

DTO. DE COMPUTACION
INFOTECA
F.C.E. y N. - U.B.A

Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis de Licenciatura

Aplicación de conceptos de la Teoría de Juegos a la relación Tester-Desarrollador

Alumno: Gabriel Reiter

Directores: Mónica Bobrowski y Ernesto Kiskurno

Diciembre de 2004

1 Resumen

En una organización que se encarga del desarrollo de sistemas, los roles del Programador y del Tester cumplen una función fundamental. Sin embargo, los objetivos de los dos actores principales del desarrollo de software son generalmente diferentes, y muchas veces opuestos.

Esta oposición de intereses trae aparejados algunos de los grandes problemas del desarrollo de software: los Desarrolladores son presionados a producir un número muy grande de líneas de código, lo que genera que el código que llega a testing sea de muy baja calidad. Esta situación obliga a los Testers a controlar todo el código recibido, sobrecargándolos de trabajo, y por lo tanto aumentando el costo del testing de la organización. Si a esto se le suma el problema de la reticencia de las empresas a la inversión en calidad, la fórmula resulta explosiva.

En este trabajo se utiliza la *Teoría de Juegos*, una teoría matemática que trata el problema de la toma de decisiones de un grupo de individuos que interactúan entre sí, como base para la presentación de un marco metodológico que permite a una organización la alineación de los objetivos de los Desarrolladores con los de los Testers.

Esta metodología de fácil implementación permite prever la actitud de los actores a partir del uso sistemático de métricas, y de esta forma disminuir la cantidad de horas de trabajo de Testers y Desarrolladores, reduciendo en forma importante la inversión de la organización en testing sin bajar la calidad del software producido.

Luego de la presentación del marco metodológico, se muestra un ejemplo basado en casos reales que muestra la facilidad de aplicación de la metodología y al mismo tiempo los beneficios que ésta posee con respecto a la no utilización de la misma.

2 Agradecimientos

A Mónica y Ernesto, por dirigirme durante todo este tiempo y saber calmarme en los momentos de desesperación.

A mis viejos, por sentir que mis éxitos y fracasos son los suyos, por apoyarme continuamente y en forma totalmente incondicional y ayudarme a lograr cada uno de los objetivos que me propongo.

A Seba, porque aunque está lejos, sé que está cerca.

A Fabi, por amarme y permitirme amarla, y por aguantarme de la mejor forma el mal humor cuando las cosas no salían.

A Pecho, Javi, Tincho, Rochi, Matías, Anita, Pablo y las demás personas que me acompañaron durante toda la carrera, porque sin ellos no podría haber llegado hasta acá.

A los muchachotes-ado, por hacer que mi vida sea cada día menos monótona.

A Vani y Jony, por estar ahí cada vez que tienen que estar.

A mis abuelos, por hacerme salir una lágrima cada vez que pienso o hablo de ellos.

A mis tíos y primos, por ser la mejor hinchada que tengo.

A todas las personas que no nombré, y que de alguna u otra forma me ayudaron a llegar a este momento tan especial de mi vida.

Gracias !!!!!!!!!!!!!!!!!!!!!!!

Índice

1	Resumen.....	2
2	Agradecimientos.....	3
3	Introducción	7
3.1	Contexto	7
3.2	Objetivos	7
3.3	Organización del trabajo.....	8
4	El testing en el proceso de desarrollo de software	9
4.1	Introducción.....	9
4.2	El proceso de desarrollo de software	9
4.3	Modelos del proceso de desarrollo de software.....	11
4.3.1	Modelo en cascada	11
4.3.2	El modelo V	12
4.4	El Testing de sistemas	13
4.4.1	Tipos de defectos.....	13
4.4.2	Tipos de pruebas.....	15
4.4.3	Visiones de los objetos de prueba	16
4.4.4	Actividades de la prueba	17
5	Teoría de Juegos	19
5.1	Introducción.....	19
5.2	Un poco de historia.....	19
5.2.1	El dilema del prisionero.	19
5.3	Aplicaciones	20
5.4	Conceptos básicos.....	21
5.5	Herramientas	22
5.5.1	Matriz de pagos	22
5.6	Tipos de juegos	23
5.6.1	Juegos simultáneos Vs. secuenciales.....	23
5.6.2	Juegos de suma cero Vs. no suma cero	24
5.7	Equilibrio de Nash.....	24
5.8	Estrategias dominantes	25
5.9	Estrategias puras y mixtas.....	25
5.9.1	Estrategias puras.....	25
5.9.2	Estrategias mixtas.....	26
6	Marco metodológico	32
6.1	Introducción.....	32
6.1.1	Aclaraciones	32
6.2	Elementos del juego.....	32
6.2.1	Jugadores.....	32

6.2.2	Acciones	33
6.2.3	Pagos	33
6.2.4	Información	38
6.3	Un primer acercamiento: modelo idealista	39
6.3.1	Supuestos generales.....	39
6.3.2	Funciones de pago	39
6.3.3	Matriz de pagos	40
6.3.4	Supuestos sobre las variables	41
6.3.5	Condiciones sobre las variables.....	41
6.3.6	Notación gráfica de supuestos y condiciones	41
6.3.7	Equilibrios	42
6.3.8	Resumen	49
6.3.9	El problema de las estrategias puras.....	52
6.4	Hacia un modelo más realista.....	53
6.4.1	Causas del desajuste	53
6.4.2	Nuevas funciones de pago	56
6.4.3	Matriz de pagos	57
6.4.4	Más equilibrios puros	58
6.4.5	El equilibrio en estrategias mixtas.....	60
6.4.6	Resumen	64
6.5	Resultados del equilibrio.....	66
6.5.1	Interpretación de los valores de equilibrio	66
6.5.2	Costo para los jugadores.....	67
6.5.3	Costo total para la empresa	68
6.5.4	Cantidad de errores en producción	68
6.5.5	Comparación con otras estrategias de testeo	69
6.6	Síntesis de la metodología	71
7	Caso de estudio	72
7.1	Introducción.....	72
7.2	Presentación de la empresa	72
7.2.1	Equipos de trabajo	73
7.3	El proyecto	76
7.4	Aplicando el modelo	77
7.4.1	Introducción	77
7.4.2	Definición de jugadores.....	77
7.4.3	Definición de unidad de pago.....	77
7.4.4	Definición de unidad de test.....	77
7.4.5	Estimación de variables independientes.....	77
7.4.6	Verificación de supuestos sobre variables	79
7.4.7	Generación de matriz de pagos.....	80
7.4.8	Cálculo de equilibrio	80
7.5	Resultados del equilibrio.....	82
7.5.1	Interpretación de valores de equilibrio	82
7.5.2	Costo para los jugadores.....	82
7.5.3	Costo total para la empresa	82
7.5.4	Cantidad de errores en producción	83
7.5.5	Análisis de sensibilidad.....	83
7.5.6	Comparación con el no uso de la metodología.....	84
7.6	Observaciones del caso.....	86

8 Conclusiones	87
9 Trabajos futuros.....	89
10 Bibliografía	90

3 Introducción

3.1 Contexto

Es innumerable la cantidad de ejemplos que se puede dar sobre disciplinas que no podrían haber alcanzado el nivel de evolución que tienen en la actualidad sin la utilización de software. Se puede nombrar, por ejemplo, la medicina, donde las tomografías computadas sirven para la detección de todo tipo de enfermedades, o la industria del transporte, en donde se requiere la posibilidad de reserva de pasajes a distancia, así como también la aviación: sin un buen software de organización, se verían todos los días noticias de aviones que chocan entre sí.

Así como en las últimas décadas los programas informáticos fueron utilizados en una gran cantidad de disciplinas que hasta ese momento no los habían incluido, se fue incrementando la complejidad de los programas a construir, al mismo tiempo que fue aumentando la necesidad de que los programas creados fueran de buena calidad. Esta necesidad cada vez más importante, fue la originadora de la disciplina de la Ingeniería de Software, que trata de dar bases y métodos para mejorar el software que se construye.

En una empresa que se dedica al desarrollo de software existen distintos roles en los equipos de trabajo que realizan los programas. Entre los roles más importantes de un equipo de desarrollo de software se encuentran el de Desarrollador (o programador), cuyo objetivo es escribir las líneas de código que conformarán cada uno de los programas que se deban realizar, y por otro lado el de Tester, cuyo objetivo es detectar la mayor cantidad posible de errores en las líneas de código escritas por el Desarrollador.

Los roles del Desarrollador y el Tester suelen tener objetivos contrapuestos en su trabajo; por un lado el Desarrollador desea producir la mayor cantidad de código posible, muchas veces no poniendo el suficiente énfasis en la calidad del mismo. En contraposición a éste, el Tester desea que el código que llegue a producción posea la menor cantidad de errores posible. De esta forma el Desarrollador usualmente termina entregando al Tester código no controlado de poca calidad, y éste se ve forzado a testear todo el código generado, lo que aumenta la cantidad de recursos necesarios para el testeo de programas.

Si la empresa puede lograr una cooperación más alta de los Desarrolladores con los Testers en el testeo de programas se puede bajar la inversión en testing necesaria para lograr software de buena calidad.

Por otro lado existe la Teoría de Juegos. Ésta es una teoría matemática elaborada por el matemático John von Neumann y el economista Oskar Morgenstern [VNE/44] que trata el problema de la toma de decisiones de un grupo de individuos racionales, en general cuando hay contraposición de intereses entre ellos.

En el contexto descripto, la teoría de juegos aparece como aplicable al problema que se plantea, y se puede utilizar como un método nuevo y aun no investigado para la alineación de los objetivos de los Desarrolladores con los de los Testers.

3.2 Objetivos

Los objetivos de este trabajo son:

- Definir un marco metodológico utilizando conceptos de la Teoría de Juegos que provea a una organización una herramienta para la alineación de los objetivos de sus programadores con respecto al de sus Testers y de esta forma mejorar

la comunicación entre ellos y bajar la inversión en testing sin bajar la calidad de los programas.

- Validar la metodología propuesta por medio de un caso de estudio que incluya todos los pasos de la misma.

3.3 Organización del trabajo

El trabajo comienza con los capítulos 4 y 5 que introducen los temas que serán necesarios para la comprensión del trabajo en general; en el capítulo 4 se presenta el proceso de desarrollo de software con todas sus etapas y se hace énfasis en la etapa del Testing. En el capítulo 5 se describe la Teoría de Juegos, que sirve de base para la propuesta del trabajo.

El capítulo 6 describe el principio del marco metodológico del trabajo. Éste comienza con los elementos básicos del modelo y va generando progresivamente el modelo completo. El primer modelo que se presenta se demuestra luego que es poco realista, para lo cual se proponen algunas modificaciones que lo mejoran y lo hacen más acorde a la realidad. Finalmente se muestran los resultados que puede generar el modelo con sus métodos de cálculo, luego de lo cual se da una síntesis de la metodología.

En el capítulo 7 se muestra un caso de estudio, presentando la empresa que se utilizará y luego aplicando uno a uno los pasos descriptos en la metodología.

El capítulo 8 muestra las conclusiones de la metodología presentada en el trabajo. Finalmente el capítulo 9 presenta las posibles ramas de continuación de investigación que se abren a partir de este trabajo.

4 El testing en el proceso de desarrollo de software

4.1 Introducción

La metodología que se presenta en este trabajo está centrada en la Ingeniería de software y en particular en la relación entre Testers y Desarrolladores en un proyecto de desarrollo de software. Por lo tanto para poder comprenderla es necesario entender qué es la Ingeniería de Software y cuál es la ubicación del testing dentro de la misma. Este capítulo intenta dar las bases para la comprensión del marco metodológico que se presenta luego.

4.2 El proceso de desarrollo de software

Cuando se presta un servicio o se crea un producto, ya sea la construcción de una casa o un auto, la preparación de un informe escrito o al realizar un viaje de negocios, siempre se sigue una secuencia de pasos o tareas para lograr los objetivos. Las tareas se realizan por lo general en el mismo orden todas las veces; por ejemplo, no se coloca la secadora antes de realizar la instalación eléctrica de una casa, ni se cocina una torta antes de mezclar correctamente sus ingredientes. El desarrollo de software no es una excepción a esta regla; también se puede descomponer en un conjunto de tareas que se deben realizar para la buena construcción de software. Podemos pensar al conjunto ordenado de tareas como un proceso; una serie de pasos que involucren actividades, restricciones y recursos que producen una determinada salida esperada [PFL/01].

El desarrollo de software implica a los siguientes participantes:

- **Clientes:** Son los que ven la necesidad de la construcción de un software y encargan la realización del mismo. También son los que pagan por el desarrollo del sistema.
- **Desarrolladores¹:** Son los encargados de construir el sistema en cuestión, utilizando las herramientas de la ingeniería de software.
- **Usuarios:** son los que van a utilizar el sistema en última instancia.

En algunas situaciones los clientes y usuarios son el mismo conjunto de personas. Tal es el caso de un banco que encarga un sistema para uso interno. Sin embargo, en ciertas ocasiones los clientes son distintos a los usuarios. Por ejemplo, el mismo banco puede encargar y pagar un sistema (por lo tanto es el cliente) para su utilización por parte de los clientes del banco (que son los usuarios finales del sistema)

Las tareas o actividades típicas que incluye el proceso de desarrollo de software son las siguientes:

- **Análisis y definición de requerimientos:** El primer paso del desarrollo de software es reunirse con el cliente para determinar los requerimientos. Estos últimos describen el sistema. Sin el debido conocimiento de datos importantes

¹ Esta es la definición más amplia de los Desarrolladores, en la que son todas las personas que se encargan de la construcción de los sistemas, en todos sus aspectos. En los capítulos posteriores del trabajo se utilizará la palabra Desarrollador para definir a cierto grupo de personas dentro del proceso de desarrollo.

como el límite, el conjunto de entidades y actividades, es imposible describir el software y cómo éste interactúa con su ambiente.

- **Diseño del sistema:** Una vez que los requerimientos están definidos, se crea un diseño del sistema para satisfacer los requerimientos especificados. El diseño del sistema representa al cliente la apariencia del sistema desde la perspectiva del usuario. De esta manera, entre otras cosas, al cliente se le presentan modelos de las pantallas que se usarán o de los informes que se generarán y cualquier otra descripción que explique la forma en que el usuario interactuará con el sistema terminado. El diseño también incluye la descripción de la estructura del sistema (arquitectura, modelo de datos, etc.).
- **Diseño de los programas:** Luego de que el diseño del sistema es aprobado por los clientes, se puede pasar al diseño de los programas individuales que conforman el sistema en forma global. Recién en esta etapa se mencionan los programas. Hasta que la funcionalidad y la apariencia estén determinadas, no tiene sentido considerar la codificación. En esta etapa se definen, entre otras cosas, los módulos que conformarán cada uno de los programas finalmente.
- **Escritura de los programas:** Esta etapa incluye la escritura del código necesario para el funcionamiento de los programas diseñados. Éste se puede escribir en distintos lenguajes de programación para cada programa, e incluso en distintos lenguajes para cada módulo dentro de cada programa.
- **Prueba Unitaria:** Cuando los programas han sido escritos, se los prueba como piezas individuales de código, antes de que puedan integrarse como un todo. Esto simplifica el testeo de los programas, ya que testear piezas de código pequeñas es más fácil que hacerlo con piezas grandes.
- **Prueba de Integración:** Una vez que se demuestra que las piezas trabajan como se desea, se las reúne y se asegura que trabajan correctamente cuando están unidas con otras. Esto se realiza con todas las partes del sistema hasta que sea operacional.
- **Prueba del sistema:** Implica la prueba del sistema completo para asegurar que las funciones y las interacciones especificadas inicialmente se han implementado correctamente. En esta fase el sistema se compara con los requerimientos especificados: el Desarrollador, el cliente y los usuarios comprueban que el sistema sirve a sus propósitos.
- **Entrega del sistema:** Implica la entrega al cliente del producto final conformado por el sistema desarrollado, junto con una serie de documentos que lo acompañan.
- **Instalación:** Incluye todas las actividades que hacen que el sistema esté en funcionamiento en el lugar y condiciones en las que finalmente trabajará. Esto incluye creación de bases de datos, instalación de programas, configuraciones en el cliente y otras actividades.
- **Mantenimiento:** A medida que se utiliza el sistema se descubren problemas y discrepancias. Si se trata de un sistema llave en mano, el cliente asume la responsabilidad por el mismo una vez entregado. Muchos sistemas no son de ese tipo, y es el Desarrollador u otra organización el que brinda el mantenimiento si algo funciona mal o si cambian las necesidades y los requerimientos. Éste es el mantenimiento correctivo. Otra situación se da cuando el cliente necesita nuevas funcionalidades en el sistema. En este caso el mantenimiento implica desarrollar estas nuevas funcionalidades.

4.3 Modelos del proceso de desarrollo de software

Se podría pensar que en el proceso de desarrollo de software las tareas descriptas se realizan una a la vez en forma secuencial, de forma que cuando se alcanza el final de la lista se habrá completado el proyecto. Sin embargo, en la realidad, son muchos los pasos que se repiten. Por ejemplo, al revisar con el cliente el diseño del sistema puede descubrirse que un requerimiento no ha sido documentado. Se puede trabajar con el cliente para agregar este requerimiento y posiblemente rediseñar el sistema. De forma similar, cuando se escribe y se prueba el código, se puede encontrar que un dispositivo no funciona como se ha descrito en su documentación. Tal vez se tenga que rediseñar el código, o reconsiderar el diseño del sistema, y aún volver a discutir con el cliente el modo de cumplimentar los requerimientos. Por esta razón se define un modelo del proceso de desarrollo de software que es una descripción del desarrollo de software que contiene todas o algunas de las tareas listadas en la sección 4.2.

Existe una gran cantidad de modelos que se han descrito en la literatura sobre Ingeniería de Software. Cada uno de ellos incluye los requerimientos del sistema como entrada y un producto entregado como salida.

A continuación se presentan algunos de los modelos existentes.

4.3.1 Modelo en cascada

Es uno de los primeros modelos propuestos presentado por Royce (1970). En este modelo las etapas se representan cayendo en cascada, desde una etapa a la siguiente (Figura 4.1). De esta forma, una etapa de desarrollo debe completarse antes de dar comienzo a la siguiente. [PFL/01]. Por ejemplo, cuando todos los requerimientos del cliente han sido identificados, analizados para comprobar su integridad y consistencia, y documentados en un documento de requerimientos, recién entonces el equipo de desarrollo puede seguir con las actividades de diseño del sistema.

El modelo en cascada puede ser muy útil, ayudando a los Desarrolladores a diagramar lo que necesitan hacer. Su simplicidad hace que sea fácil explicarlo a los clientes que no están familiarizados con el desarrollo de software. Asimismo explicita los productos intermedios que son necesarios a fin de poder comenzar con cada una de las etapas del desarrollo. Muchos otros modelos más complejos son meros retoques del modelo en cascada, que incorporan lazos de realimentación y actividades adicionales.

Existen muchas críticas hacia el modelo en cascada. Una de las más importantes es que no refleja la manera en que realmente se realiza el desarrollo del software, ya que, excepto para los problemas perfectamente comprendidos, el software se desarrolla normalmente con un alto grado de repetición. [PFL/01]

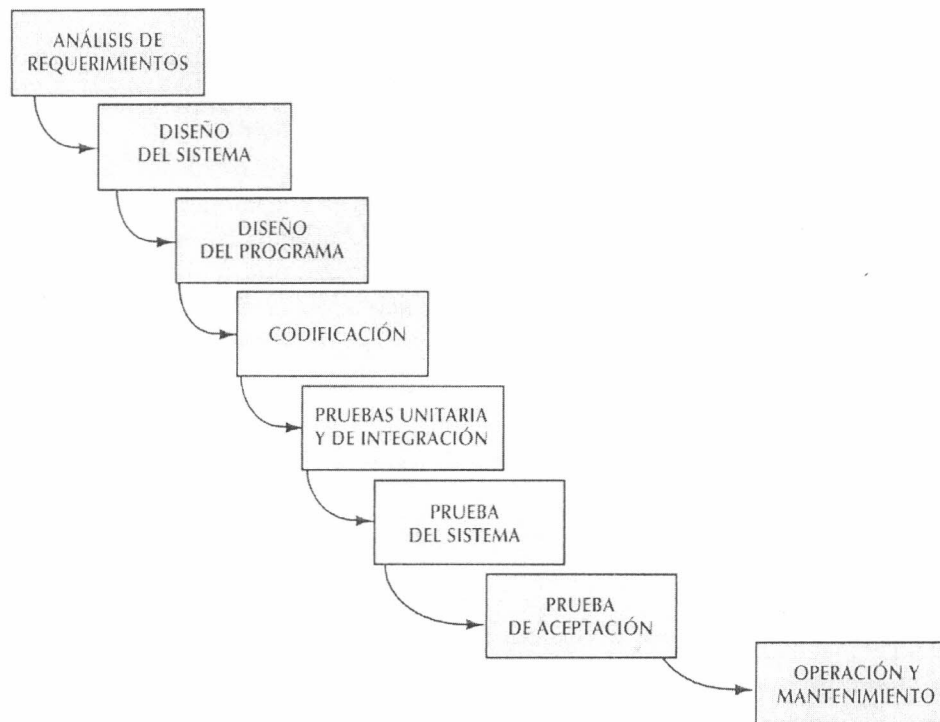


Figura 4.1. El modelo en Cascada

4.3.2 El modelo V

El modelo V es una variación del modelo en cascada que incorpora las actividades de testing desde las etapas tempranas del desarrollo de software.

Como se muestra en la figura 4.2, la codificación forma la punta de la V, con el análisis y el diseño a la izquierda y la prueba y el mantenimiento a la derecha. El modelo V sugiere que la prueba unitaria y de integración también sea utilizada para verificar el diseño del programa. Es decir, durante la prueba unitaria y de integración, los codificadores y los miembros del equipo de prueba deben asegurar que todos los aspectos del diseño del programa se han implementado correctamente en el código. De igual modo, la prueba del sistema debe verificar el diseño del sistema, asegurando que todos los aspectos del diseño del sistema están correctamente implementados. La vinculación entre los lados derecho e izquierdo del modelo V implica que, si se encuentran problemas durante la verificación y la validación, entonces el lado izquierdo de la V puede ser ejecutado nuevamente, para solucionar el problema y mejorar los requerimientos, el diseño y el código antes de retomar los pasos de prueba sobre el lado derecho. En este modelo el diseño de los casos de prueba se va haciendo a medida que finalizan las etapas de desarrollo, y esto permite la detección temprana de errores.

El modelo V hace más explícita parte de la interacción y el rehacer de tareas que están ocultas en la representación de cascada. Mientras el modelo en cascada centra su foco en los documentos y artefactos producidos, el modelo V lo pone en la actividad y la exactitud. [PFL/01]

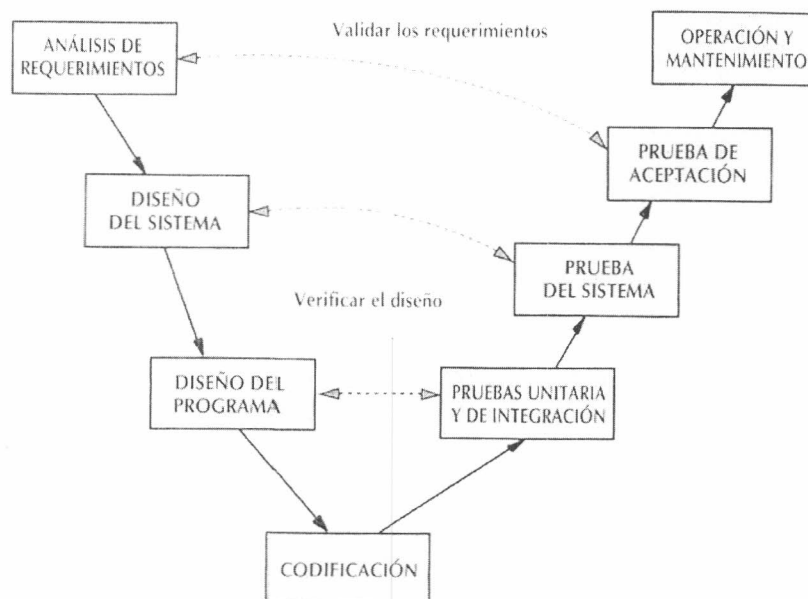


Figura 4.2. El modelo V

4.4 El Testing de sistemas

Sin duda una de las actividades más importantes del proceso de desarrollo de software, y la que más se relaciona con esta tesis, es la de Testing de programas. Ningún equipo de programación está exento de cometer errores, por más bueno que sea. Estos pueden aparecer por una serie de factores dentro del desarrollo:

- **Complejidad de los sistemas actuales:** La mayoría de los sistemas que se realizan en la actualidad operan con grandes cantidades de estados y con fórmulas, actividades y algoritmos complejos.
- **Inexactitud de requerimientos:** Muchas veces se intenta implementar una idea de un cliente sobre un sistema del que ni él mismo todavía tiene claro qué es lo que necesita exactamente.
- **Tamaño y cantidad de personas involucradas:** El creciente tamaño de los proyectos a realizar, y por lo tanto la cantidad de personas involucradas en un proyecto aumenta la complejidad del mismo, y por lo tanto la posibilidad de la existencia de errores.

Se interpreta que un software falla cuando no hace lo que especifican los requerimientos. Por ejemplo, si en la especificación dice que el sistema debe devolver cierto resultado a partir de ciertos parámetros ingresados, y en realidad devuelve otros diferentes, se dice que el sistema falló.

4.4.1 Tipos de defectos

Existen distintos tipos de defectos que puede tener un programa, y que pueden provocar fallos en el mismo. Algunos de ellos son más fáciles de encontrar, mientras que otros resultan más difíciles, y hasta pueden requerir de una prueba en forma indirecta.

Los siguientes son algunos de los tipos de defectos que puede contener un programa [PFL/01]:

- **Defecto Algorítmico:** Se produce cuando el algoritmo o la lógica de un componente no produce la salida apropiada para una entrada dada, debido a que algo está mal en los pasos del procesamiento. Estos defectos son muchas veces fácilmente detectables mediante una prueba de escritorio, pero algunas otras son de difícil detección o arreglo si es que la complejidad del algoritmo es muy grande.
- **Defecto de sintaxis:** Se produce cuando no se utilizan apropiadamente las estructuras del lenguaje de programación. Afortunadamente, los compiladores detectan la gran mayoría de estos defectos.
- **Defecto de computación o de precisión:** Se produce cuando la implementación de una fórmula es errónea o no calcula el resultado con el grado requerido de exactitud.
- **Defecto de documentación:** Se produce cuando el programa evoluciona en su comportamiento o funcionalidad y la documentación no se actualiza siguiendo esa evolución. También se puede producir cuando la documentación está mal realizada o es poco clara.
- **Defecto de estrés o sobrecarga:** Se produce cuando ciertas estructuras del programa se llenan hasta sobrepasar su capacidad especificada. Estas estructuras pueden ser colas, arreglos, vectores, buffers, tablas, etc. Los valores máximos de estas estructuras se suelen definir en la etapa de diseño cuando se especifican las necesidades máximas con respecto a variables como cantidad de usuarios, dispositivos, tiempos de respuesta máximos, etc. También se puede producir cuando el programa no maneja la concurrencia en forma apropiada con la cantidad de usuarios o procesos necesaria.
- **Defecto de capacidad o de límites:** Se produce cuando el desempeño del sistema se vuelve inaceptable a medida que la actividad alcanza su límite especificado. por ejemplo, si los requerimientos especifican que un sistema debe manejar 32 dispositivos, los programas deben ser probados para determinar el rendimiento del sistema cuando los 32 dispositivos están activos. Las condiciones de capacidad también pueden examinarse en relación al número de accesos a disco, número de interrupciones, número de tareas que ejecutan concurrentemente, número de threads que se ejecutan simultáneamente, y otras medidas relacionadas a la capacidad del sistema.
- **Defecto de sincronización o de coordinación:** Se puede producir en sistemas en tiempo real cuando se debe coordinar la ejecución de una cantidad de procesos o ejecutar una secuencia rigurosamente definida, en caso de que el código que coordina dichos eventos sea inadecuado. Suele ser un tipo de defecto difícil de identificar y corregir.
- **Defecto de rendimiento o desempeño:** Se produce cuando el sistema no opera a la velocidad prescrita en los requerimientos.
- **Defecto de recuperación:** Se produce cuando existe una falla en el sistema y éste no se recupera de forma apropiada.
- **Defecto del hardware y software de sistemas:** Se produce cuando el hardware suministrado y el software de sistema no trabajan adecuadamente de acuerdo con las condiciones operativas y procedimientos documentados.
- **Defecto de estándares y procedimientos:** Se produce cuando el código de los programas no sigue los estándares y procedimientos organizacionales definidos. Este tipo de defectos no siempre afecta la corrida de los programas,

pero puede fomentar un ambiente donde los defectos se creen a medida que el sistema es probado y modificado. Además de esto, un código que no sigue estándares es más difícil de mantener.

4.4.2 Tipos de pruebas

Existen diferentes tipos de pruebas que deben hacerse sobre un software antes de poder entregárselo a un cliente con una cierta seguridad de que no posee defectos, o por lo menos que posee la menor cantidad posible. La diferenciación entre las pruebas tiene que ver con qué es lo que se está probando: componentes, grupos de componentes, subsistemas o todo el sistema. Otra diferenciación en los tipos de pruebas tiene que ver con qué es lo que se desea saber: se puede querer conocer si el sistema trabaja de acuerdo con su diseño, si trabaja de acuerdo con sus requerimientos, o también si el funcionamiento del sistema tiene que ver con las expectativas que tienen los usuarios sobre el mismo.

Los siguientes son algunos de los tipos de prueba posibles, que en general se realizan en este orden [PFL/01]:

- **Prueba unitaria:** Cada uno de los componentes del programa se prueba en sí mismo, aislado de los demás componentes del sistema. Esta prueba verifica que el componente funciona correctamente con los tipos de entrada esperados a partir del estudio del diseño del componente.
- **Prueba de integración:** Cuando el conjunto de componentes del sistema ha superado la prueba unitaria, el siguiente paso es asegurar que las interfaces entre los componentes están definidas y se manejan correctamente. Esta prueba consiste en verificar que los componentes del sistema trabajen juntos conforme a lo descrito en las especificaciones del diseño del programa y del sistema.
- **Prueba funcional:** Una vez asegurado que la información pasa entre los componentes de acuerdo con el diseño, se prueba el sistema para asegurar que tiene la funcionalidad deseada. Una prueba funcional evalúa el sistema para determinar si las funciones descritas por la especificación de requerimientos del usuario son realmente ejecutadas por el sistema integrado.
- **Prueba de rendimiento:** Además de los requerimientos del usuario, existen otros tipos de requerimientos, los requerimientos de hardware y software. La prueba de rendimiento compara el sistema con estos requerimientos.
- **Prueba de aceptación:** Cuando la prueba de rendimiento se completa, los Desarrolladores tienen la certeza de que el sistema funciona de acuerdo con su comprensión de la descripción del sistema. El próximo paso es incluir al cliente para tener la certeza de que el sistema trabaja de acuerdo con sus expectativas. La prueba de aceptación se hace en conjunto con el cliente comprobando el sistema contra la especificación de requerimientos del cliente.
- **Prueba de instalación:** Al completar la prueba de aceptación, el sistema aceptado se instala en el ambiente en el que será utilizado y se ejecuta una última prueba de instalación para garantizar que todavía funciona como debe hacerlo.

El diagrama de la figura 4.3 muestra la relación entre los distintos tipos de prueba descriptos [PFL/01]

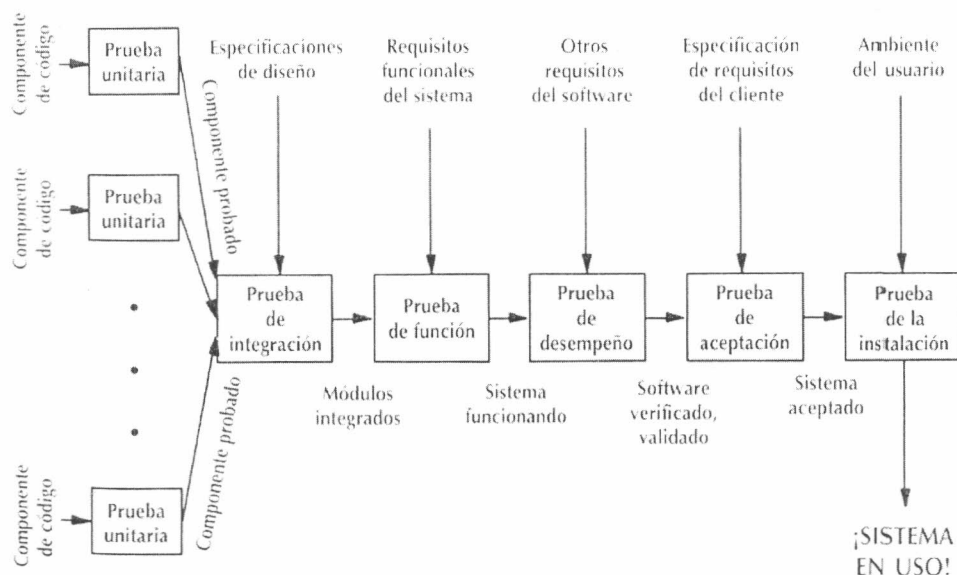


Figura 4.3. Tipos de prueba

4.4.3 Visiones de los objetos de prueba

Al probar un componente, un grupo de componentes, un subsistema o un sistema, la propia visión del objeto de la prueba puede afectar la forma en que se lleve a cabo la prueba.

Prueba de Caja Negra (o Caja Cerrada)

El objeto de prueba se ve desde afuera, sin conocer su contenido. La prueba consiste en alimentar la caja negra con entradas y anotar cuáles son las salidas que se producen. En este caso la meta de la prueba es asegurar que se ha ingresado toda clase de entradas y que la salida observada en cada caso se corresponde con la salida esperada. La ventaja que ofrece este tipo de prueba es que está libre de las restricciones impuestas por la estructura interna o la lógica del objeto de prueba. Sin embargo, esta forma de probar posee la desventaja de que no siempre permite ejecutar una prueba completa del objeto. Esto es porque no siempre el equipo de prueba puede generar un conjunto representativo de casos de prueba que demuestre la correcta funcionalidad para todos los casos.

Prueba de Caja Blanca (o Caja Abierta)

Para solucionar los problemas que posee la visión por caja negra existe la visión por caja blanca. En ella se ve el objeto de prueba como una caja abierta de la cual es posible observar su contenido. Por lo tanto se puede utilizar la estructura del objeto de prueba para probar de distintas formas. Por ejemplo, se pueden inventar casos de prueba que ejecuten todas las instrucciones o todos los caminos de control que existen en el interior del componente, para asegurar que el mismo esté trabajando en forma correcta. Esta forma de test también posee problemas propios. Por ejemplo, si un componente posee estructuras de repetición y diversas ramificaciones la cantidad de caminos formados puede llegar a ser demasiado grande como para ser plausibles de prueba.

Es importante tener en cuenta que la visión de caja negra y la de caja blanca no son las únicas posibles, sino que existen muchas visiones intermedias entre las dos que se pueden adaptar de mejor forma a ciertos componentes, dependiendo de factores como la cantidad de caminos lógicos posibles, la naturaleza de los datos de entrada, la cantidad de cómputo involucrado y la complejidad de los algoritmos.

4.4.4 Actividades de la prueba

La presente tesis deja abierta la posibilidad de utilizar distintos tipos de pruebas en la metodología, siempre que en la misma se puedan definir casos de prueba, como se explica luego. Las siguientes son posibles actividades en que pueden consistir los tipos de pruebas definidos anteriormente.

Examen de código

Cada programador codifica los programas basándose en los documentos de diseño. Por lo tanto un programa refleja necesariamente la interpretación de ese documento por parte del programador. La documentación explica lo que se supone que debe hacer el programa, por lo tanto es útil consultar con un grupo objetivo de expertos para revisar tanto el código como su documentación en busca de equivocaciones, inconsistencias y otros defectos. Este proceso es conocido como "revisión de código". En éste se forma un equipo integrado por el mismo programador y 3 o 4 expertos técnicos, y todos juntos estudian el programa de manera organizada para localizar defectos. Existen dos tipos de revisiones de código: las recorridas y las inspecciones. En una recorrida se presenta el código y la documentación correspondiente al equipo de revisión y el equipo comenta sobre su exactitud. Una inspección de código, método presentado por Fagan (1976), es similar a una recorrida pero más formal. En general involucra varios pasos. Primero, el equipo debe reunirse para hacer una revisión general del código y hacer una descripción de las metas de la inspección. Luego, los miembros del equipo se preparan individualmente para una segunda reunión del grupo. Cada inspector estudia el código y sus documentos relacionados y resalta los defectos encontrados. Finalmente, en una reunión del grupo, los miembros informan lo que han encontrado y registran los defectos que se descubren durante el proceso de discusión de los defectos descubiertos. Tanto las recorridas como las inspecciones critican el código, no al codificador, y los resultados no deben verse reflejados en la evaluación del desempeño de los programadores. [PFL/01]

Demostración de código correcto

El paso siguiente a escribir el código y examinarlo por un equipo de trabajo, es someterlo a un análisis minucioso de una forma más estructurada para establecer su exactitud.

Una manera de investigar la exactitud de un programa consiste en visualizar el código como un enunciado de flujo lógico. Si se puede reescribir el programa utilizando un sistema lógico formal, se puede probar la exactitud aplicando las expresiones de este sistema. La exactitud se interpreta en términos del diseño y se desea que las expresiones sigan los preceptos de la lógica matemática. Por ejemplo, si se puede reformular un programa como un conjunto de afirmaciones y teoremas, se puede demostrar que la verdad de los teoremas implica la exactitud del código.

La ventaja que tiene la demostración de código correcto de esta forma es que se puede probar fehacientemente la correctitud del código. Sin embargo posee la desventaja de que toma mucho trabajo llevar a cabo este tipo de pruebas. En muchos

casos toma más tiempo probar que el código es correcto que escribir el código mismo [PFL/01].

Verificación de componentes del programa

La demostración de correctitud de programas tiene mucho valor teórico y de investigación. Sin embargo, los equipos de desarrollo de las empresas están, en general, más interesados en testear sus programas que en demostrar que son correctos.

A diferencia de la demostración, que considera solamente el código y sus condiciones de entrada y salida, la prueba de programas, que será el método que se utilizará en la metodología de este trabajo, focaliza el componente desde una visión externa. En este sentido la prueba comienza con una serie de ensayos cuyos resultados se convierten en la base para decidir cómo se comporta un programa en una determinada situación. Mientras una demostración dice cómo trabajará el programa en un ambiente hipotético descrito por el diseño y los requerimientos, la prueba proporciona información que indica cómo trabaja un programa en su ambiente real de operación. [PFL/01]

Para probar un componente se seleccionan datos de entrada y condiciones, se permite que el componente manipule las condiciones y se observa la salida. La entrada se selecciona de manera que la salida demuestre algo significativo acerca del comportamiento del código.

Casos de prueba

Un caso de prueba es un conjunto particular de los datos de entrada que será utilizado en la prueba de un programa. Una prueba es un conjunto finito de casos de prueba. Es importante notar que la cantidad de casos de prueba existentes es infinita.

Una de las dificultades más grandes consiste en seleccionar, entre la cantidad infinita de posibles casos de prueba, una cantidad finita y manejable en la práctica que permita convencer a desarrolladores y clientes que el programa trabaja en forma correcta no sólo para los casos de prueba seleccionados sino que para todas las demás entradas posibles.

Lo primero que hay que hacer es definir los objetivos de una prueba. Luego se seleccionan los casos de prueba que satisfacen dicho objetivo y así se forma la prueba. Uno de los objetivos de una prueba puede ser demostrar que todas las sentencias se ejecutan correctamente. Otro puede ser mostrar que toda función realizada por el código se hace en forma correcta.

Asimismo, y como se explicó anteriormente, se puede considerar al código como una caja abierta o como una caja cerrada, examinando o no la lógica interna del componente para la realización de los casos. (Ver [PFL/01] Cap. 8 para más detalles sobre prueba de programas y generación de casos de test)

5 Teoría de Juegos

5.1 Introducción

La Teoría de Juegos es una teoría matemática que intenta describir el fenómeno socioeconómico de la interacción humana, esto es, el conflicto y la cooperación entre individuos tomadores de decisiones (llamados jugadores). [ROS/01]

La teoría trata el problema de la toma de decisiones de un grupo de individuos bajo incertidumbre y falta de información sobre algunas cuestiones como el estado del ambiente, el estado del proceso de decisión interpersonal y el estado de las habilidades e incentivos de los oponentes. Por lo tanto es inevitable la existencia de un contexto probabilístico, dado que tanto los estados de la naturaleza o ambiente como el comportamiento estratégico de los jugadores participantes son generalmente pensados como una influencia de tipo azarosa.

A pesar de que la teoría no permite resolver la totalidad de los juegos existentes, sí puede ayudar a determinar los resultados de una gran cantidad de juegos, situaciones de negociación y situaciones de conflicto de intereses que se presentan cotidianamente. Justamente en esto se basa la importancia que tiene la teoría que se va a presentar en este capítulo para muchas disciplinas diferentes.

Es sabido que las personas poseen una gran cantidad y variedad de formas de razonamiento, pensamientos, conductas, principios y motivaciones en los que se basan para tomar las decisiones. Por lo tanto, es necesario tipificar el comportamiento de las personas, utilizando un modelo sobre sus razonamientos. Para esto se utiliza una premisa básica que trata al jugador como un ente estrictamente racional y utilitario. Esta es:

"Cada jugador buscará, en todos los casos, el máximo bienestar posible"

En todo este capítulo se darán las bases referentes a la Teoría de Juegos necesarias para entender este trabajo.

5.2 Un poco de historia

La Teoría de Juegos fue diseñada y elaborada por el matemático John von Neumann y el economista Oskar Morgenstern con el fin de realizar análisis económicos de ciertos procesos de negociación. [VNE/44]

Probablemente el problema más famoso de la Teoría de Juegos sea el "Dilema del Prisionero", pensado por A.W. Tucker [LUC/57], y descrito en 5.2.1.

Uno de los adelantos más importantes de la teoría lo dio el matemático John Forbes Nash, Jr. (1928-), que creó en 1950 la noción de "Equilibrio de Nash", que corresponde a una situación en la que dos partes rivales están de acuerdo con un determinado estado del juego o negociación, y cuya alteración ofrece desventajas a ambas partes.

Otros importantes representantes de la teoría de juegos fueron el húngaro nacionalizado estadounidense John Harsanyi (1920-) y el alemán Reinhard Selten. Nash, Harsanyi y Selten recibieron el Premio Nóbel de Economía de 1994 por sus contribuciones a la Teoría de Juegos. [RUF/03], [USB/96] y [COS/00]

5.2.1 El dilema del prisionero.

El dilema del prisionero, el problema más estudiado en Teoría de Juegos, ejemplifica el clásico conflicto entre los intereses individuales y colectivos de quienes toman decisiones racionales. En éste hay dos sospechosos de haber cometido un delito,

quienes son detenidos e interrogados en celdas separadas. Las opciones de los prisioneros son:

- Si ninguno confiesa, con base en las pruebas que acumuló la policía, ambos irán a la cárcel por un año.
- Si sólo uno confiesa, y además colabora con las autoridades saldrá libre, mientras que el otro, por no colaborar, recibirá una sentencia de seis años.
- Si ambos confiesan, la sentencia será de tres años para los dos.

De acuerdo con estas opciones, si cada uno analiza qué le conviene hacer para obtener el mayor beneficio individual posible, concluirá lo siguiente:

- Si el otro no confesara, le convendría confesar, para salir libre en lugar de ir preso por un año.
- Si el otro confesara, también le convendría confesar, para lograr una rebaja en su pena e ir preso por tres años en lugar de seis.

En consecuencia, independientemente de lo que pueda hacer el otro, le convendría confesar.

Sin embargo, si se analizara el bienestar colectivo de la asociación para delinquir formada por los dos prisioneros, es decir, si se buscara cómo reducir al mínimo la suma del tiempo que podrían pasar en la cárcel sus dos integrantes, convendría que ninguno confesara, pues:

- Si ambos confesaran, los dos irían presos tres años y su asociación sufriría un total de seis años de cárcel.
- Si ninguno confesara, ambos terminarían presos por un año, con lo que el costo carcelario total ascendería a dos años, situación más conveniente para ambos.

De esta forma, el raciocinio individual los lleva a confesar a los dos lo cual constituye, desde el punto de vista de los prisioneros, una irracionalidad colectiva, ya que ambos estarían mejor guardando silencio. [AYA/99]

5.3 Aplicaciones

La Teoría de Juegos tiene aplicación en una serie de disciplinas bien diferentes entre sí. Algunas de éstas son:

- Contratos
- Guerras militares
- Guerras comerciales
- Marketing para la competencia en los mercados
- Negociaciones domésticas
- Negociaciones comerciales
- Negociaciones colectivas
- Alianzas

Debe incluirse en la lista cualquier otra situación en la que dos o más individuos requieren interactuar a fines de obtener ganancias económicas. Como el ser humano es un *homo economicus* tanto como un *homo ludicus*, puede encontrar infinidad de aplicaciones de la Teoría de Juegos. [RUF/03]

5.4 Conceptos básicos

Un **juego** es un proceso en el que dos o más personas toman decisiones y acciones, la estructura de las cuales está inscripta en un conjunto de reglas (que pueden ser formales o informales), a fines de obtener beneficio. [RUF/03]

Los elementos esenciales de un juego son:

- **Jugadores**
- **Acciones**
- **Pagos**
- **Información**

Estos elementos son llamados en conjunto las **Reglas del Juego**.

Cuando se desea aplicar la Teoría de Juegos en una situación en particular lo que se debe hacer es describir esa situación en términos de las reglas de un juego, para que la teoría nos permita explicar qué es lo que pasará en dicha situación.

Cada jugador tiene distintas decisiones y acciones que puede tomar para llegar a su objetivo, o sea maximizar sus beneficios.

Cada combinación de decisiones y acciones determina una **situación** particular, y, dado que las decisiones y acciones de los agentes involucrados (llamados jugadores) pueden ser combinadas de numerosas formas, las situaciones generadas también serán numerosas y su magnitud igual a las de las combinaciones de decisiones y acciones de los agentes. El conjunto total de situaciones posibles es el **cuadro situacional** del juego. Siguiendo con este razonamiento, encontramos que cada situación (es decir, cada punto del cuadro situacional) genera un beneficio (o perjuicio) determinado para cada uno de los jugadores. El beneficio (o perjuicio) que le da a un jugador una situación particular puede ser comparado con el beneficio (o perjuicio) que le ofrecen las otras situaciones.

De esta idea de beneficio o perjuicio que le trae una situación a un jugador sale el concepto de **pago**, que es justamente el beneficio cuantitativo o cualitativo que percibe cada uno de los jugadores para cada una de las situaciones que se pueden dar en un juego determinado.

La interacción entre los jugadores da como resultado una o más situaciones que serán las que generará el juego con más probabilidad.

Por último, el **equilibrio** del juego es el resultado que arroja el juego, o su Output y consiste en esas situaciones que genera el juego con más probabilidad, junto con las probabilidades de cada una de ellas.

Definamos bien cada uno de estos conceptos presentados:

Jugadores: son los individuos que toman decisiones. Estos pueden ser personas, o entes en general (como empresas por ejemplo). Como se dijo anteriormente, para analizar el comportamiento de un jugador se utilizará una premisa: cada jugador buscará, en todos los casos, el máximo bienestar posible.

Acciones: una acción del jugador i , que se denota a_i , es una elección que este jugador puede realizar. Justamente, el **Conjunto de Acciones** de un jugador i , denotado $A_i = \{a_i\}$, está formado por todas las acciones que el jugador puede realizar en el juego.

Combinación de acciones o Situación: es un conjunto ordenado $a = \{a_i\}$, ($i=1, \dots, n$) formado por una acción para cada uno de los n jugadores del juego.

Pagos: como se dijo anteriormente, los pagos son los beneficios que percibe un jugador ante una Situación del juego.

Los pagos se definen a partir de funciones de Utilidad (o de pagos) para cada uno de los jugadores.

La **Función de Utilidad** (o de pagos) para el jugador i , denotada $\Pi_i(S_1, \dots, S_n)$ define la utilidad esperada para el jugador i en función de la situación a la que se llegó, o sea en función de las estrategias (o acciones) seleccionadas por cada uno de los jugadores (incluido él mismo)

Información: El concepto de información se modela a partir del concepto de Conjunto de Información. El **Conjunto de información** de un jugador está formado por el conocimiento que posee sobre distintas variables en un momento particular del juego.

5.5 Herramientas

Existen distintas herramientas que se utilizan en la Teoría de Juegos. Cada una sirve para diferentes tipos de juegos o situaciones, o para diferentes tipos de análisis.

Algunas de ellas son: [RUF/03]

- **Matriz de pagos:** La Teoría de Juegos estudia una gran parte de los juegos bipersonales por medio del análisis matricial. El análisis matricial corresponde a la expresión, mediante matrices, de las situaciones que pueden ser generadas por las alternativas de decisión y acción de dos jugadores. El análisis matricial recurre a la forma llamada "Matriz de Pagos" que presenta las diversas opciones de decisión y acción de cada jugador y las resultantes situaciones.
- **Curvas de reacción:** Las curvas de reacción muestran, en un gráfico cartesiano, las combinaciones de decisiones (puede ser en las abscisas) y pagos (puede ser en las ordenadas). Un ejemplo sencillo de curvas de reacción puede verse en las curvas de oferta y demanda.
- **Árboles de resultados sucesivos:** Un diagrama de árbol de resultados sucesivos se utiliza en juegos que implican secuencias de movimientos (un movimiento es un binomio decisión-acción). En este árbol, se define un punto de partida (por ejemplo, la posición inicial del jugador A). A partir del inicio, se extienden ramas que representan los diferentes movimientos que puede realizar el jugador que inicia la competencia. Los diferentes movimientos o ramas definirán igual número de resultados o pagos, que pueden servir como punto de partida para nuevas decisiones del jugador siguiente (por ejemplo, el jugador B). El proceso se repite hasta completar el número de movimientos que A y B pueden realizar. Un juego con un movimiento para A y uno para B posee dos generaciones de ramas. Un juego con dos movimientos para A y dos movimientos para B posee cuatro generaciones de ramas. En general, un juego con m movimientos para A y n movimientos para B (el valor absoluto de $m-n$ no puede ser mayor que 1) tiene $m+n$ generaciones de ramas. Las hojas de las ramas contienen la descripción de los posibles resultados del juego.

A continuación se presenta en más detalle la matriz de pagos, que es la herramienta principal que se utiliza en este trabajo.

5.5.1 Matriz de pagos

Como se dijo anteriormente, el análisis matricial recurre a la forma llamada "Matriz de Pagos" (en inglés, Pay-Off Matrix) la cual presenta las diversas opciones de decisión y acción de cada jugador y las resultantes situaciones particulares. En efecto, la

intersección o combinación de la alternativa elegida por un jugador y la alternativa elegida por otro crea un único punto, de coordenadas (decisión del jugador A, decisión del jugador B). La situación particular definida por ese punto tiene un valor, que es la combinación de premios obtenida por los dos jugadores. A cada jugador le corresponde un premio (cuando el premio tiene valor negativo, se convierte en un castigo).

[RUF/03]

A continuación se muestra una matriz de pagos genérica.

		Jugador B	
		Acción B ₁	Acción B ₂
Jugador A	Acción A ₁	1. ($\pi_A(A_1, B_1)$; $\pi_B(A_1, B_1)$)	2. ($\pi_A(A_1, B_2)$; $\pi_B(A_1, B_2)$)
	Acción A ₂	4. ($\pi_A(A_2, B_1)$; $\pi_B(A_2, B_1)$)	3. ($\pi_A(A_2, B_2)$; $\pi_B(A_2, B_2)$)

Pagos para: (Jugador A; Jugador B)

En este ejemplo existen 2 jugadores: A y B.

Cada uno tiene dos acciones posibles: El jugador A puede elegir A₁ o A₂, y el jugador B puede elegir B₁ o B₂.

De esta forma existen dos funciones, una para cada jugador. Estás son:

$$\pi_A(A_A, A_B) \text{ y } \pi_B(A_A, A_B)$$

Cada una de estas funciones define, para un jugador en particular, el pago que le corresponde a partir de las acciones seleccionadas por cada uno de los jugadores, incluido él mismo.

Un ejemplo concreto de una matriz de pagos aparece en 5.6.2, con el dilema del prisionero.

5.6 Tipos de juegos

5.6.1 Juegos simultáneos Vs. secuenciales

En los **juegos simultáneos** los distintos jugadores toman sus decisiones en forma simultánea. De esta forma ningún jugador conoce la decisión del otro antes de tomar la propia.

Un ejemplo de juego con estas características es el conocido "Piedra, papel o tijera".

En este juego cada jugador elige, en cada turno, un elemento entre piedra, papel y tijera. Cuando los dos jugadores toman la decisión, se la muestran al otro. De esta forma, la piedra le gana a la tijera, ésta le gana al papel, y a su vez el papel le gana a la piedra. En este juego ninguno de los jugadores conoce la decisión del otro antes de tomar la suya.

En los **juegos secuenciales**, existe un orden en que los jugadores toman las decisiones. De esta forma, el segundo jugador posee el conocimiento de la acción tomada por el primero antes de decidir la suya.

Un ejemplo es el ajedrez, en el que un jugador hace su movida y luego el otro hace la suya. El segundo jugador posee el conocimiento de la jugada del primero antes de realizar la suya.

5.6.2 Juegos de suma cero Vs. no suma cero

Los **juegos de suma cero** tratan situaciones en las que los jugadores compiten por un único pago. Esto quiere decir que la suma de los pagos de todos los jugadores en cualquier situación posible es cero. En estos juegos, el premio de un jugador es exactamente igual al castigo del otro.

Esto se da, por ejemplo, en una partida de Poker. En cada mano, un jugador gana la misma cantidad de dinero que la suma de lo que pierden todos los demás jugadores juntos. De esta forma, la suma entre las ganancias (positivas o negativas) de todos los jugadores siempre da cero; no se crea ni se destruye dinero.

Los **juegos de no suma cero** tratan situaciones en las que se compete por pagos o castigos que pueden ser obtenidos simultáneamente.

El ejemplo más clásico lo constituye el dilema del prisionero, descrito en la sección 5.2.1.

La matriz de pagos de este juego es la siguiente:

		Prisionero B	
		Confesar	No Confesar
Prisionero A	Confesar	1. (-3 ; -3)	2. (-0 ; -6)
	No Confesar	4. (-6 ; -0)	3. (-1 ; -1)

Pagos, en años de cárcel, para: (**Prisionero A**; **Prisionero B**)

En este tipo de juegos la suma de las ganancias de los jugadores no es siempre cero; puede pasar que ambos jugadores pierdan, o que ambos ganen.

5.7 Equilibrio de Nash

Dada una situación cualquiera definida por una elección de A y una elección de B, si ocurre que A supone que B no modificará su elección y opta por no modificar la suya propia y, simultáneamente, B supone que A no modificará su elección y opta también por no modificar la suya, se dice que tal situación es un equilibrio Nash. Como se ve, el equilibrio Nash es una situación que presenta ventajas para los dos jugadores, y en razón de tales ventajas, ni A ni B cambiarán su decisión. Cuando se trata el problema del análisis de los juegos, la existencia de un equilibrio Nash indica un conjunto de acciones tales que ninguno de los jugadores, si considera que las acciones de su oponente están dadas, deseará cambiar su propia acción.

En un equilibrio de Nash, el jugador observará que, como la acción de su rival está predeterminada, él mismo podrá elegir su propia acción dentro de una gama de posibilidades. Si el juego ha resultado en una situación $S_{1i,2j}$ correspondiente a un equilibrio de Nash en la que el jugador 1 efectúa una acción i y el 2 una acción j, el jugador 1 rechazará cualquier posibilidad de realizar una acción distinta de i.

La conducta de dos jugadores puede dar lugar a cualquiera de las siguientes situaciones:

- **Equilibrio Nash correspondiente al resultado de aplicar estrategias puras** (es decir, decisiones y acciones que se toman con certeza, equivalente a una probabilidad del 100%).

- **Equilibrio Nash correspondiente al resultado de aplicar estrategias mixtas** (es decir, decisiones y acciones que se toman con una probabilidad inferior al 100%).
- **Existencia de dos o más equilibrios Nash dentro del juego.**
- **No existencia de equilibrios Nash.**

5.8 Estrategias dominantes

Una estrategia S_1 es **dominante** sobre otra S_2 para un jugador i si a éste le conviene siempre elegir la estrategia S_1 por sobre S_2 , sin importar la estrategia que elija el otro jugador, (o sea que sus pagos son mayores si elige S_1 que si elige S_2). En este caso, es posible eliminar la estrategia S_2 ya que se dice que está dominada por S_1 .

5.9 Estrategias puras y mixtas

Una estrategia pura es aquella decisión que se toma con certeza. En contraposición a tal concepto, una estrategia mixta es una decisión que se toma con una determinada probabilidad. Cuando un problema no alcanza una solución, o sea un equilibrio, vía estrategias puras, con frecuencia puede ser enfocado desde una perspectiva de estrategias mixtas. Así, se dice que los problemas que no tienen solución vía estrategias puras pueden tenerla vía estrategia mixtas. Ambas situaciones pueden ser vistas como soluciones ciertas versus gamas de soluciones probables.

5.9.1 Estrategias puras

Una estrategia pura le indica al jugador cuál es la acción que debe jugar. Por ejemplo, una estrategia Pura para el jugador A podría decir: jugar A_2 . Un equilibrio de Nash en estrategias puras se da cuando los dos jugadores se encuentran jugando una estrategia pura en particular, y les conviene quedarse en ella siempre que el otro jugador tampoco cambie de estrategia. A continuación se da un ejemplo de un juego que posee un equilibrio en estrategias puras.

Juego 1		Jugador B	
		Acción B_1	Acción B_2
Jugador A	Acción A_1	1. (5 ; 3)	2. (7 ; 1)
	Acción A_2	4. (0 ; 4)	3. (2 ; 0)

Pagos para: (Jugador A; Jugador B)

En el juego 1 la combinación de acciones (A_1, B_1) es un equilibrio de Nash en estrategias puras. Esto es porque en esta situación, A recibe 5. Si se cambiara a A_2 pasaría a recibir 0, lo que no le conviene. De igual forma, B recibe 3, y si se cambiara a B_2 pasaría a recibir 1, por lo cual a B tampoco le conviene cambiarse. Como a ninguno de los dos les conviene cambiar de acción mientras que el otro tampoco cambie, entonces esta combinación de acciones es un equilibrio de Nash.

Es posible que haya más de un equilibrio en un juego. Sin embargo, este no es el caso. Esto es porque:

- En la combinación (A_2, B_1) , A recibe 0. Si se cambiara a A_1 pasaría a recibir 5. Por lo tanto se cambia a A_1 . Entonces no es un equilibrio de Nash.
- En la combinación (A_1, B_2) , B recibe 1. Si se cambiara a B_1 pasaría a recibir 3. Por lo tanto se cambia a B_1 . Entonces no es un equilibrio de Nash.
- Por último, en la combinación (A_2, B_2) , B recibe 0. Si se cambiara a B_1 pasaría a recibir 4. Por lo tanto se cambia a B_1 . Entonces no es un equilibrio de Nash.

De esta forma se demuestra que el único equilibrio de Nash existente en este juego es la combinación (A_1, B_1) . Además, este equilibrio es en estrategias puras.

5.9.2 Estrategias mixtas

Existen ciertos casos en los que no se puede encontrar ningún equilibrio utilizando estrategias puras como se hizo en el punto anterior. Veamos el siguiente ejemplo:

Juego 2

		Jugador B	
		Acción B_1	Acción B_2
Jugador A	Acción A_1	1. (5 ; 3)	2. (2 ; 5)
	Acción A_2	4. (0 ; 4)	3. (4 ; 0)

Pagos para: (Jugador A; Jugador B)

Analicemos la existencia de un equilibrio de Nash mediante estrategias puras.

- En la combinación (A_1, B_1) , B recibe 3. Si se cambiara a B_2 pasaría a recibir 5. Por lo tanto se cambia a B_2 .
- En la combinación (A_1, B_2) , A recibe 2. Si se cambiara a A_2 pasaría a recibir 4. Por lo tanto se cambia a A_2 .
- En la combinación (A_2, B_2) , B recibe 0. Si se cambiara a B_1 pasaría a recibir 4. Por lo tanto se cambia a B_1 .
- En la combinación (A_2, B_1) , A recibe 0. Si se cambiara a A_1 pasaría a recibir 5. Por lo tanto se cambia a A_1 .

Dado que en ninguna de las combinaciones ambos jugadores prefieren quedarse en su acción antes de pasar a otra, entonces no existe ningún equilibrio de Nash en estrategias puras.

Cálculo de equilibrio en estrategias mixtas

Cuando un problema no alcanza una solución vía estrategias puras, con frecuencia puede ser enfocado desde una perspectiva de estrategias mixtas. Esto quiere decir buscar un equilibrio en el que los jugadores utilicen estrategias basadas en una cierta probabilidad de elegir cada una de las acciones.

Se dijo previamente que cuando se utilizan estrategias puras el equilibrio está formado por una combinación de acciones de los jugadores que hace que ninguno de ellos quiera cambiar de acción si el otro tampoco desea hacerlo. Cuando se habla de equilibrios basados en estrategias mixtas, cambia un poco el concepto. Un equilibrio en estrategias mixtas está formado por una distribución de probabilidades para cada jugador, una probabilidad para cada posible acción. Cada uno de ellos utilizará esa distribución para decidir que acción va a seleccionar. Entonces, lo primero que hay que hacer es agregar al juego dos variables de probabilidad para los jugadores como se muestra en la siguiente matriz:

Juego 3		Jugador B	
		Acción B ₁ (β)	Acción B ₂ ($1 - \beta$)
Jugador A	Acción A ₁ (α)	1. (5 ; 3)	2. (2 ; 5)
	Acción A ₂ ($1 - \alpha$)	4. (0 ; 4)	3. (4 ; 0)

Pagos para: (Jugador A; Jugador B)

En este juego se agregaron las dos variables α y β . α indica la probabilidad de que el Jugador A seleccione la acción A₁, mientras que β indica la probabilidad de que el Jugador B seleccione la acción B₁.

El próximo paso es calcular los valores de las variables para conocer el equilibrio en estrategias mixtas. Veamos como se realiza esto mediante un juego de ejemplo ([RAS/01])

El juego del bienestar social

El juego del bienestar social modela una situación en la que el gobierno de un país desea ayudar a los pobres, pero sólo a aquellos que buscan trabajo.

A continuación se muestra la matriz de pagos que representa esta situación:

Juego del bienestar social		Pobre	
		Buscar trabajo (β)	No Buscar trabajo ($1 - \beta$)
Gobierno	Ayudar (α)	1. (3 ; 2)	2. (-1 ; 3)
	No ayudar ($1 - \alpha$)	4. (-1 ; 1)	3. (0 ; 0)

Pagos para: (Gobierno; Pobre)

Se puede ver que esta situación no posee equilibrios de Nash basados en estrategias puras:

- La combinación (Ayudar, Buscar trabajo) no es equilibrio, porque el Pobre elegirá pasarse a No buscar Trabajo

- (Ayudar, No buscar trabajo) no es equilibrio, porque el Gobierno elegirá pasarse a No ayudar
- (No ayudar, Buscar trabajo) no es equilibrio, porque el Gobierno elegirá Ayudar.
- (No ayudar, No buscar Trabajo) no es equilibrio porque el Pobre elegirá Buscar Trabajo.

Con esto se demuestra que el juego no posee equilibrios en estrategias puras. Sin embargo, este juego sí posee un equilibrio en estrategias mixtas, que se calcula a continuación:

El pago para un jugador es el valor esperado de los pagos de ese jugador. Por ejemplo, si el Gobierno juega "Ayudar" con probabilidad α y el Pobre juega "Buscar trabajo" con probabilidad β , entonces el pago esperado para el Gobierno es

$$\begin{aligned}\Pi_{\text{Gobierno}} &= \alpha[3\beta + (-1)(1-\beta)] + [1-\alpha][-1\beta + 0(1-\beta)] \\ &= \alpha[3\beta - 1 + \beta] - \beta + \alpha\beta \\ &= \alpha[5\beta - 1] - \beta\end{aligned}$$

Si sólo se permiten estrategias puras, entonces α sólo puede valer 0 o 1. Sin embargo, utilizando estrategias mixtas α puede ser un valor entre 0 y 1.

Dado que el gobierno intentará maximizar sus pagos, lo que se debe hacer es maximizar el valor de Π_{Gobierno} . Esto se resuelve como un problema de maximización de funciones.

1) Se deriva la función con respecto a la variable elegida (α en este caso)

$$\frac{\partial \Pi_{\text{Gobierno}}}{\partial \alpha} = 5\beta - 1$$

2) se iguala la ecuación a 0 para obtener la condición de maximización.

$$\frac{\partial \Pi_{\text{Gobierno}}}{\partial \alpha} = 5\beta - 1 = 0$$

3) Se resuelve

$$\beta = 1/5 = 0,2$$

Por lo tanto, en el equilibrio en estrategias mixtas de este juego, el Pobre elige trabajar el 20% del tiempo.

El modo de obtener el resultado puede parecer un poco extraño por el hecho de derivar la función de pagos esperados de un jugador para calcular la probabilidad de que el otro tome alguna decisión. En realidad esta forma de cálculo es lógica, ya que la Teoría de Juegos habla de la interacción de los jugadores, entonces las decisiones de unos dependen de las decisiones de los otros.

Para obtener el valor de α , o sea la probabilidad de que el gobierno elija Ayudar, se debe realizar el mismo procedimiento, pero con la función de pagos del pobre.

$$\begin{aligned}\Pi_{\text{Pobre}} &= \beta(2\alpha + 1(1-\alpha)) + [1-\beta][3\alpha + 0(1-\alpha)] \\ &= 2\alpha\beta + \beta - \alpha\beta + 3\alpha - 3\alpha\beta \\ &= \beta(2\alpha + 1 - \alpha - 3\alpha) + 3\alpha \\ &= \beta(-2\alpha + 1) + 3\alpha\end{aligned}$$

Entonces la condición de primer orden es

$$\frac{\partial \Pi_{Pobre}}{\partial \beta} = -2.\alpha + 1 = 0$$

$$\alpha = 1/2 = 0,5$$

Esto quiere decir que el gobierno, para que exista un equilibrio de Nash, debe seleccionar la acción Ayudar con una probabilidad del 50%.

Impredecibilidad

Es importante entender la ventaja que representa para el gobierno en el ejemplo anterior la existencia de un equilibrio de Nash en estrategias mixtas. Ésta es la impredecibilidad.

A continuación se muestra por qué un equilibrio en estrategias mixtas representa impredecibilidad con el juego de los dedos, similar al anteriormente nombrado "Piedra, papel o tijera" [DIX/91]. El juego de los dedos consiste en dos jugadores, A y B, que deben elegir, en cada mano y en forma simultánea, si muestran 1 o 2 dedos. En el momento en que eligieron, se muestran uno al otro su elección, también en forma simultánea. En ese momento, si la cantidad total de dedos es impar, gana un punto el jugador A y pierde uno el jugador B. De lo contrario, si la cantidad total es par, gana un punto el jugador B y pierde uno el jugador A.

La siguiente es la matriz de pagos de este juego:

Juego de los dedos		Jugador B	
		1 dedo	2 dedos
Jugador A	1 dedo	1. (-1 ; 1)	2. (1 ; -1)
	2 dedos	4. (1 ; -1)	3. (-1 ; 1)

Pagos para: (Jugador A; Jugador B)

Analizando esta matriz se puede ver que no existen estrategias dominadas o dominantes. "1 dedo" no es dominada para A por "2 dedos" porque en caso de que B elija "2 dedos" A prefiere "1 dedo". Tampoco "2 dedos" es dominada por "1 dedo" ya que en caso de que B elija "1 dedo" A prefiere "2 dedos". De forma análoga se puede ver que B tampoco posee estrategias dominantes ni dominadas.

También se puede concluir que no existe ningún equilibrio en estrategias puras. Esto es porque:

- (1 dedo, 1 dedo) no es equilibrio, porque A elegirá pasarse a 2 dedos.
- (1 dedo, 2 dedos) no es equilibrio, porque B elegirá pasarse a 1 dedo.
- (2 dedos, 1 dedo) no es equilibrio, porque B elegirá pasarse a 2 dedos.
- (2 dedos, 2 dedos) no es equilibrio, porque A elegirá pasarse a 1 dedo.

Este resultado es lógico porque si A jugara siempre 1 dedo, B jugaría siempre 1 dedo también y le ganaría. De lo contrario, si A jugara siempre 2 dedos, B jugaría siempre 1 dedo y también le ganaría.

Esto muestra que el único equilibrio que puede existir es en estrategias mixtas, lo que implica que cada jugador deberá seleccionar cada acción con un porcentaje de probabilidad, y ésta será la única forma de hacerse impredecible ante el otro jugador. Cualquier otra estrategia que elija el jugador podrá ser explotada por el oponente a su beneficio. Para calcular estos porcentajes se agregan las dos variables de probabilidad al modelo:

Juego de los dedos		Jugador B	
		1 dedo (β)	2 dedos ($1-\beta$)
Jugador A	1 dedo (α)	1. (-1 ; 1)	2. (1 ; -1)
	2 dedos ($1-\alpha$)	4. (1 ; -1)	3. (-1 ; 1)

Pagos para: (Jugador A; Jugador B)

Luego se calcula el equilibrio:

$$\begin{aligned}
 \Pi_{JugadorA} &= \alpha \cdot [-\beta + (1-\beta)] + [1-\alpha][\beta - (1-\beta)] \\
 &= \alpha \cdot [-2\beta + 1] + [1-\alpha][2\beta - 1] \\
 &= -2\beta\alpha + \alpha + 2\beta - 2\alpha\beta + \alpha - 1 \\
 &= -4\beta\alpha + 2\alpha + 2\beta - 1
 \end{aligned}$$

$$\frac{\partial \Pi_{JugadorA}}{\partial \alpha} = -4\beta + 2 = 0$$

$$\beta = 0,5$$

El resultado es el mismo para el jugador B (ya que el juego es simétrico)

$$\alpha = 0,5$$

También se puede calcular la esperanza de ganancia, por ejemplo, del jugador A. Esta es:

$$\Pi_{JugadorA} = -4\beta\alpha + 2\alpha + 2\beta - 1 = -4 \cdot 0,5 \cdot 0,5 + 2 \cdot 0,5 + 2 \cdot 0,5 - 1 = 0$$

Esto quiere decir que si se juegan una cantidad de manos lo suficientemente grande tanto el jugador A como el B terminarán con 0 puntos.

El equilibrio calculado muestra que, no sólo es una mala jugada para A elegir siempre la misma cantidad de dedos, sino que aunque elija cada cantidad de dedos con una cierta probabilidad, el resultado siempre será peor que en el equilibrio, ya que B podrá sacar provecho de esta situación.

Por ejemplo, si A eligiera "1 dedo" con probabilidad 0,75 ($\alpha=0,75$), entonces B podría jugar siempre "1 dedo" ($\beta=1$):

$$\Pi_{JugadorA} = -4\beta\alpha + 2\alpha + 2\beta - 1 = -4 \cdot 1 \cdot 0,75 + 2 \cdot 0,75 + 2 \cdot 1 - 1 = -0,5$$

Como la esperanza de puntos es menor, entonces al jugador A le conviene la situación de equilibrio antes que la otra. Lo mismo ocurre con cualquier otro a que A pueda elegir.

Este ejemplo muestra que la situación de equilibrio en estrategias mixtas es la única que permite a los jugadores hacerse impredecibles ante el oponente.

Un ejemplo un poco más complejo con respecto a impredecibilidad sobre un partido de tenis puede encontrarse en [DIX/91], Pág. 172.

Actuar en forma azarosa

En la sección anterior se mostró lo importante que es utilizar el equilibrio en estrategias mixtas para la toma de decisiones, ya que ésta es la única forma de volverse impredecible. Esto significa asignar una probabilidad a cada una de las acciones y elegir cada acción de acuerdo a esa probabilidad.

Sin embargo, esta condición no es suficiente para volverse impredecible. Supongamos la situación del juego de los dedos. El equilibrio en estrategias mixtas dice que hay que seleccionar "1 dedo" con el 50% de probabilidad y "2 dedos" con el otro 50%. Este equilibrio podría respetarse eligiendo alternadamente "1 dedo" y "2 dedos".

Obviamente que actuar de esta forma no vuelve a un jugador impredecible frente a otro. Por lo tanto, otra condición que hay que respetar es seleccionar cada acción en cada jugada según alguna regla que sea azarosa.

Una forma posible en el juego de los dedos sería, en cada jugada, tirar una moneda. Si sale cara, elegir "1 dedo", de lo contrario elegir "2 dedos".

Sólo actuando en forma azarosa se puede lograr ser impredecible utilizando estrategias mixtas.

6 Marco metodológico

6.1 Introducción

Como se vio anteriormente, uno de los objetivos primordiales de este trabajo es diseñar una metodología de fácil utilización que permita mejorar la eficiencia del testeo de programas mediante la alineación de los objetivos de los Desarrolladores con los de los Testers. Para ello, este capítulo presenta las bases del enfoque metodológico que se utilizará, el cual estará conformado por un conjunto de pasos que una empresa o institución que desarrolla software deberá seguir para generar su propio modelo basado en la Teoría de Juegos (y en estrategias mixtas en particular), y luego utilizarlo en forma apropiada.

Dado que la metodología se basa en la Teoría de Juegos, un punto importante es definir el juego que será la base del modelo. Entonces, lo primero que se debe hacer es precisar cuáles serán los 4 elementos que componen cualquier juego posible en Teoría de Juegos, llamados en conjunto las "Reglas del Juego". Estos elementos son: **Jugadores, Acciones, Pagos e Información** (Ver capítulo 5: "Teoría de Juegos")

6.1.1 Aclaraciones

En todo el diseño de la metodología se hace referencia a la ejecución de un programa con el objetivo de detectar errores, es decir, diferencias entre el comportamiento esperado y el observado. Esta actividad es realizada tanto por los Desarrolladores como por los Testers. Para que no haya confusiones, se utiliza la palabra "Testear" para la búsqueda por parte de los Testers, y "Controlar" o "Controlar Internamente" para la búsqueda por parte de los Desarrolladores.

6.2 Elementos del juego

6.2.1 Jugadores

La Teoría de Juegos se utiliza cuando 2 o más entidades (llamadas jugadores) toman decisiones dentro de un mismo contexto, el juego, y todos ellos son conscientes de que cada una de sus decisiones afecta a los otros jugadores.

Por lo tanto, lo primero que se debe hacer para diseñar un modelo basado en esta teoría es definir cuáles son esos jugadores cuyo conflicto le da sentido al juego.

En el caso del conflicto que se quiere analizar, los jugadores son:

- **Desarrollador:** se encarga de los procesos correspondientes a la programación de los sistemas. Sabe que los programas serán puestos a prueba en una cantidad de casos de test, pero generalmente trata de maximizar la cantidad de horas que dedica a la programación y, por lo tanto, minimizar la cantidad de horas dedicadas al control interno de los programas.
- **Tester:** Su objetivo primordial es detectar la mayor cantidad posible de errores en un programa. Para eso genera una cantidad de casos de test. Dado que testearlos todos es un proceso muy largo y costoso, desea lograr que los Desarrolladores realicen un control interno lo más exhaustivo posible de sus programas.

6.2.2 Acciones

Las acciones de cada jugador son las decisiones que éste puede tomar dentro del juego. Así se define para cada jugador i su "Conjunto de acciones" A_i , conformado por todas las acciones que puede tomar.

Para el caso del Desarrollador, ante cada caso de test, puede tomar la decisión de controlarlo internamente o no.

$A_{\text{Desarrollador}} = \{\text{"Controlar Internamente"}, \text{"No Controlar Internamente"}\}$

Asimismo, para el Tester existen también dos opciones distintas ante cada caso de test de cada módulo del programa que le llega del Desarrollador; puede testear o no testear.

$A_{\text{Tester}} = \{\text{"Testear"}, \text{"No Testear"}\}$

6.2.3 Pagos

Las funciones de pago se definen por separado, una para cada jugador. La función de pago de un jugador indica, para cada combinación de acciones de cada uno de los jugadores, la utilidad (pago) esperada para ese jugador luego de que cada uno de ellos eligió dicha acción (incluido él mismo).

Las utilidades deben representar los intereses y objetivos de cada uno de los jugadores dentro del juego. De esta forma, si a un jugador le conviene más una situación (combinación de acciones) que otra diferente dados sus objetivos, entonces el pago para ese jugador debe ser mayor para la primera situación que para la segunda.

A partir del repaso de los objetivos de Desarrolladores y Testers en un proyecto de desarrollo de software, se llega a la conclusión de que son muy diferentes unos de otros, no están alineados.

Justamente la metodología que se presenta intenta alinear dichos objetivos hacia un mismo punto; mejorar la calidad del software que se realiza.

Entonces, a partir de los objetivos de cada uno de los jugadores, se debe definir una Función de Pagos $\Pi(S_1, \dots, S_n)$ para cada uno de ellos.

Unidades de pago (U.P.)

Dado que las funciones de pagos son funciones, los pagos que da como resultado deben ser medidos todos en la misma unidad.

Se pueden utilizar distintos tipos de unidades de medición, siempre que correspondan a unidades de costo (o lo que es lo mismo, de satisfacción). En este caso se proponen 2 diferentes:

- **Horas de trabajo:** Así se podría afirmar que a un Desarrollador le cuesta 2 horas de trabajo el controlar una unidad de código, o que a un Tester le cuesta 5 horas de trabajo el retrabajo correspondiente a retestear un defecto de un programa que ya se encuentra en etapa de producción.
- **Dinero:** Utilizando esta unidad se podría afirmar que a un Desarrollador le cuesta 50 pesos el desarrollar una parte de un sistema, y 30 pesos el testeo interno de un caso de test.

Este es un punto conflictivo, ya que muchas veces se hace dificultoso medir ciertos conceptos utilizando algunas unidades.

Por ejemplo, no parece resultar muy fácil, por lo menos a simple vista, medir el beneficio que le depara a un Tester el hecho de sacar a producción un código sin errores, en Horas de Trabajo. Así se pueden dar muchos otros ejemplos.

Las dos unidades presentadas parecen ser equivalentes, en el sentido de que un valor en horas siempre puede ser pasado a dinero (realizando alguna transformación que tenga en cuenta por ejemplo el sueldo de dicho empleado) y viceversa.

El hecho de que las funciones de pago sean necesarias para los modelos obliga a elegir alguna unidad para utilizar, que podría llegar a ser diferente en cada uno de los ejemplos de empresas y proyectos en los que se desee aplicar la metodología, según las características de los mismos. En este caso se ha elegido la unidad "hora de trabajo" por considerarla más simple para desarrollar los conceptos.

Variables independientes del modelo

Para definir las funciones de Pago, se utilizará una serie de variables independientes. Esto quiere decir que son valores que se pueden definir, calcular o estimar antes de comenzar el proyecto. De esta forma estas variables funcionan como el Input de la metodología.

Las variables independientes constituyen una parte esencial del problema: si se les asigna un valor equivocado o estimado en forma incorrecta, el resultado del modelo aplicado a esos valores será también inexacto. Sin embargo, no siempre es fácil estimar los valores para un proyecto en particular, y tampoco es fácil corroborar que los valores sean lógicos y por lo tanto útiles. Algo muy importante que puede mejorar la estimación y definición de valores es la experiencia sistematizada de la empresa, o sea, la utilización de buenas y variadas métricas constantemente en los distintos proyectos encarados por la empresa. Sin duda esto puede mejorar notablemente el conocimiento que la empresa tiene sobre sí misma, y así mejorar la estimación de sus variables.

A continuación se presenta el conjunto de variables de entrada a utilizar.

Ct: Costo promedio de ejecutar un caso de Test por parte de un Tester.

Es la cantidad promedio de horas que utiliza un Tester en la ejecución de un caso de test. La estimación de esta variable se puede ir mejorando mediante la utilización de métricas referentes a los costos de ejecución de casos de test en los distintos proyectos.

Ci: Costo promedio para los Desarrolladores del control interno.

Es el costo, en promedio y en horas, que le genera al Desarrollador la ejecución de un caso de test buscando errores en el programa. Se pueden sacar buenas estadísticas al respecto en los distintos proyectos que encara un equipo de desarrollo.

Rd₁: Costo promedio, en horas, del retrabajo por parte del Desarrollador cuando se detecta un error en Testing.

Este costo se genera cuando un Tester descubre un error, y avisa al Desarrollador de su existencia. Éste debe hacer las modificaciones correspondientes para arreglarlo.

Rt₁: Costo promedio, en horas, del retrabajo por parte del Tester cuando se detecta un error en Testing.

Este costo se genera en las mismas situaciones que Rd₁. Luego de que el Desarrollador realiza el retrabajo para solucionar el problema, el Tester debe retestear el (o los) casos de test en cuestión una cantidad de veces hasta que el problema se resuelve en forma definitiva.

Rd₂: Costo promedio, en horas, del retrabajo por parte del Desarrollador cuando se detecta un error en Producción.

Este costo se genera cuando se descubre un error en la etapa de producción del sistema. El Desarrollador debe realizar las modificaciones necesarias sobre los programas que se encuentran en producción, y el Tester luego debe testear todos los casos de test que corresponden a la funcionalidad que contenía un error, quizás varias veces, hasta que el problema se soluciona.

Se puede pensar en muchos aspectos que podrían tenerse en cuenta al evaluar el valor de esta variable: ¿Cómo se valoriza en dinero el costo para un banco de una transferencia entre cuentas ejecutada en forma incorrecta, o el descontento de un cliente de una aerolínea porque su boleto de avión fue emitido hacia Hong Kong en vez de hacia Madrid?

Para el modelo que estamos definiendo, sólo se tendrán en cuenta los aspectos que estén relacionados en forma directa con el costo de arreglo de un error por parte del Desarrollador.

Existen distintas dificultades al arreglar errores cuando el o los programas se encuentran en etapa de producción:

- Dificultad de arreglar un defecto cuando el programa ya se encuentra en utilización por los usuarios reales.
- Dificultad de descubrir los defectos de un programa a partir de los errores que se encuentran en el mismo.
- Un error con el sistema en producción ya funcionando puede ser que conlleve una serie de efectos colaterales que impliquen la necesidad de realizar otros arreglos estructurales, como por ejemplo en estructuras de datos.
- Nuevos testeos necesarios a partir del arreglo a realizar.
- Limitaciones en el tiempo que el sistema puede estar sin funcionar, para su arreglo en caso de que sea necesario.

Existen múltiples estadísticas en la bibliografía que indican la relación entre el costo de arreglar un error en forma temprana con respecto al de arreglar el mismo error cuando se encuentra en Producción. En la figura 6.1 se muestra un gráfico sobre este tema realizado a partir de [KRA/99]

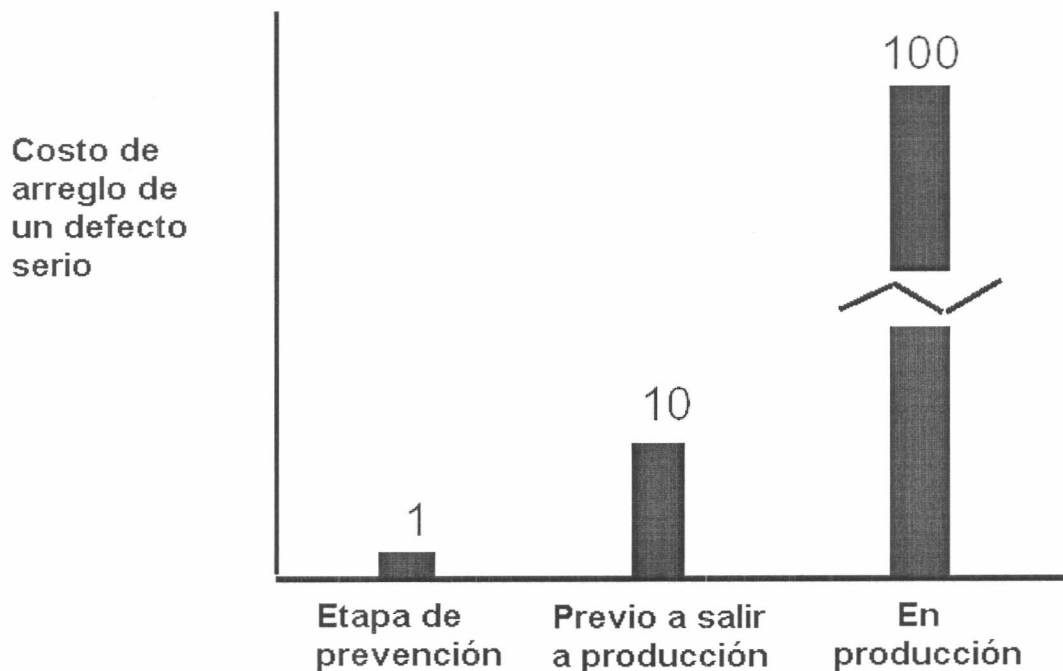


Figura 6.1. Costo de arreglo de defectos

En el gráfico se muestra que el costo de arreglar un defecto en la etapa desarrollo puede llegar a ser 10 veces menor al de arreglarlo cuando el producto ya se encuentra en producción.

De cualquier forma, para estimar este dato en forma más acertada, es útil tener estadísticas previas sobre costos de arreglo de errores y sobre todo métricas constantes sobre el avance de los proyectos. Estas métricas pueden ser sacadas tanto de la empresa como del grupo de trabajo, si es que ya trabajó en conjunto en algún proyecto anterior.

Rt₂: Costo promedio del retrabajo por parte del Tester cuando se detecta un error en Producción.

Este costo se genera cuando se descubre un error en la etapa de producción del sistema. Luego de que el Desarrollador realiza las modificaciones necesarias sobre los programas que se encuentran en producción, el Tester debe ejecutar todos los casos de test que corresponden a dicha utilidad que contenía un error, quizás varias veces, hasta que el problema se soluciona. El hecho de que el error es detectado en producción reviste un grado de dificultad aún mayor para el Tester, por motivos similares a los descriptos en Rd₂.

P: Esperanza de error en un Caso de Test sin ningún control.

El testing es una herramienta muy útil cuando es crítico generar productos que no contengan muchos de defectos. Cuando no se realiza un testeo de los programas desarrollados, existe una probabilidad de que el programa contenga defectos que luego cueste una cantidad de esfuerzo importante solucionar. Esta variable representa esa probabilidad. En otras palabras, es la probabilidad de que un caso de test cuyo código fue desarrollado por un programador en particular (o un grupo de programadores), falle cuando no se ejecutó en ningún momento anterior a salir a producción. Este dato puede ser difícil de estimar en muchos casos, y al principio se puede llegar a simplificar

con algún valor como 0,5 por ejemplo. Sin embargo, con un poco más de experiencia se puede sacar de los datos estadísticos históricos y métricas de los proyectos realizados por la compañía, de los proyectos realizados por el grupo de Desarrolladores o de los proyectos históricos en los que participaron los Desarrolladores en cuestión. El valor de P a asignar a un grupo de desarrollo depende de distintas variables, como la experiencia y calidad de cada uno de los integrantes, y la dificultad de las tareas que se asigna a cada uno de ellos.

Unidades de testeo (U.T.)

La metodología que se presenta define, para un grupo de trabajo dado y un proyecto dado, un juego. El equilibrio de ese juego le dirá a cada jugador cuál es la acción que debe seleccionar (ejecutar un caso o no hacerlo). Sin embargo, se debe definir cuál es la unidad que se utilizará para tomar como división y decidir para cada una de las unidades generadas si se testea o no.

Existen distintas unidades que se pueden utilizar, y cada una de ellas tiene sus ventajas y desventajas. A continuación se presentan dos posibles:

- **Módulos de código:** Se divide el código en módulos de una cantidad aproximada de líneas. Para cada uno de los módulos se decide si testearlo o no. La dificultad de utilizar esta unidad reside en que se hace difícil saber cuando un módulo se puede considerar como testeado completamente. Es fácil en caso de que se ejecuten todos los casos de test correspondientes. Pero ¿qué pasa si se ejecutan algunos y otros no? o ¿qué pasa si algunas pruebas del módulo fallan y otras no?

En la figura 6.2 se muestra un diagrama de un sistema dividido en módulos de código.



Figura 6.2. Testeo por módulos de código

- **Casos de test:** En esta opción primero se realiza el diseño del programa dividiéndolo en módulos separados de código. Luego se entrega el diseño a los Testers, a partir del cual generarán los casos de test correspondientes a cada módulo.

En la figura 6.3 se muestra un diagrama de un sistema dividido por casos de test.

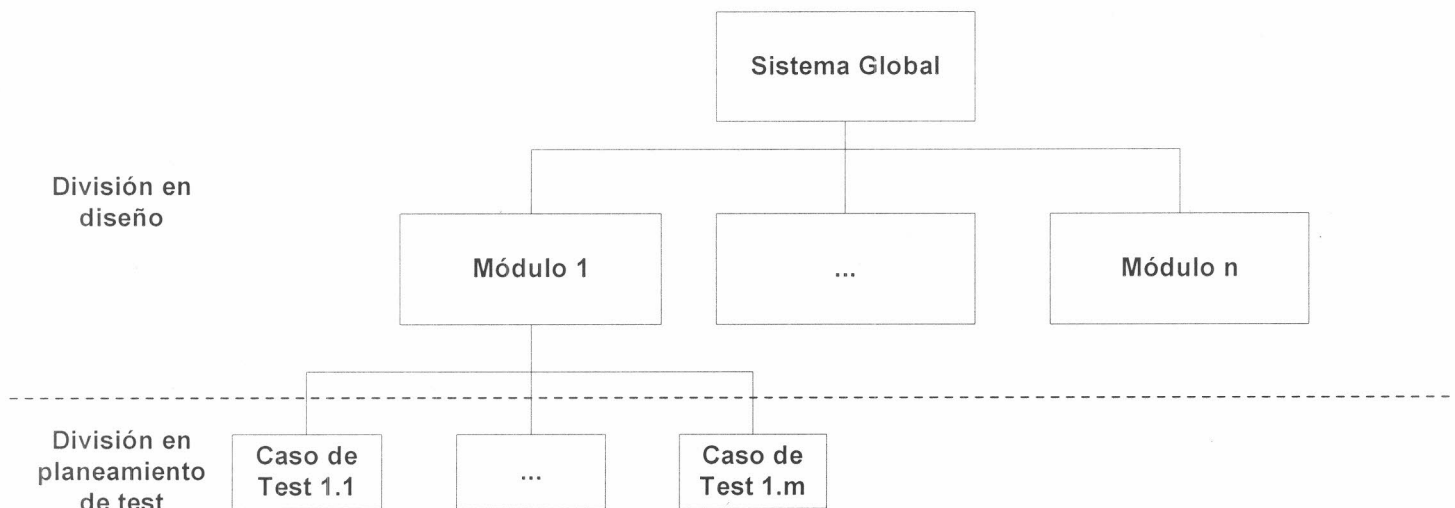


Figura 6.3. Testeo por casos de test

La segunda opción es la que se eligió como más adecuada para esta metodología. Sin embargo, ésta tiene la limitación de que requiere que la etapa de diseño del sistema sea anterior a la etapa de planeamiento de los casos de test. Esto puede o no ser un problema dependiendo del nivel de test que se está utilizando. Como la metodología propone el uso de test de unidad, esto no resulta un problema. Por lo tanto, la división por casos de test es la forma más adecuada para aplicar la metodología.

Definición de funciones de pagos

Ya presentadas las variables independientes a utilizar, se puede definir finalmente las funciones de pago correspondientes al juego. Estas son:

- $\Pi_{\text{Desarrollador}}(S_{\text{Desarrollador}}, S_{\text{Tester}})$ = Utilidad esperada para el Desarrollador
- $\Pi_{\text{Tester}}(S_{\text{Desarrollador}}, S_{\text{Tester}})$ = Utilidad esperada para el Tester

En la sección 6.3 se definirán efectivamente las funciones de pagos que se utilizarán en la metodología que se presenta, junto con distintas opciones que modelan diferentes alternativas o escenarios.

6.2.4 Información

La información se modela a partir del concepto de "Conjuntos de información". El Conjunto de información de un jugador está formado por el conocimiento que posee sobre distintas variables en un momento particular del juego.

La información ocupa un lugar central en los juegos no simultáneos, o secuenciales. En estos, el segundo jugador del turno cuenta con la información de cuál es la jugada que realizó el primer jugador antes de tomar su propia decisión. Al contrario, en los juegos simultáneos la información no posee un papel demasiado importante.

En el caso del juego que se está modelando en este momento, si bien existe una secuencialidad temporal en el sentido de que en un primer momento los Desarrolladores ejecutan internamente los casos de test y en un segundo momento posterior los Testers hacen lo propio, estos últimos no conocen cuáles fueron los casos de test que ejecutaron los Desarrolladores antes de hacer su trabajo.

Por esta razón el juego no es secuencial sino simultáneo, por lo que la información no es un factor importante para el enfoque metodológico que se está presentando.

6.3 Un primer acercamiento: modelo idealista

En este primer acercamiento se definirán las funciones de pago, y luego las matrices de pago que terminan de completar el juego que se utiliza como modelo. Asimismo se explicarán todas las decisiones tomadas con respecto al armado de las mismas. Con la matriz de pagos ya generada, se resolverá el equilibrio y se verá por qué no refleja fielmente ciertos aspectos del proceso de desarrollo de software. En la sección 6.4 se harán algunos cambios al modelo que le sumarán más realismo.

6.3.1 Supuestos generales

- Los Testers son los encargados de generar los casos de test luego de recibir el análisis de requerimientos y el diseño detallado por parte del equipo de Análisis y diseño de la empresa. Luego de generar los casos se los envían al equipo de desarrollo. Por lo tanto el equipo de testing y el de desarrollo utilizan el mismo conjunto de casos de test para buscar errores en los programas.
- Tanto los Desarrolladores como los Testers son infalibles al ejecutar un caso de test. Esto significa que si un Desarrollador ejecuta un caso lo hará correctamente. Si decide que el programa se ejecuta en forma correcta para ese caso, entonces no podrá suceder que el Tester ejecute el mismo caso y decida que el programa se comporta en forma incorrecta. De forma similar, si el Tester ejecuta un caso de test y no falla, entonces este mismo caso de test no fallará en producción.

6.3.2 Funciones de pago

Para diseñar las funciones de pago se debe definir el beneficio que recibirá cada uno de los jugadores para cada una de las situaciones (o casos) que pueden suceder. Como se explicó anteriormente, existen dos jugadores, y cada uno de ellos posee dos posibles acciones. Por lo tanto, existen cuatro casos diferentes.

En este trabajo se utilizará indistintamente la palabra "pago" y "costo", ya que un costo es un pago pero negativo.

Caso 1: El Desarrollador elige la acción "No Controlar" y el Tester la acción "Testear". (No Controlar, Testear).

Desarrollador: El Desarrollador no ejecuta el caso. Por lo tanto hay una probabilidad (P) de que ese caso falle en el testeo. Si esto sucede el Desarrollador tendrá un costo de retrabajo (Rd_1).

$$\Pi_{\text{Desarrollador}} = -Rd_1 \cdot P$$

Tester: El Tester tiene dos costos diferentes: por un lado un Costo de Testing del caso (Ct) y por el otro el Costo promedio de retrabajo (Rt_1) asociado al tiempo de retesteo una vez que se reparó un error, en caso de que se encuentre uno. Esto sucede con probabilidad P.

$$\Pi_{\text{Tester}} = -Ct - Rt_1 \cdot P$$

Caso 2: (Controlar, Testear)

Desarrollador: El Desarrollador ejecuta el caso internamente, con su consiguiente costo (Ci). Así se asegura que el caso no va a fallar en testing ni en producción, y por lo tanto no tendrá ningún otro costo.

$$\Pi_{\text{Desarrollador}} = -C_i$$

Tester: El Tester ejecuta el caso y esto le genera un costo (Ct). Dado que el Desarrollador ejecutó el caso previamente, éste no fallará y por lo tanto el Tester no tendrá ningún otro costo adicional.

$$\Pi_{\text{Tester}} = -C_t$$

Caso 3: (Controlar, No testear)

Desarrollador: El Desarrollador ejecuta el caso internamente, con su consiguiente costo (Ci). Así se asegura que el caso no va a fallar en testing ni en producción, y por lo tanto no tendrá ningún otro costo.

$$\Pi_{\text{Desarrollador}} = -C_i$$

Tester: Dado que el Desarrollador ya controló internamente el caso, es seguro que éste no fallará cuando el sistema llegue a producción. Por lo tanto el Tester no tendrá costo alguno de testeo.

$$\Pi_{\text{Tester}} = 0$$

Caso 4: (No controlar, No testear)

Ninguno de los dos ejecuta el caso de Test. Esto da como resultado un caso que se pone en producción sin ejecutar. Hay una probabilidad (P) de que ese caso falle en producción.

Desarrollador: El Desarrollador tiene el costo de retrabajo con el error encontrado en producción (Rd₂). Y esto pasa con probabilidad P.

$$\Pi_{\text{Desarrollador}} = -R_{d_2} \cdot P$$

Tester: El Tester también tiene el costo de retrabajo con el error encontrado en producción (Rt₂). Y esto sucede con la misma probabilidad (P).

$$\Pi_{\text{Tester}} = -R_{t_2} \cdot P$$

6.3.3 Matriz de pagos

A partir de los elementos definidos anteriormente se puede construir la matriz de pagos, que es un resumen de todas las características del juego.

		Desarrollador	
		No controlar	Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

6.3.4 Supuestos sobre las variables

Existen ciertos supuestos sobre las variables presentadas. Estos supuestos responden en su mayoría al sentido común:

- 1) $Rd_2 > Rd_1$: El costo de retrabajo para el Desarrollador por errores encontrados en producción es siempre mayor que el costo de retrabajo en caso de que el error se encuentre en testing. Hay muchas razones que hacen que esto sea verdadero, entre otras la regeneración de ambientes de prueba, tiempos de desarrollo acotados, Etc.
- 2) **P es una probabilidad**: Esto implica que $0 \leq P \leq 1$
- 3) **Todos los valores de las variables deben ser positivos.**

6.3.5 Condiciones sobre las variables

Así como existen ciertos supuestos sobre las variables que responden a situaciones que se dan siempre o simplificaciones del modelo que no afectan el realismo del mismo, también existen ciertas condiciones que se pueden cumplir o no, dependiendo de cada situación y que generan casos independientes que se deben analizar por separado.

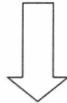
- a) $Ct+Rt_1.P \leq Rt_2.P$: Esta condición indica que para un Tester es menos costoso (o a lo sumo igualmente costoso) testear un caso y devolverlo al Desarrollador para su arreglo en caso de que contenga un error ($Ct+Rt_1.P$) que correr con los costos de dejar pasar el caso y después testearlo cuando se encontró el error en producción ($Rt_2.P$)
- b) $Rd_1.P \leq Ci$: Esta condición indica que para un Desarrollador la esperanza de costo de arreglar un defecto cuando el error se encuentra en testing ($Rd_1.P$) es menor o igual que el costo de ejecutar el caso de test correspondiente a ese error (Ci).
- c) $Rd_2.P \leq Ci$: Esta condición indica que para un Desarrollador la esperanza de costo de arreglar un defecto cuando el error se encuentra en producción ($Rd_2.P$) es menor o igual que el costo de ejecutar el caso de test correspondiente a ese error (Ci).

6.3.6 Notación gráfica de supuestos y condiciones

Algunos de los supuestos y condiciones definidos se pueden representar mediante flechas sobre la matriz de pagos que facilitan la comprensión de las distintas

situaciones. Una flecha entre un pago y otro significa que el jugador correspondiente elige el pago apuntado por la flecha antes que el otro.

El siguiente es un ejemplo de una matriz de pagos enriquecida con una flecha correspondiente a la condición $C_t > 0$.

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-C_t - R_{t1}.P$; $-R_{d1}.P$)	2. ($-C_t$; $-C_i$)
	No testear	4. ($-R_{t2}.P$; $-R_{d2}.P$)	3.  (0 ; $-C_i$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

6.3.7 Equilibrios

Como se dijo anteriormente, existen tres condiciones sobre las variables que se pueden cumplir o no según los valores que se le asignen a las diferentes variables. Esto da como resultado una serie de casos diferentes generados por distintas combinaciones de que se cumplan o no cada una de las condiciones. El equilibrio de cada uno de los casos se calcula en forma diferente, por lo que luego de estimar los valores de todas las variables se debe chequear en cuál de los casos entran los valores estimados, y calcular el equilibrio en la forma que corresponda.

En la figura 6.4 se muestra un árbol de condiciones con las diferentes combinaciones de valores de las variables y los distintos casos que se generan.

Es importante notar que la división de los casos define clases de equivalencia. Esto quiere decir que los casos no se superponen entre si, y además que cualquier combinación de valores de las variables entra en alguno de los casos definidos.

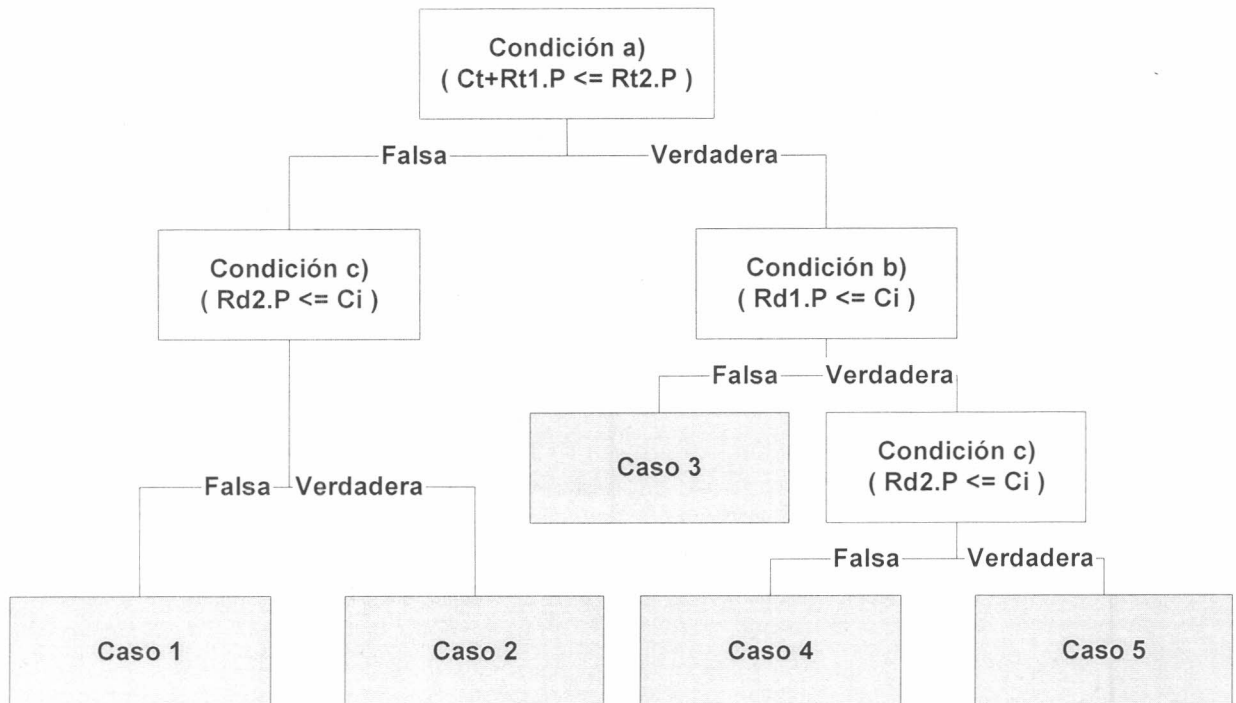


Figura 6.4. Árbol de condiciones.

A continuación se resuelven los equilibrios de cada uno de los 5 casos definidos y de esta forma se cubren todas las posibilidades.

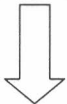
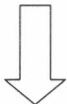
Caso 1

En este caso las condiciones a) y c) son falsas.

Como a) es falsa, entonces:

- $Ct + Rt_1.P > Rt_2.P$

Se puede agregar la condición a) a la matriz gráficamente de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4.  ($-Rt_2.P$; $-Rd_2.P$)	3.  (0 ; $-Ci$)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Se puede ver a qué condición corresponde cada una de las flechas:

- **De 2. a 3.:** Corresponde a Supuesto 3), todos los valores de las variables deben ser positivos.
- **De 1. a 4.:** Corresponde a Condición a) falsa, o sea $Ct + Rt_1.P > Rt_2.P$

Observando la matriz y la dirección de las flechas se puede ver que existe una **estrategia dominante:** "No Testear", y por lo tanto una dominada: "Testear". Esto es así porque el Tester elige siempre la estrategia "No Testear", sin importar la elección del Desarrollador.

Cuando se encuentra una estrategia dominada lo que se debe hacer es quitarla del rango de estrategias posibles; esto es porque, al no elegirse nunca, ya no tiene sentido.

La siguiente matriz es la que queda cuando se quita la estrategia dominada "Testear".

		Desarrollador	
		No Controlar	Controlar
Tester	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

En este momento hay dos posibilidades, dependiendo de la condición c) que se resuelven de forma similar, siempre generando un equilibrio en estrategias puras.

Dado que la condición c) es falsa, entonces:

- **$Rd_2.P > Ci$**

Esta condición genera la siguiente matriz con una nueva estrategia dominante: "Controlar".

		Desarrollador	
		No Controlar	Controlar
Tester	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

Realizando el mismo procedimiento que en el paso anterior, se quita la estrategia dominada, quedando la siguiente matriz:

		Desarrollador
		Controlar
Tester	No Testear	3. (0 ; -Ci)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

De esta forma, existe un equilibrio de Nash en estrategias puras, que es la combinación de acciones (No Testear, Controlar), que genera a los jugadores un pago de (0 ; -Ci).

Este equilibrio significa que, con el modelo generado, la elección del Desarrollador es siempre la de ejecutar internamente todos los casos de test y la del Tester es no ejecutar ningún caso de test.

Caso 2

El Caso 2 es análogo al 1, con la diferencia que la condición c) es verdadera. Esto implica que se cumple:

- $Rd_2.P \leq Ci$

Por lo tanto se genera la siguiente matriz:

		Desarrollador	
		No Controlar	Controlar
Tester	No testear	4. (-Rt ₂ .P ; -Rd ₂ .P)	3. (0 ; -Ci)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Luego se descarta la estrategia dominada "Controlar" quedando la matriz:

		Desarrollador
		No Controlar
Tester	No Testear	4. (-Rt ₂ .P ; -Rd ₂ .P)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Por lo tanto existe un equilibrio de Nash en estrategias puras en (No Testear, No Controlar), que genera a los jugadores un pago de (-Rt₂.P ; -Rd₂.P).

Caso 3

En este caso la condición a) es verdadera pero la b) es falsa:

- $Ct + Rt_1.P \leq Rt_2.P$
- $Rd_1.P > Ci$

Se pueden agregar gráficamente las condiciones a la matriz de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Se puede ver a qué condición corresponde cada una de las flechas:

- **De 2. a 3.:** Corresponde a Supuesto 3), todos los valores de las variables deben ser positivos.
- **De 1. a 2.:** Corresponde a Condición b) falsa, o sea $Rd_1.P > Ci$
- **De 4. a 3.:** Corresponde a Condición b) falsa ($Rd_1.P > Ci$) y Supuesto 1) ($Rd_2 > Rd_1$), ya que los dos implican $Rd_2.P > Ci$
- **De 4. a 1.:** Corresponde a Condición a) verdadera, o sea $Ct + Rt_1.P \leq Rt_2.P$

Observando la matriz y la dirección de las flechas se puede ver que existe una **estrategia dominante**: "Controlar", y por lo tanto una dominada: "No Controlar". Esto es así porque el Desarrollador elige siempre la estrategia "Controlar", sin importar la elección del Tester.

La siguiente matriz es la que queda cuando se quita la estrategia dominada "No Controlar".

		Desarrollador
		Controlar
Tester	Testear	2. ($-Ct$; $-Ci$)
	No testear	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Luego de quitar la estrategia "No Controlar" se genera una nueva estrategia dominada, esta vez del Tester: "Testear". Realizando el mismo procedimiento que en la estrategia dominada anterior, se quita, quedando la siguiente matriz:

		Desarrollador
		Controlar
Tester	No Testear	1. (0 ; -Ci)

Pagos, medidos en horas, para: (Tester; Desarrollador)

De esta forma, existe un equilibrio de Nash en estrategias puras en (No Testear, Controlar), que genera a los jugadores un pago de (0 ; -Ci). Este equilibrio significa que, con el modelo generado, la elección del Desarrollador es siempre la de ejecutar internamente todos los casos de test y la del Tester es no ejecutar ningún caso.

Caso 4

En este caso son verdaderas tanto la condición a) como la b), pero c) es falsa, o sea que:

- $Ct + Rt_1.P \leq Rt_2.P$
- $Rd_1.P \leq Ci < Rd_2.P$

La matriz correspondiente es la siguiente:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. (-Ct-Rt ₁ .P ; -Rd ₁ .P)	2. (-Ct ; -Ci)
	No testear	4. (-Rt ₂ .P ; -Rd ₂ .P)	3. (0 ; -Ci)

Pagos, medidos en horas, para: (Tester; Desarrollador)

Se puede ver a qué condición corresponde cada una de las flechas:

- **De 2. a 3.:** Corresponde a Supuesto 3), todos los valores de las variables deben ser positivos.
- **De 2. a 1.:** Corresponde a Condición b) verdadera, o sea $Rd_1.P \leq Ci$
- **De 4. a 3.:** Corresponde a Condición c) falsa ($Rd_2.P > Ci$)
- **De 4. a 1.:** Corresponde a Condición a) verdadera, o sea $Ct + Rt_1.P \leq Rt_2.P$

Observando la matriz y la dirección de las flechas se puede ver que no existen estrategias dominantes. Por lo tanto se deben buscar equilibrios en estrategias puras. Para que exista un equilibrio de Nash en estrategias puras debe existir una combinación de acciones de los jugadores que haga que a todos ellos les convenga quedarse en su situación actual antes que pasarse a una diferente. Esto quiere decir que el pago que reciben en la combinación actual es mayor al que reciben si se cambian.

En este caso existen dos equilibrios en estrategias puras: (Testear, No Controlar) y (No Testear, Controlar)

Por un lado, (Testear, No Controlar) es un equilibrio porque ni al Desarrollador le conviene pasarse a Controlar ni al Tester le conviene pasarse a No Testear.

Por otro lado (No Testear, Controlar) también es un equilibrio porque en caso de encontrarse en esa situación a ninguno de los dos jugadores le conviene pasarse a la otra acción.

Asimismo, se puede demostrar que los juegos de 2x2 que tienen dos equilibrios en estrategias puras, tienen también uno en estrategias mixtas.

Por lo tanto, este caso también posee un equilibrio en estrategias mixtas. El mismo se calculará en secciones posteriores

Caso 5

En este caso la condiciones a), b) y c) son verdaderas. O sea que

- $Ct + Rt_1.P \leq Rt_2.P$
- $Rd_1.P \leq Rd_2.P \leq Ci$

Se pueden agregar gráficamente las condiciones a la matriz de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

Se puede ver a qué condición corresponde cada una de las flechas:

- **De 2. a 3.:** Corresponde a Supuesto 3), todos los valores de las variables deben ser positivos.
- **De 2. a 1.:** Corresponde a Condición b) verdadera, o sea $Rd_1.P \leq Ci$
- **De 3. a 4.:** Corresponde a Condición c) verdadera, o sea $Rd_2.P \leq Ci$
- **De 4. a 1.:** Corresponde a Condición a) verdadera, o sea $Ct + Rt_1.P \leq Rt_2.P$

Observando la matriz y la dirección de las flechas se puede ver que existe una **estrategia dominante**: "No Controlar", y por lo tanto una dominada: "Controlar". Esto es así porque el Desarrollador elige siempre la estrategia "No Controlar", sin importar la elección del Tester.

La siguiente matriz es la que queda cuando se quita la estrategia dominada "Controlar".

		Desarrollador
		No Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)
	No testear	4.  ($-Rt_2.P$; $-Rd_2.P$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

Luego de quitar la estrategia "Controlar" se genera una nueva estrategia dominada, esta vez del Tester: "No Testear". Realizando el mismo procedimiento que en la estrategia dominada anterior, se quita, quedando la siguiente matriz:

		Desarrollador
		No Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

De esta forma, existe un equilibrio de Nash en estrategias puras en (Testear, No Controlar), que genera a los jugadores un pago de ($-Ct-Rt_1.P$; $-Rd_1.P$). Este equilibrio significa que, con el modelo generado, la elección del Desarrollador sería siempre la de no ejecutar internamente todos los casos de test y la elección del Tester sería en todos los casos ejecutar los casos de test.

6.3.8 Resumen

En este capítulo se generó la matriz de pagos que sirve de modelo para el marco metodológico. Luego se definieron ciertos supuestos y condiciones sobre las variables que la componen. Los supuestos se cumplen siempre y permiten que el modelo sea razonable, mientras que las condiciones pueden cumplirse o no según los valores que se estimen para las diferentes variables definidas. Las condiciones son:

a) $Ct + Rt_1.P \leq Rt_2.P$

b) $Rd_1.P \leq Ci$

c) $Rd_2.P \leq Ci$

A partir de estas condiciones se definieron 5 casos que cubren todo el espectro de valores de las variables y se resolvió cada uno de ellos en forma diferente llegando a los siguientes resultados:

Caso	Condiciones (Falsa/Verd./ Indiferente)			Situación	Equilibrio
	a	b	c		
1	F	I	F	• $(Ct + Rt_1.P > Rt_2.P)$ Para el Tester es menor la esperanza de costo de dejar pasar el caso y después solucionar las consecuencias de los errores que se encuentren en producción, que el costo de testear el caso y devolverlo al Desarrollador en caso de que contenga un error. • $(Rd_2.P > Ci)$ Para el Desarrollador es menor el costo de ejecutar un caso que la esperanza de costo de dejar que el caso llegue a producción con posibles errores y solucionarlos en ese momento.	En estrategias Puras en: (No Testear;Controlar)
2	F	I	V	• $(Ct + Rt_1.P > Rt_2.P)$ Para el Tester, Ídem Caso 1. • $(Rd_2.P \leq Ci)$ Para el Desarrollador es menor la esperanza de costo de dejar que el caso llegue a producción con posibles errores y solucionarlos en ese momento que el costo de ejecutar un caso.	En estrategias Puras en: (No Testear;No Controlar)
3	V	F	I	• $(Ct + Rt_1.P \leq Rt_2.P)$ Para el Tester es menor el costo de testear el caso y devolverlo al Desarrollador en caso de que contenga un error que la esperanza de costo de dejar pasar el caso y después solucionar las consecuencias de los errores que se encuentren en producción. • $(Rd_1.P > Ci)$ Para el Desarrollador es menor el costo de ejecutar un caso que la esperanza de costo de dejar que el caso llegue a testing con posibles errores, que el Tester se los devuelva y solucionarlos en ese momento.	En estrategias Puras en: (No Testear;Controlar)
4	V	V	F	• $(Ct + Rt_1.P \leq Rt_2.P)$ Para el Tester, Ídem Caso 3. • $(Rd_1.P \leq Ci < Rd_2.P)$ Para el Desarrollador el costo de ejecutar un caso es menor que la esperanza de costo de dejar que el caso llegue a producción con posibles errores, pero mayor que la esperanza de costo de dejar que el caso llegue a testing con posibles errores.	Dos equilibrios en estrategias Puras en: (No Testear;Controlar) y (Testear;No Controlar) , y un equilibrio en estrategias mixtas.
5	V	V	V	• $(Ct + Rt_1.P \leq Rt_2.P)$ Para el Tester, Ídem Caso 3. • $(Rd_2.P \leq Ci)$ Para el Desarrollador el costo de ejecutar un caso no sólo es mayor que la esperanza de costo de dejar que el caso llegue a Testing con posibles errores, sino que también mayor que la esperanza de costo de dejar que el caso llegue a producción con posibles errores.	En estrategias Puras en: (Testear;No Controlar)

6.3.9 El problema de las estrategias puras

Un equilibrio en estrategias puras implica que, para una combinación de acciones en particular, a ninguno de los dos jugadores le conviene cambiar de acción, y por lo tanto los dos jugadores se quedan en la acción que conforma el equilibrio.

En el caso de Desarrolladores y Testers, esto implica que sucede alguna de las siguientes situaciones:

- El Tester ejecuta todos los casos y el Desarrollador ninguno.
- El Desarrollador ejecuta todos los casos y el Tester ninguno.
- Tanto el Desarrollador como el Tester no ejecutan ningún caso.

En la realidad de las empresas, estas situaciones son muy raras, sino imposibles. En general en los proyectos de software se dan situaciones en las que los Desarrolladores ejecutan una cantidad de los casos definidos (no todos) y los Testers ejecutan otra cantidad (no todos tampoco). Estas situaciones, en las que los jugadores eligen sus acciones con una cierta probabilidad corresponden a los equilibrios en estrategias mixtas.

Como se vio en el resumen de la sección anterior, de los 5 casos definidos que cubren todo el espectro de valores de las variables, todos ellos tienen equilibrios en estrategias puras. De hecho, salvo el 4, todos los demás casos sólo tienen equilibrios en estrategias puras.

Ahora, si se toma el caso 3, por ejemplo, éste dice que:

- Para el Desarrollador es menor el costo de ejecutar un caso que la esperanza de costo de dejar que el caso llegue a testing con posibles errores, que el Tester se los devuelva y solucionarlos en ese momento.
- Para el Tester es menor el costo de testear el caso y devolverlo al Desarrollador en caso de que contenga un error que la esperanza de costo de dejar pasar el caso y después solucionar las consecuencias de los errores que se encuentren en producción.

Este caso se da muy comunmente en un proyecto de desarrollo de software. Sin embargo, el modelo da como resultado un equilibrio en estrategias puras, lo que no suele darse en la realidad. Esto muestra que, para una gran cantidad de casos, el modelo generado es poco realista, modela situaciones que no se dan en la realidad de las empresas.

Esta situación hace que sea necesario realizar ciertos cambios en el modelo para que éste sea más realista y se puedan sacar mejores conclusiones a partir de él.

Estos cambios se realizarán a continuación en la sección 6.4.

6.4 Hacia un modelo más realista

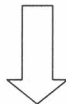
En la sección 6.3 se vio que el modelo presentado no se ajusta a la realidad de las empresas. En esta se hacen algunos cambios que permiten tener un modelo más acorde al trabajo real en las organizaciones.

6.4.1 Causas del desajuste

En la sección 6.3.9 se demostró que la matriz de pagos diseñada genera equilibrios en estrategias puras en muchos de los casos importantes generados. También se mostró que esto genera un modelo en el que muchas de las situaciones son poco reales. Para generar situaciones más reales se debe adaptar el modelo a un juego de monitoreo. En estos, el supervisado (en este caso el Desarrollador) desea "portarse bien" (en este caso Controlar) cuando el Supervisor (en este caso el Testear) supervisa (Testea), mientras que desea "portarse mal" (o sea no controlar) cuando el supervisor no supervisa (no testea). De este modo, el juego contiene un equilibrio en estrategias mixtas.

Para esto se debe realizar alguna modificación a la matriz. La cuestión más importante es detectar los factores que hacen que el modelo esté lejos de la realidad, y definir las modificaciones que lo acercan.

El modelo generado anteriormente era el siguiente:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3.  (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

La flecha que está fija de 2 a 3 está por el hecho de que $Ct > 0$.

Para que este juego se convirtiera en uno de monitoreo, y por lo tanto más real, se debería formar un "circuito" conformado por las flechas que denotan las condiciones, de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Ahora, las flechas que van de 4 a 1 y de 1 a 2 son posibles si se toman las siguientes condiciones:

- Condición a) Verdadera ($Ct+Rt_1.P \leq Rt_2.P$). Esto significa que para el Tester es más conveniente testear un caso y devolverlo al Desarrollador en caso de que posea errores, que dejarlo pasar y pagar las consecuencias de que aparezca un error en producción.
- Condición b) Falsa. ($Rd_1.P > Ci$). Esto significa que para el Desarrollador es más conveniente controlar un caso que dejarlo pasar y realizar el retrabajo necesario en caso de que el Tester encuentre un error.

El problema es que si la condición b) es falsa ($Ci < Rd_1.P$) y teniendo en cuenta el supuesto 1:

- Supuesto 1: el costo de retrabajo del Desarrollador con errores en testing es menor que el costo de retrabajo con errores en producción ($Rd_1.P < Rd_2.P$)

entonces necesariamente el costo de control interno debe ser menor que la esperanza de costo de retrabajo con error en producción ($Ci < Rd_2.P$), lo que genera una flecha que va de 3 a 4, y por lo tanto un equilibrio en estrategias puras de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. (-Ct-Rt ₁ .P ; -Rd ₁ .P)	2. (-Ct ; -Ci)
	No testear	4. (-Rt ₂ .P ; -Rd ₂ .P)	3. (0 ; -Ci)

Pagos, medidos en horas, para: (Tester; Desarrollador)

Para que exista un equilibrio en estrategias mixtas, se debe dar la condición opuesta, o sea que en caso de que el Tester elija "No Testear", entonces el Desarrollador elija "No Controlar". Esto implica que el costo del Desarrollador está incorrectamente definido en el casillero (Controlar; No Testear) o lo está en el casillero (No Controlar; No Testear). El pago en (Controlar; No Testear) parece correcto, ya que, si el Desarrollador controla, entonces tiene el costo de controlar (Ci) y se asegura que no se encuentren errores, y por lo tanto que no tendrá ningún costo más. Por lo tanto, el pago que está incorrectamente definido es el del Desarrollador en el casillero (No Controlar; No Testear).

Tan racionales no...

El factor por el cual se explica el cambio que hay que hacer sobre el pago al Desarrollador en la situación (No Controlar; No Testear) es el siguiente: Los Desarrolladores, en el momento de ejecutar los casos, no siempre tienen en cuenta en su total magnitud la esperanza de costo de arreglar los errores cuando se encuentran en producción. Esto se da por una serie de factores como ser:

- Confianza de que los Testers encontrarán todos sus errores.
- Subestimación de los costos de arreglo de los errores en producción.
- Subestimación de su tasa de error como programadores.
- Privilegio de tareas urgentes que poseen con respecto a la tarea de ejecución interna de casos de test.
- Factores externos, como presiones que reciben de la empresa para producir resultados.

Estos factores hacen pensar que el Desarrollador, cuando decide si testea o no un caso de test, no es tan racional. Si lo fuera, ante cada disyuntiva de ejecutar un caso o no, llegaría a la conclusión de que es menos costoso ejecutarlo, y por lo tanto ejecutaría todos los casos disponibles.

Se podría esperar que si los desarrolladores y los testers participan en muchos proyectos de desarrollo de software, esta irracionalidad generada por errores de percepción del desarrollador se debería reducir. De esta forma, el modelo generado a

continuación solo sería válido en un corto y mediano plazo, mientras que en un largo plazo se esperaría un modelo más parecido al utilizado en la sección anterior. Es importante notar que a partir de ahora existirán dos matrices de pagos diferentes:

- **Matriz de Pagos real de la empresa:** es la matriz que se viene utilizando hasta este momento y muestra los pagos (o costos) reales de la empresa a partir de los valores que se les asigne a las variables del modelo. Esta matriz no servirá para el cálculo de equilibrios (por las razones vistas anteriormente) pero sí se deberá utilizar para el cálculo de la fórmula de costos reales de la empresa cuando los equilibrios ya se encuentren calculados.
- **Matriz de Pagos para toma de decisiones:** es la matriz de pagos real de la empresa con las modificaciones que se realizarán a continuación. Esta matriz es la que tienen en cuenta los Desarrolladores y Testers a la hora de definir su comportamiento, y por lo tanto es la que se debe utilizar para el cálculo de equilibrios. Sin embargo, esta matriz no sirve para calcular los costos reales de la empresa a partir del equilibrio calculado.

Resumiendo, cuando los jugadores deban tomar una decisión se basarán en la matriz de toma de decisiones, pero cuando se desee calcular los costos reales de la empresa se deberá utilizar la fórmula de costos calculada a partir de la matriz de pagos real de la empresa.

6.4.2 Nuevas funciones de pago

Para generar la matriz de pagos para toma de decisiones, se debe reflejar los factores de subestimación de errores por parte del Desarrollador descriptos en la sección anterior. Entonces, si CR es el costo real de arreglo de un error cuando se detecta en producción, y CE es el costo de arreglo de un error estimado por el Desarrollador,

entonces se puede definir una nueva variable M como $\frac{CE}{CR}$, es decir, la tasa de

subestimación de costos por parte de los Desarrolladores.

Para que se comprenda mejor este efecto de subestimación y la inclusión de la variable M para reflejarlo, al final de esta sección se presenta una analogía de este efecto con un ejemplo referente al sistema impositivo de un país.

A partir de la nueva variable M se debe cambiar el pago para el Desarrollador en el caso **(No Controlar, No testear)**, multiplicando el pago por esta variable, y de esta forma el pago pasa de ser **-Rd₂.P** a **-Rd₂.P.M**. Así, el caso 4 quedaría definido de la siguiente forma:

Caso 4: (No controlar, No testear)

Desarrollador: Se multiplica el costo original por la variable M.

$$\Pi_{\text{Desarrollador}} = -Rd_2.P.M$$

Tester: Se mantiene el costo original.

$$\Pi_{\text{Tester}} = -Rt_2.P$$

El efecto de subestimación: Analogía con el sistema impositivo de un país

Es posible que el efecto de subestimación y la nueva variable M sean más fáciles de entender utilizando una analogía con el sistema impositivo de un país.

En cualquier país, existen los impuestos, y se aplican penalidades a aquellos que no los pagan correctamente. Cada mes el contribuyente elige si paga o no los impuestos que corresponden, de la misma forma que en un proyecto de desarrollo de software el Desarrollador elige si controla o no cada uno de los casos de test definidos.

Asimismo, el organismo de recaudación de impuestos (AFIP en Argentina) elige cada mes si inspecciona o no a cada uno de los contribuyentes de la misma forma en que el Tester elige si testea o no cada uno de los casos de test.

De esta forma se puede comparar la labor de los contribuyentes con la de los Desarrolladores y la labor del ente de recaudación con la de los Testers.

Así como en un proyecto de software los Testers no pueden testear cada uno de los casos de test porque es muy caro, la AFIP tampoco puede inspeccionar a cada uno de los contribuyentes en cada período, por lo que debe elegir en forma azarosa a cuál de ellos controla y a cual de ellos no.

Ahora, para evitar que los contribuyentes evadan impuestos, se define una penalidad suficientemente grande como para que el costo de correr el riesgo de no cumplir sea más grande que el de cumplir. Pero si esto fuera así y los contribuyentes fueran completamente racionales, todos ellos pagarían sus impuestos en su totalidad y se llegaría a un equilibrio de Nash en la situación (Cumplir; No inspeccionar). Cuál es la razón por la que hay muchos contribuyentes que no cumplen con sus impuestos? Y más aún, cuál es la razón por la que el porcentaje de contribuyentes que cumple en los distintos países es diferente?

Evidentemente los contribuyentes no se comportan en forma completamente racional, y muchos de ellos menosprecian el poder de control de los inspectores en cada uno de los países. Si esto no fuera así, todos pagarían los impuestos sin problemas. Sin embargo, no todas las personas menosprecian la posibilidad de que los descubran de la misma forma, así que se puede estimar una tasa de menosprecio para cada uno. La variable M que se utiliza en la metodología se puede comparar con esta tasa definida. Cuánto más baja la variable, más menosprecia la persona la posibilidad de que lo descubran no pagando los impuestos o, en el contexto de este trabajo, no controlando los casos de Test.

Una diferencia que se puede establecer entre la irracionalidad de los contribuyentes de un sistema impositivo y la del Desarrollador en un proyecto de desarrollo de software es que los primeros sólo menosprecian la probabilidad de ser inspeccionados (ya que las penalidades las conocen correctamente), mientras que los desarrolladores no solo suelen menospreciar la probabilidad de ser testeados, sino que también la cantidad de horas que les puede llevar arreglar los errores cometidos. Sin embargo, los efectos que tienen los dos tipos de menosprecio son igualmente comparables.

6.4.3 Matriz de pagos

La nueva variable M se incorpora a la matriz de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. (-Ct-Rt ₁ .P ; -Rd ₁ .P)	2. (-Ct ; -Ci)
	No testear	4. (-Rt ₂ .P ; -Rd ₂ .P.M)	3. (0 ; -Ci)

Pagos, medidos en horas, para: (Tester; Desarrollador)

La variable M, por su definición, no es fácilmente estimable. Esto es porque es difícil estimar a priori la subestimación que tiene un Desarrollador con respecto a los costos de arreglo de errores. Sin embargo, se vio antes que si sucede que $-Rd_2.P.M < -Ci$, entonces se está frente a uno de los casos vistos en la sección 6.3 que se resuelven mediante estrategias puras.

Por lo tanto, sin perder casos, se puede agregar un nuevo supuesto que se da cuando $-Rd_2.P.M > -Ci$ o sea que $Rd_2.P.M \leq Ci$.

Supuesto 4: $0 \leq M \leq \frac{Ci}{Rd_2.P}$

Aplicando este nuevo supuesto, la matriz queda de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. (-Ct-Rt ₁ .P ; -Rd ₁ .P)	2. (-Ct ; -Ci)
	No testear	4. (-Rt ₂ .P ; -Rd ₂ .P.M)	3. ↓ (0 ; -Ci)

Pagos, medidos en horas, para: (Tester; Desarrollador)

6.4.4 Más equilibrios puros

En forma similar a la sección 6.3.7, para esta nueva matriz se forman distintos casos según los valores de las variables y las condiciones a) y b) descriptas anteriormente (la condición c) se deja de usar).

En la figura 6.5 se muestra el nuevo árbol de condiciones.

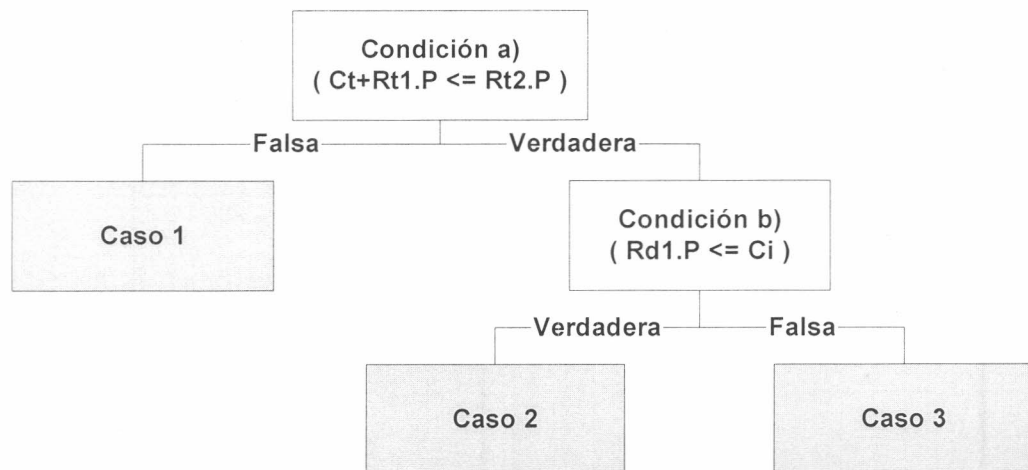


Figura 6.5. Árbol de condiciones

A continuación se resuelve el equilibrio de cada uno de los 3 casos existentes.

Caso 1

En este caso la condiciones a) es falsa. Entonces:

- $Ct + Rt_1.P > Rt_2.P$

Esta condición indica que, para un Tester, es más costoso ejecutar un caso y devolverlo al Desarrollador para su arreglo (en caso de que contenga un error) que correr con los costos de dejar pasar el caso y después testearlo cuando se encontró el error en producción (si es que tiene error).

Se puede agregar la condición a) a la matriz gráficamente de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ↓ ($-Rt_2.P$; $-Rd_2.P.M$)	3. ↓ (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

En forma similar a la sección 6.3.7, se forma un equilibrio en estrategias puras en (No Controlar ; No Testear)

Caso 2

En este caso las condiciones a) y b) son verdaderas. Entonces:

- $Ct + Rt_1.P \leq Rt_2.P$

Esta condición indica que, para un Tester, es menos costoso ejecutar un caso y devolverlo al Desarrollador para su arreglo (en caso de que contenga un error), antes que correr con los costos de dejar pasar el caso y después testearlo cuando se encontró el error en producción (si es que tiene error).

- $Rd_1.P \leq Ci$

Esta condición indica que para el Desarrollador es más costoso ejecutar un caso de test que dejarlo pasar y correr con los costos de arreglar el error, en caso de que el Tester lo encuentre en la etapa de testing.

Se pueden agregar estas condiciones a la matriz gráficamente de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P.M$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

También en este caso se forma un equilibrio en estrategias puras, esta vez en (No Controlar; Testear)

6.4.5 El equilibrio en estrategias mixtas

El último caso de los 3 definidos corresponde a las situaciones en las que la condición a) es verdadera y la condición b) es falsa. Esto significa que:

- $Ct + Rt_1.P \leq Rt_2.P$

Esta condición indica que, para un Tester, es menos costoso ejecutar un caso y devolverlo al Desarrollador para su arreglo (en caso de que contenga un error), antes que correr con los costos de dejar pasar el caso y después testearlo cuando se encontró el error en producción (si es que tiene error).

- $Rd_1.P > Ci$

Esta condición indica que para el Desarrollador es menos costoso ejecutar un caso de test que dejarlo pasar y correr con los costos de arreglar el error, en caso de que el Tester lo encuentre en la etapa de testing.

Se pueden agregar estas condiciones a la matriz gráficamente de la siguiente forma:

		Desarrollador	
		No Controlar	Controlar
Tester	Testear	1. ($-Ct - Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P.M$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (**Tester**; **Desarrollador**)

Se puede ver en la matriz anterior que se forma el circuito que se necesita para que exista sólo un equilibrio en estrategias mixtas, así como se vio en la sección 6.4.1. De cualquier forma, a continuación se verifica que no existan estrategias dominantes o dominadas, luego que no existan equilibrios en estrategias puras, y luego, finalmente, se calcula el equilibrio en estrategias mixtas del juego.

Estrategias dominantes

El juego generado no posee ninguna estrategia dominada. Esto se debe chequear caso por caso:

- El **Tester** prefiere
 - **Testear**: en caso de que el Desarrollador no controle ($Ct + Rt_1.P \leq Rt_2.P$ por condición a) verdadera)
 - **No testear**: en caso de que el Desarrollador sí controle ($0 > -Ct$ por supuesto 3)
- El **Desarrollador** prefiere:
 - **No controlar**: en caso de que el Tester no testee ($Rd_2.P.M \leq Ci$ por supuesto 4)
 - **Controlar**: en caso de que el Tester sí testee ($Ci < Rd_1.P$ por Condición b) falsa)

Dado que ninguno de los jugadores prefiere una estrategia con respecto a la otra sin importar la estrategia elegida por el otro jugador, entonces no existen estrategias dominantes sobre otras para alguno de los jugadores.

Estrategias puras

El siguiente paso es verificar que no exista ningún equilibrio de Nash en estrategias puras. Para que exista un equilibrio de Nash en estrategias puras debe existir una combinación de acciones de los jugadores que haga que a todos ellos les convenga quedarse en su situación actual antes que pasarse a una diferente. Esto quiere decir

que el pago que reciben en la combinación actual es mayor al que reciben si se cambian.

Para esto se debe controlar cada una de las combinaciones (equivalentes a los cuadros de la matriz) para ver si alguna cumple con la condición anterior:

- **Cuadro 1.** El Desarrollador no controla y recibe $-Rd_1 \cdot P$ por lo cual se verá tentado a empezar a controlar, con lo que recibirá $-Ci$ (por condición b falsa)
- **Cuadro 2.** El Tester testea y recibe $-Ct$, por lo que se pasará a "No Testear", con un aumento del pago a 0 (por supuesto 3)
- **Cuadro 3.** El Desarrollador recibe $-Ci$, por lo que se pasará a "No Controlar", con lo que pasará a recibir $-Rd_2 \cdot P \cdot M$ (por supuesto 4)
- **Cuadro 4.** El Tester recibe $-Rt_2 \cdot P$, por lo que intentará pasar a "Testear", con su consiguiente pago de $-Ct - Rt_1 \cdot P$ que es mayor (por condición a verdadera)

Como para cada una de las combinaciones de acciones hay algún jugador que se quiere "cambiar" de acción, entonces se puede verificar que no existe ningún equilibrio en estrategias puras en este juego.

Estrategias mixtas

Dado que el juego que se diseñó no posee ningún equilibrio en estrategias puras, se debe buscar un equilibrio en estrategias mixtas.

Para eso se agregan dos variables al modelo, que representan las probabilidades de elegir cada una de las acciones para cada jugador.

Se agrega α como la probabilidad de que el Tester ejecute un caso y β como la probabilidad de que el Desarrollador no ejecute un caso.

La siguiente es la matriz de pagos con las nuevas variables agregadas:

		Desarrollador	
		No Controlar (β)	Controlar ($1 - \beta$)
Tester	Testear (α)	1. ($-Ct - Rt_1 \cdot P$; $-Rd_1 \cdot P$)	2. ($-Ct$; $-Ci$)
	No testear ($1 - \alpha$)	4. ($-Rt_2 \cdot P$; $-Rd_2 \cdot P \cdot M$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

De esta manera, el equilibrio deja de estar formado por una combinación de acciones de los jugadores y pasa a estar formado por los valores de estas variables adicionales, o sea por una probabilidad para cada jugador de elegir cada una de sus acciones posibles.

Cálculo del equilibrio en estrategias mixtas

Como se dijo anteriormente, el equilibrio está compuesto por dos valores (en este caso α y β) que corresponden a las probabilidades que deben utilizar cada uno de los jugadores para que se llegue a un equilibrio. A continuación se calculan esos valores a partir de los valores que se asignen a las variables de Input del modelo.

Cálculo de β

$$\begin{aligned}\Pi_{Tester} &= \alpha.(\beta.(-Ct - Rt_1.P) + (1 - \beta).(-Ct)) + (1 - \alpha).(\beta.(-Rt_2.P)) \\ &= \alpha.(-\beta.Ct - \beta.Rt_1.P - Ct + \beta.Ct) + (1 - \alpha).(\beta.(-Rt_2.P)) \\ &= \alpha.(-\beta.Rt_1.P - Ct) - \beta.Rt_2.P + \alpha.\beta.Rt_2.P \\ &= \alpha.(-\beta.Rt_1.P - Ct + \beta.Rt_2.P) - \beta.Rt_2.P\end{aligned}$$

$$\frac{\partial \Pi_{Tester}}{\partial \alpha} = -Ct + \beta.(-Rt_1.P + Rt_2.P) = 0 \Rightarrow \beta = \frac{Ct}{P(Rt_2 - Rt_1)}$$

Cálculo de α

$$\begin{aligned}\Pi_{Desarrollador} &= \beta.(\alpha.(-Rd_1.P) + (1 - \alpha).(-Rd_2.P.M)) + (1 - \beta).(\alpha.(-Ci) + (1 - \alpha).(-Ci)) \\ &= \beta.(\alpha.(-Rd_1.P) - Rd_2.P.M + \alpha.Rd_2.P.M) + (1 - \beta).(-\alpha.Ci - Ci + \alpha.Ci) \\ &= \beta.(-\alpha.Rd_1.P - Rd_2.P.M + \alpha.Rd_2.P.M) + (-Ci) + \beta.Ci \\ &= \beta.(-\alpha.Rd_1.P - Rd_2.P.M + \alpha.Rd_2.P.M + Ci) - Ci \\ &= \beta.(\alpha(-Rd_1.P + Rd_2.P.M) - Rd_2.P.M + Ci) - Ci\end{aligned}$$

$$\frac{\partial \Pi_{Desarrollador}}{\partial \beta} = \alpha(-Rd_1.P + Rd_2.P.M) - Rd_2.P.M + Ci = 0 \Rightarrow \alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M}$$

Por lo tanto se llega a que

$$\beta = \frac{Ct}{P(Rt_2 - Rt_1)} \quad y \quad \alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M}$$

¿Son α y β probabilidades?

Se explicó anteriormente que el equilibrio del juego está compuesto por los valores de α y β , que son las probabilidades de que cada jugador elija alguna de sus dos acciones posibles.

Se debe demostrar que estas dos variables, calculadas como se mostró anteriormente, son realmente probabilidades:

- β se calcula como la relación entre Ct y $P(Rt_2 - Rt_1)$. Por condición a) verdadera, se sabe que $Ct + Rt_1.P \leq Rt_2.P$, y por lo tanto, $Ct \leq P.(Rt_2 - Rt_1)$. Por otro lado el supuesto 3 indica que todas las variables son positivas. Estas dos condiciones

implican que $\frac{Ct}{P(Rt_2 - Rt_1)}$ es un valor entre 0 y 1, y por lo tanto una probabilidad.

- α se calcula como la relación entre $Ci - Rd_2.P.M$ y $Rd_1.P - Rd_2.P.M$. Se sabe que $Ci - Rd_2.P.M$ es positivo por el supuesto 4. También que $Rd_1.P > C$ por la condición b falsa. Por lo tanto, $Rd_1.P - Rd_2.P.M$ también es positivo. Esto

demuestra que $\frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} > 0$

Ahora, por la condición b) falsa, $C < Rd_1.P$. Esto implica que $Ci - Rd_2.P.M < Rd_1.P - Rd_2.P.M$

Por lo tanto,

$$\frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} < 1$$

Entonces,

$$0 < \alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} < 1, \text{ o sea que } \alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} \text{ es un valor}$$

correcto de probabilidad.

6.4.6 Resumen

En la sección 6.3 se había definido un primer modelo para la metodología, y se había mostrado que se debían realizar algunos cambios para que quedara más acorde a la realidad.

En esta sección se mostró la razón de los desajustes del primer modelo y se propusieron ciertos cambios para solucionarlos. De esta forma, y utilizando las mismas 3 condiciones del modelo anterior, se definieron 3 casos que cubren todo el espectro de valores de las variables y se resolvió cada uno de ellos en forma diferente llegando a los siguientes resultados:

Caso	Condiciones (Falsa/Verd./ Indiferente)			Situación	Equilibrio
	A	b	c		
1	F	I	I	• $(Ct + Rt_1.P > Rt_2.P)$ Para el Tester es menor la esperanza de costo de dejar pasar el caso y después solucionar las consecuencias de los errores que se encuentren en producción, que el costo de testear el caso y devolverlo al Desarrollador en caso de que contenga un error.	En estrategias Puras en: (No Testear; No Controlar)
2	V	V	I	• $(Ct + Rt_1.P \leq Rt_2.P)$ Para el Tester es menor el costo de testear el caso y devolverlo al Desarrollador en caso de que contenga un error, que la esperanza de costo de dejar pasar el caso y después solucionar las consecuencias de los errores que se encuentren en producción. • $(Rd_1.P \leq Ci)$ Para el Desarrollador el costo de ejecutar un caso es mayor que la esperanza de costo de dejar que el caso llegue a testing con posibles errores.	En estrategias Puras en: (Testear; No Controlar)
3	V	F	I	• $(Ct + Rt_1.P \leq Rt_2.P)$ Para el Tester, Ídem Caso 2. • $(Rd_1.P > Ci)$ Para el Desarrollador es menor el costo de ejecutar un caso que la esperanza de costo de dejar que el caso llegue a testing con posibles errores, que el Tester se los devuelva y solucionarlos en ese momento.	En estrategias Mixtas en: -Testear con Prob. α -No Controlar con Prob. β

En la sección 6.3.9 se mostró que los juegos resueltos con estrategias puras no sirven para explicar los comportamientos de los Desarrolladores y Testers en la mayoría de los casos.

El caso 3, que se resuelve mediante estrategias mixtas, es el caso más "interesante" de todos, en el sentido que modela de mejor forma la mayor cantidad de situaciones que se presentan en el desarrollo de sistemas. Sin embargo, los equilibrios de los otros casos se pueden utilizar en caso de que las características de la situación lo exija.

6.5 Resultados del equilibrio

Una vez calculado el equilibrio formado por los valores de α y β , se puede llegar a realizar las siguientes preguntas:

- Cuál es la interpretación de este equilibrio ?.
- Cuál es el costo para cada uno de los jugadores en el equilibrio ?
- Cuál es el costo total para la empresa en el equilibrio ?
- Cuál es la esperanza de cantidad de errores en producción cuando se está en situación de equilibrio.
- Cuál es la ventaja del equilibrio ? Qué ventaja hay en la utilización de la metodología con respecto a la utilización de otras técnicas de testeo ?

Todos estos resultados se calculan a continuación y sirven para verificar la utilidad de la metodología en una organización para la mejora de sus procesos.

6.5.1 Interpretación de los valores de equilibrio

En la descripción de la metodología se demostró que tanto α y β , los valores del equilibrio, son probabilidades. Pero ¿cómo pueden interpretarse dichos valores a partir de las interpretaciones de las variables independientes?

β se calcula como $\frac{Ct}{P(Rt_2 - Rt_1)}$, o sea la relación entre el costo de ejecutar un caso de

test para el Tester con respecto a la diferencia entre la esperanza de costo de retrabajo del Tester con un error detectado en producción, y el costo cuando se detecta en testing.

$$\begin{aligned} \beta = P(\text{No Controlar}) &= \frac{Ct}{P(Rt_2 - Rt_1)} \\ &= \frac{\text{Costo de Testear}}{\text{Diferencia entre Esperanza de costos de Retrabajo con error en producción y en testing}} \\ &= \frac{\text{Costo de Testear}}{E \text{ de Costo de retrabajo en produccion} - E \text{ de Costo de retrabajo en testing}} \\ &= \frac{\text{Costo de Testear}}{\text{Pr obabilidad de error} \cdot (\text{Costo de retrabajo en produccion} - \text{Costo de retrabajo en testing})} \end{aligned}$$

Entonces, la probabilidad de no ejecutar un caso de test por parte del Desarrollador, o sea estar en falta, será mayor si:

- Aumenta el costo del testing: Si aumenta el costo de testeo, posiblemente el Tester decida probar menos unidades de código, lo que hará más tentador arriesgarse a no controlar internamente.
- Disminuye la diferencia entre el costo de retestear un caso cuando el programa se encuentra en producción y el costo de retestearlo si el defecto se encuentra en testing: si el beneficio de testear antes el caso es menor para el Tester, esta condición tendrá el mismo efecto que la suba del costo de testeo: hará que el Tester ponga menos énfasis en la cantidad de unidades de código a probar.

- Disminuye la probabilidad de error de un caso cuando no se hacen controles sobre el mismo. Esta probabilidad es menor cuanto más experiencia y capacidad tenga el Desarrollador. O sea, cuanto mejor el Desarrollador, más confía en sí mismo, y más cosas deja pasar.

Por otro lado, α se calcula como $\frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M}$, o sea la relación entre el costo del

control interno de un caso de test por parte de los Desarrolladores decrementado por la expectativa de esperanza de costo del Desarrollador cuando el error se encuentra en producción, y la esperanza de Costo de retrabajo de los mismos cuando el error se encuentra en testing decrementado por la expectativa de esperanza de costo del Desarrollador cuando el error se encuentra en producción.

$$\alpha = P(\text{Testear}) = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} =$$

$$= \frac{\text{Costo de Control Interno} - \text{Expectativa de costo en producción}}{\text{Esperanza de Costo de retrabajo en testing} - \text{Expectativa de costo en producción}}$$

Entonces, la probabilidad de ejecutar un caso de test por parte de los Testers aumentará si:

- Aumenta el costo de control interno para el Desarrollador: Si aumenta el costo de control interno para el Desarrollador, entonces posiblemente éste decida controlar menos unidades de código. Por lo tanto, el Tester debe desconfiar (testear) más.
- Disminuye la probabilidad de error: Si disminuye la probabilidad de error del Desarrollador habrá más posibilidades de que éste, confiadamente, se arriesgue a no controlar el código que programa.
- Disminuye el costo de retrabajo para el Desarrollador: Si disminuye este costo, se hará más atractiva para el Desarrollador la posibilidad de no controlar el código como corresponde.
- Disminuye la expectativa del Desarrollador con respecto a la esperanza de costo de retrabajo cuando el error se descubre en producción; Si la expectativa de costo del Desarrollador es menor, éste se arriesgará más a no ejecutar un caso.

6.5.2 Costo para los jugadores

Luego de interpretar los valores del equilibrio se puede calcular el costo (o pago) para cada uno de los jugadores en el equilibrio.

Esto se puede calcular reemplazando los valores de α y β en las funciones de pago de cada uno de los jugadores.

Es importante recordar que en la definición del marco metodológico se presentaron dos matrices de pagos diferentes: por un lado la matriz de pagos para la toma de decisiones de los jugadores y por otro lado la matriz de pagos real de la empresa.

Se mostró que la primera sirve para el cálculo de los equilibrios, pero no para el cálculo de costos reales. Por lo tanto, para calcular los costos finales, tanto de los jugadores como de la empresa, se utilizará la fórmula de costo calculada a partir de la segunda matriz, mientras que los equilibrios serán calculados a partir de la primera.

La función de pago del Tester de la matriz de pagos de costo real es:

$$\Pi_{Tester} = \alpha.(-\beta.Rt_1.P - Ct + \beta.Rt_2.P) - \beta.Rt_2.P$$

Por otro lado, la función de pago del Desarrollador de esta misma matriz es:

$$\begin{aligned}\Pi_{Desarrollador} &= \beta.(\alpha(-Rd_1.P) + (1-\alpha).(-Rd_2.P)) + (1-\beta)(\alpha.(-Ci) + (1-\alpha).(-Ci)) \\ &= \beta.(\alpha(-Rd_1.P) + (1-\alpha).(-Rd_2.P)) - Ci + \beta.Ci \\ &= \beta.(-\alpha.Rd_1.P - Rd_2.P + \alpha.Rd_2.P) - Ci + \beta.Ci \\ &= \beta.P.(-\alpha.Rd_1 - Rd_2 + \alpha.Rd_2) - Ci + \beta.Ci\end{aligned}$$

Reemplazando los valores de α , β del equilibrio y de las variables independientes en cada una de las fórmulas se puede calcular el costo de los jugadores por cada caso de test.

6.5.3 Costo total para la empresa

La finalidad de la metodología es alinear los objetivos del Desarrollador con los del Tester, y al mismo tiempo bajar la inversión de la empresa en la prueba de los sistemas manteniendo la calidad de los mismos. A continuación se calcula el costo para la empresa de un caso de test.

$$Costo\ Total = \Pi_{Tester}.S_{Tester} + \Pi_{Desarrollador}.S_{Desarrollador}$$

donde S_{Tester} es el salario por hora del Tester y $S_{Desarrollador}$ es salario por hora del Desarrollador

En este costo total, la cantidad de horas de cada empleado está ponderada por el salario del mismo para dar como resultado el costo total de la empresa. Este resultado muestra que no siempre los beneficios para el Tester y el Desarrollador significan beneficio para la empresa.

Supongamos que el salario por hora de un Tester es, en promedio, mucho más bajo que el salario promedio de un Desarrollador. Bajo estas condiciones es posible que no tenga sentido forzar al Desarrollador a realizar más control interno de los programas, utilizando horas de alto costo, sino que lo mejor puede ser evitar todo tipo de control interno por parte del Desarrollador y asignar al Tester todas las tareas referentes a la prueba de programas, utilizando sus horas de bajo costo.

De igual forma, si el salario del Desarrollador es mucho menor al del Tester, se podría llegar a la conclusión de que es preferible asignar al Desarrollador todas las responsabilidades sobre el testeo de programas y dejar al Tester tareas referentes a la supervisión del trabajo del Desarrollador.

De cualquier forma, dado que en la realidad los salarios de Desarrolladores y Testers no difieren en gran medida, se podría llegar a la conclusión de que en general el menor costo del Desarrollador y el Tester implica el menor costo de la empresa.

6.5.4 Cantidad de errores en producción

Otro resultado que aporta la metodología es la esperanza de cantidad de errores que habrá en producción si es que se mantiene el equilibrio.

Se puede calcular el porcentaje de casos de test que llegan a producción sin controles de ningún tipo como la probabilidad de la situación (No Controlar, No Testear) en el equilibrio.

Este valor se puede calcular como:

$$\begin{aligned}\text{Pr ob}(\text{NoControlar}, \text{NoTestar}) &= \text{Pr ob}(\text{NoControlar}) \cdot \text{Pr ob}(\text{NoTestar}) \\ &= \beta \cdot (1 - \alpha) \\ &= \frac{Ct}{P(Rt_2 - Rt_1)} \cdot \left(1 - \frac{Ci - Rd_2 \cdot P \cdot M}{Rd_1 \cdot P - Rd_2 \cdot P \cdot M}\right)\end{aligned}$$

Esto es así porque la probabilidad de ejecución de casos del Desarrollador y del Tester son sucesos independientes.

De esta forma, el valor calculado indica la probabilidad de que un caso llegue a producción sin haber sido ejecutado ni por el Tester ni por el Desarrollador.

Ahora, P es la probabilidad de que un caso de test que no ha sido ejecutado falle en producción. Entonces La probabilidad de error para un caso si se sigue el equilibrio sería:

$$\begin{aligned}\text{Pr ob}(\text{Error}) &= \text{Pr ob}(\text{NoControlar}, \text{NoTestar}) \cdot P \\ &= \frac{Ct}{P(Rt_2 - Rt_1)} \cdot \left(1 - \frac{Ci - Rd_2 \cdot P \cdot M}{Rd_1 \cdot P - Rd_2 \cdot P \cdot M}\right) \cdot P \\ &= \frac{Ct}{Rt_2 - Rt_1} \cdot \left(1 - \frac{Ci - Rd_2 \cdot P \cdot M}{Rd_1 \cdot P - Rd_2 \cdot P \cdot M}\right)\end{aligned}$$

Este valor se puede utilizar para predecir la cantidad de errores que habrá en producción siempre que se siga el equilibrio.

6.5.5 Comparación con otras estrategias de testeo.

Para analizar el aporte de la metodología al proceso de desarrollo de software se la debe comparar con otras estrategias para el testeo de programas que no la utilicen. La comparación de estrategias se puede realizar dejando fija la probabilidad de error en producción, y midiendo el costo total de la empresa para mantener esa tasa de errores en cada estrategia de testeo.

Como ejemplo se pueden utilizar las siguientes estrategias de testeo para su comparación:

1. *Utilización de la metodología presentada:* Si se utiliza la metodología presentada en este trabajo, el costo total del testeo para la empresa y la probabilidad de error son las calculadas en las secciones 6.5.3 y 6.5.4 respectivamente.
2. *Sólo el Desarrollador testea:* En esta estrategia el Desarrollador hace su control interno normalmente pero el Tester no ejecuta los casos en primera instancia sino que sólo ejecuta los casos a los que se encontraron errores en producción.
3. *Sólo el Tester testea:* Esta estrategia es similar a la anterior, salvo que esta vez es el Tester, y no el Desarrollador, el que testea, y se libera al Desarrollador de la tarea de control interno.

En el capítulo 7 se mostrará un ejemplo de aplicación de la metodología a partir de un caso basado en la realidad. Al final del mismo se calcularán los resultados y se los comparará con la aplicación, en ese mismo ejemplo, de las distintas estrategias de testeo descriptas anteriormente. De esta forma se podrán visualizar las ventajas resultantes de la aplicación de la metodología.

6.6 Síntesis de la metodología

A continuación se da una síntesis de los pasos a seguir para la aplicación del marco metodológico presentado en este capítulo en un ejemplo real en una empresa que se dedica al desarrollo de software.

1. **Definición de jugadores:** Se debe definir la lista de los Desarrolladores y Testers que participarán en el desarrollo del sistema. Se debe categorizar a cada uno de ellos según su experiencia, su eficiencia, su capacidad, etc.
2. **Definición de unidad de pagos:** Se debe definir la unidad en la que se medirán los pagos para cada uno de los jugadores. Ésta puede ser montos de dinero, horas, o alguna otra unidad de satisfacción. Las horas de trabajo suelen ser una buena unidad de medición, pero pueden utilizarse otras.
3. **Definición de unidad de test:** Se debe definir cuál va a ser la unidad que se utilizará para el testeo de los programas. La división en casos de test asignados previamente a módulos de código es la más aconsejable y es la elegida para el modelo presentado.
4. **Estimación de variables independientes:** Se debe dar una estimación de cada una de las variables independientes. Estas son: **Ct, Ci, Rd₁, Rt₁, Rd₂, Rt₂, P y M.**
5. **Verificación de supuestos sobre variables:** Se deben verificar los supuestos sobre las variables para asegurarse que los valores estimados o definidos sean lógicos
6. **Generación de matriz de pagos:** Se debe generar la matriz de pagos en la que se incluyen todos los datos recavados anteriormente.
7. **Cálculo de equilibrio:** Se debe controlar las condiciones sobre las variables para saber en qué caso entra el contexto actual. En el caso en que corresponda, se debe calcular el equilibrio en estrategias mixtas correspondiente.
8. **Cálculo de resultados:** Se deben calcular los distintos resultados de la metodología para su utilización en la mejora de los procesos de la empresa.
9. **Cálculo de métricas:** Es una actividad constante que debe realizar la empresa para poseer mejores mediciones y de esta forma estimar mejor las variables para tener resultados más precisos.

7 Caso de estudio

7.1 Introducción

Este capítulo tiene como objetivo mostrar, mediante un ejemplo concreto, la aplicación del enfoque metodológico presentado. Este ejemplo está basado en el funcionamiento de una empresa real y permitirá comprender la metodología en forma más concreta.

7.2 Presentación de la empresa

DotSys es una empresa que se dedica al desarrollo de software. Posee distintos departamentos divididos mediante un criterio funcional, cada uno con una serie de equipos de trabajo.

La división en equipos de trabajo dentro de los departamentos se realiza por criterios de proyecto.

Asimismo, cada departamento trabaja en forma independiente de los demás, y cada equipo de trabajo funciona coordinado por un jefe, que responde al líder de proyecto del proyecto en particular al que fue asignado su equipo de trabajo. Los integrantes de los equipos de trabajo suelen mantenerse juntos para los diferentes proyectos que encara la empresa en distintos momentos.

El diagrama de la figura 7.1 muestra el organigrama de la empresa.

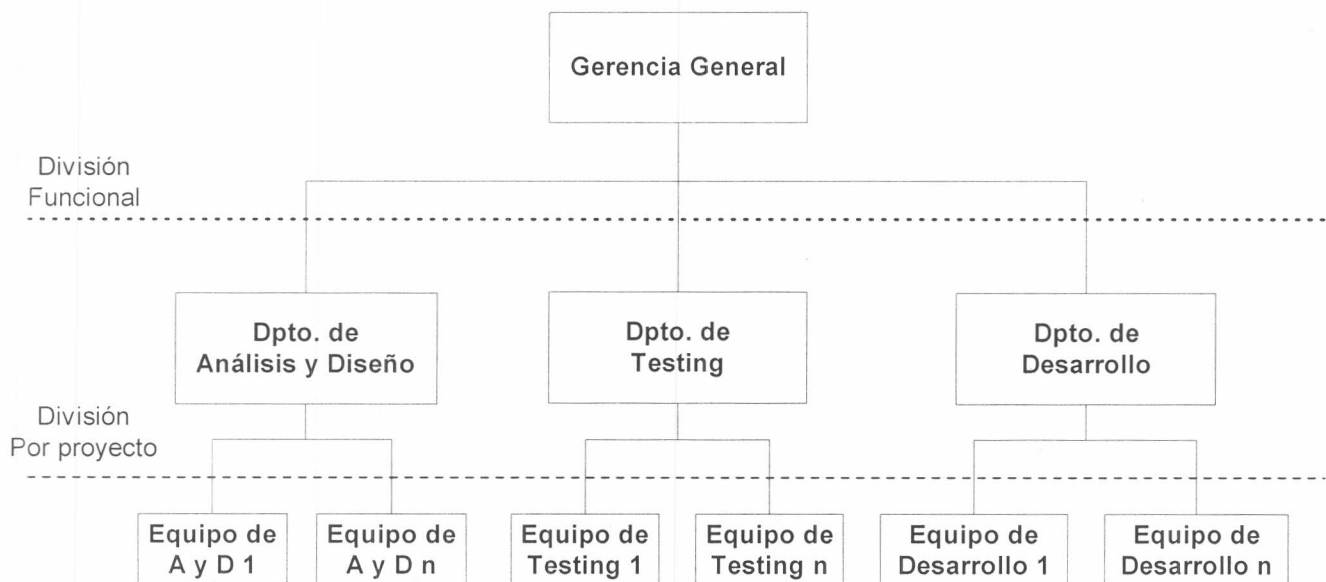


Figura 7.1. Organigrama de la empresa

En los últimos tiempos la gerencia de sistemas ha notado que se invierte una gran cantidad de recursos en el testeo de los programas y sin embargo se ha recibido un número cada vez mayor de quejas por parte de los clientes sobre la calidad de los mismos. Esto les ha traído una cantidad de implicancias negativas tanto en el prestigio de la empresa para el exterior como en los crecientes costos de retrabajo. Esta

situación ha hecho sospechar a los cargos gerenciales que los recursos que se invierten en testeos pueden no estar siendo usados eficientemente. Por otro lado la empresa está confiada en que los profesionales con los que cuenta poseen una buena preparación para el trabajo que hacen. Sin embargo no pueden sacar conclusiones sobre la calidad de sus empleados porque no llevan estadísticas sobre su trabajo.

7.2.1 Equipos de trabajo

En la presentación de la metodología se mostró que la misma se aplica a un proyecto específico de una empresa. Por lo tanto, en este ejemplo se tendrá en cuenta sólo un equipo de trabajo de cada uno de los departamentos de la empresa, que será el encargado del proyecto en cuestión. Los siguientes son los equipos de trabajo:

Equipo de relevamiento, análisis y diseño.

El equipo de relevamiento, análisis y diseño pertenece al departamento de análisis y diseño de la empresa. Trata los proyectos en su parte más temprana y se encarga de hacer el análisis de requerimientos en el cliente, analizando cuáles son las características y requerimientos, tanto funcionales como no funcionales, del sistema que se deberá desarrollar. Asimismo, realiza el diseño del sistema, dividiéndolo en una cantidad de módulos funcionales.

Luego de realizado su trabajo prepara 2 documentos: uno correspondiente al análisis de requerimientos, que podrá utilizar técnicas como los casos de uso, y otro correspondiente al diseño del sistema.

Los dos documentos se entregan a los equipos de Desarrollo y Testing para su posterior utilización.

El equipo de trabajo está compuesto por las siguientes personas:

- 1 Analista senior que coordina la etapa de análisis y diseño.
- 1 Analista júnior.

Equipo de desarrollo

El equipo de desarrollo pertenece al departamento de desarrollo de la empresa. Se encarga de realizar toda la etapa de programación de los sistemas.

Los Desarrolladores reciben los documentos de requerimientos y análisis del proyecto que utilizan como base para la programación de los programas que son parte del proyecto. Luego de programados los módulos, deben ser controlados internamente para detectar en forma temprana la mayor cantidad de errores posibles y facilitar de esta forma el trabajo al equipo de testing. Eso último lo hacen a partir del documento de diseño de casos de prueba que les entrega el departamento de testing.

El resultado de su trabajo es un conjunto de módulos desarrollados para cada uno de los programas que se realizan, junto con una serie de documentos de uso. Tanto el código fuente de estos módulos como sus archivos ejecutables son entregados al equipo de testing para su posterior testeo.

El equipo de trabajo está compuesto por las siguientes personas:

- 1 Desarrollador senior
- 2 Desarrolladores júnior.

Departamento de testing

El equipo de testing pertenece al departamento de testing de la empresa. Una vez recibidos el documento de análisis de requerimientos y el documento de diseño del

sistema, se encarga de realizar el planeamiento de los casos de test que se utilizarán para la prueba de los programas. A partir de este planeamiento se prepara un documento que se entrega al departamento de desarrollo.

Luego de desarrollado cada módulo, se encarga de realizar la ejecución de los casos de test correspondientes a cada uno de los módulos del sistema. Los módulos están compuestos por una cantidad de líneas de código, y cada módulo tiene una cantidad de casos de test con los que estará relacionado (ya que el planeamiento de los casos de test se realiza posteriormente al diseño del sistema). Al final del testeo de cada módulo se realiza un documento en donde se informa al equipo de desarrollo cuáles fueron los errores encontrados, para su posterior resolución.

El equipo de trabajo está compuesto por las siguientes personas:

- Coordinador de testing
- 2 Testers

En el diagrama de la figura 7.2 se puede ver la relación entre los distintos equipos de trabajo funcionales en medio de las distintas etapas de desarrollo de un producto en particular.

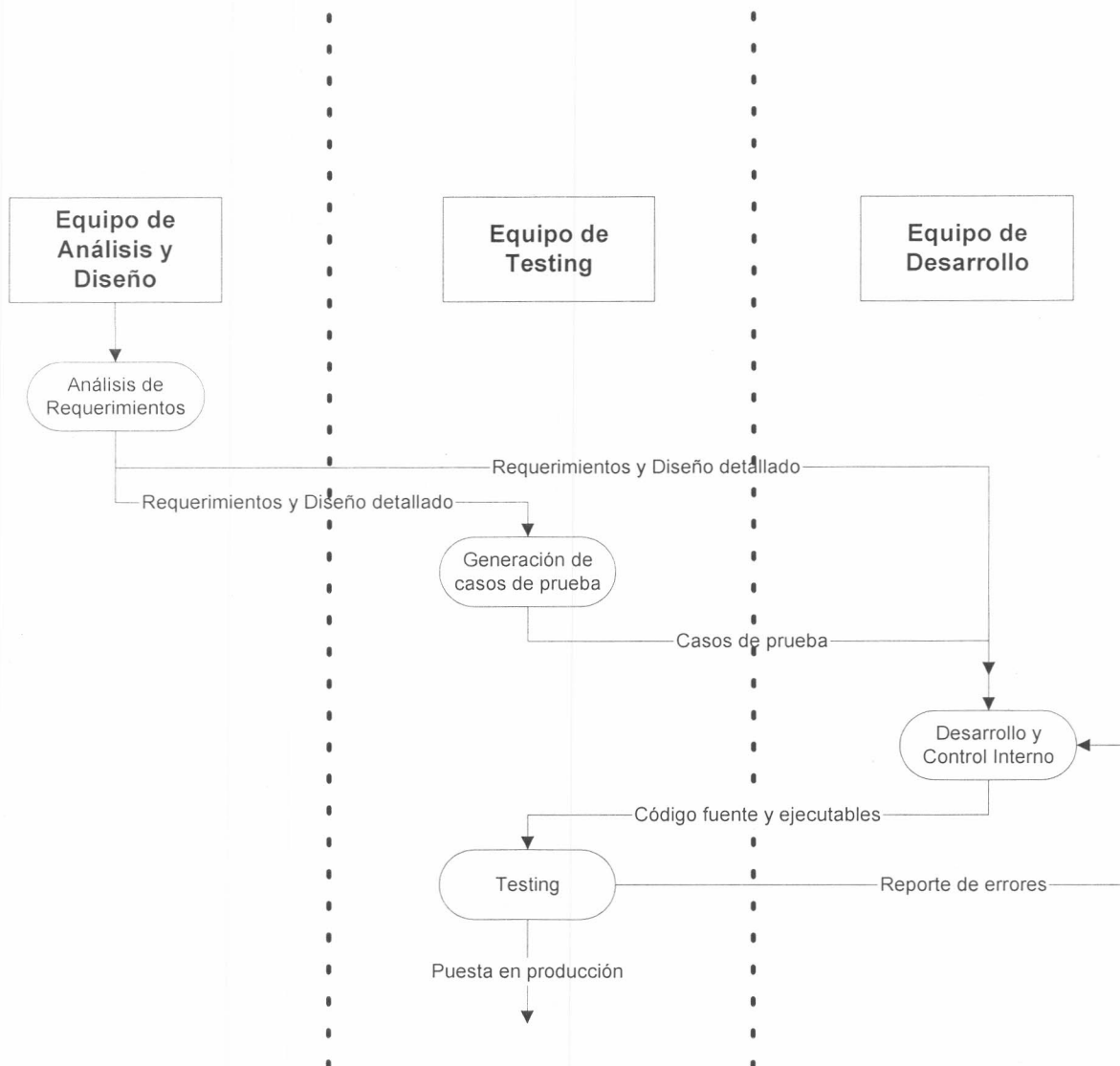


Figura 7.2. Relación entre equipos de trabajo en un proyecto

En la primera etapa los analistas realizan el análisis de requerimiento del sistema a realizar y el diseño del mismo. Le entregan los documentos de análisis y diseño tanto al departamento de testing como al de desarrollo. El departamento de testing realiza la planificación del testing, que entrega al departamento de desarrollo, mientras éste se encarga de codificar y luego controlar internamente cada uno de los módulos del sistema. Luego se entrega al departamento de testing el código fuente y los archivos ejecutables del sistema para la ejecución de los casos de test planificados anteriormente.

En caso de que se encuentren errores en testing, se entrega un reporte al departamento de desarrollo para su arreglo, luego de lo cual se devuelve al departamento de testing. Este circuito puede ser realizado varias veces, hasta que se decide poner en producción el sistema.

7.3 El proyecto

La empresa DotSys es convocada para realizar un proyecto para el estudio contable "Rodinni & Gutiérrez". El proyecto consiste en un sistema contable completo que tenga la posibilidad de manejar la contabilidad de cada uno de los clientes del estudio en forma independiente.

Luego de la realización de un relevamiento por parte del equipo de análisis, se realiza un diseño del sistema dividido en los siguientes módulos:

- Compras
- Ventas
- Pagos
- Cobros
- Cheques
- Contabilidad
- Sueldos

La empresa considera a este cliente como muy importante, y por lo tanto desea hacer todo lo posible para entregarle un producto de buena calidad con una baja cantidad de errores en producción pero sin aumentar la inversión en testing. Para esto decide utilizar la metodología presentada en este trabajo por primera vez.

7.4 Aplicando el modelo

7.4.1 Introducción

A partir de conocer las características del proyecto y de los equipos de trabajo, se puede empezar a aplicar el modelo. Para esto se deben seguir los pasos descriptos en la sección 6.6.

7.4.2 Definición de jugadores

Los integrantes de los equipos de desarrollo y testing que se encargarán de la programación de los módulos correspondientes y su testeo son los siguientes:

- Desarrollador Semi-Senior
- Desarrollador Júnior
- Tester Semi-Senior

7.4.3 Definición de unidad de pago

Como fue explicado anteriormente, se utilizará el tiempo como unidad de pago (ver sección 6.2.3). De esta forma, todas las variables que se definan o estimen posteriormente, como los costos y beneficios por ciertas tareas, serán medidas en las horas de trabajo que le cuesta a una persona realizar la tarea descripta.

7.4.4 Definición de unidad de test

Como se explicó en el marco metodológico, la unidad de testeo más conveniente es el caso de test (ver sección 6.2.3). En esta opción primero se realiza el diseño del programa dividiéndolo en módulos, y luego el equipo de testing realiza el diseño de los casos de test para cada módulo. Este mismo conjunto de casos de test será utilizado tanto por los Desarrolladores para control interno como para los Testers para el testing del programa.

7.4.5 Estimación de variables independientes

Luego de decidir cuáles van a ser las unidades de pago y de testeo, se debe empezar a estimar los valores de cada una de las variables de entrada del modelo.

Como se dijo anteriormente, la empresa DotSys no lleva ningún tipo de registro estadístico de los proyectos anteriormente desarrollados. Por lo tanto, los valores estimados a continuación para las variables son sólo aproximaciones que la empresa podrá mejorar notablemente mediante el registro constante de métricas.

Ct: Costo promedio de ejecutar un caso de test por parte de un Tester

Se calcula que un Tester puede tardar aprox. unas 2 horas en promedio en ejecutar cada caso de test, teniendo en cuenta que puede llegar a necesitar que un caso se testee más de una vez si previamente tuvo que ser arreglado. Entonces:

$$\mathbf{Ct = 2\ Hs.}$$

Ci: Costo promedio de los Desarrolladores del control interno.

Se calcula que un Desarrollador puede tardar aprox. 1 hora en promedio en ejecutar cada caso de test, menos de lo que tarda un Tester. Esto es porque se tiene en cuenta

que el Tester puede llegar a tener que ejecutar cada caso más de una vez. Por lo tanto:

$$C_i = 1 \text{ Hs.}$$

Rd₁: Costo promedio del retrabajo por parte del Desarrollador cuando se detecta un error en testing.

En las estadísticas presentadas en el marco metodológico, se menciona que el costo de arreglar un error cuando se encuentra en testing puede ser hasta 10 veces mayor que arreglarlo cuando se detecta en la etapa de prevención (ver 6.2.3). Se puede estimar esta tasa en 5, la mitad del máximo tiempo que puede llevar (con respecto al tiempo que se tarda en arreglarlo en forma temprana). Si arreglar un error en forma temprana le lleva 2 Hs. al Desarrollador, entonces se puede calcular el valor de Rd₁ como 10 Hs.

$$Rd_1 = 10 \text{ Hs.}$$

Rt₁: Costo promedio del retrabajo por parte del Tester cuando se detecta un error en testing.

Se mencionó anteriormente que si el Tester detecta un error en un caso de test, se lo debe devolver al Desarrollador para su corrección. Luego de la corrección de los errores el Desarrollador reentrega el programa al Tester para su nuevo testeo. Este circuito puede darse hasta 3 o 4 veces. Incluso se puede dar el caso de que se tengan que probar distintas partes de un programa de nuevo porque arreglar un error puede derivar en la aparición de nuevos defectos en el programa.

Se puede estimar este costo como un promedio de 3 veces más alto que el costo de testeo de un caso por parte del Tester.

$$Rt_1 = 3 * Ct = 6 \text{ Hs.}$$

Rd₂: Costo promedio del retrabajo por parte del Desarrollador cuando se detecta un error en producción.

En las estadísticas presentadas en el marco metodológico, se ve que el costo de arreglar un error cuando ya se encuentra en producción puede ser hasta 10 veces mayor al costo de arreglarlo cuando el error se descubre en etapa de testing (ver 6.2.3). Además existen otros tipos de beneficios que representa para el Tester el hecho de que no se descubran problemas con el sistema en producción. En definitiva éste es justamente su trabajo más importante.

Se estimará el valor de Rd₂ como 5 veces el costo de retrabajo con un error descubierto en la etapa de testing.

$$Rd_2 = 5 * Rd_1 = 50 \text{ Hs.}$$

Rt₂: Costo promedio del retrabajo por parte del Tester cuando se detecta un error en producción.

Varios factores, explicados anteriormente, hacen que sea más difícil retestear un caso cuando se detecta un error en producción que cuando se lo detecta en testing.

Se puede estimar este costo como aproximadamente 3 veces el valor de Rt₁.

$$Rt_2 = 3 * Rt_1 = 18 \text{ Hs.}$$

Supuestos sobre las variables:

- 1) $Rd_2 > Rd_1$: $50 > 10$. Se cumple.
- 2) $0 < P < 1$: $0 < 0,3 < 1$. Se cumple.
- 3) **Todos los valores de las variables son positivos**: Se cumple.
- 4) $0 \leq M \leq \frac{Ci}{Rd_2.P}$: $0 \leq 0,01 \leq \frac{1}{50.0,3} = 0,067$ Se cumple.

Dado que se cumplen todos los supuestos, es que se está en condiciones de aplicar el modelo en el ejemplo.

7.4.7 Generación de matriz de pagos

Luego de chequear que se verifiquen todos los supuestos del modelo se debe generar la matriz de pagos en la que se incluyen los datos recavados anteriormente. Se debe tomar como base la matriz de pagos generada en el marco metodológico, y reemplazar en ella los valores asignados a las variables.

La matriz de la metodología es la siguiente:

		Desarrollador	
		No controlar	Controlar
Tester	Testear	1. ($-Ct-Rt_1.P$; $-Rd_1.P$)	2. ($-Ct$; $-Ci$)
	No testear	4. ($-Rt_2.P$; $-Rd_2.P.M$)	3. (0 ; $-Ci$)

Pagos, medidos en horas, para: (Tester; Desarrollador)

La matriz resultante luego de reemplazar los valores asignados a las variables es la siguiente:

		Desarrollador	
		No controlar	Controlar
Tester	Testear	1. ($-3,8$; -3)	2. (-2 ; -1)
	No Testear	4. ($-5,4$; $-0,15$)	3. (0 ; -1)

Pagos, medidos en horas, para: (Tester; Desarrollador)

7.4.8 Cálculo de equilibrio

En el marco metodológico se definieron 3 condiciones que se pueden cumplir o no según los valores que se le asignan a las variables. Según la combinación de condiciones que se cumplen o no, se genera un equilibrio que se calcula en forma

diferente. Por lo tanto, antes de poder calcular el equilibrio para los valores estimados en el ejemplo, se debe chequear en cuál de los casos se está.

Condiciones sobre las variables:

- a) **$Ct + Rt_1.P \leq Rt_2.P$** : $2 + 6 * 0,3 \leq 18 * 0,3$ entonces $3,8 \leq 5,4$. La condición es verdadera.
- b) **$Rd_1.P \leq Ci$** : $10 * 0,3 \leq 1$ entonces $3 \leq 1$. La condición es falsa.
- c) **$Rd_2.P \leq Ci$** : $50 * 0,3 \leq 1$ entonces $15 \leq 1$. La condición es falsa.

Dado que la condición a) es verdadera y la b) y c) son falsas, el criterio indica que para el equilibrio del ejemplo se aplica el caso 3 (ver figura 6.5 en la sección 6.4.4). Este es el caso que se resuelve con estrategias mixtas.

Como se dijo anteriormente, este equilibrio está formado por los valores de α y β que corresponden a las probabilidades con las que cada uno de los Desarrolladores y Testers ejecutará cada caso de test.

Estos valores se pueden calcular a partir de las fórmulas a las que se llegó en el marco metodológico. Estas son:

$$\beta = \frac{Ct}{P(Rt_2 - Rt_1)} \quad y \quad \alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M}$$

Equilibrio resultante

$$\beta = \frac{Ct}{P(Rt_2 - Rt_1)} = \frac{2}{0,3.(18 - 6)} = \frac{2}{3,6} = 0,55$$

$$\alpha = \frac{Ci - Rd_2.P.M}{Rd_1.P - Rd_2.P.M} = \frac{1 - 50.0,3.0,01}{10.0,3 - 50.0,3.0,01} = \frac{0,85}{2,85} = 0,30$$

7.5 Resultados del equilibrio

7.5.1 Interpretación de valores de equilibrio

El equilibrio calculado indica dos cosas:

Por un lado la probabilidad que utilizará el Desarrollador para no controlar cada caso de test será de 0,55, o sea que **el Desarrollador controlará el 45% de los casos**. Por otro lado, la probabilidad que utilizará el Tester para testear un caso será 0,30, o sea que **el Tester controlará el 30% de los casos**.

7.5.2 Costo para los jugadores

Costo para el Tester

El costo para el Tester se calcula como

$$\begin{aligned}\Pi_{Tester} &= \alpha.(-\beta.Rt_1.P - Ct + \beta.Rt_2.P) - \beta.Rt_2.P \\ &= 0,30.(-0,55.6.0,3 - 2 + 0,55.18.0,3) - 0,55.18.0,3 \\ &= 0,30.(-1 - 2 + 3) - 3 \\ &= -3\end{aligned}$$

Costo para el Desarrollador

El costo para el Desarrollador se calcula como

$$\begin{aligned}\Pi_{Desarrollador} &= \beta.P.(-\alpha.Rd_1 - Rd_2 + \alpha.Rd_2) - Ci + \beta.Ci \\ &= 0,55.0,3.(-0,30.10 - 50 + 0,30.50) - 1 + 0,55.1 \\ &= 0,55.0,3.(-3 - 50 + 15) - 0,44 \\ &= 0,55.0,3.(-38) - 0,44 \\ &= -6,71\end{aligned}$$

7.5.3 Costo total para la empresa

El costo total para la empresa es la cantidad de dinero que le cuesta a la empresa, en promedio, cada caso de test.

Este se puede calcular como

$$\text{Costo Total} = \Pi_{Tester}.S_{Tester} + \Pi_{Desarrollador}.S_{Desarrollador}$$

donde S_{Tester} es el salario por hora del Tester y $S_{Desarrollador}$ es salario por hora del Desarrollador

Si se supone que el salario por hora de un Tester es de \$20 y el de un Desarrollador es de \$15, entonces

$$S_{Tester} = \$20$$

$$S_{Desarrollador} = \$15$$

$$\text{Costo Total} = \Pi_{Tester}.S_{Tester} + \Pi_{Desarrollador}.S_{Desarrollador} = -3.\$20 - 6,71.\$15 = \$-160,65$$

El costo total por caso para la empresa, en promedio, es de \$160,65

7.5.4 Cantidad de errores en producción

Una información muy útil que da la metodología es el la esperanza de porcentaje de errores que tendrá la empresa en el proyecto en el que utiliza la metodología si los jugadores siguen el equilibrio.

Esto se calcula como

$$Prob(Error) = Prob(NoControlar, NoTestar).P$$

$$= \beta.(1 - \alpha).P$$

$$= 0,55.(1 - 0,30).0,3 = 0,11$$

Entonces, el código en producción tendrá una tasa de error del 11%

7.5.5 Análisis de sensibilidad

El siguiente es un análisis de sensibilidad de los valores de equilibrio con respecto a las variables de entrada. Para ello se toman los valores originales de las variables de entrada en el ejemplo y se las varía en un 10%, en más y en menos, para luego calcular los nuevos valores de las variables de equilibrio y su variación con respecto al equilibrio original.

Variable α

El valor de la variable de equilibrio α depende de las variables Ci, P, Rd1, Rd2 y M. Por lo tanto esas son las variables que se varían.

Variación	Ci	P	Rd ₁	Rd ₂	M	α	Variación de α	$\pi(\text{Desarr.})$ (Horas)	Var. de $\pi(\text{Desarr.})$	$\pi(\text{Empresa})$ (\$)	Var. de $\pi(\text{Empresa})$
Original	1	0,3	10	50	0,01	0,298		-6,789		-161,842	
Ci+10%	1,1	0,3	10	50	0,01	0,333	+11,76%	-6,600	-2,79%	-159,000	-1,76%
Ci-10%	0,9	0,3	10	50	0,01	0,263	-11,76%	-6,979	2,79%	-164,684	1,76%
P+10%	1	0,33	10	50	0,01	0,266	-10,70%	-7,658	12,79%	-174,868	8,05%
P-10%	1	0,27	10	50	0,01	0,337	+13,07%	-5,921	-12,79%	-148,816	-8,05%
Rd ₁ +10%	1	0,3	11	50	0,01	0,270	-9,52%	-7,024	3,45%	-165,357	2,17%
Rd ₁ -10%	1	0,3	9	50	0,01	0,333	+11,76%	-6,500	-4,26%	-157,500	-2,68%
Rd ₂ +10%	1	0,3	10	55	0,01	0,295	-1,24%	-7,402	9,02%	-171,032	5,68%
Rd ₂ -10%	1	0,3	10	45	0,01	0,302	+1,23%	-6,183	-8,93%	-152,749	-5,62%
M+10%	1	0,3	10	50	0,011	0,295	-1,24%	-6,814	0,37%	-162,213	0,23%
M-10%	1	0,3	10	50	0,009	0,302	+1,23%	-6,765	-0,36%	-161,475	-0,23%

De este análisis se desprende que:

- Las variables a las que α es más sensible son Ci (Costo de control interno del Desarrollador), P (Probabilidad de error del desarrollador sin control) y Rd₁ (Costo de retrabajo del Desarrollador con error en testing).
- La variable a la que son más sensibles los costos del Desarrollador y de la empresa es P (Probabilidad de error del desarrollador sin control) y en segundo lugar Rd₂ (Costo de retrabajo del Desarrollador con error en producción).

- El costo total de la empresa es más sensible a la variable P (Probabilidad de error del desarrollador sin control).

Variable β

El valor de la variable de equilibrio β depende de las variables Ct, P, Rt₁ y Rt₂ y M. Por lo tanto esas son las variables que se varían.

Variación	Ct	P	Rt ₁	Rt ₂	β	Variación de β	$\pi(\text{Desarr.})$ (Horas)	Var. de $\pi(\text{Desarr.})$	$\pi(\text{Empresa})$ (\$)	Var. de $\pi(\text{Empresa})$
Original	2	0,3	6	18	0,556		-6,789		-161,842	
Ct+10%	2,2	0,3	6	18	0,611	+10,00%	-6,600	-2,79%	-167,842	3,707
Ct-10%	1,8	0,3	6	18	0,500	-10,00%	-6,979	2,79%	-155,842	-3,707
P+10%	2	0,33	6	18	0,505	-9,09%	-7,658	12,79%	-161,842	0,000
P-10%	2	0,27	6	18	0,617	+11,11%	-5,921	-12,79%	-161,842	0,000
Rt ₁ +10%	2	0,3	6,6	18	0,585	+5,26%	-7,024	3,45%	-165,000	1,951
Rt ₁ -10%	2	0,3	5,4	18	0,529	-4,76%	-6,500	-4,26%	-158,985	-1,765
Rt ₂ +10%	2	0,3	6	19,8	0,483	-13,04%	-7,402	9,02%	-159,233	-1,612
Rt ₂ -10%	2	0,3	6	16,2	0,654	+17,65%	-6,183	-8,93%	-165,372	2,181

De este análisis se desprende que:

- Las variables a las que β es más sensible son Ct (Costo de prueba del Tester), P (Probabilidad de error del desarrollador sin control) y Rt₂ (Costo de retrabajo del Tester con error en producción).
- La variable a la que es más sensible el costo del Tester es P (Probabilidad de error del desarrollador sin control) y en segundo lugar Rt₂ (Costo de retrabajo del Tester con error en producción).
- Por otro lado, el costo total de la empresa es más sensible a la variable Ct (Costo de prueba del Tester).

7.5.6 Comparación con el no uso de la metodología

Hasta ahora se mostraron los resultados que se generan utilizando la metodología. Como se habló en la sección 6.5.5, es posible comparar la metodología presentada con otras estrategias de testeo de programas.

Para comparar los distintos métodos de testeo se puede dejar fija la cantidad de errores en producción y verificar cuál de las estrategias conlleva menor costo para la empresa teniendo como máximo esa cantidad de errores.

Las distintas estrategias presentadas son las siguientes (Ver 6.5.5 para un detalle de las mismas):

1. *Utilización de la metodología presentada.*
2. *Sólo el Desarrollador testea.*
3. *Sólo el Tester testea.*

A continuación se comparará la metodología con las otras estrategias.

La esperanza de error en producción se mantiene fija y se compara el costo total para la empresa de las distintas estrategias.

Utilizando la metodología presentada en el ejemplo se llegó a que la tasa de error en producción es del 11%. Entonces, éste será el valor que se mantendrá fijo.

Estrategia 1 (metodología)

$$\Pi_{Tester} = -3$$

$$\Pi_{Desarrollador} = -6,71$$

$$Costo\ Total = -160,65$$

Estrategia 2

El Desarrollador es el único que realiza Testing. Para mantener la tasa de error del 11%, y dado que la probabilidad de que un caso que no se ejecuta previamente llegue a producción con errores (P) es de 0,3, entonces la tasa de casos sin controlar debería ser a lo sumo del 37%.

Si el Desarrollador es el único que testea, entonces debe ejecutar por lo menos el 63% de los casos.

Por lo tanto:

$$\alpha = \text{Prob}(\text{Testear}) = 0$$

$$\beta = \text{Prob}(\text{No Controlar}) = 0,37$$

$$\begin{aligned}\Pi_{Tester} &= \alpha \cdot (-\beta \cdot Rt_1 \cdot P - Ct + \beta \cdot Rt_2 \cdot P) - \beta \cdot Rt_2 \cdot P \\ &= 0 \cdot (-0,37 \cdot 6 \cdot 0,3 - 2 + 0,37 \cdot 18 \cdot 0,3) - 0,37 \cdot 18 \cdot 0,3 \\ &= -2\end{aligned}$$

$$\begin{aligned}\Pi_{Desarrollador} &= \beta \cdot P \cdot (-\alpha \cdot Rd_1 - Rd_2 + \alpha \cdot Rd_2) - Ci + \beta \cdot Ci \\ &= 0,37 \cdot 0,3 \cdot (-50) - 1 + 0,37 \cdot 1 \\ &= -5,55 - 1 + 0,37 \\ &= -6,18\end{aligned}$$

$$\begin{aligned}Costo\ Total &= \Pi_{Tester} \cdot S_{Tester} + \Pi_{Desarrollador} \cdot S_{Desarrollador} \\ &= -2 \cdot S_{Tester} - 6,18 \cdot S_{Desarrollador} \\ &= -2 \cdot 20 - 6,18 \cdot 15 \\ &= -132,66\end{aligned}$$

Estrategia 3

El Desarrollador no controla ningún caso, por lo que el Tester es el único que testea. Para mantener la tasa de error del 11%, y dado que la probabilidad de que un caso que no se ejecuta previamente llegue a producción con errores (P) es de 0,3, entonces al igual que en la estrategia 2, la tasa de casos sin controlar debería ser a lo sumo del 37%.

Si el Tester es el único que testea, entonces debe ejecutar por lo menos el 63% de los casos.

Por lo tanto:

$$\alpha = \text{Prob}(\text{Testear}) = 0,63$$

$$\beta = \text{Prob}(\text{No Controlar}) = 1$$

$$\begin{aligned}\Pi_{\text{Tester}} &= \alpha \cdot (-\beta \cdot Rt_1 \cdot P - Ct + \beta \cdot Rt_2 \cdot P) - \beta \cdot Rt_2 \cdot P \\ &= 0,63 \cdot (-1.6 \cdot 0,3 - 2 + 1.18 \cdot 0,3) - 1.18 \cdot 0,3 \\ &= 0,63 \cdot (-1,8 - 2 + 5,4) - 5,4 \\ &= -4,39\end{aligned}$$

$$\begin{aligned}\Pi_{\text{Desarrollador}} &= \beta \cdot P \cdot (-\alpha \cdot Rd_1 - Rd_2 + \alpha \cdot Rd_2) - Ci + \beta \cdot Ci \\ &= 1.0,3 \cdot (-0,63 \cdot 10 - 50 + 0,63 \cdot 50) - 1 + 1.1 \\ &= 0,3 \cdot (-6,3 - 50 + 31,5) \\ &= -7,44\end{aligned}$$

$$\begin{aligned}\text{Costo Total} &= \Pi_{\text{Tester}} \cdot S_{\text{Tester}} + \Pi_{\text{Desarrollador}} \cdot S_{\text{Desarrollador}} \\ &= -4,39 \cdot S_{\text{Tester}} - 7,44 \cdot S_{\text{Desarrollador}} \\ &= -4,39 \cdot 20 - 7,44 \cdot 15 \\ &= -199,40\end{aligned}$$

7.6 Observaciones del caso

Se utilizó la metodología presentada en el análisis de un caso basado en la realidad. Se confirmó que la metodología es fácil de utilizar, con fórmulas no muy complejas para el cálculo de los resultados.

Los resultados obtenidos de la comparación de técnicas de testeo mostraron que la técnica en la que sólo el Desarrollador realiza el testeo de los programas es la más económica para la empresa. Sin embargo, a los Desarrolladores suele disgustarle la tarea de Testing, además de que suelen estar mucho menos capacitados que los Testers para la misma. Por lo tanto la utilización de esta técnica puede tener otro tipo de consecuencias negativas para la organización.

Asimismo, se vio que el costo utilizando la metodología es 20% menor que el costo en la táctica de testeo en la que sólo los Testers realizan el testeo de programas.

8 Conclusiones

El objetivo de este trabajo es definir un marco metodológico que permita alinear el objetivo de los Desarrolladores con el de los Testers utilizando técnicas de Teoría de Juegos, en el contexto de un proyecto de ingeniería de software en una empresa. Luego de definido el marco se propuso validarlo mediante un caso de estudio. Las conclusiones que se ha podido extraer de este trabajo son las siguientes:

- La relación entre Desarrolladores y Testers en un proyecto de ingeniería de software suele ser un punto conflictivo en las empresas. Esto es porque los dos roles poseen intrínsecamente objetivos o intereses distintos. Sin embargo, la buena comunicación entre ellos es un factor muy importante para el éxito de los proyectos de software, por lo que alinear sus objetivos es una tarea sobre la que vale la pena trabajar.
- La Teoría de Juegos se aplica a situaciones en donde un grupo de individuos tienen intereses contrapuestos para lograr un objetivo. La oposición de objetivos de Desarrolladores y Testers hace que el problema se adapte bien al dominio de aplicación de esta teoría.
- El marco metodológico propuesto provee a la empresa una forma de análisis del problema de los costos de Testing en una empresa en forma sistematizada y racional. Además le provee un standard de trabajo que permite al Tester hacerse impredecible ante el Desarrollador quitándole la necesidad de ejecutar todos los casos de test diseñados. De esta forma la metodología le indica al Tester, mediante estrategias mixtas de Teoría de Juegos, el porcentaje de casos de test que debe ejecutar y de qué forma los debe elegir para lograr dicho objetivo.
- La metodología presentada es aplicable por cualquier empresa que produzca software. Sólo exige la utilización de ciertas métricas para la estimación de las variables de entrada de la metodología, además de la adaptación de la forma de trabajo a ciertos standards. La utilización de buenas y variadas métricas constantemente en los distintos proyectos encarados por una empresa mejora sensiblemente la exactitud del equilibrio resultante.
- Puede pensarse que el objetivo de la empresa está más representado en los objetivos de los Testers que en los de los Desarrolladores, y por lo tanto si la metodología se alinea a los objetivos del Tester también lo hace con respecto a los de la organización. Sin embargo esto no se cumple en todos los casos. Se vio que no siempre el equilibrio que calcula la metodología beneficia a la empresa. Esto depende, entre otras cosas de los salarios de los Desarrolladores y Testers dentro de la misma.
- El modelo más intuitivo utilizado al principio de la metodología no resulta suficientemente cercano a la realidad. Esto es porque los Desarrolladores no actúan de una forma totalmente racional, por lo menos en el corto y mediano plazo en un proyecto de software. Para hacer más real el modelo se lo debe modificar cambiando ciertos parámetros que tienen que ver con las expectativas del Desarrollador con respecto a su función de pagos. Se puede esperar que en el largo plazo los participantes modifiquen sus errores de percepción y comiencen a actuar más racionalmente.
- La metodología provee a la empresa una serie de resultados que son útiles para su funcionamiento y la mejora de sus procesos. Estos resultados se dan cuando

se pone en práctica el equilibrio en estrategias mixtas de la metodología. Entre estos resultados están el costo en horas de Desarrolladores y Testers con cada caso de test, el costo total para la empresa de cada caso y la tasa de errores a registrar en producción.

- Se aplicó la metodología a un caso concreto y se pudo verificar la facilidad de aplicación por parte de una empresa, y la facilidad de cálculo de los resultados finales.
- Los resultados obtenidos de la comparación de técnicas de testeo mostraron que en el caso estudiado, el costo total para la empresa utilizando la metodología es 20% menor que el costo utilizando la táctica de testeo en la que sólo los Testers realizan la prueba de los programas.

10 Bibliografía

- [AYA/99] Ayala Ruiz, Luis Eduardo y Arias Amaya, Ramiro. 1999. Gerencia de Mercadeo, Apuntes de clase.
- [BEI/95]: Beizer, Boris. 1995. Black-Box Testing.
- [COS/00]: Costales, Felipe. 2000. Monografía sobre Teoría de juegos
- [DIX/91]: Dixit, Vainas K. y Nalebuff, Barry J. 1991. Thinking Strategically.
- [KRA/99]: Krasner, Herb. 1999. Using The Cost of Quality Approach for Software Development
- [LUC/57] Luce, R. D. y Raiffa, H. 1957. Games and Decisions
- [NAL/96]: Nalebuff, Barry J. y Brandenburger, Adam M. 1996. Coopetencia.
- [PFL/01]: Pfleeger, Shari Lawrence. 2001. Ingeniería de Software, Teoría y práctica.
- [RAS/01]: Rasmusen, Eric. 2001. Games and Information, Third Edition - An introduction to Game Theory.
- [ROS/01]: Rosenmüller, Joachim y Trockel, Walter. 2001. Game Theory
- [RUF/03]: Rufasto, Augusto. 2003. Manual de Teoría de Juegos
- [USB/96]: Estrategias de juego en las decisiones económicas. 1996. Publicado en Economía #20. Universidad de San Buenaventura.
- [VNE/44]: Von Neumann, John y Morgenstern, Oskar. 1944. Theory of Games and Economic Behavior.

