre el paquete *kernel*. Cada clase tiene modificadores y observadores de sus atributos. Las invariantes de clase serian, por ejemplo, en las clases CuentaCorriente y CuentaAhorros que el saldo nunca sea negativo, en el caso del cliente, que nunca tenga mas de una cuenta de un tipo especifico, etc.. Con base en este diagrama, el grafo de pruebas sería:

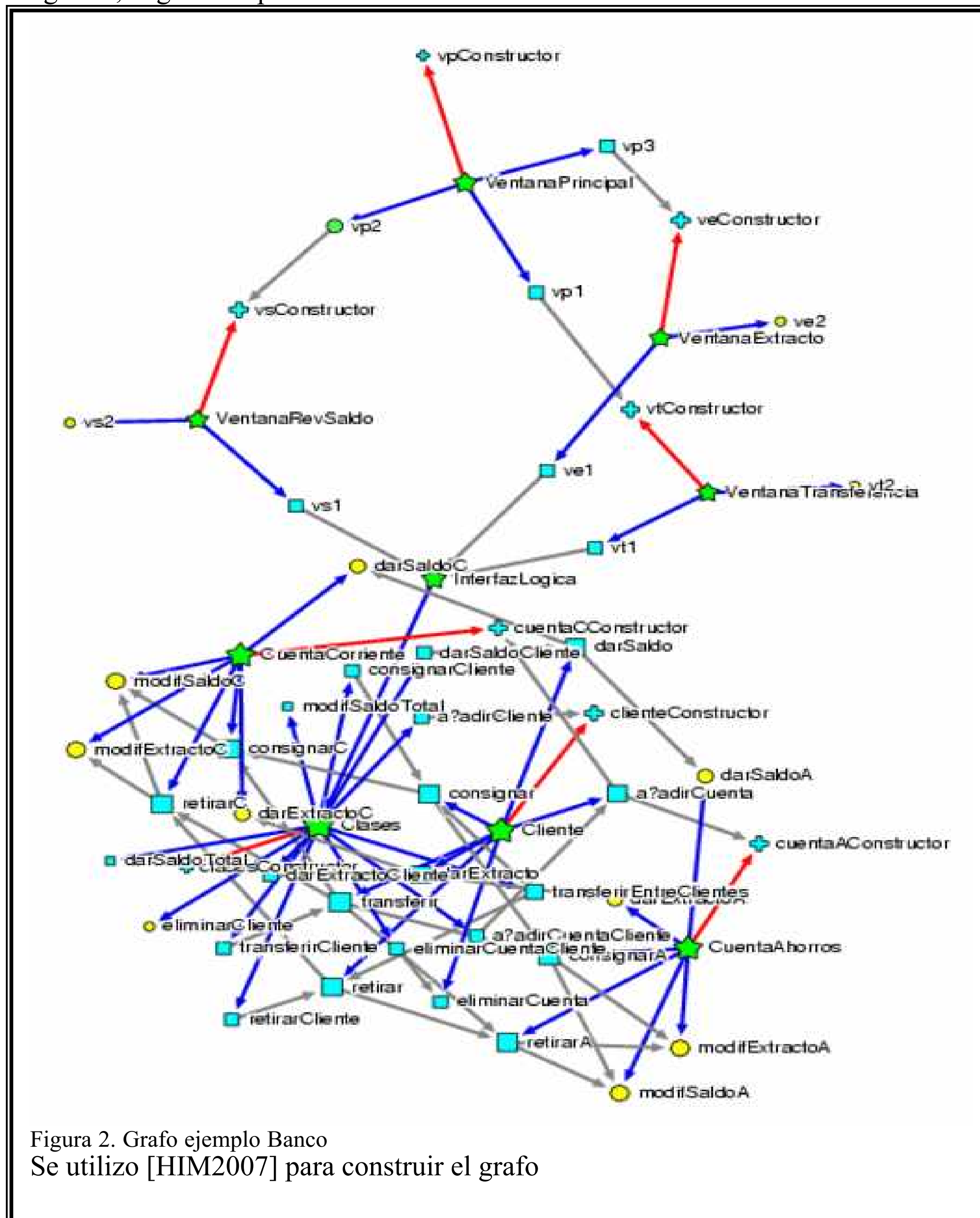


Figura 2. Grafo ejemplo Banco
Se utilizo [HIM2007] para construir el grafo

Las estrellas verdes representan las clases (y por tanto la prueba de la invariante de clase), los cuadrados aguamarina representan métodos que dependen de otros métodos, las cruces aguamarina representan los constructores de las clases, y los círculos amarillos representan métodos terminales (no dependen de otros métodos).

La forma en la que se manejarían las pruebas será discutida más adelante, pero a grandes rasgos se trata de recorrer el grafo desde las hojas hacia las raíces.

2.1.2 Corrección *normal*

Como ya fue descrito, por medio de las pruebas se puede verificar que se este cumpliendo la especificación. Y como ya fue señalado, la especificación puede ser tan amplia o tan simple como se desee, y el secreto de una buena prueba está en una buena especificación.

Dentro de la especificación se pueden considerar casos de terminación excepcional, es decir casos en los que el método que está siendo ejecutado aborte abruptamente. En una prueba, un caso de estos no representa necesariamente un error del método que se está probando, pero si implica que es necesario generar mas pruebas que determinen la razón de la terminación excepcional y que indiquen si este tipo de terminación se considera en la especificación [QUI2006].

Esta investigación no considerará los casos de terminación excepcional, serán manejados de acuerdo con la especificación, independientemente de si son un defecto o no. Es decir mientras se cumpla la especificación, no importa si se generó una terminación excepcional. De este modo, de acá en adelante se hablará de corrección y pruebas *normales*, que no tienen en cuenta la forma de terminación.

2.2 Herramientas

2.2.1 JUnit: Pruebas unitarias

Las pruebas unitarias automáticas tienen como ventaja sobre las demás pruebas el que pueden ejecutarse de nuevo con facilidad, y suele haber *frameworks* que facilitan la ejecución de la prueba, al compilar la aplicación y correr las pruebas automáticamente [HAN2007].

JUnit es un *framework* creado por Erich Gamma y Kent Beck que es utilizado para hacer pruebas unitarias de aplicaciones Java [JUN2007].

JUnit es la librería de pruebas unitarias estándar para el lenguaje Java. JUnit impulsó la popularización del proceso de pruebas. Gracias a JUnit el código Java tiende a ser más robusto, confiable y con menos defectos que antes. Además, JUnit ha inspirado la creación de varias herramientas para pruebas unitarias, para un amplio rango de lenguajes. La importancia de JUnit no radica en su *framework* sino en la idea central de la herramienta, es decir la programación guiada por las pruebas. El hecho de que las pruebas se generen antes del código del programa ha probado mejorar la calidad del código [HAR2005]

El *framework* de JUnit permite ejecutar clases Java de manera que sea posible evaluar el funcionamiento de los métodos de dicha clase. De este modo el programador define valores de entrada y evalúa el resultado de la ejecución contra un valor esperado. Si se cumple la especificación, JUnit retornará una señal de éxito, de lo contrario devolverá un fallo, indicando donde sucedió el error [HAR2005].

Siendo por ejemplo *A* la clase que se va a probar, se construye una clase de nombre *TestA* en la que se implementa un conjunto de métodos dirigidos a realizar pruebas. En cada uno de estos métodos se crea manualmente un objeto de clase *A*, que se corresponderá con el resultado esperado de la prueba; también en cada método se construye un objeto de clase *A* ejecutando las operaciones definidas en esta clase, que se corresponde con el resultado obtenido. Si el objeto construido manualmente es igual que el construido automáticamente, entonces la prueba ha sido superada.[JUN2007]

Aunque en este proyecto se hace uso de la filosofía detrás de JUnit y se hace uso de dicha herramienta, debe ser claro que no es la única librería para pruebas unitarias existente. A continuación se expone una corta lista de herramientas [UNI2007]:

- *Rational Test RealTime Unit Testing.* Puede generar pruebas unitarias para C, C++, Ada 83 y 95. Realiza pruebas unitarias de caja negra para evaluar funcionalidad.

- *C++ Test.* Es una herramienta para C/C++ que prueba automáticamente clases funciones o componentes.

- *AQtest.* Herramienta que soporta pruebas automáticas funcionales unitarias y de regresión. Funciona para VC++, VB, Delphi, C++Builder, Java o VS.NET. También soporta pruebas de caja blanca para métodos y propiedades.

- *EasyMock.* Librería que permite usar objetos maniquí (*mock*) para representar las unidades con las que pueda colaborar el método que se está probando, y además de probar el método, prueba también que el objeto maniquí sea correcto.
- *GrandTestAuto.* Herramienta de pruebas unitarias para Java. Corre automáticamente todas las pruebas unitarias para una aplicación y al mismo tiempo revisa que dichas pruebas provean un cubrimiento suficiente para asegurar la corrección del programa.

2.2.2 JML: Lenguaje de especificación y prueba

Un lenguaje de especificación es un lenguaje formal que se utiliza durante el análisis del sistema, de los requerimientos y el diseño. Los lenguajes de especificación están pensados para describir el *qué* de cada objeto, no el *cómo*, por lo que usualmente este tipo de lenguaje no requiere de ejecución [SPE2007].

Estos lenguajes tienen una sintaxis y semántica claramente especificada, por lo que la ambigüedad se elimina, mejorando la intuición y aumentado la consistencia, permitiendo detectar errores durante el diseño, antes de la implementación.

La especificación puede ser tratada de dos formas [SPE2007]:

- *Orientada a propiedades.* En este caso la especificación consiste en axiomas lógicos dentro de un sistema en el que la equivalencia tiene un rol principal, describiendo las propiedades que las funciones deben satisfacer.
- *Orientada a modelos.* En esta especificación se busca definir el comportamiento requerido por cada modelo.

El lenguaje de modelado en Java (Java Modeling Language, JML [JML2007]) es un lenguaje de especificación formal de comportamiento para Java. Como tal, permite al usuario especificar tanto la interfaz sintáctica del código Java como su comportamiento. La interfaz sintáctica del código Java consiste en nombres, visibilidad y otros modificadores, e información sobre el chequeo de tipos. Por ejemplo, la interfaz sintáctica de un método puede ser vista en la cabecera del método, la cual lista los modificadores, sus nombres, su tipo de retorno, los parámetros formales, y los tipos de las excepciones que pueda sacar. El comportamiento del código Java describe que debe pasar en tiempo de ejecución cuando el código es usado. Por ejemplo el comportamiento de un método describe que debe pasar cuando el método es llamado. Este comportamiento suele especificarse utilizando precondiciones y poscondiciones. [LEA2006]
JML usa expresiones Java para escribir predicados usados en aserciones, como precondiciones y poscondiciones. La ventaja de utilizar la notación de Java en las aserciones es que es mas sencillo de aprender para los programadores y menos intimidante que lenguajes que usan notación matemática para propósitos específicos. Para los casos en los que se necesita mas expresividad en las expresiones de Java, JML extiende las expresiones con varias construcciones de especificación como los cuantificadores. [LEA2006]
JML además incorpora varias ideas y conceptos de los lenguajes de especificación orientados al modelo, lo que permite escribir especificaciones que son apropiadas para la verificación formal. [LEA2006]

La forma en la que se utiliza JML es descrita en el anexo A.

2.3 Elementos de estadística

La *estadística* es una ciencia matemática que se dedica a la recolección, análisis, interpretación y presentación de datos. [MEY1976]
Específicamente, la *estadística de inferencia* o *inductiva* concierne el uso de estadística para inferir sobre un parámetro de la población. En este caso la población son conjuntos de datos de entrada para un método específico y el parámetro es la probabilidad de que un programa se desarrolle correctamente ante uno de estos conjuntos. [MEY1976]

2.3.1 Algunos conceptos

Se considera la estadística como centro del proyecto, pues con base en técnicas estadísticas de muestreo e intervalos de confianza, se busca no revisar la totalidad de los caminos de ejecución, sino probar cada método con diferentes parámetros de entrada y verificar que no haya errores. Por esta razón es necesario revisar estos conceptos, lo que se hará a continuación.

2.3.1.1 Muestreo

Es la parte de la estadística que trabaja con la selección de observaciones individuales que pretenden inferir algún conocimiento sobre una población específica.
Los procesos de muestreo consisten en cinco etapas [MAK2007]:

- Definición de la población de interés.
- Especificación de un marco muestral, un conjunto de elementos o eventos que se pueden medir.
- Especificación del método de muestreo para escoger elementos del marco.
- Muestreo y colección de datos.
- Revisión del proceso de muestreo.

Entre las técnicas más comunes de muestreo se encuentran:

- *Aleatorio*: Cada miembro de la población tiene la misma probabilidad de ser elegido.
- *Sistemático*: Una población consiste en `N` objetos. La meta es obtener una muestra de tamaño `n` tal que `N=nk`, donde `k` es el número de grupos. De manera aleatoria, se selecciona un punto de inicio entre el primer y el `k`-esimo objeto (incluido). Se selecciona cada `k`-esimo objeto después, hasta que tenga una muestra de tamaño `n`. Si el orden de la población es aleatorio, entonces el muestreo sistemático puede ser analizado tal como si fuera muestreo aleatorio simple.
- *Estratificado*: Una población puede ser dividida en sub poblaciones llamadas *estratos*. Se escoge de forma aleatoria un miembro de cada estrato. Dentro de cada estrato, los objetos son homogéneos, pero los estratos son heterogéneos.
- *Clusters*: Una población se divide en secciones llamadas *clusters*. Se escoge de manera aleatoria un cluster completo y se revisan todos los objetos dentro del cluster. Dentro de un cluster, los objetos son heterogéneos, pero los clusters son homogéneos.

En este trabajo se tiene que la población a muestrear son todos los posibles conjuntos de parámetros de entrada del método que se quiere verificar. El propósito del método puede dar indicios sobre la distribución de estos parámetros. Debido a la naturaleza de los datos, puede ser más sencillo dividir en grupos de parámetros equi-probables de ser utilizados en el método, y dentro de cada uno de estos grupos realizamos una serie de muestreos aleatorios simples.

2.3.1.2 Intervalos de Confianza

Para este proyecto específico son de interés los *intervalos de confianza*, que son rangos de valores, calculados a través de una muestra, entre los que se encuentra el verdadero valor del parámetro con una probabilidad determinada. Esta probabilidad se denomina *nivel de confianza* y se denota como $1-\alpha$. Para construir un intervalo de confianza se utilizan las tablas de cuantiles de la distribución normal (por lo que se requiere que el parámetro sobre el que se infiere se distribuya normalmente). La formula que define el intervalo establece $Z_{\alpha/2} \leq \frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}} \leq Z_{1-\alpha/2}$, donde x es un estimador de X, μ es la media de X y σ es su desviación estándar. [MEY1976]

De modo que si una variable X tiene distribución $N(\mu, \sigma^2)$, entonces el $1-\alpha/2$ de las veces se cumple la formula dada, con n ensayos.

Específicamente, son de interés los intervalos de confianza para proporciones, dada la naturaleza del problema de verificación. Estos intervalos se construyen para una proporción o porcentaje poblacional.

Si $X_1, X_2, ..., X_n$ son variables aleatorias independientes, idénticamente distribuidas, de valor medio μ y varianza σ^2, entonces la distribución de la variable $Z = \frac{(X_1 + X_2 + \cdots + X_n) - n\mu}{\sigma\sqrt{n}}$ se aproxima a la de una variable Normal Estándar $N(0,1)$; la calidad de la aproximación mejora a medida que n aumenta.

Luego, procediendo en forma análoga al caso de la media, se puede construir un intervalo de $1-\alpha$ de confianza para la proporción poblacional p, con lo que se tendría: $|p - \hat{p}| \leq Z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$

Con un $p\hat{}=0.005$, una confianza de $.99$, y un n calculado en 664, la distribución del estimador sería:

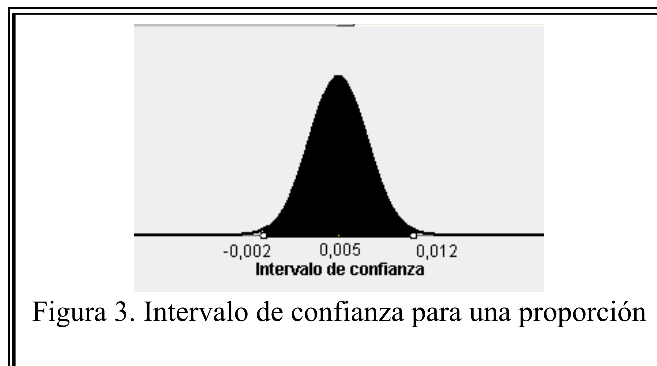


Figura 3. Intervalo de confianza para una proporción

2.3.1.3 Distribuciones probabilísticas

Una *variable aleatoria* se define como el resultado de un experimento, no necesariamente numérico. Las variables aleatorias asocian un valor numérico único a cada salida del experimento, por lo que puede verse como una función $X: \Omega \rightarrow \mathbf{R}$ que califica los eventos de un espacio muestral Ω con un número real.
El valor de la variable suele cambiar cada vez que se repite el experimento. Las variables aleatorias tienen un valor esperado y una varianza. El valor esperado (μ) indica el valor promedio de la variable, mientras que la varianza (σ^2) es un número no negativo que indica que tan amplio es el rango de valores que toma la variable [EAS2007].

Una variable aleatoria puede tomar valores discretos o continuos. En el primer caso los valores son números contables, usualmente son conteos. Si la variable solo puede tomar un número finito de valores, debe ser discreta. Las variables continuas pueden tomar un número infinito de valores. Usualmente son medidas.

La distribución de probabilidad de una variable aleatoria discreta es una lista de las probabilidades asociadas con cada uno de los valores que puede tomar la variable. Es una función que da la probabilidad $p(x_i)$ de que una variable equivalga a x_i [EAS2007].

$$p(x_i) = P(X=x_i)$$

Se satisface a través de las siguientes condiciones:

- $0 \leq p(x_i) \leq 1$
- $\Sigma\ p(x_i) = 1$

La función de distribución acumulativa es la función que da la probabilidad de que la variable aleatoria X sea menor o igual que x, para todo valor de x.

$$F(x) = P(X \leq x) \quad \text{para } -\infty < x < \infty$$

La función de densidad de probabilidad de una variable aleatoria continua es una función que puede ser integrada para obtener la probabilidad de que la variable tome un valor en la integral dada [EAS2007].

De este modo, la función de densidad de probabilidad $f(x)$ de una variable aleatoria continua X es la derivada de la distribución acumulada $F(x)$.

$$\int_a^b f(x)dx = P(a < X < b)$$

$f(x)$ debe obedecer dos condiciones:

- $\int_{-\infty}^{\infty} f(x) = 1$
- $f(x)>0$ para todo x.

2.3.1.3.1 Algunas Distribuciones

A continuación se enumeran algunas de las distribuciones más comunes, como punto de referencia para la comprensión del concepto de distribución [EAS2007], [MAT2007]:

Distribución Normal. Esta es una distribución de variables continuas.
Se dice que X se distribuye Normal con media μ y desviación σ^2 ($N(\mu,\sigma^2)$) si

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

El caso mas simple de la distribución normal es la $N(0,1)$, llamada la normal estándar.

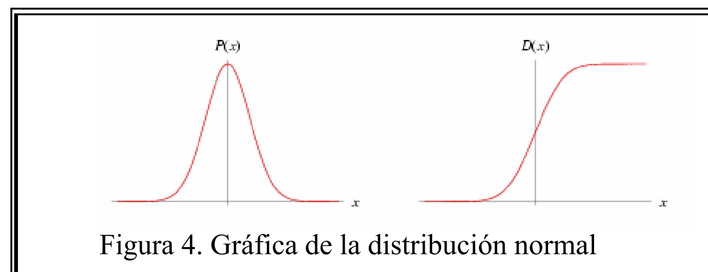


Figura 4. Gráfica de la distribución normal

Distribución Poisson. Es una distribución de variables aleatorias discretas.
Usualmente una variable que se distribuye de esta forma representa el conteo del número de eventos que ocurren en un intervalo de tiempo.
Se dice que X se distribuye Poisson con parámetro m, $X \sim Po(m)$, si

$P(X = x) = \frac{m^x}{x!} e^{-m}$, donde $0 \leq x \leq n$ y $m>0$.
El parámetro de la Poisson representa la tasa en la que se presenta el evento. Tanto el valor esperado como la varianza equivalen a m.

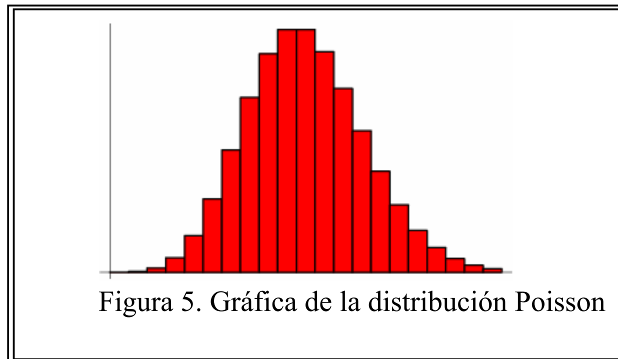


Figura 5. Gráfica de la distribución Poisson

Distribución Binomial. Es una distribución de variables discretas. Usualmente es el número de éxitos en una serie de intentos. Se dice que $X \sim Bi(n,p)$ si

$P(X = x) = \binom{n}{x} p^x (1 - p)^{n-x}$, donde $0 \leq x \leq n$, $n > 0$, $p =$ probabilidad de éxito. El valor esperado de la binomial es np y la varianza es $np(1-p)$.

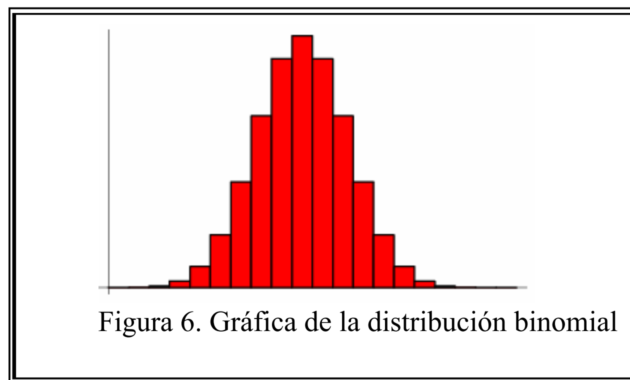


Figura 6. Gráfica de la distribución binomial

Distribución Bernoulli. Es una distribución discreta que tiene dos posibles salidas: $n=0$ y $n=1$, donde la primera indica falla, y ocurre con probabilidad p y la segunda indica éxito, y ocurre con probabilidad $q=1-p$. La función de probabilidad es $P(n)=p^n(1-p)^{1-n}$

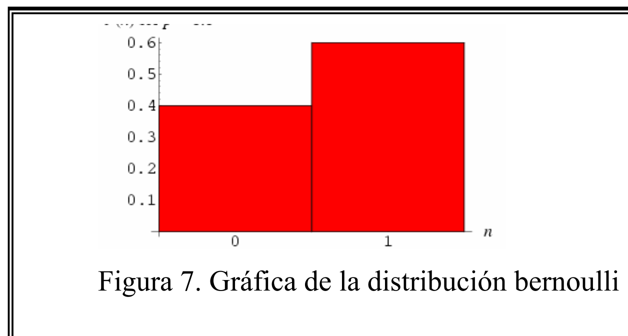


Figura 7. Gráfica de la distribución bernoulli

Distribución Geométrica. Es una distribución para variables discretas. Se define como el número de intentos necesarios para obtener la primera falla. Una variable X sigue una distribución geométrica $X \sim Ge(p)$, si

$P(X = x) = p^{x-1}(1 - p)^{x}$, donde $x>0$ y p es la probabilidad de éxito. El valor esperado es $1/(1-p)$ y la varianza es $p/(1-p)^{2}$

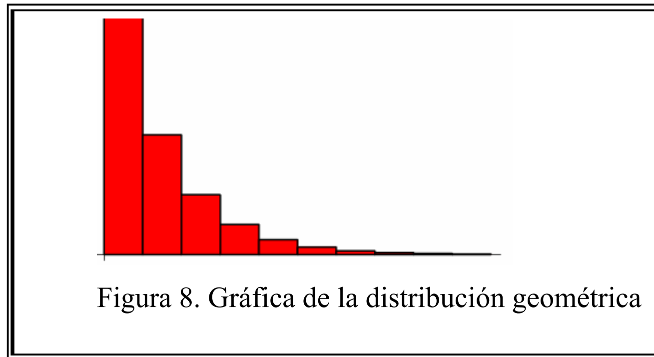


Figura 8. Gráfica de la distribución geométrica

Distribución Uniforme. Esta distribución se modela sobre variables continuas y sobre variables discretas. Una variable X sigue una distribución uniforme con parámetros a y b si:

$P(X = x) = \frac{1}{b - a}$, donde $x>0$. La distribución uniforme tiene probabilidad igual en cada uno de sus valores. El valor esperado es $(a+b)/2$ y la varianza es $(b-a)^{2}/12$.

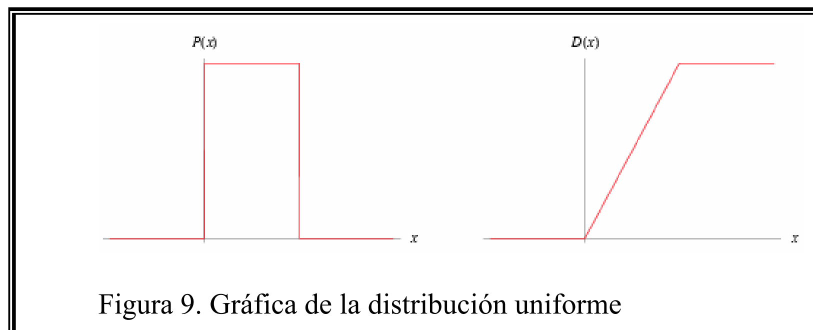


Figura 9. Gráfica de la distribución uniforme

2.3.1.4 Datos para aplicaciones de informática

En este documento las distribuciones probabilísticas van a definir la generación de datos para las pruebas. Por ello es importante determinar la distribución a elegir en cada caso. Como se señaló anteriormente, el programador puede no saber como se distribuyen los datos de entrada de un método específico, y acogerse a una distribución “por defecto” (v.gr., la uniforme) distorsiona el cálculo de la confianza.

Una solución plausible sería hacer una simulación que permitiera extraer datos para ajustarlos a alguna distribución. Esto puede ser costoso y muy complicado, sobre todo, si el software no está terminado.

De este modo, solo quedaría dejar a la subjetividad del probador el escoger una distribución. Aunque parezca algo arbitrario, se debe tener en cuenta que el programador conoce la especificación, lo que le da indicios para definir los límites en los que se mueve la variable, y con ayuda de los requerimientos (y casos de uso) conoce el objetivo de cada método, por lo que tendrá conocimiento de los valores que pueden tomar esos datos. Por ejemplo puede estimar si la distribución que usará es discreta o continua, con lo que el problema será establecer la distribución, que podría ser diferente para diferentes segmentos de datos. Lastimosamente no hay otra forma de definir esa distribución diferente a la intuición. De nuevo, la intuición del programador, unida al conocimiento del propósito del método puede darle indicios.

En el peor de los casos, en aquel en el que el conocimiento del programador no ofrezca ninguna ayuda para escoger la distribución de los datos, se recomienda (como último recurso) elegir una distribución uniforme. Aunque tal situación no es la ideal, al menos se define una distribución que sirve para efectos de las pruebas, y como se observara más adelante, la generación de datos aleatoria, unida a la partición del espacio, mejora la confianza de la prueba.

Capítulo 3

Validación Estadística

Resumen. Este capítulo propone una metodología de validación que utiliza los conceptos estadísticos del capítulo 2 y la generación de pruebas para determinar si un producto de software es correcto, estando esta determinación sujeta a un porcentaje de error. Se hace uso de los intervalos de confianza para determinar la cantidad de datos necesaria para que la prueba tenga una confiabilidad dada. La metodología se describe por partes. La primera parte considera la validación estadística de métodos, para lo que se exponen dos modalidades de prueba: contra especificación y contra oráculo. En esta primera parte se discute el problema de la generación de instancias de clases para poder ejecutar los casos de prueba que se generen, y se discute la forma en la que es posible transformar las funciones de distribución de probabilidad de modo que se amplíe la gama de posibilidades para la generación de datos aleatorios. La segunda parte de la metodología describe la validación estadística de clases como la suma de la validación estadística de métodos más la validación de las invariantes de la clase. De nuevo se discute la forma en la que se deberían generar las instancias de la clase que se está probando, y la forma en la que estas instancias deben ser validadas. Esta segunda parte discute los planes de pruebas de integración, al ser estos parte de la validación estadística de clases. La tercera parte de la metodología trata la validación estadística de productos de software. En esta parte se explican los planes de prueba de sistema, definiendo un producto correcto como la sinergia de todas sus clases correctas. Finalmente se describen los procesos de reutilización de pruebas dentro de la metodología, considerando el almacenamiento y regresión de las mismas, y exponiendo como mediante un DAG se puede generar una topología de pruebas que permita organizar el proceso de corrección de un producto de software.

En este proyecto se denomina *validación estadística* al proceso de asegurar la confiabilidad de un producto de software con un nivel de confianza determinado. La validación estadística es una metodología para definir y administrar pruebas, haciendo que sean suficientes para poder afirmar calidad con confianza (o bien "corrección estadística"). Dicha validación se lleva a cabo utilizando conceptos estadísticos, empezando por el nivel de confianza, para poder generar una cantidad de pruebas que, una vez ejecutadas, retornarán si el producto es o no correcto. La generación de los datos para las pruebas considera distribuciones probabilísticas para los parámetros, y con base en éstas genera la cantidad de datos necesaria, según la confianza requerida.

Como se comentó en el capítulo anterior, el principal problema del uso de la técnica de generación aleatoria de datos es la dificultad para predecir la distribución que siguen los datos en la realidad. Este problema afecta principalmente la confianza de la prueba, pues puede generar datos por fuera de los rangos reales solamente y considerarse el programa

correcto, cuando puede fallar en los datos no probados. El problema no es exclusivo de la generación aleatoria, puesto que todos los modelos de pruebas no exhaustivos están sujetos al mismo problema. Por lo anterior, parece sensato utilizar todo el conocimiento de los requerimientos y hacer la abstracción de la realidad (modelo OO) pensando en que datos de esa realidad van a ser útiles en la definición de las distribuciones. Esta idea no asegura la calidad de la prueba, pero propende porque el error en la confianza sea mínimo.

Este capítulo establecerá la idea de la validación estadística y desglosará cada una de las partes que es necesario tener en cuenta para llevar dicha validación a cabo.

3.1 La idea fundamental

El centro del modelo de validación estadística está en el éxito de la aplicación de modelos estadísticos. Ese éxito permite asegurar la confianza de las pruebas. El ejercicio consiste entonces en una serie de experimentos Bernoulli[2], donde la variable aleatoria representa el éxito o el fracaso de una prueba. Cada prueba tiene asociada una serie de datos, estos datos, generados a través de una distribución dada, son independientes en cada prueba. Las pruebas son entonces independientes e idénticamente distribuidas.

El primer problema es definir que es una *prueba* dentro de esta metodología. Como se definió en la sección 2.1.1.2, un caso de prueba es un conjunto de entradas para un producto específico que producen un resultado de éxito o falla al verificar el producto. Una prueba se define como un conjunto de casos de prueba ejecutable, que debe considerar:

- El producto a probar.
- El conjunto de datos de entrada que se usa en cada caso de prueba.
- La forma en la que se valida el producto (en el caso de este trabajo, la especificación o en oráculo).
- La confianza que se espera del resultado de la prueba.
- Los elementos que se usan en cada caso de prueba para ejecutarlos.
- La forma en la que se generan los datos.

Otro problema está en definir la cantidad de casos de prueba necesarios para que, con una confianza dada, se pueda afirmar la corrección del programa. Este número esta dado por la fórmula de un intervalo de confianza para proporciones, que se revisó en la sección 2.3.1.2. Esta fórmula requiere que el parámetro a investigar se distribuya de manera normal, pero eso no es conocido en este caso, pues se estaría afirmando que el error de un método tiene una distribución normal, cuando se sabe que es Bernoulli: o tiene errores o no los tiene.

En este punto se aprovecha el hecho de que las pruebas son independientes e idénticamente distribuidas, ya que se puede utilizar el *teorema del límite central*. Este teorema puede interpretarse como que si se tiene un grupo 'grande' de variables independientes e

[2] Ver sección 2.3.1.3.1. La media de la distribución Bernoulli es `p` y la varianza es `p(1-p)`.

idénticamente distribuidas, la suma de ellas se distribuye según una distribución normal. Un nuevo problema está en determinar lo que significa 'grande'. Algunas aproximaciones frecuentes establecen que $n>9/p(1-p)$, considerando el peor de los casos, es decir aquel en que $p=0.5$, se tendría que $n=36$. [MAT2007]

De este modo, es posible afirmar que con un $n>36$ la distribución de la suma de variables de tipo Bernoulli puede ser aproximada a una distribución normal con media p y varianza $(p(1-p))/n$. Así, se tiene una nueva variable aleatoria X que representa el promedio del resultado (éxito $=1$, fracaso $=0$) de n pruebas.

Con ayuda del supuesto de la aproximación se puede calcular el n necesario para la confianza dada, bajo el precepto de que si el n calculado es menor a 36, el supuesto es inválido. Sea α la confianza con la que el programador espera asegurar la corrección del programa, y Z_α es el resultado de aplicar la función normal estándar inversa sobre α, se tendría $Z_{\alpha/2} \leq \dfrac{|\hat{p} - p|}{\dfrac{\sqrt{\hat{p}(1 - \hat{p})}}{\sqrt{n}}} \leq Z_{1-\alpha/2}$

Despejando n se tiene que $n = \dfrac{Z^2_{1-\alpha/2}\,\hat{p}(1 - \hat{p})}{(\hat{p} - p)^2}$

Es decir, se ha obtenido el mínimo n necesario para alcanzar la confianza dada.
En esta fórmula, $\hat{p}$ es el estimador de la probabilidad p de que el programa no tenga errores. $\hat{p}-p$ es el error entre el estimador y el parámetro.
Al empezar ninguno de estos datos es conocido, por lo que se tiene que considerar el peor caso posible, es decir, se considera que el producto a probar "adivina" el resultado, por lo que la probabilidad de que adivine bien es del 50%, con lo que se tiene $\hat{p}=0.5$. El error $|\hat{p}-p|$ también se supone, considerándolo como un número muy pequeño, en este caso se tomó 0.0025.

Para efectos de un ejemplo, considérese que la confianza deseada es del 99%, es decir, $\alpha=0.01$. Con estos datos, n debería ser, mínimo, 664. El procedimiento a seguir consistiría en generar 664 casos de prueba con datos independientes. En cada caso de prueba se evalúa si se cumple la especificación y se cuenta cuantas fallas y cuantos éxitos se obtienen. Para el caso tratado en este documento, con una sola falla sería suficiente para detener el experimento, pues se determinaría que el programa no es correcto y debe ser modificado. Un caso distinto sucede cuando el experimento no presenta ninguna falla, es decir que el resultado de todas las pruebas realizadas fue consistente con la especificación; en el caso del ejemplo indicaría que con un 99% por ciento de confianza el producto probado fue correcto. Es decir que hay un 1% de probabilidad de que haya errores no detectados, bajo el supuesto de pruebas independientes. Este supuesto está fundamentado en la generación de datos para las pruebas, de lo que se hablará mas adelante.
Aunque estadísticamente se puede llegar a una confianza del 100% en la realidad ese nivel de confianza solo se alcanzaría con una exploración exhaustiva. Pero también es claro que

entre más pruebas independientes se hagan mayor confianza se tendrá. De este modo es posible repetir el experimento, es decir generar y ejecutar n nuevos casos de prueba, y asegurar con ello el cumplimiento del supuesto de independencia, reafirmando el nivel de confianza requerido para la prueba.

La validación estadística considera las pruebas como una forma de mejoramiento continuo, de manera que cada prueba que se realice debe ser guardada para su uso posterior, no solo para probar el sistema completo, sino para probar cualquier cambio que se realice al mismo. En un plan de pruebas adecuado, las pruebas empiezan desde muy temprano en el proceso de desarrollo, por lo que puede que un mismo producto se cambie varias veces antes de lograr estabilidad, la especificación, en cambio, suele estar disponible desde el principio del proyecto y no sufre cambios importantes a través del mismo, lo que permite utilizar las pruebas que se realicen para asegurar la corrección del producto a lo largo de su avance, y evitar el efecto "bola de nieve" que tienen los errores en el proceso de desarrollo.

3.2 Una metodología de validación

Como se había afirmado en el capítulo anterior, cualquier metodología de validación requiere un plan de pruebas bien diseñado. En el caso de la programación orientada a objetos, este plan debería considerar la arquitectura general del software, es decir la división del mismo en subsistemas, paquetes, clases, atributos y métodos.
La metodología de validación que se propone consiste en validar cada una de esas partes y luego validar la integración de las mismas.

La validación de un sistema de software equivale a la validación de sus subsistemas, más la validación de la integración de dichos subsistemas. A su vez, la validación de un subsistema deberá considerar la validación de cada paquete dentro del mismo y la validación de la integración de dichos paquetes. Si hubiese requerimientos específicos para el subsistema, también será necesario validarlos. Del mismo modo sucede para la validación de un paquete con sus clases. En el caso de la validación de una clase, será necesario validar sus métodos, y para validar la integración de sus métodos será necesario considerar la validación de la invariante de clase y de su funcionamiento contra las responsabilidades establecidas para la misma en el diseño. De este modo para probar un sistema hay que probar previamente sus subsistemas, y a su vez, para validad un subsistema se deben validar sus paquetes, dentro de estos sus clases y dentro de éstas sus métodos.

La metodología propuesta sugiere empezar por la validación de las partes más pequeñas (los métodos) para luego pasar a la validación de partes complejas.

Aquí es justamente donde entra el plan de pruebas en forma de grafo. Los métodos hacen parte de los nodos terminales del grafo, de modo que el primer paso consiste en validar dichos nodos, en el caso de la Figura 2 se empieza por validar los métodos representados por círculos amarillos. Sin embargo, para poder probar un método es necesario tener una instancia de la clase a la que este método pertenece, por lo que el primer paso realmente consiste en probar los constructores, si es que estos son nodos terminales del grafo. De lo

contrario, es necesario probar los métodos usando objetos generados por constructores vacíos (de modo que debería obligarse a que exista al menos un constructor vacío). El problema que esto representa esta en la incapacidad de probar mas que instancias de cada clase. Ese es uno de los aspectos del paradigma OO, es imposible considerar la clase para una prueba, siempre que se quiera ejecutar dicha clase se deberá generar una instancia de la misma.

Volviendo a la metodología, el algoritmo es:

1. Probar los constructores que sean nodos terminales del grafo.
2. Probar los métodos que están en los nodos terminales, pero esto solo es posible si se tiene una instancia de la clase que contiene dichos métodos. De este modo antes de probar los métodos de los nodos terminales es necesario asegurarse de que es posible crear una instancia correcta de la clase. Los métodos de las hojas no hacen llamados a ningún otro método lo que hace que el proceso de validación dependa solamente de la lógica del método mismo.
3. Probar los constructores que sea posible, utilizando los métodos en nodos terminales ya probados.
4. Probar los métodos en nodos terminales que sea posible, utilizando los constructores ya probados.
5. Repetir los pasos 3 y 4 hasta que no haya más nodos terminales para probar.
6. De acá en adelante la metodología de prueba consiste en seguir el grafo e ir probando de afuera hacia adentro. Ningún elemento del grafo puede ser probado a menos que se hayan probado todos los elementos de los que este dependa.

Una vez validados los métodos se puede probar la clase, en la Figura 2 las clases están representadas por estrellas. La metodología para validar una clase será descrita posteriormente, se trata de considerar las invariantes de clase y las responsabilidades de la misma. De esta forma se puede seguir todo el grafo del producto, en el caso del ejemplo, un paquete.

Como se explicó en la sección 2.1.1.2.4.3, es posible ordenar el DAG de plan de pruebas de manera topológica, lo que hace que el procedimiento descrito sea adecuado. De esta forma el algoritmo de prueba no se queda en ciclo y es posible recorrer todo el grafo desde los nodos terminales hasta las raíces, probando todo el sistema.

La validación depende, como ya es claro, de la especificación. Esta metodología considera JML [JML2007] cómo la forma de especificar cada una de las restricciones, invariantes, responsabilidades y características del sistema.

En este trabajo se considera la validación de un producto de software (un paquete, subsistema o sistema de software), la validación de una clase y la validación de un método. Se define la metodología para dicha validación y se exponen los puntos focales que optimizarían la confianza en un proceso de aseguramiento de corrección.

3.2.1 Validación estadística de métodos

La validación estadística de métodos es la forma en la que se puede asegurar la corrección de un método con un nivel dado de confianza.
Aunque la metodología de prueba considera la división del sistema en sus partes para facilitar la validación, no se puede apartar completamente ninguna de esas partes del sistema, pues puede estar interconectada con muchas más partes del sistema. De este modo es necesario conocer todo el producto para la prueba de un método.

Como se comentó previamente, la metodología propuesta necesita tener la instancia de la clase, de modo que el primer elemento a probar son los constructores.
En esta sección se considerarán, en términos de estrategia de validación, iguales los constructores y los métodos, y se les llamará métodos indistintamente. Más adelante se discutirá como generar la instancia de la clase necesaria para la prueba de los métodos, en la determinación del ambiente de prueba.

3.2.1.1 Modalidades de prueba

Una vez considerado el sistema al que pertenece el método a probar, y escoger el método especifico que se quiere probar, se puede escoger la forma en la que se va a probar.

Contra especificación. La *prueba contra especificación* ejecuta el método y compara el resultado de dicha ejecución contra una especificación formal. El mínimo de especificación requerido consiste en la precondición y poscondición del método, la corrección de la prueba dependerá de que dicha especificación sea correcta y suficiente. Cualquier otra aserción que se utilice dentro de la especificación del método será verificada también. Este trabajo utiliza JML como lenguaje para las especificaciones, por lo que para la prueba contra especificación se requiere que todo método tenga especificado, mínimo, la precondición y poscondición. Todas las aserciones JML del método son revisadas.

Contra oráculo. La *prueba contra oráculo* ejecuta el método a probar con unos datos específicos y además considera un método oráculo que también se ejecuta con los mismos datos. La comparación de los resultados de las ejecuciones de ambos métodos es la que determinará el éxito o fracaso de la prueba. En esta modalidad también se considera la especificación del método a probar, pero la decisión sobre el éxito de la prueba es exclusiva del oráculo. La especificación del método oráculo no se tiene en cuenta, por lo que se considera que los datos que funcionan para el método a probar, deben satisfacer la especificación del método oráculo.
Una limitación de las pruebas contra oráculo está en la prueba de métodos con respuesta no funcional, pues si no hay respuesta no es posible hacer comparación, y por lo tanto no es posible verificar la corrección del método.

En ambas modalidades cada uno de los datos generados para ejecutar el método a probar, debe cumplir la precondición de la especificación JML de dicho método.

3.2.1.2 Determinación del ambiente de prueba

El *ambiente de prueba* de un método se refiere a la instancia de la clase que se necesita para que dicho método pueda llamarse y a todas las variables y objetos que el método pueda necesitar para ejecutarse. A estos objetos se les llama *mock-objects* [PAN1999]; usualmente estos objetos se generan 'a mano' utilizando el conocimiento sobre el software que se esta probando, aunque hay varias herramientas que facilitan su construcción.

Todo método que no sea constructor necesita la generación de una instancia de la clase a la que pertenece, para poder ejecutarlo. Dicha instancia puede ser generada solamente por el llamado a un constructor sin parámetros, pero también puede ser generada con una llamada a un constructor complejo, o mediante la llamada a un constructor y a algunos métodos para definir los atributos del objeto.
Sin embargo, la generación de dicha instancia solo permite validar el método sobre un objeto, de modo que la clase en sí no puede ser probada. Puede que el método funcione correctamente en una instancia de la clase generada con un constructor, pero en la instancia generada con otro no, lo que envilecería la confianza de la prueba.

Para contrarrestar estas dificultades, se propone, en la creación del ambiente de prueba, considerar el uso de todos los constructores (todos los que no hagan llamados al método que se está probando) y llamarlos de manera aleatoria. En el caso en que las características de la instancia se definan a través de métodos, la aleatoriedad debe definir dichas características, de modo que el método se valide contra diferentes instancias de la clase.

La definición del ambiente de prueba presenta otro problema, y es que todo método o constructor que sea llamado para construir la instancia de la clase, debe haber sido previamente validado. El problema está en que la forma en que se construye la instancia que se busca, puede querer llamar métodos que no han sido probados. Una posible salida es estratificar el caso de prueba, de modo que se prueban primero los métodos con un ambiente generado por constructores simples (sin parámetros, fácilmente validables a través de pruebas). Una vez se tienen estas pruebas es posible subir uno de los métodos un escalón y utilizar un constructor con llamados, esos llamados consideraran los métodos generados por los constructores simples y así sucesivamente hasta que se logren probar todos los casos.

En el caso en que se quiera probar un constructor, no se necesitará definir un ambiente específico, pues es claro que el constructor genera la instancia, y no es necesario que exista una instancia previamente.

3.2.1.3 Generación de datos de entrada

La generación de datos de entrada para el método que se está validando es un reto interesante, pues no solo deben generarse de manera aleatoria sino que deben cumplir los requerimientos de la precondición del método.

Se generan datos para cada uno de los parámetros del método[3] que se está probando. Se designa con n la cantidad de casos de prueba que se necesitan para satisfacer un nivel α de confianza. De este modo se generarán n conjuntos de datos, cada conjunto contiene tantos datos como parámetros tenga el método. Si el método no tiene parámetros, el conjunto es vacío.
La generación de datos depende del tipo del parámetro, se esperan diferentes entradas para poder realizar dicha generación:

- Si el tipo es un número (representado dentro de los tipos básicos *int*, *double*, *float,* o en los objetos *Integer,Double* o *Float*) se espera la distribución de probabilidad de que un número sea elegido.
- Si se trata de *char* se espera la distribución de que un carácter específico sea elegido, de alguna manera es como una distribución numérica truncada entre 33 y 255.
- Si se trata de un *boolean* se espera la distribución de probabilidad de que sea verdadero o falso.
- Si el tipo es *String* se espera la distribución de probabilidad de la cantidad de caracteres que tendrá y se genera una distribución uniforme sobre los caracteres posibles.
- Si el tipo es un arreglo o una lista (representados con [] o en objetos *List* o *Array*) se espera, además de la distribución de los elementos que la conforman, la distribución de probabilidad de que un tamaño dado sea elegido.
- Si los parámetros no caben dentro de los tipos anteriormente mencionados (denominados *simples* de ahora en adelante), se consideran *recurrentes*. En este caso, se proponen al usuario tres posibilidades: utilizar un constructor del objeto, generar un constructor o utilizar más de un constructor para el objeto. Esto se describe en detalle en la sección 3.2.1.3.1

La generación de datos simples se hace a través de las formulas de distribuciones. Se genera un dato numérico en todos los casos y se transforma a las necesidades del dato.

3.2.1.3.1 Generación de objetos recurrentes

Los objetos recurrentes son aquellos cuya construcción requiere como parámetro un objeto del mismo tipo que el objeto que se está creando. Por ejemplo, un árbol se puede construir vacío, con una raíz, o con una raíz y 2 árboles hijos. El problema está en como generar de manera aleatoria esos árboles.

---

[3] De nuevo se utiliza la palabra 'método' para referirse a todos los métodos incluyendo los constructores.

Se propone hacer la generación de objetos recurrentes a través de tres modalidades:

- **Uso de un constructor**
Todos los objetos tienen constructores, por lo que se escoge uno de esos constructores para generar la instancia. Si el constructor que se escoge tiene parámetros, se debe dar una distribución para cada uno de ellos, del mismo modo que para los parámetros del método.

- **Combinación de constructores (recurrencia)**
Cuando hay más de un constructor y se quiere asegurar aleatoriedad en la construcción de los objetos, se propone combinar los constructores, poniéndole una probabilidad a cada constructor de ser utilizado. De este modo se utiliza la distribución uniforme para escoger uno de los constructores para cada caso de prueba. Cuando se da el caso de que un constructor tenga como parámetros datos del mismo tipo que el objeto que construye, es decir constructores recursivos, los parámetros internos se generan con la misma combinación de constructores.

- **Generación de un constructor**
Puede darse el caso de que los constructores existentes no generan la instancia con todos los atributos que se necesitan, es decir es necesario generar código para generar un método que construya el objeto que se necesita. También se puede generar un nuevo constructor combinando los constructores y métodos estáticos de la clase.

Si los constructores tienen parámetros de tipo recurrente, se procederá recursivamente hasta que se puedan generar los objetos con datos simples.

3.2.1.4 Distribuciones de probabilidad para datos

La distribución de probabilidad de cada uno de estos datos debe ser definida por el programador. Como ya se discutió, el problema de decidir cual distribución tiene un grupo de datos no es trivial. Y aun suponiendo que se tienen datos suficientes para inferirla, esta puede no ser una distribución 'común', puede estar dividida por segmentos, de modo que un segmento se distribuye de un modo y otro de otro, o puede estar distribuida con una distribución común, pero no en todo el rango de datos, sino en una parte, por ejemplo `N(0,1)` entre `0.2` y `0.8`.

Considérese una variable aleatoria X que cuenta con una distribución conocida, es decir se conocen su función acumulativa F_x y su densidad de probabilidad f_x. Por la definición de función acumulativa se tiene que:

$$F_X(u) = Pr\{X \le u\} = \int_{-}^{u} fX(x)dx \quad , \; u \in \mathbf{R}$$

Todas las distribuciones comunes tienen un F_X y un f_X conocido.

Típicamente, se conoce cómo se distribuyen variables aleatorias de una variable real [CAR2006]:

$X \sim U(0,1)$:	$F_X(x) = x$	, $0 \le x \le 1$
	$f_X(x) = 1$	, $0 \le x \le 1$
$X \sim N(0,1)$:	$F_X(x) = \Phi(x)$	, $x \in \mathbf{R}$
	$f_X(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$	, $x \in \mathbf{R}$
$X \sim Exp(1)$:	$F_X(x) = 1-e^{-x}$	, $x \ge 0$
|	$f_X(x) = e^{-x}$	, $x \ge 0$
$X \sim T(0,1)$:	$F_X(x) = x^2$	, $0 \le x \le 1$
	$f_X(x) = 2x$	, $0 \le x \le 1$
$X \sim Tr(a)$:	$F_X(x) = (1-a)x^2 + ax$	, $0 \le x \le 1$
	$f_X(x) = 2(1-a)x+a$	, $0 \le x \le 1$

etc.

El *soporte* de una función $h:\mathbf{R}\to\mathbf{R}$ es el conjunto $s(h) = \{x:\mathbf{R} \mid h(x)\neq 0\}$. Nótese que la integral de f sobre un conjunto fuera de su soporte es nula.
En general, se puede suponer que $s(f_X) \subseteq [a,b]$, con $a,b\in\mathbf{R}\cup\{-\infty,\infty\}$ [CAR2006].

El problema está en cómo generar datos en las diferentes distribuciones. En el caso en que los datos estén uniformemente distribuidos en el intervalo $[0,1]$ no habría problema, pues la función que genera estos datos se encuentra en casi todas las plataformas computacionales.

El problema esta en la generación de datos para otras distribuciones. A través de la distribución uniforme es posible generar datos de otras distribuciones, tal y como lo demuestra [MEY1976], un caso especial de este teorema se muestra en [CAR2006]:

> Dadas una variable aleatoria $X: \Omega \to \mathbf{R}$, y una función $H: \mathbf{R} \to \mathbf{R}$, la función $Y = H(X)$ es también una variable aleatoria. La distribución de Y depende de la de X y de la forma de H.
>
> *Caso especial*
>
> Se tratará el caso en que:
>
> $X \sim U(0,1)$
>
> H creciente
>
> Existe una inversa izquierda para H, i.e., $H^{-1}(H(x)) = x$
>
> En este caso:
>
> $F_Y(y)$
>
> $=$
>
> $P(Y \le y)$
>
> $=$

$P(H(X) \le y)$

$=$ $\langle H^{-1}(H) = Id, H \text{ creciente} \rangle$

$P(X \le H^{-1}(y))$

$=$

$F_X(H^{-1}(y))$

$=$ $\langle X \sim U(0,1) \rangle$

$H^{-1}(y)$

Ahora, si se comienza con

$X \sim U(0,1)$

$Y \sim D$, donde D es una distribución de probabilidad conocida

y la pregunta es cuál H puede servir, se propone, precisamente:

$H: [0,1] \to \mathbf{R}$

tal que $H(x) = F_Y^{-1}(x)$.

Como F_Y podría ser no estrictamente creciente, la inversa no existe en el caso general. Sin embargo, se puede convenir en entender que

$$F_Y^{-1}(x) = (\inf y| : F_Y(y) = x)$$

entendiendo, además, que puede darse que $F_Y^{-1}(x)$ no sea finito.

Nótese que, en este caso, H^{-1} es no decreciente y cumple que $H^{-1}(y) = F_Y(y)$.

Ejemplo: Suponga que se quieren generar datos para una distribución $N(0,1)$.

La distribución de la función acumulativa normal se conoce como Φ. Usualmente se conocen tablas para Φ, i.e., $\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-u^2/2} du$.

Se debe elegir $H(x) = \Phi^{-1}(x)$. En otras palabras, una vez generado un valor $x \in [0,1]$, se busca en la tabla de Φ qué valor z es tal que $\Phi(z) = x$. Éste es el valor generado en la muestra de valores para Y.

Así, si se genera $x = .0104$, se comprueba que $\Phi(-2.31) = .0104$, de manera que -2.31 sea parte de la muestra para la variable normal.

3.2.1.5 Generación de datos aleatorios con una distribución dada

Si se quiere generar datos para una variable aleatoria $Y \sim D$, para la que es conocida su distribución de probabilidad, se define

$H: [0,1] \to \mathbf{R}$

como la inversa de la función acumulativa F_Y.

Ahora, partiendo de una variable $X \sim U(0,1)$, se genera una muestra aleatoria para X. Si los datos obtenidos se transforman con H, se obtendrá una muestra aleatoria para Y, distribuida según la probabilidad deseada.

Pero como se indicó antes, también se quieren poder generar construcciones sobre estas distribuciones, transformarlas para lograr que sea mas apropiada la distribución que se escoja para los datos.

Una de las construcciones posibles sobre las distribuciones consiste en la transformación lineal de variables aleatorias. Se quiere poder alterar la función de distribución de modo que se pueda escalar con un factor k y desfasar con un factor r.
La demostración de la posibilidad de realizar transformaciones lineales de variables aleatorias se muestra a continuación [CAR 2006]:

Para $k>0$, $r\in\mathbf{R}$, considérese la función $g: \mathbf{R} \to \mathbf{R}$ definida por

$$g(z) = k^{-1} f_X(\frac{z-r}{k})$$

La función g tiene las siguientes propiedades:

(g1) $g \geq 0$

(g2) $s(g) \subseteq [ka+r, kb+r]$

(g3) $\int_{-\infty}^{\infty} g(y)dy = 1$

(g1) es trivial.

Para mostrar (g2):

$z \notin [ka+r, kb+r]$

$=$

$z<ka+r \lor z>kb+r$

$=$

$\frac{z-r}{k}<a \lor \frac{z-r}{k}>b$

$\Rightarrow$ $\langle s(f) \subseteq [a,b]\rangle$

$f_X(\frac{z-r}{k})=0$

$\Rightarrow$

$g(z) = 0$

Y, para mostrar (g3):

$\int_{-\infty}^{\infty} g(y)dy$

$=$ $\langle s(g) \subseteq [ka+r, kb+r] \rangle$

$\int_{ka+r}^{kb+r} g(y)dy$

$=$ $\langle y:= kx+r;\ dy = k\ dx\rangle$

$\int_{a}^{b} k\ g(kx+r)dx$

$=$ $\langle \text{Def } g\rangle$

$\int_{a}^{b} f_X(x)dx$

$=$

1

En otras palabras, g es una densidad de probabilidad. Sea Y una variable aleatoria distribuida de acuerdo con la densidad de probabilidad g. Entonces:

$F_Y(u)$

$=$

$$\int_{-\infty}^{u} g(y)\,dy$$

$= \langle y:= kx+r;\ dy = kdx\rangle$

$$\int_{-\infty}^{(u-r)/k} k\ g(kx+r)\,dx$$

$=$

$$\int_{-\infty}^{(u-r)/k} f_X(x)\,dx$$

$=$

$$F_X\left(\frac{u-r}{k}\right)$$

Esta relación permite establecer una relación entre X e Y, ya que:

$\Pr\{Y \le u\}$

$=$

$F_Y(u)$

$=$

$F_X\left(\frac{u-r}{k}\right)$

$=$

$\Pr\{X \le \frac{u-r}{k}\}$

$=$

$\Pr\{kX+r \le u\}$

es decir[4], $Y = kX+r$.

Ejemplo 1

Sean $X \sim U(0,1)$, $k=3$, $r=0$. Se quieren determinar f_{3X} y F_{3X}. En este caso, para $0 \le x \le 3$:

$f_{3X}(x) = 1/3$

$F_{3X}(x) = x/3$

Claramente, se ve que: $3X \sim U(0,3)$.

Ejemplo 2

Sean $X \sim N(0,1)$, $k=\sigma$, $r=\mu$.

Se quieren determinar f_Y y F_Y sabiendo que $Y = \sigma X+\mu$. Entonces:

$f_Y(z)$

$=$

$f_X\left(\frac{z-\mu}{\sigma}\right)/\sigma$

$=$

$$\frac{\exp\left(-\left(\frac{z-\mu}{\sigma}\right)^2/2\right)}{\sqrt{2\pi}\sigma}$$

Y también:

$F_Y(u)$

$=$

[4] Excepto, a lo sumo, en un conjunto de probabilidad nula.

$\Phi(\frac{u-\mu}{\sigma})$

Es decir: $Y \sim N(\mu,\sigma^2)$.

Ejemplo 3

Sean $X \sim \exp(1)$, $k=\alpha^{-1}$, $r=0$.

En este caso: $Y = \alpha^{-1}X$. Por tanto:

$$f_Y(z) = f_X(\frac{z}{\alpha^{-1}})/\alpha^{-1} = \alpha\, e^{-\alpha x}$$

Y además:

$$F_Y(u) = F_X(\frac{u}{\alpha^{-1}}) = 1-e^{-\alpha x}$$

Es decir: $Y \sim \exp(\alpha)$.

También es posible hacer la transformación monótona derivable de variables aleatorias [CAR 2006]. [Mey1973] establece el siguiente resultado:

Teorema

Sea X variable aleatoria continua con densidad de probabilidad f_X, con $s(f_X) = [a,b]$.

Sea $Y = H(X)$, donde H es una función estrictamente monótona y derivable.

Entonces

$$f_Y(y) = f_X(H^{-1}(y))|dx/dy|$$

es una densidad de probabilidad para Y.

Si H es creciente, $s(f_Y) = [H(a),H(b)]$. Si H es decreciente, $s(f_Y) = [H(b),H(a)]$.

∎

En términos de **1**, si se tiene $H(x) = kx+r = y$, con $k>0$, H es estrictamente creciente y derivable. Entonces:

$$f_Y(y) = f_X(\frac{y-r}{k})|\frac{dx}{dy}| = f_X(\frac{y-r}{k})\ \frac{1}{k}$$

que coincide con los resultados de **1**.

Otra forma de manipular las distribuciones es truncandolas entre dos puntos, de modo que tenga la misma forma de la distribución original, pero solo tenga como dominio el intervalo deseado.

La posibilidad de realizar esta manipulación se explica a continuación [CAR2006]:

Para $c,d\in\mathbf{R}$, con $c<d$, se puede definir una nueva variable $X[c,d]$ que tenga una densidad de la misma forma que la de X en el intervalo $[c,d]$. En este caso, se

quiere que la probabilidad que cuenta X fuera del intervalo [c,d] sea agregada proporcionalmente a lo largo de este intervalo a la densidad de X[c,d].

Para definir la densidad de $X_{[c,d]}$ sea

$$m(c,d) = Pr\{ c \le X \le d \} = F_X(d)-F_X(c)$$

Para m(c,d)>0, se define:

$f_{X[c,d]}(x)$ = **if** $c \le x \le d$ **then** $f_X(x)/m(c,d)$ **else** 0 **fi**

Es claro que $f_{X[c,d]}$ es una densidad de probabilidad.

Además:

$F_{X[c,d]}(u)$

=

$\int_{-\infty}^{u} f_{X[c,d]}(y)dy$

=

$\int_{-\infty}^{u}$ (**if** $c \le y \le d$ **then** $f_X(y)/m(c,d)$ **else** 0 **fi**) dy

=

if $u < c$ **then** 0
elsf $c \le u \le d$ **then** $\int_{-\infty}^{u} f_X(y)/m(c,d)\ dy$
else 1
fi

=

if $u < c$ **then** 0
elsf $c \le u \le d$ **then** $F_X(u)/m(c,d)$
else 1
fi

Por otro lado, se puede observar que:

$Pr\{X_{[c,d]} \le u\}$

=

$F_{X[c,d]}(u)$

=

if $u < c$ **then** 0
elsf $c \le u \le d$ **then** $F_X(u)/m(c,d)$
else 1
fi

=

if $u < c$ **then** 0
elsf $c \le u \le d$ **then** $Pr\{X \le u\}/Pr\{c \le X \le d\}$
else 1
fi

=

$Pr\{X \le u \ \cap \ c \le X \le d\}/Pr\{c \le X \le d\}$

=

$Pr\{X \le u \mid c \le X \le d\}$

Es decir, X[c,d] se distribuye como $X_{|c \le X \le d}$.

Las transformaciones anteriores permiten una amplia gama de posibilidades para facilitar la definición de las distribuciones de probabilidad para los parámetros. Pero aun falta la posibilidad de combinarlas, dicha posibilidad debe establecer como factible la combinación lineal de distribuciones, como se muestra a continuación [CAR 2006]:

Sea X_i variable aleatoria, $i=1,\ldots,n$.

Sean $p_1,\ldots,p_n$ constantes tales que:

$$0 \le p_1,\ldots,p_n \le 1$$

$$(+i \mid 1\le i\le n: p_i) = 1$$

Si se define $g: \mathbf{R} \to \mathbf{R}$ por

$$g(z) = (+i \mid 1\le i\le n: p_i f_{Xi}(z))$$

es fácil ver que:

(g1) $g \ge 0$

(g2) $s(g) \subseteq [a_r,b_s]$, donde $a_r = (\min k \mid 1\le k\le n: a_k)$, $b_s = (\max k \mid 1\le k\le n: b_k)$.

(g3) $\int_{-\infty}^{\infty} g(y)\,dy = 1$

La función g es, de hecho, una densidad de probabilidad para la variable

$$Y = (+i \mid 1\le i\le n: p_i X_i)$$

Además, se tendrá:

$$F_Y(u) = (+i \mid 1\le i\le n: p_i F_{Xi}(u)).$$

En este caso, Y es una *combinación lineal convexa* de las X_i's.

Sea $[a_i,b_i]$ el soporte de la variable aleatoria X_i, $i=1,\ldots,n$. Supóngase que los soportes de las variables X_i's son disyuntos por pares:

$$[a_i,b_i] \cap [a_j,b_j] = \emptyset, \text{ si } i\neq j.$$

Entonces, vale además:

(g2') $s(g) = (\cup i \mid 1\le i\le n: [a_i,b_i])$

En este caso, Y es una *combinación lineal convexa disyunta* de las X_i's.

La propuesta que se hace para combinar distribuciones considera una serie de distribuciones, cada una con dominios que no se traslapan. Cada una de estas distribuciones tiene una probabilidad de ser escogida. Como ejemplo considérese la combinación de 3 distribuciones con funciones de densidad de probabilidad $f(x)$, $g(x)$ y $h(x)$ y probabilidades de ser escogidas α,β y σ con $\alpha+\beta+\sigma=1$. De ese modo se tiene $\int_{\lim\inf f}^{\lim\sup f} f(x)dx + \int_{\lim\inf g}^{\lim\sup g} g(x)dx + \int_{\lim\inf h}^{\lim\sup h} h(x)dx = 1$. Para generar un número aleatorio con la

distribución combinada se utiliza el mismo principio que para las distribuciones comunes, se toma un número a distribuido uniforme entre 0 y 1 y se decide cual distribución utilizar (se utiliza $f(x)$ si a está entre $\lim\inf f$ y $\lim\sup f$, se utiliza $g(x)$ si a está entre $\lim\inf g$ y $\lim\sup g$, etc.). Considérese que la función de distribución escogida es $f(x)$, se sabe que $\kappa * \int_{\lim\inf f}^{\lim\sup f} f(x)dx = \alpha$, de modo que se calcula κ y se aprovecha la linealidad de la función uniforme para transformar linealmente a a α. De este modo se calcula la integral con $x=a*\alpha$, y se continua la generación de datos igual que para las distribuciones comunes.

Esta cantidad de distribuciones posibles facilitan al programador la definición de la probabilidad de los parámetros

3.2.1.6 Número de datos de la muestra

En la sección 3.2.1 se discutió la forma de hallar el número de datos necesario n para una confianza α. Pero debido a las diversas formas de las distribuciones y a la posibilidad de no obtener los datos menos probables de la distribución, se considera la opción de utilizar muestreo aleatorio simple dentro de grupos equi-probables. Como se mencionó en el capítulo anterior, este tipo de muestreo divide el espacio muestral en grupos. El espacio muestral en este caso es el rango de probabilidades $[0,1]$ que se pueden obtener de una distribución. En este caso los grupos están definidos por terciles de probabilidad (*alta*, *media*, *baja*) de modo que para cada parámetro habría 3 divisiones. De esta manera, cada grupo estaría compuesto por una de las divisiones de cada parámetro que tenga el método, de modo que siendo k parámetros habría 3^k grupos.

Para cada grupo se generan n datos, de manera que se asegura que va a haber datos de todo el espacio muestral y habrá todas las combinaciones posibles de datos, mejorando la fiabilidad de la prueba.

La confianza del experimento está definida por el usuario, pero al tener varios grupos de prueba, ésta no es la misma confianza que se debe utilizar para calcular el n, pues la confianza del experimento está dada por la multiplicación de la confianza en cada uno de los grupos. De ésta forma, siendo α la confianza que se quiere de la prueba, se calcula β que es la confianza que se necesita dentro de cada grupo para obtener α en la totalidad del experimento: $\beta \geq e^{\frac{\log \alpha}{3^k}}$. Con ese β se calcula el número de datos necesarios n para cada grupo, como se explicó en la sección 3.2.1.

Repetir el experimento mejorará la fiabilidad del mismo. De este modo, manteniendo el mismo nivel de confianza se generan $n*3^k$ datos cada vez y se promedia el resultado de los casos de prueba. Para porcentajes de error muy pequeños, aumentar el número de

repeticiones (r) puede ayudar a encontrarlo, pues aumenta la cantidad de datos revisados del espacio muestral. La decisión sobre el número de repeticiones depende del nivel de confianza que se desee, del tamaño del espacio muestral y del costo que implique realizar cada repetición. El número de repeticiones no mejora necesariamente la confianza estadística, pero le da fiabilidad al intervalo de confianza, minimizando la posibilidad de que este sea incorrecto.

3.2.1.6.1 Convergencia del proceso

Al pensar en una cantidad finita de datos a generar ($r*n*3^k$) es claro que el proceso converge. Pero se había expuesto como un problema de la generación de datos la necesidad de que todos los datos generados cumplan la precondición definida en la especificación del método. Una forma de evitar que se generen datos inútiles es definir la distribución de los parámetros de modo que concuerde con esta especificación. Pero esto no siempre es simple, pues la especificación puede estar en términos de otros métodos o los límites pueden definirse de acuerdo con un atributo de la clase.

De este modo es ineficiente[5] asegurar la convergencia del proceso de generación de pruebas. La solución propuesta es determinar un límite de repeticiones. De esta manera, para cada repetición, en cada grupo, se generarían las n pruebas. Sea h el número de pruebas que no cumplieron la precondición. Es decir la probabilidad de no cumplir la precondición es de h/n, por lo que la segunda vez que se corra la prueba se necesitarían (n-h) pruebas, pero para asegurar que se de este número es necesario tener en cuenta la probabilidad de que no cumplan la precondición, por lo que se deben ejecutar (n*(n-h))/h pruebas. Cuando la probabilidad de falla es muy alta, el número de datos necesarios puede crecer demasiado, por lo que en este caso se harían 2*n nuevas pruebas, intentando encontrar algún dato que cumpla la precondición. En cada iteración, se recalcula la probabilidad considerando los cálculos anteriores, de modo que cada vez es más preciso el cálculo y cada vez se necesitan menos iteraciones
Bajo esta idea, se puede pensar en el peor caso posible, de manera que se pueda determinar el número de veces que debe iterar antes de decidir que no pudo encontrar n datos que cumplieran la precondición. Ya que la probabilidad se recalcula en cada iteración, el peor caso sería que en la primera iteración encontrara que la probabilidad de fracaso fue muy pequeña (menor a 0.001) y en la siguiente iteración encontrara que fue muy alta (mayor a 0.99) (un caso poco probable, pues con la misma especificación y la misma distribución, la probabilidad de no seguir la precondición no debería oscilar mucho). En este caso necesitaría muchas iteraciones para observar el error, por lo que si una probabilidad se diferencia de la siguiente en más de .5 se ignora el dato anterior y se toma el actual. En la mayoría de los casos 10 iteraciones son suficientes, pero si después de 10 iteraciones aun no se ha llegado a la cantidad n de datos necesaria, se puede aumentar la cantidad de datos en un factor determinado por dicho n, de modo que entre 10 y 20 iteraciones se aumente n/100, y entre 20 y 30, n/10. De este modo se acelera el proceso de convergencia. Aún en el peor de los casos probabilísticos, después de 30 iteraciones logra

[5] Si los datos existen, se puede realizar una verificación formal de todo el espacio, pero como se explico en la sección 2.1.1.1 este proceso es costoso en términos de tiempo y recursos.

la convergencia. Si no lo logra después de este punto puede significar un error en la especificación, o una discrepancia entre los límites de las distribuciones y las restricciones de la especificación.
De este modo se asegura la convergencia del proceso de generación de datos aleatorios, aun si no alcanza la totalidad de los datos requeridos, en cuyo caso se recalcularía la confianza y el resultado se presentaría con el nuevo cálculo de confianza.

Otro problema de convergencia en la generación de datos aleatorios está en la generación de objetos con constructores recursivos. La cantidad de veces que se recurre debe estar limitada, pues de lo contrario recurrirá (en teoría) al infinito. Una posible solución está en el uso de la técnica de generación de datos que considera la combinación de los constructores del objeto, utilizando probabilidades para decidir cual se usa. Con esta idea se puede reducir la probabilidad de utilizar constructores recurrentes en cada recursión. Sea `m` el tope máximo de recursiones, `w` la probabilidad de utilizar un constructor recursivo (si hay mas de un constructor recursivo, `w` será la suma de las probabilidades de utilizar cada uno de esos constructores), y `1-w` la probabilidad de utilizar constructores no recursivos. El objetivo de está técnica sería reducir gradualmente `w` de modo que en el `m-esimo` llamado recursivo, sea `0`. Entonces $w_k = ((m-k)/m)*w$.

3.2.2 Validación estadística de clases

Como se comentó previamente, la validación de una clase depende principalmente de la validación de todos sus métodos. Además se debe tener en cuenta al validar una clase que su invariante se mantenga a través de las diferentes instancias que pueda haber.

Es decir:

Clase correcta ≡ Todos sus métodos correctos
+ Invariante de clase mantenido al cambiar instancias

Así como la validación estadística de métodos concierne a las pruebas unitarias, la validación estadística de clases debe considerarse como una prueba de integración.

La validación de clases debe, no solo considerar la integración, sino pensar en escenarios en los que algún método pueda estar fallando, dada la especificación, y dado el hecho de que las pruebas no aseguran el `100%` de confianza.

3.2.2.1 Invariante de clase

El invariante de clase es un conjunto de aserciones que deben mantenerse como verdaderas al generarse la instancia de la clase y después de ejecutar cualquier método de la misma. El

invariante de clase considera los atributos de la clase para definir sus aserciones. Estas aserciones hacen parte de la especificación del producto de software que se este probando.

Los momentos más importantes para verificar dicha invariante son al generar la instancia de la clase y cuando se ejecuta un método modificador.

La invariante debe mantenerse en todas las posibles instancias de la clase, lo que presenta un problema para asegurar corrección, ya que los atributos de la clase pueden tener rangos muy amplios, lo que implicaría tantos casos de prueba como para hacer ineficiente el proceso de corrección. Entonces el proceso de validación es el mismo que con los métodos: generar de manera aleatoria una serie de instancias (para un nivel de confianza dado) y verificarlas. La generación de dichas instancias no debe hacerse a la ligera, deben considerarse los métodos de la clase, la forma en la que estos métodos se utilizan y la forma en la que afectan la instancia, de modo que sea posible verificar a través de ejecuciones de todos aquellos métodos la invariante de clase al generar una instancia.

3.2.2.2 Planes de prueba: Integración

Del mismo modo que las pruebas unitarias, las pruebas de integración también requieren un plan definido para asegurar la calidad del proceso de pruebas.

En el caso de las pruebas de integración, el plan puede concebirse principalmente de dos formas [CUL1996]: *top-down* y *bottom-up*.

En las pruebas *top-down* las rutinas de control de alto nivel se prueban primero, utilizando código (*stubs*) que simule el comportamiento de los objetos reales que no se han probado.
En las pruebas *bottom-up* cada modulo se prueba individualmente y se combinan en una colección que es probada. Este proceso continua hasta que la colección corresponda a la totalidad de la aplicación.

El DAG de plan de pruebas considera las pruebas de integración, ya que para poder probar una clase es necesario haber probado todos los métodos de la misma, y probar la invariante de la clase. Por lo tanto, el DAG es un tipo de prueba *bottom-up*.

3.2.3 Validación estadística de productos de software

La validación de un producto de software depende de la validación de todos sus componentes. En el nivel más pequeño, un producto de software puede ser un paquete, en cuyo caso sería necesaria la verificación de todas sus clases. La misma idea se aplica para productos de software más grandes, como subsistemas y sistemas, por lo que solo se hará referencia a productos como paquetes.

En términos generales:

Producto correcto ≡ Toda clase correcta.

La validación de productos concierne a las pruebas de sistema.

La validación estadística considera la posibilidad de generar escenarios aleatorios para cada prueba en el plan de pruebas de sistema. De nuevo, usando una confianza definida, se puede asegurar la corrección del sistema, mientras los escenarios de prueba sean independientes.

3.2.3.1 Planes de prueba: Sistema

Las pruebas de sistema verifican la corrección del producto contra los requerimientos funcionales del sistema. En esta etapa también se busca encontrar cualquier comportamiento no deseado que se haya pasado por alto en pruebas anteriores.

El plan de prueba de sistema considera las siguientes pruebas [CUL1996]:

- *Pruebas de regresión.* Se trata de utilizar de nuevo pruebas que ya fueron utilizadas con versiones anteriores del producto, asegurando que las características requeridas en la versión anterior funcionen en la nueva versión.
- *Pruebas de recuperación.* Son pruebas en las que el producto se expone a interrupciones de diversa naturaleza, como removiendo la conexión con el disco duro o apagando el computador, de manera que se asegure que el producto esta preparado para recuperarse de este tipo de situaciones sin perdida de información.
- *Pruebas de seguridad.* En estas pruebas se intenta de manera no autorizada operar el producto, obtener acceso a los datos, envilecer la instalación del producto o envilecer el producto mismo. El objetivo es encontrar posibles entradas y dificultar el acceso no autorizado al producto.
- *Pruebas de stress.* Se hacen demandas anormales al producto, aumentando la tasa en la que debe recibir datos, o la tasa en la que debe producir información, de manera que se observe cuando se están hacienda demandas excesivas al sistema operativo o a la máquina en la que se esta ejecutando el producto.
- *Pruebas de desempeño*. En este caso se chequean los requerimientos de desempeño, como el tamaño del software cuando se instala, la cantidad de memoria principal que requiere, etc.
- *Pruebas de usabilidad.* Se trata de validar la usabilidad del producto contra los requerimientos del mismo.
- *Pruebas alpha y beta*. Estas pruebas se realizan cuando el software es entregado a los usuarios finales. En la entrega inicial, alpha, se seleccionan ciertos usuarios que se espera reporten defectos y otras observaciones. Cuando la aplicación a pasado a través de las pruebas alpha se entrega una versión beta, posiblemente incorporando los cambios necesarios. Esta versión se entrega a un grupo mas grande y

representativo de los usuarios finales, en busca de defectos, que son corregidos antes de entregar la versión final.

En términos del plan de pruebas en DAG, las pruebas de sistema consistirían en la revisión de la totalidad de los nodos del grafo. Todos los tipos de pruebas necesarios en esta fase pueden considerar al DAG como centro del plan, pues a través de este es posible determinar lo que se ha hecho y lo que falta, y por tanto es posible considerar el sistema como una sinergia de partes.

3.2.4 Reutilización de pruebas

Los procesos de software buscan el mejoramiento continuo de los sistemas, y los planes globales de pruebas consideran la aplicación de las mismas durante todo el proceso de desarrollo. Pero cada vez que se modifica un producto es necesario volver a probar todos sus componentes de modo que sea posible asegurar la corrección de sus componentes para poder continuar el proceso de pruebas. [BRU2002]

Volver a generar las pruebas sería un proceso desgastante y con un costo innecesario, pues las pruebas que ya se ejecutaron y demostraron la corrección del programa contra su especificación siguen siendo validas.

Otra necesidad de reutilizar las pruebas sucede cuando estas demuestran un error en el producto. Si se encuentra una falla el primer paso sería arreglarla, y se quisiera ejecutar el mismo caso de prueba en que fallo para asegurar que las modificaciones realizadas hallan arreglado ese defecto.

Del mismo modo, la reutilización de pruebas sirve cuando hay cambios menores en la especificación, cambios que no involucren la precondición de la unidad que se está probando, pues la poscondición se deberá seguir cumpliendo con todos los datos que sigan la precondición.

La reutilización de las pruebas se observa como un punto importante de la metodología propuesta. Los dos puntos a analizar son la forma en la que dichas pruebas se deberán almacenar y la forma en la que se hará la regresión de dichas pruebas.

3.2.4.1 Almacenamiento de pruebas

El almacenamiento de pruebas debe considerar cada una de las partes de la prueba y considerar los datos con los que se generó la prueba. Se detallarán los datos que se consideran importantes, dentro de esta metodología, para guardar una prueba para cada uno de los módulos revisados en este capítulo: método, clase y producto.

Método

La validación estadística para los métodos considera $r*n*3^k$ casos de prueba, que deben ser almacenados.

En general, se necesita almacenar la ubicación del producto al que pertenece el método, la clase, el nombre del método y sus parámetros (por que puede haber mas de un método con el mismo nombre, entonces es necesario diferenciarlos a través de los tipos de datos de los parámetros). Con la información anterior ya sería claro cual es el método al que pertenecen los casos de prueba.

Cada una de las características de la prueba debe ser almacenada, como la modalidad de prueba (contra especificación o contra oráculo), el nivel de confianza con el que se realizó la prueba, el número de repeticiones del experimento que se incluyeron en la validación, la fecha en la que se realizó la prueba y el resultado de la misma (éxito o error). El ambiente de la prueba también debe guardarse, tanto el ambiente para el método de prueba como el ambiente para el método oráculo. La distribución con la que se generaron los datos debe ser guardada. La característica más importante de la prueba son los casos de prueba que conforman a la misma.

Cada caso de prueba consta de los datos generados con la distribución dada para cada parámetro. Cada uno de esos datos debe ser guardado. Lo ideal es guardar el dato de modo que pueda ser leído por el programador, como un documento de texto, pero como estos datos pueden ser objetos complejos, la única forma de guardarlos es mediante serialización. Entonces sería bueno tener dos archivos de datos, uno para leer normalmente, y uno para que sea leído por el computador para regenerar la instancia del objeto que se uso en la prueba.

Clase

Las pruebas de integración también deben considerar la localización de la clase a al que se le estén realizando las pruebas, y del producto al que pertenece la clase.

Cada uno de las instancias de la clase generadas para las pruebas debe guardarse, y de nuevo, por ser objetos complejos, la serialización representa una buena posibilidad.

Los constructores y métodos que se utilicen para generar cada instancia en cada caso de prueba, deben ser guardados en el orden en que fueron utilizados. Ya que existe la posibilidad de generar estas instancias utilizando herramientas estadísticas, las probabilidades que se asignen al uso de cada método y las distribuciones de los parámetros que se utilicen deben ser almacenadas también.

Para la prueba de la clase también debe guardarse el resultado de la prueba y la forma en la que se generaron los objetos *mock*, distintos a las instancias de la clase en si, que se necesitan para ejecutar las pruebas.

Debido a que las pruebas de integración consideran las pruebas unitarias como base para su realización, es necesario almacenar las pruebas unitarias realizadas a cada uno de los métodos dentro de la clase.

Producto

En el caso de las pruebas de sistema es necesario considerar la ubicación del producto dentro de un sistema mayor, si es que existe dicho sistema. De nuevo es necesario guardar el resultado de la prueba.

Datos como las pruebas de integración de los subsistemas del producto son de suma importancia para la generación de las pruebas de sistema, por lo que también deben ser almacenados.
Para cada tipo de prueba de sistema que se decida hacer deben almacenarse los escenarios generados, y la forma en la que dichos escenarios fueron generados, es decir las distribuciones probabilísticas utilizadas y los objetos *mock* que pudieran ser necesarios.

JUnit [JUN2007]

Como se expuso en el capítulo anterior, JUnit es un *framework* de pruebas muy utilizado, por lo que las pruebas generadas con esta metodología podrían quererse almacenar como pruebas JUnit también. JUnit solo considera pruebas unitarias, por lo que solo se podrían almacenar las pruebas que se generaron para los métodos y para las clases (JUnit considera las clases como unidades). El ideal es que de generen las pruebas JUnit y se puedan utilizar los datos generados aleatoriamente.

3.2.4.2 Regresión

El proceso de regresión dentro de la validación consiste en volver a ejecutar los casos de prueba cuando se considere necesario [BRU2002], por ejemplo cuando se hace un cambio al producto, cuando se realizan cambios en la poscondición o la invariante, cuando se completa un modulo (clase, paquete, subsistema, sistema) y se quiere revisar que todo siga funcionando. Como se expuso antes, el proceso de regresión también es útil cuando la prueba falla y se hacen correcciones, pues se quiere observar el comportamiento del producto con el mismo dato con que falló.

Así como la prueba se define a través de ciertas características diferentes para métodos, clases y productos, el proceso de regresión también en diferente en cada caso. Aunque en general se querría que la prueba se ejecutara exactamente igual que la primera vez, dependiendo del cambio que se haga puede que se quieran modificar algunas características de la misma.

Método

En el caso de los métodos, la regresión debe mantener varias características sin cambio, como el método elegido, el proyecto al que pertenece, la modalidad de prueba (especificación u oráculo), la confianza, el número de repeticiones, el ambiente generado para el método, las distribuciones utilizadas para los parámetros. Si se hacen cambios a estas características se estaría hablando de una prueba diferente, por lo que no sería regresión.
Se debe poder cambiar los datos, de modo que o sean los mismos que se utilizaron en la primera prueba o sean generados de nuevo con las mismas características que se generaron los datos anteriores.

Clase

La regresión de una prueba de integración es recursiva, ya que debe considerar la regresión de las pruebas unitarias, en este caso las pruebas de los métodos. Específicamente, la regresión de la validación de una clase debe permitir conservar las distribuciones con las que se usó cada constructor para generar cada instancia, y las probabilidades de aplicar cada método a cada uno de esos constructores. Los datos específicos de la prueba original pueden, de nuevo, ser reutilizados, o regenerarlos utilizando las distribuciones que se conservaron.

Producto

Para los productos la regresión implica considerar la regresión de las pruebas de integración, y dentro de estas las de las pruebas unitarias.

Las distribuciones con las que se generaron los diferentes escenarios fueron almacenadas, por lo que de nuevo, los datos de prueba pueden ser reutilizados o regenerados utilizando la información almacenada.

Toda modificación que sufra el producto está sujeta a la verificación a través de pruebas de regresión.

Orientada por DAG de plan de pruebas

La regresión puede estar orientada a través del DAG del que se habló anteriormente.

En este caso, la regresión consideraría, para la prueba de un modulo de software, el probar todos los hijos que este modulo tenga en el grafo.

De este modo, el plan de pruebas del DAG se revisaría al revés: en el plan de pruebas se empezaba de las hojas hacia las raíces, en el caso de las pruebas de regresión, un cambio en un modulo hace que se repitan las pruebas a los hijos de ese modulo y luego las pruebas a los ancestros de ese modulo.

Considérese el grafo utilizado como ejemplo en el capitulo anterior. En el momento en el que se haga un cambio a la clase *CuentaAhorros* es necesario revisar si las clases vinculadas a esta están o no revisadas. En la Figura 9 se muestra un fragmento de dicho grafo. Supongase que el producto estaba totalmente validado, el cambio en la clase *CuentaAhorros* requiere que se revisen muchas de las pruebas que ya se habían hecho. Los nodos que no necesitan de regresión tienen un símbolo verde que lo indica, mientras que los que si necesitan tienen un símbolo rosado. El nodo en el que se generó el cambio tiene un sol negro. Todos los nodos que dependen de este último deben ser parte de la regresión.

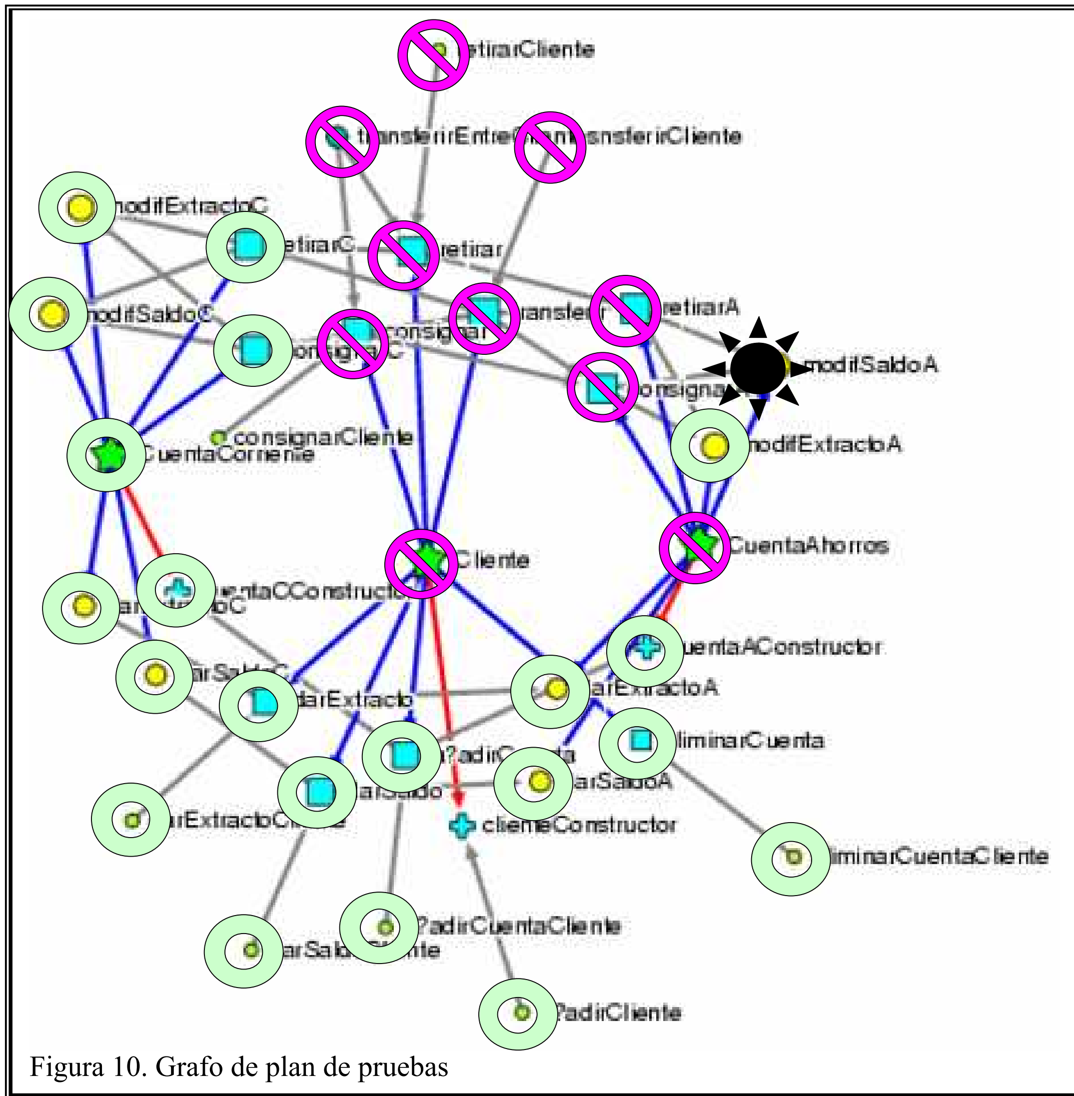


Figura 10. Grafo de plan de pruebas

Capítulo 4

JSvt: Una herramienta para validación estadística

Resumen. JSvt es un producto de software implementado en Java que utiliza JML, e implementa parte de la metodología descrita en el capítulo 3. Específicamente, JSvt implementa la validación estadística de métodos, y la reutilización de pruebas también para métodos. Se describe la forma en la que se implementó la metodología, tocando los puntos importantes y complejos del proceso, como lo son la generación de datos y la convergencia de este proceso, la generación de instancias de clases para probar los métodos, la ejecución de los casos de prueba y el almacenamiento y regresión de pruebas.

El proceso de validación estadística descrito en el capítulo 3 se implementó como un software para la generación automática de pruebas. JSvt (*Java Statistic Validation Tool*) es el resultado de esa implementación, no se implementaron todas las ideas propuestas, pero se llegó a una herramienta interesante que será descrita a continuación.

JSvt es una herramienta que permite generar casos de prueba estadísticamente para la validación de métodos (únicamente).

Como se afirmó en el primer capítulo, muchos autores han considerado que la generación de datos utilizando distribuciones de probabilidad puede disminuir la confianza de la validación, al no poder definir claramente las distribuciones de los parámetros, razón por la que JSvt considera todos los puntos expuestos en el capítulo pasado sobre la generación de datos aleatorios, incluyendo las transformaciones a las distribuciones que fueron expuestas allí. [DUI2004]

JSvt permite también la regresión de pruebas, de modo que permite almacenar las pruebas que se le hagan a los métodos, y los datos de dichas pruebas. La regresión permite utilizar los mismos datos que se utilizaron en la prueba original y también permite volver a generar los datos.

En JSvt también se pueden generar pruebas JUnit, de modo que con los datos salvados y los cascarones de las pruebas se puedan generar las pruebas JUnit y correrlas en ese *framework*.

La funcionalidad de JSvt esta dirigida a la corrección de errores. Por lo tanto una parte importante de las características de JSvt es la posibilidad de encontrar el dato con el que se generó el error. También hace parte de esta funcionalidad la posibilidad de ver todos los datos en los que se generó error.

Los objetivos y alcance de la herramienta, junto a su arquitectura, requerimientos y decisiones de implementación se definirán a lo largo de este capítulo.

4.1 Objetivos y alcance

El objetivo principal de esta herramienta es automatizar la generación de pruebas unitarias de métodos, de manera que facilite la validación estadística que se describió en el capítulo 3.

Esta herramienta permite escoger un método específico, escoger una modalidad de prueba (oráculo o especificación), escoger una distribución para cada parámetro del método, escoger un nivel de confianza y un número de repeticiones.

La primera versión de JSvt solo se encarga de pruebas a métodos. Las pruebas de clases y productos de software hacen parte del trabajo futuro. Sin embargo, es posible revisar el invariante de clase durante la prueba de cada método, de modo que si se incumple dicha invariante se presenta un mensaje de error exponiendo el problema. De este modo, es posible generar un plan de pruebas que considere el uso de estas pruebas unitarias para definir diferentes instancias de la clase. El modelo específico que permite la generación aleatoria de instancias no está implementado aún.

4.2 Requerimientos

JSvt está desarrollado en Java, por lo que corre en varias plataformas. Se ha probado a satisfacción en Linux y en Windows.

JSvt requiere Java 1.5 para su correcto funcionamiento. Es necesario que esté instalado el ambiente de desarrollo (jdk), no solo el ambiente de ejecución (jre). Esto debido a que JSvt hace llamados a la máquina virtual de Java y por lo tanto hace uso del compilador de Java

Una de las funcionalidades de JSvt permite generar los archivos Java que hacen uso de los *frameworks* de JUnit, generando pruebas que utilizan JML y que pueden ser corridas

utilizando la interfaz gráfica de JUnit. Esta versión de JSvt utiliza JUnit 3.8.1, por que es la versión compatible con JML.

La versión actual de JML es la 5.2 y JSvt es compatible con dicha versión.

Eclipse se utilizó como herramienta de desarrollo, por lo que JSvt tiene archivos *.project* que permiten que sea fácilmente importado a Eclipse y fácilmente ejecutado desde este programa.

4.3 Organización del software

Para que sea posible para JSvt generar las pruebas, el software debe estar organizado y anotado de manera particular.

La organización del software debe ser en subsistemas, y dichos subsistemas deben estar divididos en paquetes. La idea de que estén organizados en subsistemas viene de la necesidad de ejecución de cada método que se esté probando. Para ejecutar código Java sin problemas es pertinente que todas las clases que se comuniquen sean conocidas. Por lo tanto es necesario conocer cual es el subsistema para poder compilar y ejecutar dicho subsistema dentro de JSvt, y poder generar las pruebas. Si hubiese librerías o clases adicionales que necesite el software que se está probando, estas deberían ser adicionadas al *classpath* cuando se este haciendo la prueba, por lo que JSvt ofrece la opción de añadir dichas librerías al *classpath* que se usa para generar y ejecutar las pruebas.

La distribución en paquetes se pide para facilitar la localización del método que se está probando, y para facilitar la localización de las pruebas y datos generados, además por que se considera que la distribución del software en paquetes es una buena práctica para facilitar la ejecución de un buen plan de pruebas.

4.3.1 Anotación de especificaciones

El software a probar debe estar anotado. Toda especificación que se haya determinado que el software debe cumplir, debe estar escrita en forma de aserciones JML, tanto para los métodos como para las clases.

Como se declaró en el capítulo anterior, la fiabilidad de la prueba depende de qué tan buena sea la especificación. JSvt pide como mínimo, para asegurar la bondad del experimento estadístico, que el software esté anotado con precondiciones y poscondiciones.

Si las clases se anotan con invariantes de clase, las pruebas que se ejecuten a los métodos considerarán estas invariantes como parte de la prueba, de modo que aun si el método es correcto, si se incumple la invariante de clase, se generará un error, y dicho error especificará su naturaleza como error de invariante y no error de precondición o poscondición. JML también permite añadir aserciones dentro de los métodos, para especificar invariantes de ciclos o aserciones sobre las variables que se utilizan dentro del método. Si estas aserciones existen, JSvt también verificará la corrección del método contra estas especificaciones.

Si el software no está anotado, y se le pide a JSvt que pruebe contra especificación, el resultado será una prueba que indique que el método probado es correcto, pues su precondición y poscondición son supuestas como *true*.

La forma en la que se anota con JML se describe en el Anexo A.

4.4 Arquitectura de JSvt

JSvt es un sistema que considera dos paquetes (interfaz y kernel) y una librería de distribuciones, en términos generales.
Los diagramas de clases de cada paquete y de la librería, junto a una corta descripción de las responsabilidades de sus clases, se presentarán a continuación.

4.4.1 Librería de distribuciones

Se utilizó la librería de distribuciones probabilísticas realizada por Kyle Siegrist [SIE2004]. Esta librería permite generar distribuciones de diversos tipos, y encontrar los valores de sus funciones de distribución y sus funciones acumuladas. Específicamente se utilizó el paquete *edu.uah.math.distributions*, el cual contiene encapsulamientos Java de objetos matemáticos, incluyendo distribuciones de probabilidad y estructuras de datos. La clase *Distribution* es una clase abstracta que es padre de todas las demás clases que representan distribuciones probabilísticas. Las clases *Data* e *IntervalData* representan estructuras de datos, estas dos clases no son utilizadas por JSvt. La clase *Domain* representa la partición en intervalos del dominio de una distribución. La clase *Functions* es una colección de métodos estadísticos para calcular las funciones especiales que la mayoría de las distribuciones utilizan. [SIE2004]

Esta librería fue modificada para permitir la realización de transformaciones lineales sobre las funciones de distribución de probabilidad, y para permitir la combinación lineal de dichas funciones. En la siguiente sección se explicará como se implementó esta modificación.

Las distribuciones permitidas en JSvt son las que aparecen en el diagrama de clases a continuación. La librería utilizada ofrece otras distribuciones que no se tuvieron en cuenta.

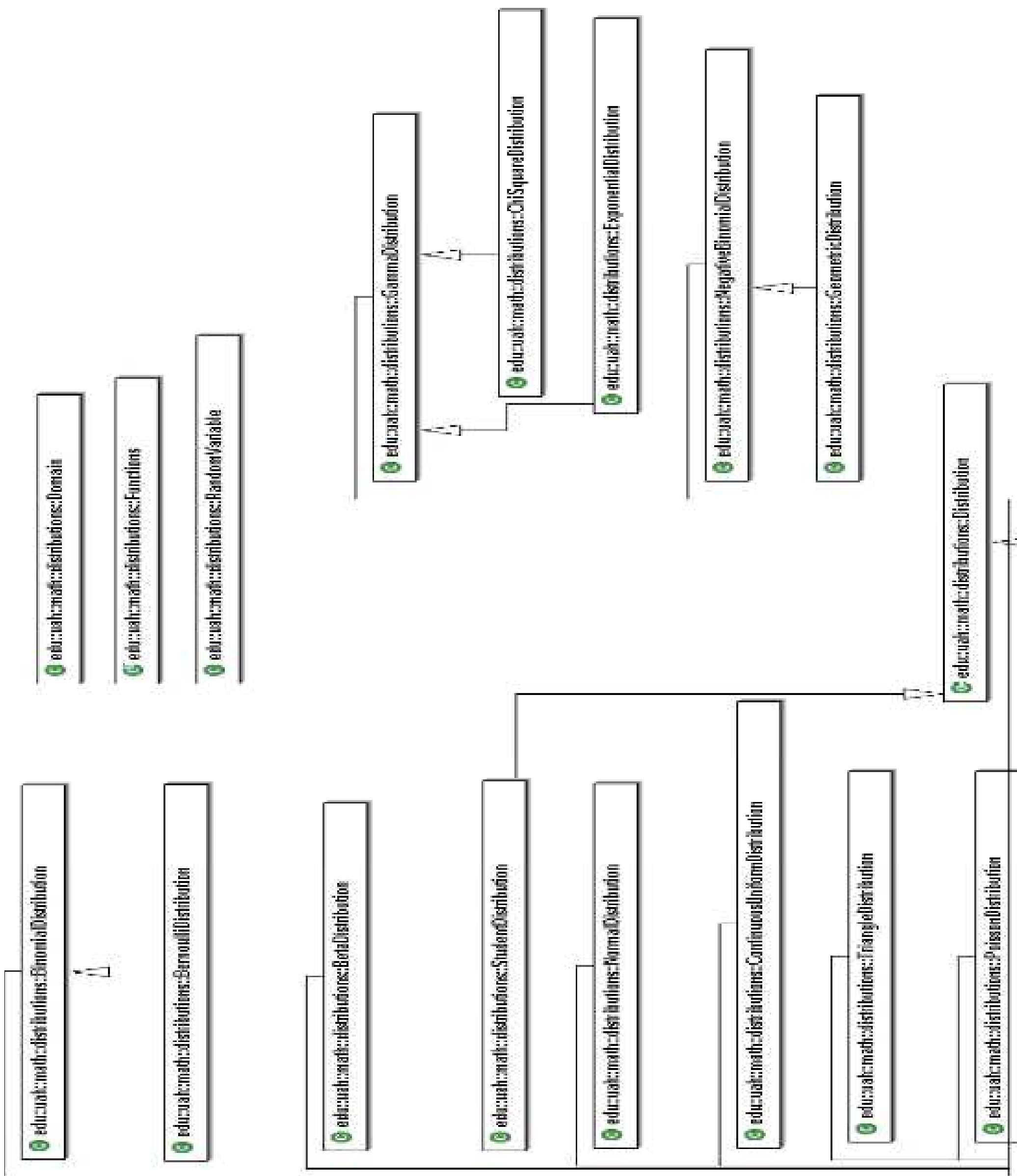


Figura 11. Diagrama de clases de la librería de distribuciones.

edu::uah::math::distributions::Distribution

CONTINUOUS: int
DISCRETE: int
MIXED: int

buscarProb()
buscarProbTrun()
getCDF()
getCDFTable()
getCDFTrun()
getCDFTrunTable()
getDensity()
getDensityTable()
getDensityTrun()
getDensityTrunTable()
getFailureRate()
getMGF()
getMaxDensity()
getMean()
getMedian()
getMoment()
getMoment()
getPGF()
getQuantile()
getQuantileTable()
getQuantileTrun()
getQuantileTrunTable()
getSD()
getVariance()
simulate()
toString()

Figura 12. Clase *Distribution*

4.4.2 Kernel

La lógica de la aplicación se centra en este paquete. Se utilizó el patrón de *fachada* para su construcción.
La clase *ConstrGenerados* es una clase que funciona como cascaron, que se re-escribe con cada validación que se realiza, de modo que en esta clase se implementan constructores para objetos complejos (esto se explicará mas adelante).
La clase *ConjConst* representa conjuntos de constructores, es decir varios constructores de un mismo objeto que se utilizan con diferentes probabilidades para generar instancias de dicho objeto.
La clase *Parametro* representa las características necesarias para poder generar un dato aleatorio para usar como parámetro en un método.
La clase *Prueba* contiene atributos para almacenar todas las características de una prueba. Por medio de esta clase es posible guardar y recuperar una prueba.
La clase *Fachada* contiene el centro de la funcionalidad de JSvt. A través de esta clase es posible crear y ejecutar una prueba, generar datos aleatorios para cada parámetro que lo necesite, guardar y recuperar pruebas, generar y ejecutar pruebas JUnit-JML para una clase, y conducir el experimento estadístico que permite validar el software.

4.4.3 Interfaz

La interfaz de JSvt está planeada para ofrecer al usuario facilidad en la selección de características para una prueba específica.

En esta sección se presentarán algunas de las interfaces de JSvt, mientras se comenta la secuencia de pasos de la generación de una prueba.

La ventana principal de JSvt permite seleccionar un sistema y dentro de ese sistema una clase especifica sobre la que se va a realizar la prueba. En esta ventana es posible seleccionar la modalidad de prueba (especificación u oráculo), y en caso de que se escoja trabajar con un oráculo se debe escoger el sistema y la clase que contiene al método oráculo. También se define se quiere generar los datos aleatorios o se quiere utilizar unos datos serializados ya existentes.
Una vez seleccionada la clase, se presenta una ventana con la lista de métodos que esa clase contiene, de modo que el usuario seleccione uno de ellos para la prueba. Si se ha seleccionado probar contra un oráculo, entonces se presentará la opción de escoger el método oráculo.

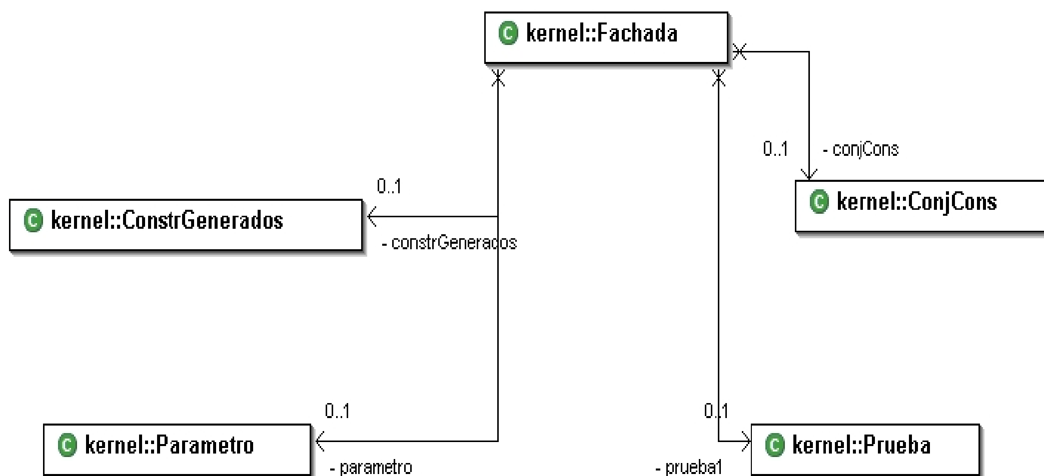


Figura 13. Diagrama de clases del paquete *kernel*

Según el método elegido, se presenta una lista con sus parámetros. Para cada parámetro se debe definir una distribución para poder continuar con el proceso de validación. Si el parámetro es numérico*, String, char* o *boolean*, la ventana que se presenta es muy parecida,

donde se le pide al usuario escoger la distribución y parámetros de dicha distribución para el dato que se esta observando. Si se trata de un arreglo o una lista se presentan 2 ventanas, una para seleccionar la distribución de la cantidad de datos y otra para seleccionar la distribución para generar los objetos dentro de la lista. Si el tipo del parámetro es un objeto complejo se presenta al usuario la posibilidad de utilizar uno de sus constructores, de generar un método que construya el objeto o de combinar sus constructores con diferentes probabilidades.

Una vez definida la forma en la que se van a generar los datos, se pide al usuario definir el ambiente del método que se va a probar. En este caso el usuario debe codificar la forma en la que se va a generar la instancia de la clase que contiene el método que se está probando. Si existe un método oráculo, se deberá definir el ambiente para este.

Finalmente se le pide al usuario definir la confianza que quiere para la prueba, el número de repeticiones y la cantidad de errores que quiere ver. Esto último por que puede que solo necesite ver el primer error, y con eso ya quiera hacer cambios, o puede querer ver todos los errores y modificar la aplicación respecto a ellos.

Se presenta una ventana de progreso, y al terminar se presenta una ventana que indica cual es la probabilidad de que el programa este errado dada la confianza deseada.

En la ventana inicial también es posible hacer regresión de una prueba. Se le pide al usuario la ubicación de dos archivos: uno que contiene el *path* en el que se encuentra la clase a probar y otro que contiene los datos específicos de la prueba. La razón de que sean dos archivos es que el primero es una archivo de texto que el usuario puede cambiar manualmente si modifica la ubicación de las carpetas, mientras que el segundo es un archivo serializado que no es posible modificar manualmente.
Una vez escogidos estos archivos se pasa a una ventana que muestra los datos de la prueba y que permite volver a correr la prueba.

4.5 Validación de métodos con JSvt

Ya que el objetivo principal de JSvt es permitir la validación estadística de métodos, todos los puntos que se trataron en el capítulo anterior debieron ser implementados.

La implementación de la validación estadística de métodos debe considerar como un punto focal la generación de datos aleatorios. El problema principal de esta generación de datos es el sinnúmero de tipos a los que estos datos pueden pertenecer. De este modo la implementación debe ser lo suficientemente flexible como para crear cualquier objeto posible. Es claro que todo objeto del que se quieran generar instancias debe ser conocida como parte del proyecto.

La implementación de JSvt implicó considerar el uso interno de la maquina virtual de java, para poder compilar de nuevo todas las clases que hiciesen parte del proyecto, y también para compilar en tiempo de ejecución utilizando las librerías JML.

4.6 Cómo se implantó la validación estadística de métodos

Para cada uno de los puntos importantes de JSvt se comentará sobre la forma en que se implementó.

4.6.1 Generación de datos

Para poder generar datos para las diferentes distribuciones fue necesario considerar las formulaciones explicadas en la sección 3.2.1.4. De manera práctica se generó, con distribución uniforme, un número aleatorio a entre 0 y 1, lo que es fácil, ya que Java incluye esta función.
La dificultad que se presenta está en contar con la función acumulada inversa para todas las distribuciones. Con ayuda del paquete de distribuciones de [SIE2004] se sorteó esta dificultad.

- **Distribuciones Transformadas**

  Para generar datos de distribuciones transformadas se utilizo el concepto de transformación lineal que se describió en la sección 3.2.1.5. En términos generales el usuario no define los parámetros específicos de la transformación, pero si define los límites en los que va a truncar la distribución.
  El desfase de funciones de distribución no es permitido en la primera version de JSvt.

- **Distribuciones Combinadas**

  El usuario elige los límites, que no se traslapen, para cada una de las distribuciones que quiere combinar. Utilizando los conceptos de la sección 3.2.1.5 se generan los datos para este tipo de distribuciones.

La forma en la que se generan objetos recurrentes se implemento tal y como fue descrita en 3.2.1.3.1.

4.6.2 Definición del ambiente de prueba

En JSvt, el ambiente de prueba se define a través de la codificación de un método que retorna una instancia de la clase que contiene el método, de modo que a través de esa instancia se hacen los llamados al método durante la ejecución de las pruebas.

JSvt ofrece el espacio para la codificación de ese método, y permite compilarlo.

Si existe un oráculo, se ofrece espacio para codificar la instancia de la clase que contiene el método oráculo.

4.6.3 Ejecución de los casos de prueba

La ejecución de los casos de prueba debe hacerse con el uso de JML para poder verificar la especificación. Por esta razón la ejecución de dichos casos se debe hacer utilizando el compilador de JML, y luego ejecutarlas con el *runtime assertion checker* de JML (jmlrac). La ejecución del compilador de JML implica un llamado a la consola de comandos del sistema a través de la maquina virtual de Java.

4.6.3.1 Uso de jmlrac

Para usar el *runtime assertion checker* de JML es necesario ejecutar una clase que tenga un *main*. Los llamados a este ejecutable son costosos en términos de tiempo, por lo que se decidió ejecutar tantas pruebas como sea posible por vez.
Las llamadas a jmlrac se hacen después de generar los n datos, y como se explico antes, se vuelve a ejecutar con una cantidad de datos para tapar las fallas en la precondición.
Es necesario crear en ejecución una clase nueva que contenga el *main* y que considere el ambiente que se definió para el método y que considere los parámetros que se requieren.

4.6.3.2 Uso de JVM para oráculos

Cuando se utilizan oráculos no basta ejecutar el jmlrac, pues es necesario ejecutar el método oráculo y el método que se prueba y comparar los resultados de ambos métodos para confirmar la corrección.
La ejecución de los oráculos no se hace a través de JML sino a través del compilador de Java. Los objetos resultado de las ejecuciones de los dos métodos se compara utilizando el método *equals*.

4.6.4 Reutilización de las pruebas

La reutilización de las pruebas se hace a través del almacenamiento de pruebas y la recuperación de las mismas. El almacenamiento y recuperación de las pruebas se hace a través de archivos serializables. Se guarda un objeto de tipo *Prueba* serializado y dentro de este objeto están todas las características de la prueba, como los parámetros, los directorios de los archivos, la modalidad de la prueba, etc. La recuperación de la prueba se hace en dos pasos, el primero consiste en recuperar los archivos que hacen parte del sistema en prueba, estos se compilan y en el segundo paso se carga la prueba. La razón para hacerlo en dos pasos es que la prueba puede contener parámetros complejos y estos parámetros pueden depender de objetos en el sistema, por lo que es necesario que el sistema este cargado y compilado antes de cargar la prueba.

4.6.5 Pruebas para JSvt

JSvt ha sido probado con diferentes programas, tomados de los ejercicios de la clase de Algorítmica y programación por objetos I, de la Universidad de los Andes.

El software no graba la prueba sino hasta el final de la misma, por lo que en caso de una falla del sistema, los datos de la prueba se perderían.

El programa se ha probado con errores inyectados, que ha encontrado en la totalidad de los casos.

Conclusiones

El principal aporte de este proyecto es la generación de una propuesta de metodología de validación, que combina la estadística con las técnicas de prueba para mejorar la calidad y confianza del proceso de validación.

La validación estadística de software se presenta como una metodología que pretende utilizar el conocimiento de la ingeniería y arquitectura de software para establecer un plan de pruebas que considera los estándares de los procesos de verificación y validación y que considera los pasos del desarrollo de pruebas que han demostrado ser efectivos. Esta metodología no se trata solamente de generar datos y casos de prueba, sino de considerar el sistema como un todo sinérgico, cuyas unidades y la integración entre las mismas requiere de procesos claros de prueba, que permitan al programador tener confianza en los productos que realiza y evitar los efectos de bola de nieve que generan los errores que se pasan entre una etapa de desarrollo y la siguiente. Una parte importante de la metodología es la posibilidad de utilizar un DAG para el plan de pruebas, lo que permite hacer un orden topológico para las pruebas, y facilitar la organización en la prueba de clases y productos de software.

La generación de casos de prueba considera un método sencillo, pero involucra la estadística para darle robustez a esa generación. Se presenta una propuesta para generar objetos recurrentes de manera aleatoria, que demostró funcionar correctamente. Dentro de la metodología esos casos de prueba se integran utilizando intervalos de confianza, lo que permite utilizar la estadística para darle solidez y fiabilidad a la prueba.

Se mostró que al utilizar la estadística como parte del proyecto, las distribuciones probabilísticas juegan un papel importante. Dentro de la generación de datos para las pruebas, dichas distribuciones deben reflejar en la mejor forma la realidad de la selección, por lo que poder transformar las distribuciones comunes en diversas formas puede facilitar el realismo en la generación de datos. Dentro de este tema, un punto importante es la demostración de que cualquier dato con una función acumulada conocida puede ser generado utilizando un generador de datos uniforme, como los que se encuentran el la mayoría de plataformas.

El espacio muestral, que se define a través de las distribuciones, se modela en grupos permitiendo mayor representatividad de los datos, lo que implica más casos de prueba pero

más datos, por lo que aumenta la posibilidad de encontrar los errores que pueda haber en el software.

JSvt, el software que implementa la primera parte de dicha metodología, ha permitido demostrar que la validación estadística es posible, y aun mejor, es efectiva. JSvt permite validar métodos con alta precisión y bajos costos.
La implementación de JSvt permitió observar los problemas de la metodología en la práctica, exponiendo necesidades y soluciones. JSvt también permitió visualizar el alcance de la metodología, y las posibilidades de expandir la herramienta y de fortalecer la metodología al combinarla con otras técnicas de prueba. JSvt hace un aporte importante a la generación automática de pruebas, al involucrar la estadística y al ser una implementación orientada a objetos, que considera muchas de los conceptos que se aplican a la programación actualmente. Los módulos de esta implementación consideran la generación de casos de prueba, incluyendo la generación de datos, y la reutilización de esas pruebas, con lo que se facilita el proceso de regresión. Esto significa que JSvt puede utilizarse en varias etapas del desarrollo.

A través de este trabajo es posible observar que las pruebas pueden asegurar corrección en un producto de software. Sin embargo debe ser claro que los procesos de pruebas no son 100% confiables, pues dependen de la definición del producto, de la solidez y completitud de la especificación, de la definición de las distribuciones, y del buen diseño del plan de pruebas. A pesar de las dependencias, seguir la metodología con precisión permitirá desarrollar software correcto con mayor facilidad, y se asegura que el desarrollo de pruebas evitará que haya errores no detectados.

El proceso de validación estadística requiere que la especificación sea correcta, si hay errores en la especificación estos se verán reflejados en los errores que encuentre la aplicación. De este modo la estrategia utilizada para realizar el proceso de pruebas no solo debe considerar la metodología propuesta, sino la revisión de esta especificación, no solo para revisar que sea correcta, sino para revisar que contenga todas las aserciones que sean necesarias para poder asegurar la corrección de un método.

La posibilidad de utilizar diferentes distribuciones de probabilidad para la generación de datos, y poder transformar dichas distribuciones facilita la representación de los datos. Sigue siendo problemático el desconocer el comportamiento estadístico de los datos. Pero pensar en ello abre posibilidades de aseguramiento de calidad, e incluso, de mejora.

El uso de oráculos es una técnica bastante usada. Su combinación con métodos estadísticos y con validación por especificación permite hacer pruebas mucho más robustas y confiables. Sin embargo, los oráculos presentan el problema de necesitar ser funcionales, de modo que se puedan comparar los resultados de la prueba y el oráculo.

JSvt requiere que el software que se esté probando esté dividido en paquetes, ya que el método a probar se ejecuta para verificar su corrección. Esta prueba dinámica implica que los objetos y métodos llamados y las librerías o subsistemas necesarios deben estar ya codificados de manera que la prueba no genere errores de compilación y que la prueba no

esté sesgada a solo casos de objetos vacíos. De esta forma la metodología no permite generar las pruebas antes de generar el software, como muchos de los planes de pruebas, pero al generar datos aleatorios reduce el riesgo de sesgar las pruebas al hacerlas después de tener el código.

La definición del ambiente de prueba también es un punto importante. Si este ambiente se determina para generar siempre objetos simples, cuando existe la posibilidad de generar objetos complejos, la prueba estará incompleta, y su resultado no será fiable.

JSvt no implementa la totalidad de la metodología. El primer paso en el trabajo futuro será permitir que JSvt maneje pruebas para clases y productos. Por lo tanto deberá implementarse la generación del grafo de pruebas, la generación aleatoria de instancias de clases, los procesos de integración y un módulo que permita ver que pruebas han dado como resultado éxito y que significan estas pruebas sobre el grafo de pruebas.

Otras tareas importantes como trabajo futuro sobre JSvt son:

- Probar JSvt con sistemas más complejos y ampliar su funcionalidad de modo que acepte pruebas a sistemas de diferentes clases, como sistemas web o sistemas distribuidos.
- Permitir la prueba de corrutinas y sistemas *multi-thread*.
- Añadir a JSvt otras posibilidades para la generación de datos, como transformaciones de distribuciones a través de desfases.
- Permitir en JSvt la generación de instancias de clases para el ambiente automáticas, utilizando distribuciones estadísticas.
- Permitir la verificación formal en casos donde así lo requiera el usuario.
- Adicionarse a Eclipse[6] como un *plugin*, para facilitar la revisión del software que esta siendo probado.

También es importante ampliar la metodología para que considere pruebas de aceptación y con ello se pueda apoyar el proceso de pruebas con el usuario. La metodología debe ser implementada en procesos de desarrollo de software, para verificar la influencia de esta en la calidad del producto.

Este documento pretende apoyar la ampliación tanto de la metodología como del software, de modo que sean posibles futuras investigaciones sobre el proceso de pruebas y la calidad del software basada en los procesos de pruebas.

[6] Eclipse es una plataforma que utiliza diversas librerías y *frameworks* para apoyar el proceso de construcción, ejecución y manejo del software en todas las etapas de desarrollo.[ECL2007]

Bibliografía

[BEI1995] Beizer, B., *Black-box Testing: techniques for functional testing of software and systems*. New York : Wiley, c1995. ISBN: 0471120944 Physical description: xxv, 294 p.: ill. ; 23 cm.

[BRU2002] Bruegge, B., Dutoit, A., *Ingeniería de software orientado a objetos.* Ed. Prentice Hall, 2002.

[CAR1991] Cardoso, R., *Verificación y Desarrollo de Programas*, Universidad de los Andes, 1991.

[CAR2006] Cardoso, R., *Construcción de distribuciones*, Universidad de los Andes, 2006

[CLA1999] Clarke, E.M., Grumberg O., Peled D., *Model Checking.* Capitulos 1-10. The MIT Press, Cambridge, Massachusetts, Londres, Inglaterra. 1999

[CUL1996] Culwin, F. *ADA: A Developmental Approach*. Ed. Prentice Hall. 2a edición. 1996.

[CUP2007] *Proyecto Cupi2*. URL: cupi2.uniandes.edu.co (consultado:enero 2007)

[DUI2004] Duitama, J.A., *Selección de datos de entrada para pruebas de caja blanca*. Tesis de Maestría, Bogotá: Uniandes, 2004.

[EAS2007] Easton, V., McColl, J. *Statistics Glossary v1.1*. STEPS. URL: http://www.stats.gla.ac.uk/steps/glossary/probability_distributions.html (consultado: enero 2007)

[ECL2007] *Página oficial del proyecto Eclipse*. URL: http://www.eclipse.org/ (consultado: enero 2007)

[ESA1991] *ESA Software Engineering Standards*, ESA PSS-05-0 Issue 2, February 1991.

[ESA1995] ESA Board for Software Standardisation and Control (BSSC). European Space Agency. *Guide to software verification and validation.* ESA PSS-05-10, Issue 1 Revision 1,

March 1995. URL: http://styx.esrin.esa.it/premfire/Docs/PSS0510.pdf (consultado: enero 2007)

[FUS1996] Fussell, M.L.. *A Good Architecture for Object-Oriented Information Systems Structures, Designs, and Patterns.* ChiMu Corporation 1996. URL: http://www.chimu.com/publications/oopsla96tutorial23/index.html (consultado: enero 2007)

[HAN2007] Hansen, K.H. *Unit Testing Java Programs*. URL: http://javaboutique.internet.com/tutorials/UnitTesting/ (consultado: enero 2007)

[HAR2005] Harold, E., *An early look at JUnit 4:Upcoming release promises evolution in testing*. Polytechnic University. 13 Sep 2005

[HIM2007] *Himmeli v3.0 web interface*. URL: http://www.artemis.kll.helsinki.fi/himmeli/ (consultado:enero 2007)

[HOL2001] Holzmann, G.J. *Economics of Software Verification*. Bell Laboratories. 2001

[IEE1990] *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 610.12-1990

[IEE1989] *IEEE Std.982-1989* IEEE Standard for Software Verification and Validation. 1989.

[IEE1998] *IEEE Std 1012-1998* IEEE Standard for Software Verification and Validation. 1998.

[JAV2007] *Página oficial de Java*. URL: http://java.sun.com/docs/books/tutorial/java/concepts/ (consultado: enero 2007)

[JML2007] *Página oficial de JML*. URL: http://www.cs.iastate.edu/~leavens/JML/ (consultado: enero 2007)

[JUN2007] *Página Oficial de JUnit*. URL: www.junit.org (consultado: enero 2007)

[KAN2001] Kan, S. H., Parrish, J., Manlove, D. . *In-process metrics for software testing*. IBM SYSTEMS JOURNAL, VOL 40, NO 1, 2001

[KOU2000] Kourie, D., *Software Testing.* Department of Computer Science. Pretoria University. 2000

[LEA2006] Leavens, G.T., Cheon Yoonsik, *Design by Contract with JML,* June 5, 2006. URL: ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf

[MAL2003] Malveau, R., Mowbray, T.J. *Software Architect Bootcamp*. Software Architecture: Basic Training. Ed. Prentice Hall, 2a edición, 2003.

[MAK2007] *Institute of Statistics and Applied Economics*, Makerere University. URL: http://www.makerere.ac.ug/statistics/intro.htm (consultado: enero 2007)

[MAT2007] *MathWorld*. URL: http://mathworld.wolfram.com/ (consultado : enero 2007)

[MEY1976] Meyer, P., *Probabilidad y aplicaciones estadísticas*, Delaware : Addison-Wesley, 1976.

[NAS2007] Software assurance. Nasa. URL: http://satc.gsfc.nasa.gov/assure/agbsec5.txt (consultado: enero 2007)

[PER2000] William P., *Efective Methods for Software Testing*, 2nd edition, Wiley Computer Publishing, 2000.

[PER1990] Perry, W.E., *A standard for testing application software*, 1990.

[PAN1999] Pan, J., *Software Testing*. Carnegie Mellon University. Dependable Embedded Systems, Spring 1999

[QUI2006] Quiroga, A. F., *Jprove : una plataforma de verificación para Java*. Tesis (Magíster en Ingeniería Sistemas) -- Universidad de los Andes, Director: Rodrigo Cardoso. 2006

[SIE2004] Siegrist, K., *Probabilistic Distributions Package.* Department of Mathematical Sciences. University of Alabama in Huntsville. URL: http://www.math.uah.edu/stat/ (consultado:enero 2007)

[SPE2007] *Specification Languages.* URL: http://portal.etsi.org/mbs/Languages/introduction.asp (consultado: enero 2007)

[UNI2007] *Unit test tools*. URL:http://www.testingfaqs.org/t-unit.html(consultado: enero 2007)

Anexo A

JML

El lenguaje de modelado JML [JML2007] permite escribir aserciones dentro del código Java, de manera que se pueda especificar la precondición, poscondición e invariantes de métodos y clases.
Este anexo pretende exponer algunos de los puntos más importantes de JML y definir que características de este lenguaje son importantes para esta investigación.

Instalación

Los pasos para la instalación básica que se necesita en este trabajo son:

1. Descargar JML de http://sourceforge.net/project/showfiles.php?group_id=65346
2. Se obtiene un archivo *tar* que debe descomprimirse en el directorio que se desee.
3. En el archivo *readme* que se extrae hay mas instrucciones de instalación. Para este trabajo solo es necesario cambiar los archivos *jmlenv.bat* y *jmlrac.bat* donde hay que escribir el directorio en el que se encuentra instalado Java y JUnit.

Lo necesario de JML para este trabajo.

JML fue creado como una herramienta para programar a través de la técnica de diseño por contrato (DBC, por sus siglas en inglés). En esta técnica se considera la inclusión de un contrato entre una clase y sus clientes, de manera que los clientes garantizan cierta condición antes de llamar a un método y en retorno la clase garantiza ciertas propiedades que mantendrán después de la llamada. Estas condiciones son las que llamaremos precondición y poscondición. JML permite hacer estos contratos ejecutables, de modo que es posible verificar la especificación del método utilizando las herramientas que acompañan este lenguaje.

Las especificaciones JML se escriben utilizando comentarios de anotación, que son comentarios Java que empiezan con un signo @. De esta forma las aserciones que se encuentren después de //@, o entre /*@ y */@ se consideran código JML. El signo @ debe

estar inmediatamente después de // o de /*, si hay espacios se ignorará el comentario. Las especificaciones JML se escriben justo antes del encabezado del método.

Se utiliza la cláusula *requires* para indicar la precondición y la cláusula *ensures* para la poscondición. En el caso de las excepciones que puede lanzar el método que se está especificando, se utiliza la cláusula *signals* para especificarlas. También es posible escribir aserciones utilizando la cláusula *assert*. Para referirse al resultado del método dentro de las aserciones se utiliza el vocablo *\result*. Los cuantificadores se pueden representar a través de las expresiones *(\forall ...)* y *(\exists ...)*, los cuantificadores generalizados se pueden escribir con *\sum, \product, \min, \max*. Estas son solo algunas de las diversas cláusulas y vocablos que utiliza JML. Un ejemplo de especificación con JML puede verse en la figura siguiente. JSvt puede revisar todas las cláusulas JML, pero las que acá se describen son las más comunes.

Las especificaciones JML son una buena forma de especificar y a la vez documentar el código. El lenguaje es bastante amplio, y por ello permite alta precisión en la definición de la especificación.

```
package bank;
public class BankingExample {

    public static final int MAX_BALANCE = 10000;
    public int balance;
    public boolean isLocked = false;

    //@ invariant balance >= 0 && balance <= MAX_BALANCE;

    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() {balance=0; }

    //@ requires amount > 0;
    //@ ensures balance == \old(balance) + amount;
    //@ assignable balance;
    public int credit(int amount) { balance=balance+amount ; return balance;}

    //@ requires amount > 0;
    //@ ensures balance == \old(balance) - amount;
    //@ assignable balance;
    public void debit(int amount) { balance=balance-amount;}

    //@ ensures isLocked == true;
    public void lockAccount() { isLocked = true; }
```

Herramientas

Las herramientas principales que ofrece JML son:

- El compilador de JML (*jmlc*) es una extensión de un compilador de Java, y permite compilar programas de Java anotados con especificaciones JML y generar código binario Java. El código generado incluye instrucciones de chequeo de las aserciones en tiempo de ejecución, de modo que se puedan chequear las precondiciones, poscondiciones e invariantes.
- La herramienta de pruebas unitarias (*jmlunit*) que combina el compilador de JML con JUnit. LA herramienta permite al programador no hacer el código que decide el éxito o falla de una prueba, sino que genera código que chequea las clases a través de la especificación JML.
- El generador de documentación (*jmldoc*) produce archivos HTML que contienen tanto los comentarios Java como las especificaciones JML.
- El verificador estático extendido (*escjava2*) puede encontrar errores en el código Java de manera rápida. Es especialmente bueno en la búsqueda de posibles excepciones de apuntador nulo (*null-pointer*) y de operaciones que indexan un arreglo por fuera de sus límites (*array-out-of-bounds*). Usa anotaciones JML para apagar los *warnings* y propaga y chequea las especificaciones JML.
- El verificador de tipos (*jml*) es otra herramienta que chequea las especificaciones JML, como un sustituto rápido de jmlc, ya que no necesita compilar el código.
- El verificador de aserciones en tiempo de ejecución (*jmlrac*) permite ejecutar el código compilado con *jmlc* y mientras este se ejecuta, va verificando que la especificación de los métodos y clases ejecutadas se cumpla.

JSvt hace uso de *jmlc*, de *jmlrac* y de *jmlunit*.

Descripción gramatical

En esta sección se muestra un breve esquema de lo que es una aserción JML. Las posibilidades que se definen no son todas las que ofrece JML, pero si las mas comunes, y las de mayor relevancia en este trabajo.

Una *aserción* en JML puede ser:

- Una aserción entre paréntesis.
- Una *cláusula tipo 1* seguida de una *expresión booleana.*
- Una *cláusula tipo 2* seguida de una *variable*.
- Una *cláusula de tipo 3* seguida de 0 ó más *clausulas de tipo3.*

Una *cláusula de tipo 1* puede ser:

- *requires*
- *ensures*
- *assert*
- *invariant*

Una *cláusula de tipo 2* pude ser:

- *assignable*
- *signals_only*
- *throws*

Una *cláusula de tipo 3* pude ser:

- *pure*
- *also*
- *non_null*
- *null*
- *spec_public*

Una *expresión booleana* puede ser:

- Un '(', seguido de un *cuantificador* seguido de una *expresión de cuantificación* seguido de ')'.
- Una *expresión booleana* seguida de '&&' seguida de una *expresión booleana*.
- Un llamado a una función que retorne un valor booleano.
- Un llamado a una función o una *variable*, seguida de un *comparador*, seguido de una *variable* o de un llamado a una función.
- Una variable de tipo booleano.

Un *cuantificador* puede ser:

- *\forall*
- *\exists*
- *\sum*
- *\product*
- *\min*
- *\max*

Una *expresión de cuantificación* es:

- Un tipo de una variable y un nombre para la variable seguido de ';' seguido de una *expresión booleana* seguido de ';' seguido de otra *expresión booleana*. La sintaxis de la expresión de cuantificación es la misma que en lógica: la primera expresión booleana indica el rango y la segunda el predicado.

Un *comparador* puede ser:

- ==
- !=
- >
- <
- >=
- <=
- Una función de comparación (v.gr., *equals*).

Una *variable* se refiere a las variables en el mismo sentido que en Java.

Anexo B

Manual de usuario para JSvt

Este manual describe la funcionalidad de JSvt, explicando los pasos necesarios para realizar una prueba, definir distribuciones, definir ambientes, hacer regresión a las pruebas, etc.

Este manual habla de la instalación de JSvt y de uno de los posibles pasos para generar y ejecutar pruebas.

Como usar JSvt

1. Se debe instala el ambiente de compilación de java (jdk) 1.5 (java.sun.com), JUnit 3.8 (www.junit.org) y JML 5.2 (http://www.cs.iastate.edu/~leavens/JML/).
2. Instalar JSvt.
3. Se puede generar una nueva prueba o abrir una prueba existente.
    a. Para generar una nueva prueba se debe seleccionar el método que va a probar y las condiciones de prueba. JSvt va pidiendo datos que el usuario debe determinar para generar la prueba. Cada uno de esos pasos se explica en las secciones siguientes.
    b. Si se quiere abrir una prueba existente se debe seleccionar el archivo que guarda el *path* de la prueba, y el archivo que guarda el objeto 'prueba'.
4. Una vez generada o abierta una prueba, es posible ejecutarla, de modo que se verifica cada paso de prueba y se ofrece una respuesta al usuario.
5. Cuando ya se conoce la respuesta, es posible guardar la prueba que se generó (o que se abrió y se modificó). También se puede generar archivos de pruebas JUnit, utilizando los botones que así lo indican.

Instalación

La instalación de JSvt requiere de la instalación previa del jdk 1.5 de Java, JUnit 3.8 y JML 5.2.

Para instalar el programa solo es necesario descomprimirlo, modificar el archivo *properties* con el *path* en el que se instaló JUnit y el path de JML.

Una vez hecho esto, con solo ejecutar el archivo *run.bat* es posible correr el programa.

Generación de una nueva prueba

Para generar una nueva prueba es necesario definir:

- El directorio donde se encuentra el producto de software. Este directorio se busca con ayuda del botón que está al lado del campo designado para este fin. Este boton abre un explorador de archivos que permite buscar la localización en el equipo del producto de software. Este *path* debe ir hasta la primera carpeta del paquete, por ejemplo, si la carpeta *src* contiene a la clase *uniandes.tesis.sistemas.prueba.Principal* el *path* debe ir …/src/uniandes.
- La clase que contiene el método a probar. Para llenar este campo también hay un boton que permite explorar el equipo. La clase debe estar dentro del directorio escogido en el punto anterior. Al escoger la clase esta pasa de tener '/' a tener '.', de modo que no queda como el directorio a explorar sino que queda como el nombre completo de la clase, incluyendo su paquete.
- La modalidad de prueba. Utilizando las casillas, se puede escoger la verificación contra la especificación o contra un oráculo.
    - Si se elige la verificación contra un oráculo, será necesario escoger el directorio y la clase donde está el método oráculo. El procedimiento es el mismo que para el método a probar.
- Los datos para la prueba. Se puede escoger generar los datos para la prueba ó utilizar un archivo de datos serializado que ya exista. En el primer caso solo es necesario escoger esta opción, en el segundo caso se debe dar también la localización de dicho archivo, utilizando el botón de exploración que se encuentra al lado de este campo.

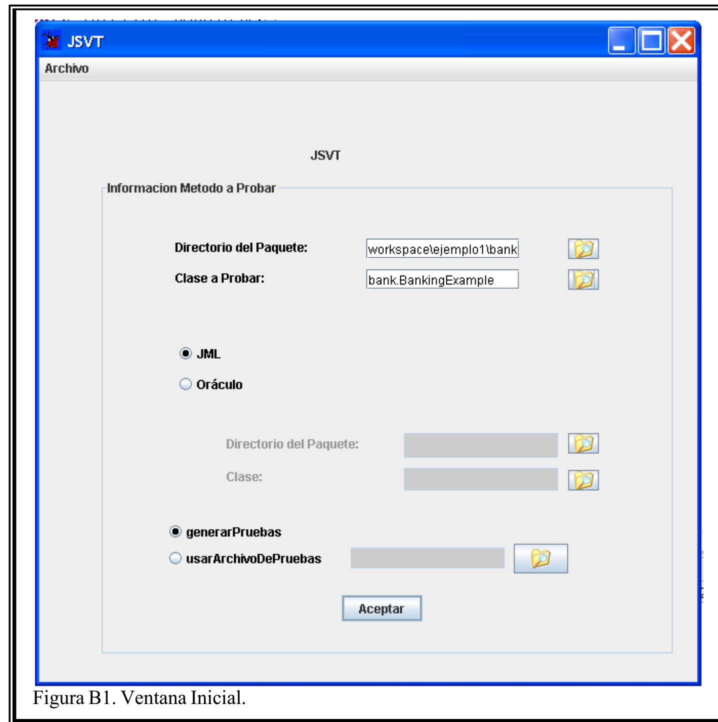


Figura B1. Ventana Inicial.

Una vez definidos estos datos se pasa a la selección del método a probar. En esta ventana se observa una lista con todos los métodos de la clase elegida, incluyendo los constructores. Si se eligió utilizar un oráculo, también aparece una lista con los métodos de la clase oráculo.

Modificación del classpath

Si se desea se pueden escribir librerías y clases para añadir al *classpath* para utilizarlas durante el proceso de compilación. Estas librerías se añaden al *classpath* solo durante la ejecución de la prueba.

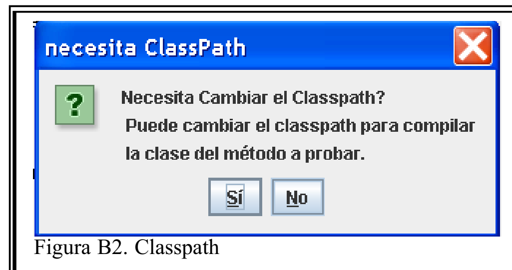


Figura B2. Classpath

Definición de las distribuciones para los parámetros

Caso 1: Sin parámetros. Si el método elegido no tiene parámetros se pasa automáticamente a la definición de la confianza.

Caso 2: Parámetros de tipo 'simple'. Los parámetros de tipo simple son *int, float, double, char, boolean, String, Integer, Double, Float*. Para cada uno de estos tipos hay ventanas que permiten definir la distribución. Esta ventana tiene una lista de las posibles distribuciones. Según la distribución que elija se muestran los campos que debe llenar como parámetros.
Para utilizar transformaciones de estas distribuciones se elige la opción *Segmentos*. Si se trata de utilizar solo una fracción de una distribución se escriben los limites inferior y superior (si se trata de -∞ ó ∞ se utiliza MININT y MAXINT ó MINDOUBLE y MAXDOUBLE según sea el caso), el nombre de la distribución que se quiere utilizar, seguido de paréntesis y los parámetros separados por comas (por ejemplo `normal(0,1)`) y la probabilidad de utilizarla, que en este caso sería `1`.
Si se quiere combinar varias distribuciones se siguen los mismos pasos que para una sola distribución, teniendo en cuenta que los limites no deben traslaparse, y que la probabilidad total debe ser `1`.

Caso 3: Parámetros de tipo 'complejo'. Los parámetros complejos son aquellos que no son simples. Para la definición de los parámetros 'complejos' se ofrecen tres posibilidades:

- Escoger un constructor para ese tipo. Según el tipo del parámetro, se ofrece la lista de sus constructores y el usuario puede escoger uno de ellos para generar estos objetos.
- Utilizar todos o algunos de los constructores para este tipo. Si el tipo del parámetro tiene mas de un constructor, estos se pueden combinar utilizando probabilidades de uso para cada uno de ellos. La suma de estas probabilidades debe ser `1`.
- Generar un constructor. Se puede codificar un método que genere el objeto que se necesita como lo desee el usuario. También puede combinar constructores y métodos para generar la instancia del objeto como la desee.

En todos los casos siempre se tendrá que dar distribuciones para los parámetros de los constructores, y si estos parámetros son objetos complejos se recurrirá buscando constructores hasta que los parámetros sean de tipo simple.

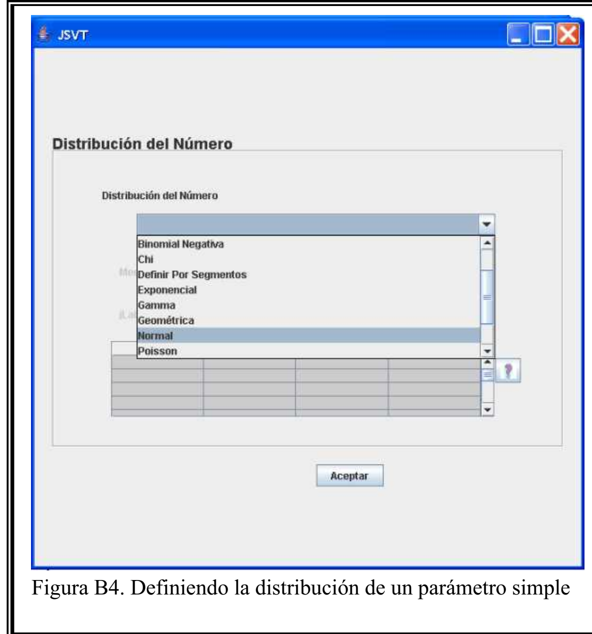


Figura B4. Definiendo la distribución de un parámetro simple

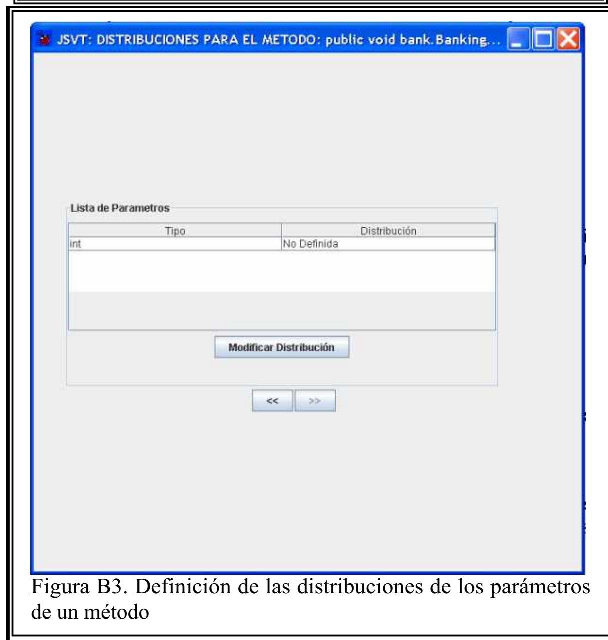


Figura B3. Definición de las distribuciones de los parámetros de un método

Figura B6. Distribución de un número utilizando una distribución conocida

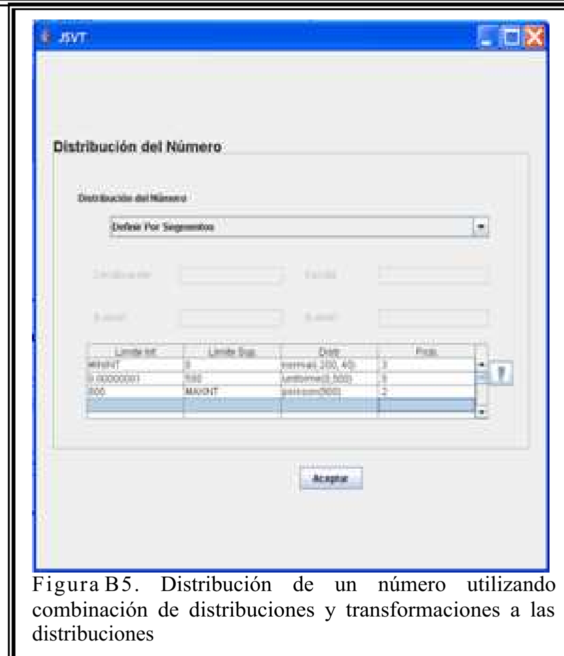


Figura B5. Distribución de un número utilizando combinación de distribuciones y transformaciones a las distribuciones

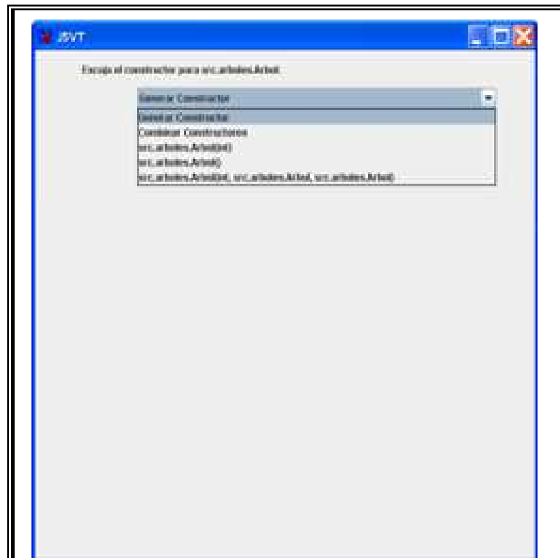


Figura B6. Posibilidades para definir la distribución de un objeto complejo

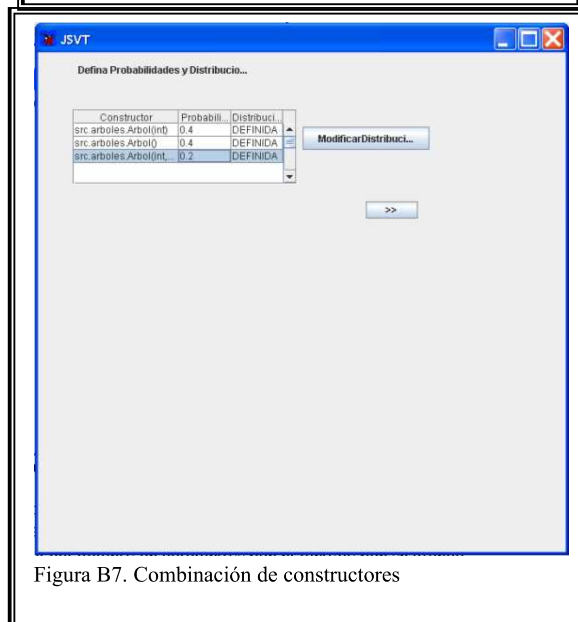


Figura B7. Combinación de constructores

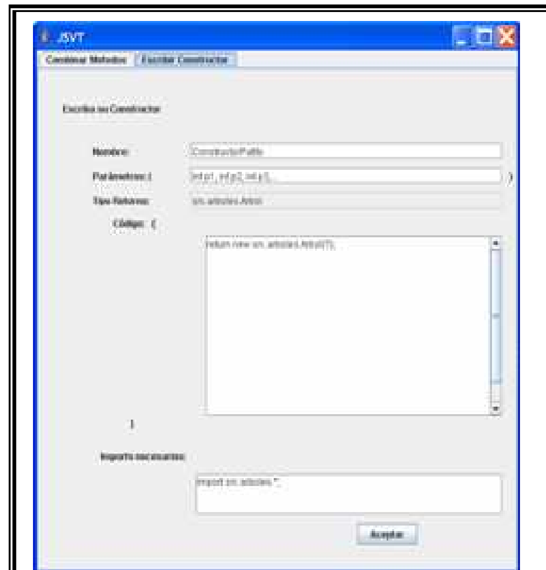


Figura B9. Generación de un constructor a través de código

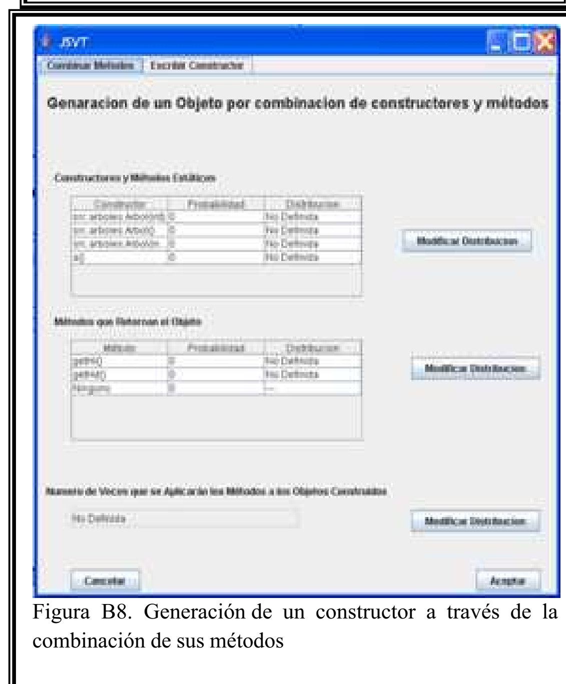


Figura B8. Generación de un constructor a través de la combinación de sus métodos

Correspondencia de parámetros

Si se utiliza un oráculo será necesario especificar como corresponden los parámetros del método a probar con los parámetros del método oráculo. El método oráculo debe tener menos o igual numero de parámetros que el método que se prueba.
Para definir esa correspondencia solo es necesario escoger un parámetro en uno y otro en el otro y hacer clic en el botón *añadir*.

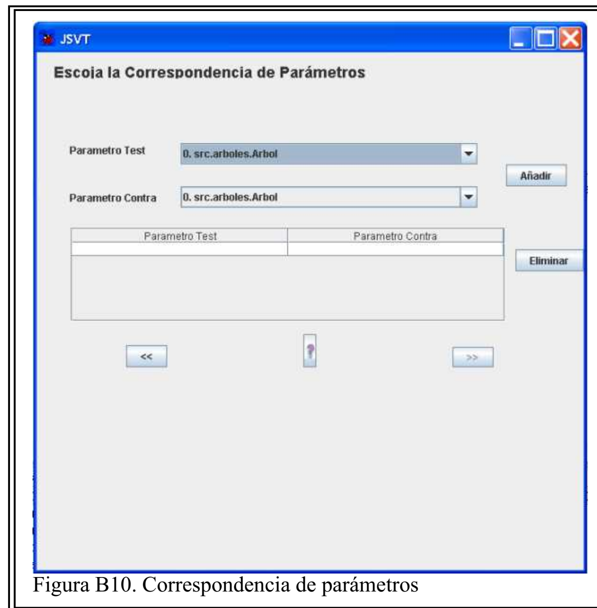


Figura B10. Correspondencia de parámetros

Definición de la confianza

Una vez definida la forma en la que se vana generar los datos es necesario definir la confianza con la que se realizará la prueba. Entre mayor sea la confianza elegida mas tiempo va a tomar la prueba.
En esta misma ventana es posible definir el número de repeticiones que se van a realizar a la prueba.
También es posible elegir el número de errores a mostrar. El número elegido debe ser mayor a cero. En caso de que se presenten errores, el programa parará la prueba en el momento en que se alcance el número de errores que el usuario eligió. Si se elige un número solo es necesario escribirlo en el campo designado para ello, no hay problema si se producen menos errores que el número elegido. También se pueden mostrar todos los errores, eligiendo esta opción.

Si se utilizó un archivo de datos existente, la confianza y las repeticiones no se pueden definir, sino que serán calculadas según la cantidad de datos que se ejecuten.

Si el método que se está probando es un constructor, se saltará el paso de definir un ambiente, por lo que en esta ventana se generará la compilación JML. Si existe algún error en esta compilación se informará al usuario y este tendría que arreglar el error y volver a empezar la prueba.

Figura B11. Definición de la confianza

Definición del ambiente de prueba

Si el método en prueba no es un constructor, se debe definir la forma en la que se generará la instancia de la clase sobre la que se ejecutará el método. En esta ventana el usuario debe escribir código Java para retornar un objeto del tipo solicitado. La ventana ofrece inicialmente la instrucción *return new* seguida del nombre del objeto.
Al hacer clic en el botón aceptar se compilará el código que se escribió en esta ventana. Si existe algún error se le comunicará al usuario para que lo arregle en esta misma ventana.
Si no hay errores en esta compilación se realizará la compilación JML. Si existe algún error en esta compilación se le comunicará al usuario para que haga las modificaciones necesarias y reinicie la prueba.
Si existe un método oráculo, que no sea un constructor, se deberá definir el ambiente para este método. El proceso es el mismo que con el método a probar.

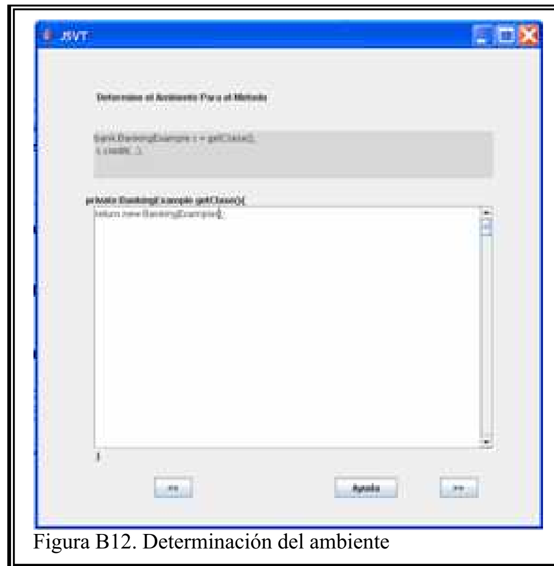


Figura B12. Determinación del ambiente

En espera de los resultados

Durante la generación de los datos y la ejecución de las pruebas se presenta al usuario una ventana con 3 o 4 barras de estado.
La primera barra indica cuantos datos se han generado respecto al total de datos que se deben generar para satisfacer la confianza elegida. La segunda barra indica el porcentaje de estos datos que no han pasado la precondición. La tercera barra indica el porcentaje de datos que no han pasado la prueba. La última barra solo aparece cuando la prueba se hace contra un oráculo, e indica la cantidad de datos que no ha pasado la validación contra el oráculo.

Esta ventana también presenta un campo de texto en el que se especifica en palabras la cantidad de datos que no han cumplido la precondición.

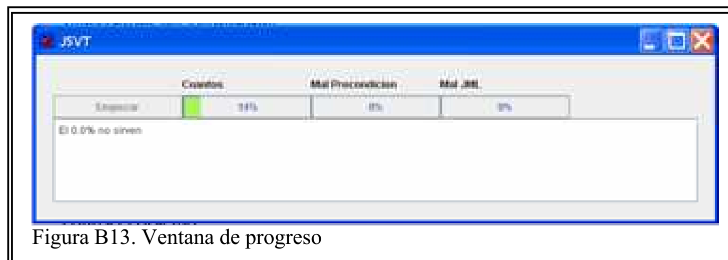


Figura B13. Ventana de progreso

En la ventana de resultados

La ventana de resultados expone al usuario el resultado de la prueba, diciendo que con una confianza dada se encontró que el producto en prueba tiene un porcentaje x de errores.
En esta ventana se presentan 4 opciones:

- Volver a la ventana inicial sin guardar la prueba.
- Salir. Esta opción pregunta si se quiere guardar la prueba.
- Generar pruebas JUnit. Esta opción genera los cascarones para las pruebas Junit.
- Correr pruebas JUnit. Una vez generadas las pruebas y modificadas las partes que hay que modificar, se puede correr desde aca las pruebas JUnit-JML.

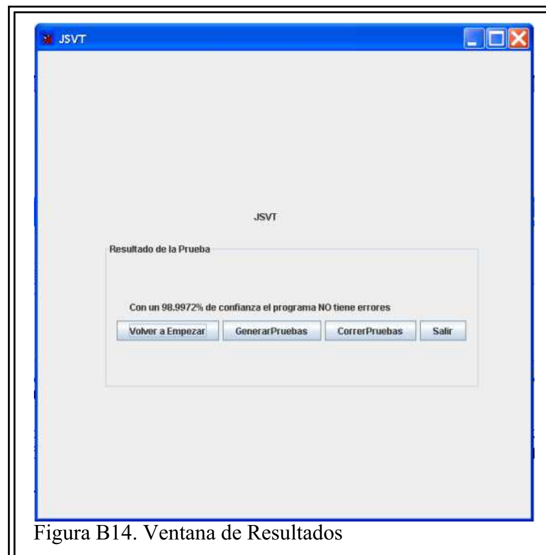


Figura B14. Ventana de Resultados

Guardar una prueba

Cuando se elige guardar la prueba, JSvt pide el directorio en el que la va a guardar. Guarda un archivo de datos serializados, un archivo de texto con los datos (para que los datos puedan ser leidos, los tipos de los datos deben tener implementado el método *.toString()*), un archivo serializado que contiene la prueba, un archivo de texto con los *path* necesarios y los archivos de pruebas JUnit, si estos fueron generados.

Pruebas JUnit-JML

Al generar las pruebas JUnit utilizando JSvt se generan los archivos *.java* con las pruebas. Estos archivos deben ser modificados manualmente para hacer que reciban los datos

generados con JSvt. Una vez modificados los archivos es posible compilar y ejecutar estas pruebas a través de JSvt.

Regresión

Desde la ventana principal se puede abrir un archivo de prueba ya existente a través de *Archivo*→ *Abrir*. Al elegir esta opción se piden dos archivos. El primer archivo que se pide es el de los *path* de la prueba, este es un archivo de extensión *jsvtdir*. El segundo archivo que se pide es el de la prueba, de extensión *JSvt*. Cuando se pide el primer archivo se hace una copia del sistema que se está probando, y se compila este sistema. Luego de pedir el segundo archivo se compila el método utilizando JML. Si existe algún error se informa al usuario y se suspende la regresión. Si no hay errores se presenta al usuario una ventana que expone todas las características de la prueba, incluyendo las distribuciones que se eligieron y el ambiente de la prueba. Es posible repetir la prueba tal como está al hacer clic en el botón *correr prueba*. También se puede modificar la prueba para que se generen de nuevo los datos, y para que se muestren más o menos errores.

Al hacer clic en el botón *correr prueba* se pasa a esperar los resultados.

Figura B15. Ventana de Regresión

Anexo C

Ejemplos de uso de JSvt

Ejemplo 1. El banco.

Esta aplicación sencilla simula las transacciones básicas de un banco como el depósito y retiro de dinero y el mantener y recuperar el balance. Esta aplicación consta de solamente una clase, que aunque no tiene errores aparentes, pero no maneja los casos de balance negativo en las transacciones, sino que lo deja pasar, generando incongruencias frente al invariante de clase, que establece que dicho balance no puede ser negativo.

```
package bank;
public class BankingExample {

    public static final int MAX_BALANCE = 1000;
    public int balance;
    public boolean isLocked = false;

    //@ invariant balance >= 0 && balance <= MAX_BALANCE;

    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() {balance=0; }

    //@ requires amount > 0;
    //@ ensures balance == \old(balance) + amount;
    //@ assignable balance;
    public int credit(int amount) { balance=balance+amount ; return balance;}

    //@ requires amount > 0;
    //@ ensures balance == \old(balance) - amount;
    //@ assignable balance;
    public void debit(int amount) { balance=balance-amount;}

    //@ ensures isLocked == true;
    public void lockAccount() { isLocked = true; }


    public /*@ pure @*/ int getBalance() { return balance; }
…
```

En este caso el orden de prueba debe considerar primero el constructor, que es un constructor sencillo, luego el método observador *getBalance*, y luego los demás métodos.
Suponiendo que ya está probado el constructor, es posible hacer la prueba del método *credit*. Para hacer la prueba es necesario saber como se van a distribuir los datos que entran a ese método. Por la especificación se sabe que los este método se utiliza casi siempre con un valor de 600, pero se concentran entre 200 y 1000. De este modo se decide utilizar una Normal(600,100)(100 por la forma en la que se distribuye la normal). Se utiliza una confianza del 99% y se define el ambiente utilizando solamente el constructor.

Varios datos de los generados no cumplen la precondición, pues algunos de ellos son menores a cero, pero son muy pocos por lo que la convergencia se consigue rápidamente.
El resultado de la prueba es de fallo, ya que los datos que entran mayores a 1000 generan un error que no es considerado por el código del método.
En el caso del método *debit*, utilizando la misma distribución, y utilizando el mismo ambiente el resultado de la prueba es de fallo, pues con el balance en 0, debitar de la cuenta deja el balance en menos de 0, lo que incumple la invariante de clase. Al arreglar el problema y rehacer la prueba, es necesario considerar un cambio de ambiente, y generar un ambiente en el que además del constructor se utilice el método *credit* (previamente probado) con un dato que se acomode a la distribución seleccionada. En este caso podría ser 600.

El resultado de la prueba también es de falla, pues los datos generados mayores a `600` generan un error, de modo que es necesario considerar en el método que se ignoren los datos que dejan el balance en menos de 0, o que se genere un error.

En ambos casos solo hay un parámetro, por lo que se generan solo 3 grupos (probabilidad baja, media y alta) y con la confianza del 99%, son 664 datos por grupo.

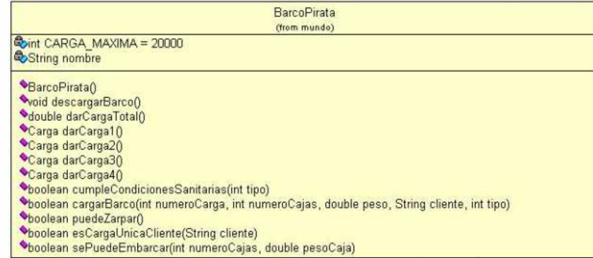
Ejemplo 2. El barco pirata

Este ejemplo es tomado de los ejercicios de nivel de la clase Algorítmica y Programación por Objetos 1 de la Universidad de los Andes [CUP2007]. Fue modificado para añadir la especificación en JML. Este ejemplo puede encontrarse, incluyendo los requerimientos y código, en cupi2.uniandes.edu.co.

Enunciado tomado del proyecto cupi2:

> Morgan es el capitán encargado de administrar la carga que se embarca en el barco "El Pirata". La embarcación solo puede transportar 20.000 kilos de carga por viaje. "El Pirata" presta sus servicios únicamente a 4 grandes compañías de la región, que constituyen sus clientes especiales. Es posible que en cada viaje se embarque carga de todos o de algunos de los clientes especiales. Por cada carga se conoce el número de cajas a transportar, el peso por caja (se supone que todas las cajas pesan lo mismo), el cliente a quien pertenece y el tipo de carga. Se han definido tres tipos de carga: Peligrosa, Perecedera y No Perecedera.
>
> Antes de aceptar una nueva carga en "El Pirata" se deben verificar las siguientes condiciones: Por reglas de sanidad, no se puede transportar en un mismo viaje una carga de tipo PERECEDERA y una carga de tipo PELIGROSA. Tampoco se pueden transportar más de 20.000 kilos por viaje y finalmente no se puede transportar más de una carga del mismo cliente.
>
> A los clientes no les gusta transportar sus cargas parcialmente, así que si no es posible transportar la totalidad de la carga, no la envían en la embarcación.
>
> Para evitar pérdidas el dueño de "El Pirata" ha establecido la siguiente política: El barco solo puede zarpar si se tiene mínimo dos cargas o el peso total de las cargas es mayor a 12.000 kilos.
>
> Morgan requiere una aplicación que le permita administrar las cargas de su barco teniendo en cuenta las reglas de negocio citadas anteriormente. La aplicación debe permitir: (1) Cargar el barco, teniendo en cuenta las condiciones especificadas anteriormente, (2) Descargar el barco para iniciar un nuevo viaje, (3) Dar el peso total de la carga que transporta el barco y (4) Mostrar si es posible que el barco zarpe o no.
>
> La interfaz y el modelo conceptual del *kernel* se muestran a continuación.

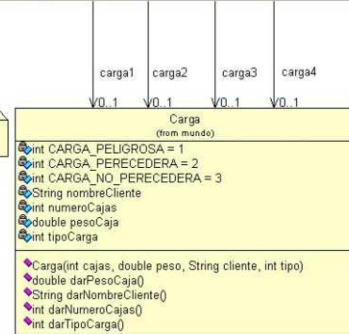
BarcoPirata
Clase que representa la embarcación del capitán Morgan.



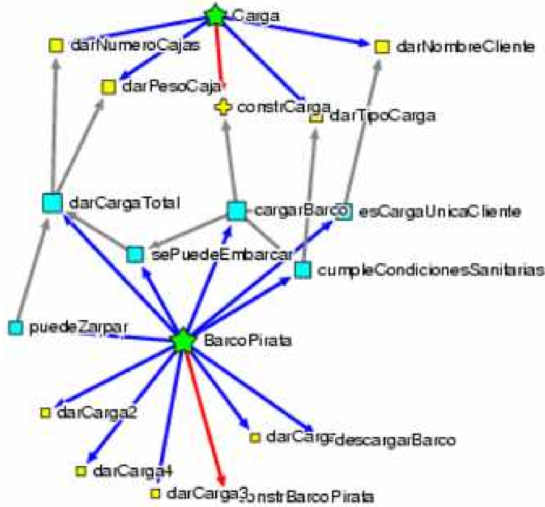
carga1 carga2 carga3 carga4

0..1 0..1 0..1 0..1

Carga
Esta clase representa una carga del barco pirata.



Considérese el grafo del *kernel*.



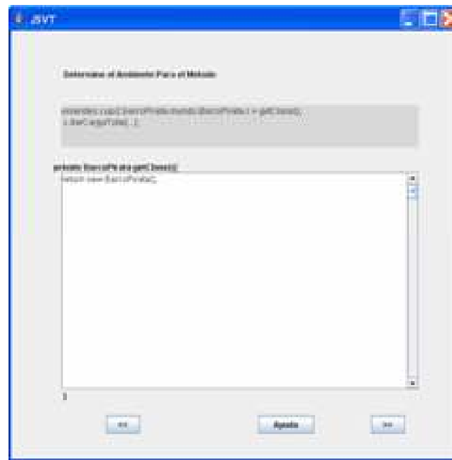

```
/**
 * Suma el peso de todas las cargas embarcadas y lo retorna. <br>
 * @return el peso total cargado en el barco.
 */
/*@ ensures \result==
(((carga1!=null)?carga1.pesoCaja*carga1.numeroCajas:0)+((carga2!=null)?carga2.pesoCaja*carga2.numeroCajas:0)+((carga3!
=null)?carga3.pesoCaja*carga3.numeroCajas:0)+((carga4!=null)?carga4.pesoCaja*carga4.numeroCajas:0));
 @*/
public double darCargaTotal( )
{
   double peso=0;
   if(carga1 !=null)
         peso+=carga1.darPesoCaja()*carga1.darNumeroCajas();
   if(carga2 !=null)
         peso+=carga2.darPesoCaja()*carga2.darNumeroCajas();
   if(carga3 !=null)
         peso+=carga3.darPesoCaja()*carga3.darNumeroCajas();
   if(carga4 !=null)
         peso+=carga4.darPesoCaja()*carga4.darNumeroCajas();

   return peso;
}
```

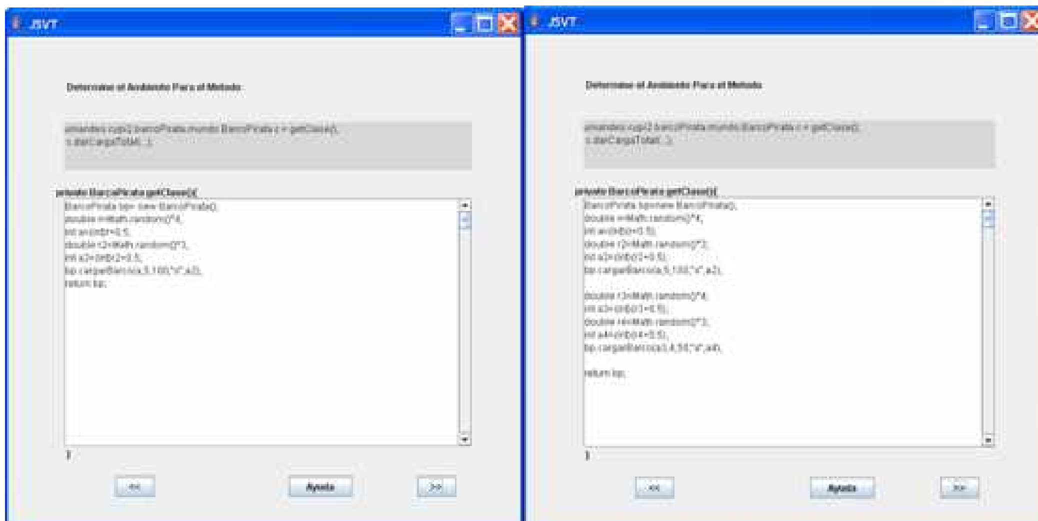
Se va a probar el método *darCargaTotal*, para lo que es necesario haber probado el constructor de la clase *BarcoPirata* y los métodos de la clase *Carga: darPesoCaja* y *darNumeroCajas* y por extensión, se necesita haber probado el constructor de la clase *Carga*.

Suponiendo que todos estos métodos ya han sido probados, se podría probar el método *darCargaTotal*. No es necesario dar una distribución para los datos por que el método que se va a probar no tiene parámetros. El ambiente, en cambio, es bastante importante para esta prueba. La razón es que si se utiliza el constructor del *BarcoPirata*, se generaran 4 cargas nulas, en donde el método debería retornar `0`, pero este no es el único

caso que se quiere probar, se quiere probar que este método funcione con cualquier cantidad de cajas.

Este ambiente requiere que se haya probado el método *cargarBarco* y todos los métodos a los que llama, incluido el método *darCargaTotal*, de modo que es necesario hacer pruebas en escalas, es decir, se prueba *darCargaTotal* con un *BarcoPirata* sin cargas, luego se prueba el método *cargarBarco* sobre ese barco vacío, una vez probado, se puede generar el ambiente para probar el *darCargaTotal* con una carga, y hecho esto, se puede probar *cargarBarco* con un barco que ya tiene una carga, y así sucesivamente hasta llegar a las 4 cargas.



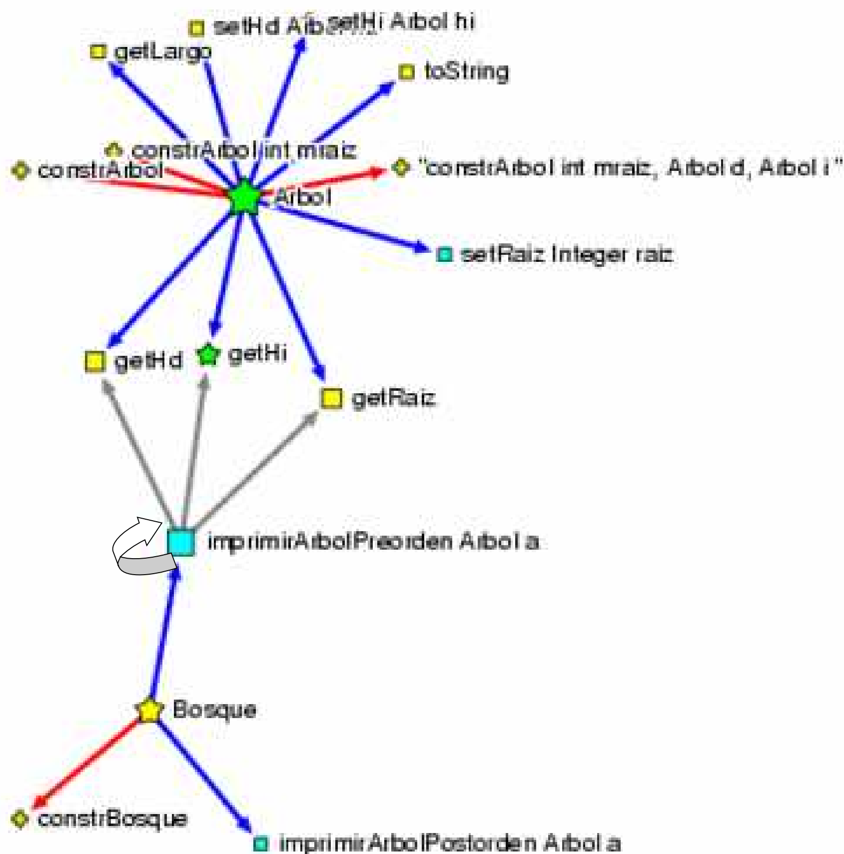


Se escoge una confianza del 99%, como no hay parámetros se generan solo 669 datos (no hay sino 1 grupo). El resultado de los diferentes escenarios de prueba es positivo.

Ejemplo3. El Bosque

Este es un ejemplo que tiene dos clases *Bosque* y *Arbol*. Los árboles tienen hojas enteras, y pueden generarse de tres formas: un árbol vacío, un árbol con una raíz donde se auto-generan hijos (árboles) vacíos y un árbol con una raíz y dos árboles como hijos. El *Bosque* permite imprimir en pre-orden y pos-orden los árboles, y otras operaciones que no se considerarán en este ejemplo.

El grafo de dependencias es el siguiente:



Se va a probar el método que imprime en pre-orden utilizando recursión. Este se va a probar contra un oráculo que no utiliza recursión.

El primer paso es determinar la forma en la que se distribuye el parámetro de tipo *Arbol*. Por ser un tipo complejo, es necesario recurrir a sus constructores. Ya que la generación de árboles es recursiva (el constructor recibe árboles para generar árboles) se eligió combinar los constructores, dándole probabilidades a cada uno de ellos. Se le dio probabilidad de `0.3` al constructor vacío, de `0.3` al constructor con solo la raíz y de `0.4` al constructor recursivo (estos datos se basan en el tipo de prueba que se quiere hacer).

Una vez definidas estas probabilidades, se pasa a definir las distribuciones de los parámetros de los constructores que se usaron. El constructor vacío no tiene parámetros por lo que salta esta fase, el constructor que tiene una raíz recibe un entero, al que se le da una distribución `Normal(400,200)`, el constructor que recibe una raíz entera y dos árboles se distribuye de la siguiente forma: la raíz se distribuye `Normal(800,100)` y los árboles que vienen como parámetro tienen la misma distribución que el árbol que era el parámetro original, de modo que estos árboles también se generarán con las probabilidades que se definieron.

Este proceso de prueba es largo, pues se necesita tiempo para que el proceso de recursión converja. Se generan `3` grupos, de probabilidad alta, media y baja para el parámetro del método *imprimirArbolPreorden*. De este modo, con una probabilidad del `99%` se necesitan 664 datos en cada grupo.

La correspondencia de parámetros es sencilla, pues tanto el método original como el método oráculo solo tienen un parámetro.

El ambiente que se define para el método original solo requiere utilizar el constructor simple de la clase *Bosque*, mientras que el del método oráculo llama al constructor de la clase *BosqueNR*.

El resultado de la prueba es positivo.