10. Testabilidad

La prueba conduce al fracaso, y el fracaso lleva a la comprensión
—Burt Rutan

Las estimaciones de la industria indican que entre el 30 y el 50 por ciento (o en algunos casos, incluso más) del costo del desarrollo de sistemas bien diseñados se realiza mediante pruebas. Si el arquitecto de software puede reducir este costo, la recompensa es grande.

La capacidad de prueba del software se refiere a la facilidad con la que se puede hacer que el software demuestre sus fallas a través de pruebas (típicamente basadas en la ejecución). Específicamente, la capacidad de prueba se refiere a la probabilidad, asumiendo que el software tiene al menos una falla, que fallará en su próxima ejecución de prueba. Intuitivamente, un sistema es comprobable si "abandona" sus fallas fácilmente. Si una falla está presente en un sistema, queremos que falle durante la prueba lo más rápido posible. Por supuesto, el cálculo de esta probabilidad no es fácil y, como verá cuando analicemos las medidas de respuesta para la comprobabilidad, se utilizarán otras medidas.

La Figura 10.1 muestra un modelo de prueba en el que un programa procesa la entrada y produce la salida. Un oráculo es un agente (humano o mecánico) que decide si la salida es correcta o no comparando la salida con la especificación del programa. La salida no es solo el valor producido funcionalmente, sino que también puede incluir medidas derivadas de atributos de calidad, como el tiempo que llevó producir la salida. La Figura 10.1 también muestra que el estado interno del programa también se puede mostrar al oráculo, y un oráculo puede decidir si eso es correcto o no, es decir, puede detectar si el programa ha entrado en un estado erróneo y emitir un juicio en cuanto al Corrección del programa.

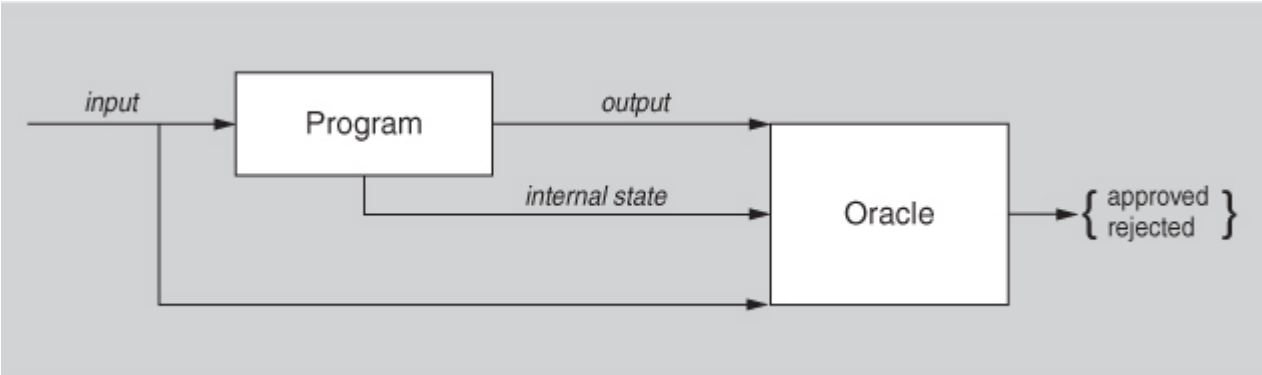

Figura 10.1. Un modelo de prueba.

Establecer y examinar el estado interno de un programa es un aspecto de las pruebas que ocupará un lugar destacado en nuestras tácticas de verificación.

Para que un sistema sea verificable adecuadamente, debe ser posible controlar las entradas de cada componente (y posiblemente manipular su estado interno) y luego observar sus salidas (y posiblemente su estado interno, ya sea después o en el camino para calcular las salidas). Con frecuencia, este control y observación se realiza mediante el uso de un arnés de prueba, que es un software especializado (o en algunos casos, hardware) diseñado para ejercer el software bajo prueba. Los arneses de prueba vienen en varias formas, como la capacidad de grabación y reproducción de datos enviados a través de varias interfaces, o un simulador para un entorno externo en el que se prueba una pieza de software integrado, o incluso durante la producción (vea la barra lateral). El arnés de prueba puede proporcionar asistencia para ejecutar los procedimientos de prueba y registrar la salida.

Las pruebas se llevan a cabo por varios desarrolladores, usuarios o personal de control de calidad. Se pueden probar partes del sistema o todo el sistema. Las medidas de respuesta para la capacidad de prueba se relacionan con la efectividad de las pruebas en el descubrimiento de fallas y el tiempo que lleva realizar las pruebas hasta el nivel deseado de cobertura. Los casos de prueba pueden ser escritos por los desarrolladores, el grupo de prueba o el cliente. Los casos de prueba pueden ser una parte de las pruebas de aceptación o pueden impulsar el desarrollo como lo hacen en ciertos tipos de metodologías ágiles.

Simian Army de Netflix

Netflix distribuye películas y programas de televisión tanto a través de DVD como a través de transmisión de video. Su servicio de transmisión de video ha sido extremadamente exitoso. En mayo de 2011, el flujo de video de Netflix representó el 24 por ciento del tráfico de Internet en América del Norte. Naturalmente, la alta disponibilidad es importante para Netflix.

Netflix aloja sus servicios informáticos en la nube de Amazon EC2, y utilizan lo que llaman un "Ejército Simiano" como parte de su proceso de prueba. Comenzaron con un mono caos, que mata al azar los procesos en elsistema en ejecución Esto permite monitorear el efecto de los procesos fallidos y brinda la capacidad de garantizar que el sistema no falle o sufra una degradación grave como resultado de una falla del proceso.

Recientemente, el Chaos Monkey consiguió que algunos amigos asistieran en las pruebas. Actualmente, el ejército de Netflix Simian incluye estos:

• El Latency Monkey induce retrasos artificiales en la capa de comunicación cliente-servidor para simular la degradación del servicio y mide si los servicios de nivel superior responden adecuadamente.

• El Conformity Monkey encuentra instancias que no se adhieren a las mejores prácticas y las apaga. Por ejemplo, si una instancia no pertenece a un grupo de escalado automático, no se escalará adecuadamente cuando aumente la demanda.

• El Doctor Mono aprovecha las comprobaciones de estado que se ejecutan en cada instancia, así como también supervisa otros signos externos de salud (por ejemplo, la carga de la CPU) para detectar instancias poco saludables.

• Janitor Monkey garantiza que el entorno de nube de Netflix se ejecute sin desorden y desperdicio. Busca recursos no utilizados y dispone de ellos.

• El mono de seguridad es una extensión de Conformity Monkey. Encuentra infracciones o vulnerabilidades de seguridad, como grupos de seguridad configurados incorrectamente, y termina las instancias ofensivas. También garantiza que todos los certificados de gestión de derechos digitales (DRM) y SSL sean válidos y no se renueven.

• El 10-18 Monkey (localización-internacionalización) detecta problemas de configuración y tiempo de ejecución en instancias que atienden a clientes en múltiples regiones geográficas, utilizando diferentes idiomas y conjuntos de caracteres. El nombre 10-18 proviene de *L10n-i18n* , una especie de taquigrafía para la *localización* e *internacionalización de* palabras .

Algunos de los miembros del Ejército de Simia utilizan la inyección de fallas para colocar fallas en el sistema en ejecución de forma controlada y monitoreada. Otros miembros monitorean diversos aspectos especializados del sistema y su entorno. Ambas técnicas tienen una aplicabilidad más amplia que solo Netflix.

No todas las fallas son iguales en términos de severidad. Debe ponerse más énfasis en encontrar las fallas más graves que en encontrar otras fallas. El

Ejército de Simia refleja una determinación de Netflix de que las fallas que buscan son las más serias en términos de su impacto.

Esta estrategia ilustra que algunos sistemas son demasiado complejos y adaptables para ser probados completamente, porque algunos de sus comportamientos son emergentes. Un aspecto de las pruebas en ese campo es el registro de los datos operacionales producidos por el sistema, de modo que cuando se producen fallas, los datos registrados se pueden analizar en el laboratorio para tratar de reproducir las fallas. Arquitectónicamente, esto puede requerir mecanismos para acceder y registrar cierto estado del sistema. El Ejército Simiano es una forma de descubrir y registrar el comportamiento en sistemas de esta clase.

- *LB*

La prueba de código es un caso especial de validación, que consiste en asegurarse de que un artefacto diseñado cumpla con las necesidades de sus partes interesadas o sea adecuado para su uso. En el capítulo 21 discutiremos las revisiones de diseño arquitectónico. Este es otro tipo de validación, donde el artefacto que se está probando es la arquitectura. En este capítulo solo nos preocupa la capacidad de prueba de un sistema en ejecución y de su código fuente.

10.1. ESCENARIO DE PROBABILIDAD GENERAL

Ahora podemos describir el escenario general para la testabilidad.

• *Fuente de estímulo*. Las pruebas son realizadas por probadores de unidades, probadores de integración o probadores de sistemas (en el lado de la organización en desarrollo), o probadores de aceptación y usuarios finales (por el lado del cliente). La fuente podría ser humana o un probador automatizado.

• *Estímulo* . Se ejecuta un conjunto de pruebas debido a la finalización de un incremento de codificación como una capa de clase o un servicio, la integración completa de un subsistema, la implementación completa de todo el sistema o la entrega del sistema al cliente.

• *Artefacto* . Una unidad de código (correspondiente a un módulo en la arquitectura), un subsistema o todo el sistema es el artefacto que se está probando.

• *Medio ambiente* . La prueba puede realizarse en el momento del desarrollo, en el momento de la compilación, en el momento del despliegue o mientras el sistema se está ejecutando (tal vez en uso de rutina). El entorno también puede incluir el arnés de prueba o los entornos de prueba en uso.

• *Respuesta* . El sistema se puede controlar para realizar las pruebas deseadas y se pueden observar los resultados de la prueba.

• *Medida de respuesta* . Las medidas de respuesta están destinadas a representar la facilidad con la que un sistema en prueba "abandona" sus fallas. Las medidas pueden incluir el esfuerzo involucrado en encontrar una falla o una clase particular de fallas, el esfuerzo requerido para probar un porcentaje dado de declaraciones, la longitud de la cadena de prueba más larga (una medida de la dificultad de realizar las pruebas), las medidas de esfuerzo para realizar las pruebas, medidas de esfuerzo para encontrar fallas, estimaciones de la probabilidad de encontrar fallas adicionales y el tiempo o la cantidad de esfuerzo para preparar el entorno de prueba.

Quizás una medida es la facilidad con la que el sistema puede ser llevado a un estado específico. Además, se pueden usar medidas de la reducción del riesgo de los errores restantes en el sistema. No todas las fallas son iguales en términos de su posible impacto. Las medidas de reducción de riesgo intentan evaluar la gravedad de las fallas encontradas (o por encontrarlas).

La figura 10.2 muestra un escenario concreto de comprobabilidad. El probador de la unidad completa una unidad de código durante el desarrollo y realiza una secuencia de prueba cuyos resultados se capturan y que proporciona una cobertura de ruta del 85 por ciento dentro de las tres horas de la prueba.

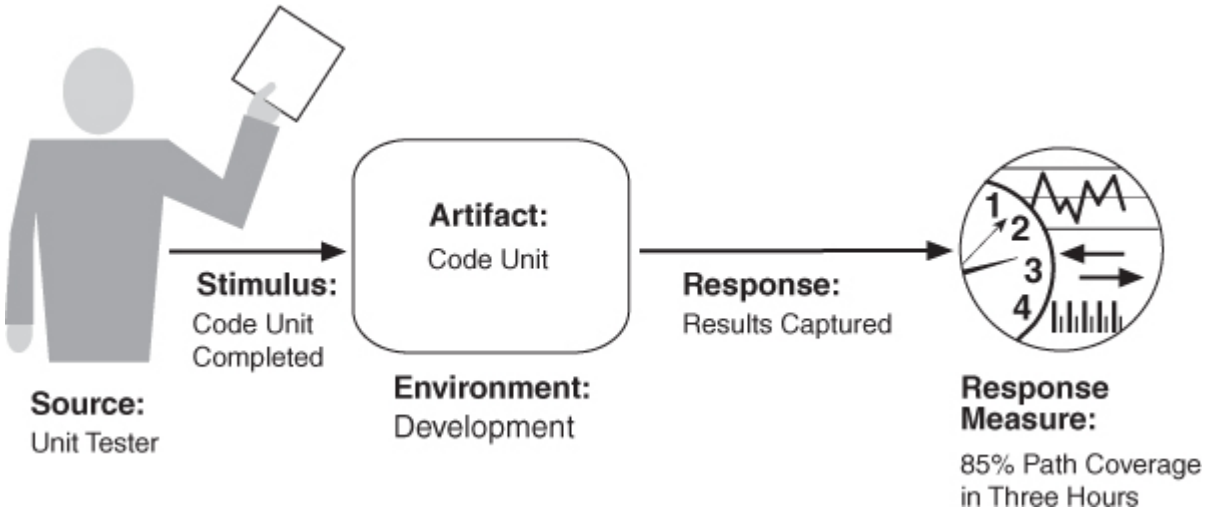

Figura 10.2. Escenario concreto de prueba de muestra.

La tabla 10.1 enumera los elementos del escenario general que caracterizan la capacidad de prueba.

Tabla 10.1. Escenario de probabilidad general

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results, capture activity that resulted in the fault, control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure (size(loss) × prob(loss))

10.2. TÁCTICAS PARA LA TESTABILIDAD

El objetivo de las tácticas para la capacidad de prueba es permitir pruebas más fáciles cuando se completa un incremento del desarrollo de software. La figura 10.3 muestra el uso de tácticas para la prueba. Las técnicas de arquitectura para mejorar la capacidad de prueba del software no han recibido tanta atención como las disciplinas de atributos de calidad más maduras, como la modificabilidad, el rendimiento y la disponibilidad, pero como dijimos anteriormente, cualquier cosa que pueda hacer el arquitecto para reducir el alto costo de las pruebas dará como resultado beneficio.

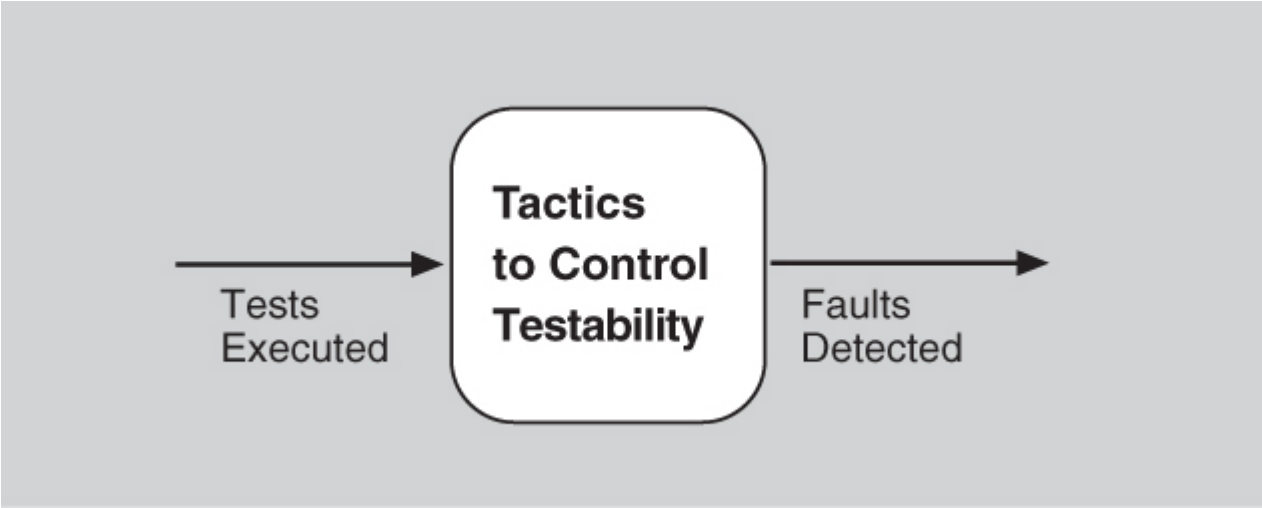

Figura 10.3. El objetivo de las tácticas de testabilidad.

Hay dos categorías de tácticas para la testabilidad. La primera categoría trata de agregar control y observabilidad al sistema. El segundo se ocupa de limitar la complejidad en el diseño del sistema.

Controlar y observar el estado del sistema

El control y la observación son tan fundamentales para la capacidad de prueba que algunos autores incluso definen la capacidad de prueba en esos términos. Los dos van de la mano; no tiene sentido controlar algo si no puedes observar lo que sucede cuando lo haces. La forma más simple de control y observación es proporcionar un componente de software con un conjunto de entradas, dejar que haga su trabajo y luego observar sus salidas. Sin embargo, la categoría de control y observación del estado del sistema de las tácticas de prueba proporciona información sobre el software que va más allá de sus entradas y salidas. Estas tácticas hacen que un componente mantenga algún tipo de información de estado, permita a los evaluadores asignar un valor a esa información de estado y / o haga que esa información sea accesible para los evaluadores que lo soliciten. La información del estado puede ser un estado operativo, el valor de alguna variable clave, la carga de rendimiento, los pasos del proceso intermedio, o cualquier otra cosa útil para recrear el comportamiento de los componentes. Las tácticas específicas incluyen lo siguiente:

• *Interfaces especializadas*. Tener interfaces de prueba especializadas le permite controlar o capturar valores variables para un componente a

través de un arnés de prueba o mediante una ejecución normal. Ejemplos de rutinas de prueba especializadas incluyen estas:

• Un método de *establecer* y *obtener* variables, modos o atributos importantes (métodos que de otra manera no estarían disponibles, excepto para propósitos de prueba)

• Un método de *informe* que devuelve el estado completo del objeto.

• Un método de *restablecimiento* para establecer el estado interno (por ejemplo, todos los atributos de una clase) en un estado interno específico

• Un método para activar resultados detallados, varios niveles de registro de eventos, instrumentación de rendimiento o monitoreo de recursos

Las interfaces y los métodos de prueba especializados deben identificarse claramente o mantenerse separados de los métodos de acceso e interfaces para la funcionalidad requerida, de modo que puedan eliminarse si es necesario. (Sin embargo, en sistemas críticos para el rendimiento y algunos sistemas críticos para la seguridad, es problemático presentar un código diferente al que se probó. Si elimina el código de prueba, ¿cómo sabrá que el código que presenta tiene el mismo comportamiento, particularmente el mismo? comportamiento de sincronización, como el código que probó? Para otros tipos de sistemas, sin embargo, esta estrategia es efectiva.)

• *Grabación / reproducción* . El estado que causó una falla a menudo es difícil de recrear. La grabación del estado cuando cruza una interfaz permite que ese estado se use para "reproducir el sistema" y para volver a crear la falla. La grabación / reproducción se refiere a la captura de información que atraviesa una interfaz y se usa como entrada para pruebas adicionales.

• *Localizar almacenamiento de estado* . Para iniciar un sistema, subsistema o módulo en un estado arbitrario para una prueba, es más conveniente si ese estado se almacena en un solo lugar. Por el contrario, si el estado está enterrado o distribuido, esto se vuelve difícil, si no imposible. El estado puede ser de grano fino, incluso a nivel de bits, o de grano grueso para representar abstracciones amplias o modos operativos generales. La elección de la granularidad depende de cómo se utilizarán los estados en las pruebas. Una forma conveniente de "externalizar" el almacenamiento de estado (es decir, para poder manipularlo a través de las características de la interfaz) es usar una

máquina de estados (u objeto de máquina de estados) como mecanismo para rastrear e informar el estado actual.

• *Fuentes de datos abstractos*. Al igual que para controlar el estado de un programa, controlar fácilmente los datos de entrada facilita la prueba. El resumen de las interfaces le permite sustituir los datos de prueba más fácilmente. Por ejemplo, si tiene una base de datos de transacciones de clientes, puede diseñar su arquitectura para que sea fácil apuntar su sistema de prueba a otras bases de datos de prueba, o posiblemente incluso a archivos de datos de prueba, sin tener que cambiar su código funcional.

• *Caja de arena* . "Sandboxing" se refiere a aislar una instancia del sistema del mundo real para permitir la experimentación que no está limitada por la preocupaciónSobre tener que deshacer las consecuencias del experimento. Las pruebas son ayudadas por la capacidad de operar el sistema de tal manera que no tiene consecuencias permanentes, o para que cualquier consecuencia pueda revertirse. Esto se puede utilizar para el análisis de escenarios, entrenamiento y simulación. (El marco de Spring, que es bastante popular en la comunidad Java, viene con un conjunto de utilidades de prueba que lo admiten. Las pruebas se ejecutan como una "transacción", que se revierte al final).

Una forma común de sandboxing es virtualizar recursos. Probar un sistema a menudo implica interactuar con recursos cuyo comportamiento está fuera del control del sistema. Usando un sandbox, puede construir una versión del recurso cuyo comportamiento está bajo su control. Por ejemplo, el comportamiento del reloj del sistema generalmente no está bajo nuestro control (se incrementa un segundo por segundo), lo que significa que si queremos que el sistema piense que es medianoche del día en que se supone que todas las estructuras de datos se desbordarán, necesitamos una forma de hacerlo, porque esperar es una mala elección. Al tener la capacidad de abstraer la hora del sistema de la hora del reloj, podemos permitir que el sistema (o los componentes) se ejecute a una velocidad mayor que la del reloj de pared. y para permitir que el sistema (o los componentes) se prueben en límites de tiempo críticos (como el próximo turno de encendido o apagado del horario de verano). Podrían realizarse virtualizaciones similares para otros recursos, como memoria, batería, red, etc. Los stubs, las simulaciones y la inyección de dependencia son formas simples pero efectivas de virtualización.

• *Afirmaciones ejecutables*. Usando esta táctica, las aserciones (generalmente) se codifican a mano y se colocan en las ubicaciones deseadas para indicar cuándo y dónde un programa se encuentra en un estado defectuoso. Las aserciones a menudo están diseñadas para verificar que los valores de los datos satisfacen restricciones específicas. Las aserciones se definen en términos de declaraciones de datos específicas y deben colocarse donde se hace referencia o se modifican los valores de los datos. Las afirmaciones se pueden expresar como condiciones previas y posteriores para cada método y también como invariantes de nivel de clase. Esto se traduce en un aumento de la observabilidad, cuando una afirmación se marca como que ha fallado. Las aserciones insertadas sistemáticamente donde los valores de los datos cambian se pueden ver como una forma manual de producir un tipo "extendido". Esencialmente, el usuario está anotando un tipo con un código de verificación adicional. Cada vez que se modifica un objeto de ese tipo, el código de verificación se ejecuta automáticamente, y se generan advertencias si se viola alguna condición. En la medida en que las aserciones cubren los casos de prueba, efectivamente incorporan el oráculo de la prueba en el código, asumiendo que las aserciones son correctas y están codificadas correctamente.

Todas estas tácticas agregan capacidad o abstracción al software que (si no estuviéramos interesados en probar), de lo contrario no estarían allí. Se puede considerar que reemplazan el software básico para hacer el trabajo con un software más elaborado que tiene campanas y silbidos para las pruebas. Hay una serie de técnicas para efectuar este reemplazo. Estas no son tácticas de prueba, per se, sino técnicas para reemplazar un componente con una versión diferente de sí mismo. Incluyen los siguientes:

• Reemplazo de componentes, que simplemente cambia la implementación de un componente con una implementación diferente que (en el caso de la capacidad de prueba) tiene características que facilitan las pruebas. El reemplazo de componentes se realiza a menudo en los scripts de compilación de un sistema.

• Macros de preprocesador que, cuando se activan, se expanden al código de informe de estado o activan sentencias de sondeo que devuelven o muestran información, o devuelven el control a una consola de prueba.

• Aspectos (en programas orientados a aspectos) que manejan la preocupación transversal de cómo se informa el estado.

Complejidad límite

El software complejo es más difícil de probar. Esto se debe a que, según la definición de complejidad, su espacio de estado operativo es muy grande y (siendo todo lo demás igual) es más difícil recrear un estado exacto en un espacio de estado grande que hacerlo en un espacio de estado pequeño. Debido a que las pruebas no solo consisten en hacer que el software falle, sino en encontrar la falla que causó el error para poder eliminarlo, a menudo nos preocupa que el comportamiento sea repetible. Esta categoría tiene tres tácticas:

• *Limitar la complejidad estructural*. Esta táctica incluye evitar o resolver dependencias cíclicas entre componentes, aislar y encapsular dependencias en el entorno externo y reducir las dependencias entre componentes en general (por ejemplo, reducir la cantidad de accesos externos a los datos públicos de un módulo). En los sistemas orientados a objetos, puede simplificar la jerarquía de herencia: limite el número de clases de las que se deriva una clase o el número de clases derivadas de una clase. Limite la profundidad del árbol de herencia y el número de hijos de una clase. Limitar el polimorfismo y las llamadas dinámicas. Una métrica estructural que se ha demostrado empíricamente que se correlaciona con la capacidad de prueba se llama *respuesta*de una clase. La respuesta de la clase C es un conteo del número de métodos de C más el número de métodos de otras clases que son invocados por los métodos de C. Mantener esta métrica baja puede aumentar la capacidad de prueba.

Tener una alta cohesión, un acoplamiento flexible y una separación de inquietudes: todas las tácticas modificables (ver Capítulo 7) También puede ayudar con la capacidad de prueba. Son una forma de limitar la complejidad de los elementos arquitectónicos al dar a cada elemento una tarea enfocada con una interacción limitada con otros elementos. La separación de las preocupaciones puede ayudar a lograr la capacidad de control y la observabilidad (así como a reducir el tamaño del espacio estatal del programa en general). La capacidad de control es fundamental para hacer que las pruebas sean manejables, como señaló Robert Binder: "Un componente que puede actuar independientemente de los demás es más fácilmente controlable. . . . Con un alto acoplamiento entre clases, normalmente es más difícil controlar la clase bajo prueba, lo que reduce la capacidad de

prueba. . . . Si las capacidades de la interfaz de usuario están entrelazadas con funciones básicas, será más difícil probar cada función " [Carpeta 94] .

Además, los sistemas que requieren una consistencia completa de los datos en todo momento son a menudo más complejos que aquellos que no lo hacen. Si sus requisitos lo permiten, considere construir su sistema bajo el modelo de "consistencia eventual", donde tarde o temprano (pero quizás no ahora) sus datos alcanzarán un estado consistente. Esto a menudo hace que el diseño del sistema sea más sencillo y, por lo tanto, más fácil de probar.

Finalmente, algunos estilos arquitectónicos se prestan a comprobabilidad. En un estilo de capas, puede probar primero las capas inferiores y luego probar las capas superiores con confianza en las capas inferiores.

• *Limitar el no determinismo*. La contraparte de limitar la complejidad estructural es limitar la complejidad del comportamiento, y cuando se trata de pruebas, el no determinismo es una forma muy perniciosa de comportamiento complejo. Los sistemas no deterministas son más difíciles de probar que los sistemas deterministas. Esta táctica implica encontrar todas las fuentes de no determinismo, como el paralelismo sin restricciones, y eliminarlas lo más posible. Algunas fuentes de no determinismo son inevitables, por ejemplo, en sistemas de subprocesos múltiples que responden a eventos impredecibles, pero para tales sistemas, otras tácticas (como la grabación / reproducción) están disponibles.

La figura 10.4 proporciona un resumen de las tácticas utilizadas para la prueba.

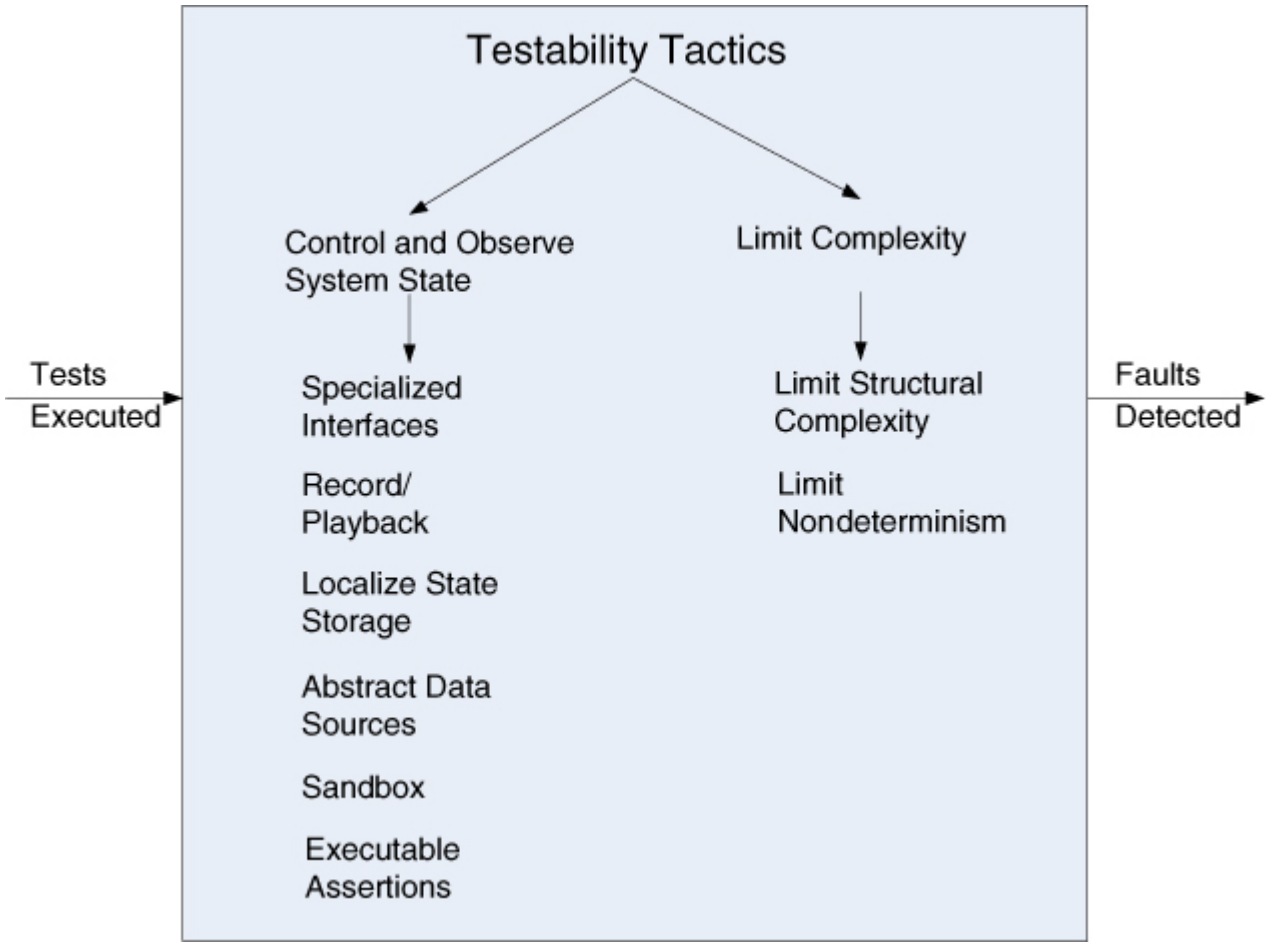

Figura 10.4. Tácticas de testabilidad

10.3. UNA LISTA DE VERIFICACIÓN DE DISEÑO PARA LA PROBABILIDAD

La Tabla 10.2 es una lista de verificación para respaldar el proceso de diseño y análisis para la capacidad de prueba.

Tabla 10.2. Lista de verificación para apoyar el proceso de diseño y análisis para la capacidad de prueba

Category	Checklist
Allocation of Responsibilities	Determine which system responsibilities are most critical and hence need to be most thoroughly tested. Ensure that additional system responsibilities have been allocated to do the following: ▪ Execute test suite and capture results (external test or self-test) ▪ Capture (log) the activity that resulted in a fault *or* that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault ▪ Control and observe relevant system state for testing Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.
Coordination Model	Ensure the system's coordination and communication mechanisms: ▪ Support the execution of a test suite and capture the results within a system or between systems ▪ Support capturing activity that resulted in a fault within a system or between systems ▪ Support injection and monitoring of state into the communication channels for use in testing, within a system or between systems ▪ Do not introduce needless nondeterminism

Data Model	Determine the major data abstractions that must be tested to ensure the correct operation of the system. ▪ Ensure that it is possible to capture the values of instances of these data abstractions ▪ Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created ▪ Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured
Mapping among Architectural Elements	Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, and modules to components) so that the desired test response is achieved and potential race conditions identified. In addition, determine whether it is possible to test for illegal mappings of architectural elements.
Resource Management	Ensure there are sufficient resources available to execute a test suite and capture the results. Ensure that your test environment is representative of (or better yet, identical to) the environment in which the system will run. Ensure that the system provides the means to do the following:

	▪ Test resource limits ▪ Capture detailed resource usage for analysis in the event of a failure ▪ Inject new resource limits into the system for the purposes of testing ▪ Provide virtualized resources for testing
Binding Time	Ensure that components that are bound later than compile time can be tested in the late-bound context. Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure. Ensure that the full range of binding possibilities can be tested.
Choice of Technology	Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on? Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.

Ahora que tu arquitectura está lista para ayudarte a probar. . .

Por Nick Rozanski, coautor (con Eoin Woods) de Arquitectura de sistemas *de* software: Trabajando con partes interesadas utilizando puntos de vista y perspectivas

Además de diseñar su sistema para que sea apto para pruebas, deberá superar dos desafíos más específicos y desalentadores al probar sistemas muy grandes o complejos, a saber, datos de prueba y automatización de pruebas.

Datos de prueba

Su primer desafío es cómo crear *conjuntos de datos de prueba* grandes, consistentes y útiles . Este es un problema importante en mi experiencia, especialmente para las pruebas de integración (es decir, probar varios componentes para confirmar que funcionan juntos correctamente) y las pruebas de rendimiento (confirmando queel sistema cumple con los requisitos de rendimiento, latencia y tiempo de respuesta). Para las pruebas unitarias, y generalmente para las pruebas de aceptación del usuario, los datos de la prueba generalmente se crean a mano.

Por ejemplo, es posible que necesite 50 productos, 100 clientes y 500 pedidos en su base de datos de prueba, de modo que pueda probar los pasos funcionales involucrados en la creación, modificación o eliminación de pedidos. Estos datos deben ser lo suficientemente variados para que las pruebas valgan la pena,

deben cumplir todas las reglas de integridad referencial y otras restricciones de su modelo de datos, y debe poder calcular y especificar los resultados esperados de las pruebas.

He visto, y he estado involucrado en, dos formas de hacer esto: usted escribe un sistema para generar sus datos de prueba, o captura un conjunto de datos representativos del entorno de producción y los anonimiza según sea necesario. (La anonimización de los datos de prueba implica eliminar cualquier información confidencial, como datos personales de personas u organizaciones, detalles financieros, etc.)

Crear sus propios datos de prueba es lo ideal, porque sabe qué datos está utilizando y puede asegurarse de que cubra todos sus casos de borde, pero es un gran esfuerzo. Capturar datos del entorno en vivo es más fácil, asumiendo que ya hay un sistema allí, pero no sabe qué datos y, por lo tanto, qué cobertura obtendrá, y es posible que tenga que tener mucho cuidado para cumplir con la privacidad y Legislación de protección de datos.

Esto puede tener un impacto en la arquitectura del sistema de varias maneras, y el arquitecto debe darle la debida atención desde el principio. Por ejemplo, el sistema puede necesitar capturar transacciones en vivo o tomar "instantáneas" de datos en vivo, que pueden usarse para generar datos de prueba. Además, el sistema de generación de datos de prueba puede necesitar una arquitectura propia.

Automatización de pruebas

Su segundo desafío es alrededor de la *automatización de pruebas* . En la práctica, no es posible probar sistemas grandes a mano debido a la cantidad de pruebas, su complejidad y la cantidad de verificación de resultados que se requiere. En el mundo ideal, crea un marco de automatización de prueba para hacer esto automáticamente, que alimenta con datos de prueba y configura la ejecución todas las noches, o incluso se ejecuta cada vez que se registra algo (el modelo de integración continua).

Esta es un área que recibe muy poca atención en muchos proyectos grandes de desarrollo de software. A menudo, no está presupuestado en el plan del proyecto, con el supuesto no escrito de que el esfuerzo necesario para construirlo puede, de alguna manera, “absorberse” en los costos de desarrollo. Un marco de automatización de prueba puede ser una cosa significativamente compleja en sí misma (¡lo que plantea la cuestión de cómo se prueba!). Debe estar dentro del alcance y planeado como cualquier otro proyecto entregable.

Debería prestarse la debida atención a cómo el marco invocará las funciones en el sistema que se está probando, en particular para probar las interfaces de usuario, que es casi sin excepción una pesadilla. (La ejecución de una prueba de interfaz de usuario depende en gran medida del diseño de las ventanas, el orden de los campos, etc., que generalmente cambia mucho en los sistemas muy centrados en el usuario. A veces es posible ejecutar los controles de las ventanas

mediante programación, pero en en el peor de los casos es posible que tenga que grabar y volver a reproducir las pulsaciones de teclado o los movimientos del mouse.)

Hay muchas herramientas para ayudar con esto hoy en día, como Quick Test Pro, TestComplete o Selenium para pruebas, y CruiseControl, Hudson y TeamCity para una integración continua. Una lista completa en la web se puede encontrar aquí: en.wikipedia.org/wiki/Test_automation .

10.4. RESUMEN

Asegurarse de que un sistema sea fácilmente comprobable tiene beneficios tanto en términos del costo de las pruebas como de la confiabilidad del sistema. Un vehículo usado a menudo para ejecutar las pruebas es el arnés de prueba. Los arneses de prueba son sistemas de software que encapsulan recursos de prueba, como casos de prueba e infraestructura de prueba, por lo que es fácil volver a aplicar las pruebas a través de iteraciones y es fácil aplicar la infraestructura de prueba a nuevos incrementos del sistema. Otro vehículo es la creación de casos de prueba antes del desarrollo de un componente, para que los desarrolladores sepan qué pruebas deben pasar sus componentes.

Controlar y observar el estado del sistema es una clase importante de tácticas de prueba. Proporcionar la capacidad de realizar una inyección de fallas, registrar el estado del sistema en partes clave del sistema, aislar el sistema de su entorno y abstraer varios recursos son tácticas diferentes para respaldar el control y la observación de un sistema y sus componentes.

Los sistemas complejos son difíciles de probar debido al gran espacio de estado en el que tienen lugar sus cálculos y al mayor número de interconexiones entre los elementos del sistema. En consecuencia, mantener el sistema simple es otra clase de tácticas que apoya la capacidad de prueba.