TESTABILIDAD

Las estimaciones de la industria indican que entre el 30 y el 50 por ciento (o en algunos casos, incluso más) del costo de desarrollar sistemas bien diseñados se asume 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 el software para demostrar sus fallas a través de pruebas (generalmente basadas en la ejecución). Específicamente, la probabilidad 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 se puede probar si "Renuncia" a sus defectos fácilmente. Si hay una falla en un sistema, queremos que falle durante la prueba lo más rápido posible. Por supuesto, calcular esta probabilidad no es fácil y, como verá cuando analicemos las medidas de respuesta para la comprobabilidad, otras medidas se utilizarán.

La figura 10.1 muestra un modelo de prueba en el que un programa procesa la entrada y produce 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 cuánto tiempo llevó producir La salida. La figura 10.1 también muestra que el estado interno del programa también puede ser mostrado al oráculo, y un oráculo puede decidir si eso es correcto o no, puede detectar si el programa ha entrado en un estado erróneo y generar un juicio en cuanto a la corrección del programa.

Establecer y examinar el estado interno de un programa es un aspecto de las pruebas que figurará de forma destacada en nuestras tácticas de testabilidad.

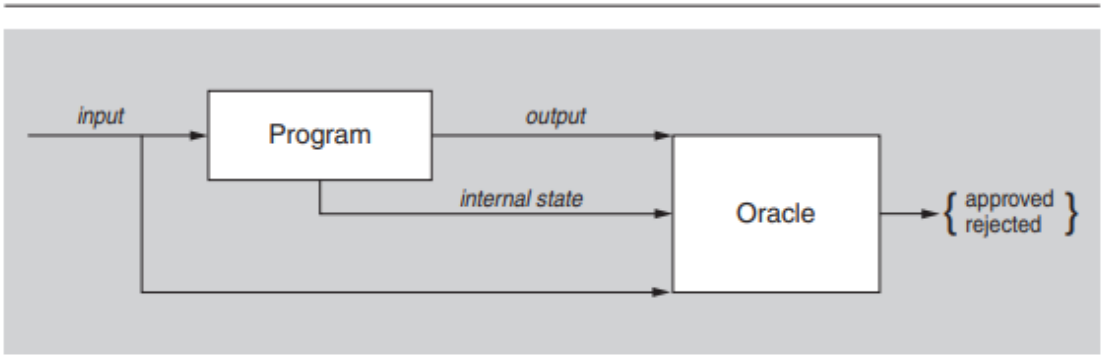

FIGURE 10.1 A model of testing

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

ayuda para ejecutar los procedimientos de prueba y registrar la salida. Un arnés de prueba puede ser una pieza sustancial de software por derecho propio, con su propia arquitectura, partes interesadas y requisitos de atributos de calidad.

Las pruebas las llevan a cabo varios desarrolladores, usuarios o personal de control de calidad. Se pueden probar partes del sistema o todo el sistema. La medida de respuesta para la capacidad de prueba se refiere a la eficacia de las pruebas para descubrir fallas y cuánto tiempo lleva realizar las pruebas hasta el nivel de cobertura deseado. Prueba los casos 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.

Ejército Simio de Netflix

Netflix distribuye películas y programas de televisión tanto en DVD como transmitiendo video. Su servicio de transmisión de video ha tenido un gran éxito. En mayo de 2011, la transmisión de video de Netflix representaba 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 utiliza lo que ellos llaman un "**Ejército Simio"** como parte de su proceso de prueba.

Comenzaron con un Chaos Monkey, que mata aleatoriamente los procesos en el sistema en funcionamiento. Esto permite el seguimiento del efecto de los procesos fallidos y brinda la capacidad de garantizar que el sistema no falle o sufra graves degradaciones como resultado de una falla en el proceso. Recientemente, el Chaos Monkey consiguió algunos amigos para ayudar en las pruebas.

Actualmente, el Ejército Simio de Netflix incluye estos:

■ El Latency Monkey induce retrasos artificiales en el cliente-servidor capa de comunicación para simular la degradación del servicio y medidas si los servicios ascendentes responden adecuadamente.

■ Conformity Monkey encuentra instancias que no se adhieren mejor practica y las cierra. Por ejemplo, si una instancia no pertenece a un grupo de escalado automático, no se escalará adecuadamente cuando la demanda aumenta.

■ El Doctor Monkey aprovecha las comprobaciones de estado que se ejecutan en cada instancia, así como monitorea otros signos externos de salud (por ejemplo, carga de CPU) para detectar instancias insalubres.

■ Janitor Monkey garantiza que el entorno de nube de Netflix esté corriendo libre de desorden y desperdicio. Busca recursos no utilizados y se deshace de ellos.

■ Security Monkey es una extensión de Conformity Monkey. Encuentra violaciones de seguridad o vulnerabilidades, como configuraciones incorrectas grupos de seguridad y termina las instancias infractoras. También asegura que todos los certificados SSL y de gestión de derechos digitales (DRM) son válidos y no se van a renovar.

■ El mono 10-18 (localización-internacionalización) detecta problemas de configuración y tiempo de ejecución en instancias que atienden a clientes en múltiples regiones geográficas, usando diferentes idiomas y caracteres conjuntos. El nombre 10-18 proviene de L10n-i18n, una especie de abreviatura del localización e internacionalización de palabras.

Algunos de los miembros del Ejército Simio usan inyección de fallas para colocar fallas en el sistema en ejecución de una manera controlada y monitoreada.

Otros miembros monitorean varios aspectos especializados del sistema y su medio ambiente. Ambas técnicas tienen una aplicabilidad más amplia que solo Netflix.

No todas las fallas son iguales en términos de gravedad. Debería hacerse más énfasis colocado en encontrar las fallas más severas que en encontrar otras fallas. Los Simian Army refleja la determinación de Netflix de que las fallas que buscan son los más graves en cuanto a su impacto.

Esta estrategia ilustra que algunos sistemas son demasiado complejos y adaptables para ser probados por completo, porque algunos de sus comportamientos son emergentes. Un aspecto de las pruebas en ese ámbito es el registro de datos operativos producidos por el sistema, de modo que cuando ocurran fallas, los datos registrados puedan ser analizados en el laboratorio para intentar reproducir las fallas. Desde el punto de vista arquitectónico, esto puede requerir mecanismos para acceder y registrar cierto estado del sistema. El ejército simio es unidireccional para descubrir y registrar el comportamiento en sistemas de este tipo.

—LB

La prueba de código es un caso especial de validación, que asegura que un artefacto diseñado satisface las necesidades de sus partes interesadas o es 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 prueba es la arquitectura. En este capítulo se refieren únicamente a la capacidad de prueba de un sistema en ejecución y de su código fuente.

Escenario general de testeabilidad

Ahora podemos describir el escenario general de testeabilidad.

■ **Fuente de estímulo**. La prueba es realizada por probadores unitarios, integración probadores o probadores de sistemas (en el lado de la organización en desarrollo), o probadores de aceptación y usuarios finales (del lado del cliente). La fuente podría ser humano o un probador automático.

■ **Estímulo**. Se ejecuta un conjunto de pruebas debido a la finalización de un incremento de codificación como una capa de clase o 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 el sistema completo es el artefacto que se está probando.

■ **Medio ambiente**. La prueba puede ocurrir en el tiempo de desarrollo, en el tiempo de compilación, en tiempo de implementación, o mientras el sistema está funcionando (quizás en uso rutinario). 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 tienen como objetivo representar la facilidad con la que un sistema sometido a prueba "abandona" sus fallas. Las medidas pueden incluir 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 más larga cadena de pruebas (una medida de la dificultad de realizar las pruebas), medidas de esfuerzo para realizar las pruebas, medidas de esfuerzo para encontrar realmente fallas, estimaciones de la probabilidad de encontrar fallas adicionales y el período de tiempo o cantidad de esfuerzo para preparar el entorno de prueba.

Tal vez una medida sea la facilidad con la que se puede llevar el sistema a un estado específico. Además, se pueden utilizar medidas de 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 del riesgo intentan calificar la gravedad de fallas encontradas (o por encontrar).

La figura 10.2 muestra un escenario concreto para la comprobabilidad. El probador de unidades completa una unidad de código durante el desarrollo y realiza una secuencia de prueba cuyos resultados

se capturan y eso brinda una cobertura de ruta del 85 por ciento dentro de las tres horas posteriores a la prueba.

La Tabla 10.1 enumera los elementos del escenario general que caracterizan testabilidad.

Tabla 10.1 Escenario general de probabilidad

Porción de escenario	**valores posibles**
FUENTES	Probadores de unidades de origen, probadores de integración, probadores de sistemas, aceptación probadores, usuarios finales, ya sea ejecutando pruebas manualmente o usando herramientas de prueba automatizadas
ESTÍMULO	Se ejecuta un conjunto de pruebas debido a la finalización de una codificación incremento como una capa de clase o servicio, la completada integración de un subsistema, la implementación completa del todo el sistema, o la entrega del sistema al cliente.
ENTORNO	Tiempo de diseño, tiempo de desarrollo, tiempo de compilación, tiempo de integración, tiempo de implementación, tiempo de ejecución
ARTEFACTOS	La parte del sistema que se está probando

RESPUESTA	Uno o más de los siguientes: ejecute el conjunto de pruebas y capture resultados, captura la actividad que dio lugar a la falla, control y monitorear el estado del sistema
MEDIDA **de** RESPUESTA	Uno o más de los siguientes: esfuerzo para encontrar una falla o clase de fallas, esfuerzo para lograr un porcentaje dado de espacio de estado cobertura, probabilidad de que la falla sea revelada por la siguiente prueba, tiempo para realizar pruebas, esfuerzo para detectar fallas, duración de cadena de dependencia más larga en prueba, tiempo de preparación entorno de prueba, reducción de la exposición al riesgo (tamaño (pérdida) × prob (pérdida)

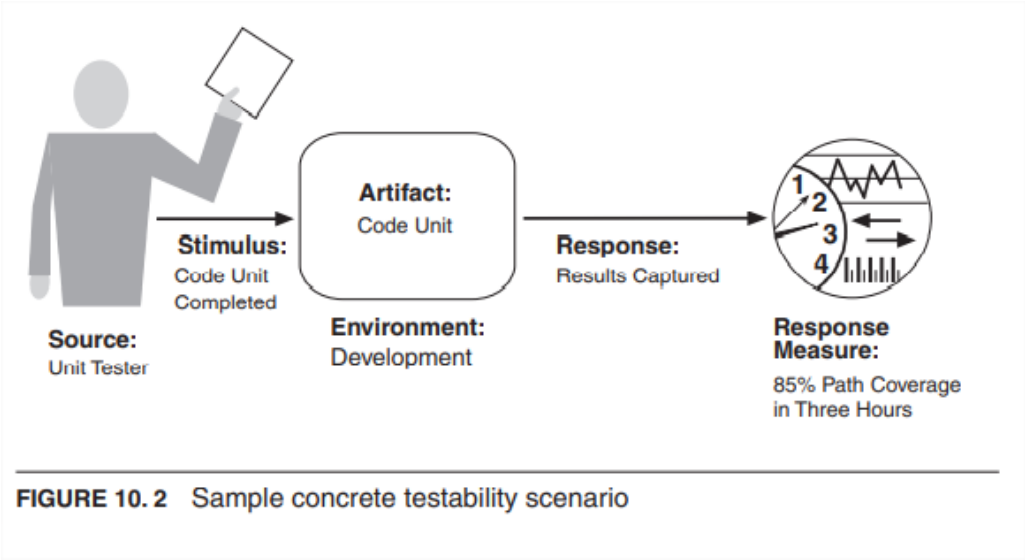

FIGURE 10. 2 Sample concrete testability scenario

10.2 TACTICAS DE TESTEABILIDAD

El objetivo de las tácticas para la capacidad de prueba es permitir una prueba más fácil cuando un incremento del desarrollo de software se completa. La figura 10.3 muestra el uso de tácticas para testabilidad. Las técnicas arquitectónicas 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 antes, cualquier cosa que el arquitecto pueda hacer para reducir el alto costo de las pruebas producirá una significativa beneficio.

Hay **dos categorías de tácticas para la prueba**. La primera categoría trata de agregar controlabilidad y observabilidad al sistema y la segunda limita la complejidad en el diseño del sistema.

Control y observación del estado del sistema

El control y la observación son tan fundamentales para la testeabilidad que algunos autores incluso definen testabilidad 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, déjelo hacer su trabajo y luego observe sus resultados. Sin embargo, el control y la observación del sistema de categoría de estado de tácticas de capacidad de prueba

proporciona información sobre el software que va más allá de sus entradas y salidas. Estas tácticas hacen que un componente mantenga algunos tipos de información de estado, permite a los evaluadores asignar un valor a esa información de estado, y / o hacer que esa información sea accesible a los probadores a pedido. La información del estado puede ser un estado operativo, el valor de alguna variable clave, el rendimiento de carga, pasos intermedios del proceso o cualquier otra cosa útil para recrear el componente comportamiento. Las **tácticas específicas incluyen las siguientes:**

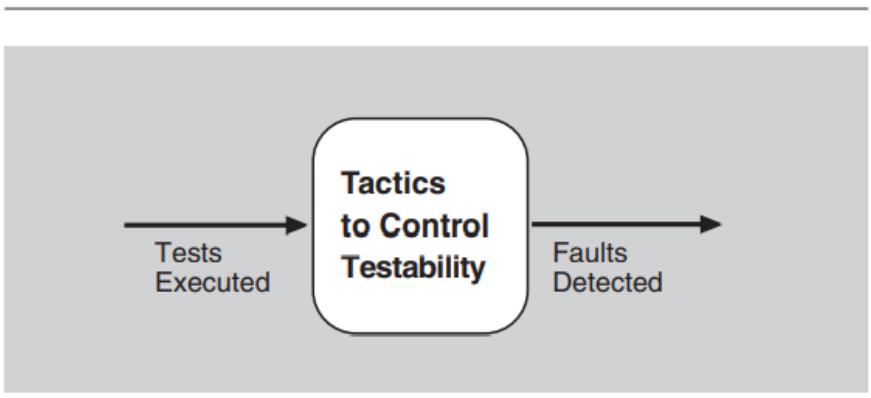

FIGURE 10.3 The goal of testability tactics

■ **Interfaces especializadas**. Tener interfaces de prueba especializadas le permite para controlar o capturar valores de variable para un componente ya sea a través de una prueba arnés o mediante ejecución normal. Ejemplos de rutinas de prueba especializadas incluir estos:

- ■ Un método **set y get** para variables, modos o atributos importantes (métodos que de otro modo podrían no estar disponibles excepto para pruebas propósitos)
- ■ Un método de **report** que devuelve el estado completo del objeto.
- ■ Un método **reset** para establecer el estado interno (por ejemplo, todos los atributos de un class) a un estado interno especificado
- ■ Un método para activar la salida detallada, varios niveles de registro de eventos, instrumentación de rendimiento o supervisión de recursos

Las interfaces y métodos de prueba especializados deben estar claramente identificados o mantenerse separados de los métodos de acceso y las interfaces para la funcionalidad requerida, de modo que puedan eliminarse si es necesario.

■ **Grabar / reproducir**. El estado que causó una falla a menudo es difícil de recrear. Registrar el estado cuando cruza una interfaz permite utilizar ese estado para "reproducir el sistema" y recrear la falla. Se refiere a grabación / reproducción tanto para capturar información que cruza una interfaz como para usarla como entrada para pruebas adicionales.

■ **Localizar el estado del almacenamiento**. 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 estado del almacenamiento (es decir, para que pueda ser manipulado a través de las características de la interfaz) es utilizar una máquina de estado como el mecanismo para rastrear e informar el estado actual.

■ **Fuentes de datos abstractas**. Similar a controlar el estado de un programa, controlar fácilmente sus datos de entrada hace que sea más fácil de probar. La abstracción de las interfaces permite sustituye 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 en su lugar, pruebe los datos, sin tener que cambiar su código funcional.

■ **Sandbox**. "Sandboxing" se refiere a aislar una instancia del sistema del mundo real para permitir una experimentación que no esté restringida por la preocupación sobre tener que deshacer las consecuencias del experimento. Esto puede utilizarse para análisis de escenarios, formación y simulación.

Una forma común de sandboxing es virtualizar recursos. Probando un sistema a menudo implica interactuar con recursos cuyo comportamiento está fuera el control del sistema. Con una caja de arena, puede crear una versión del recurso cuyo comportamiento está bajo su control. Por ejemplo, el comportamiento del reloj normalmente no está bajo nuestro control, incrementa un segundo cada segundo, lo que significa que si queremos hacer que el sistema cree que es medianoche de día,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 ejecuten más rápido que el reloj de pared. Se pueden realizar virtualizaciones para otros recursos, como memoria, batería, red, etc. Los stubs, simulacros y la inyección de dependencia son simples, pero formas efectivas de virtualización.

■ **Aserciones ejecutables.** Usando esta táctica, las afirmaciones son (generalmente) codificadas a mano y colocados en las ubicaciones deseadas para indicar cuándo y dónde se encuentra en un programa un estado de falla. Las afirmaciones a menudo están diseñadas para verificar que los valores de los datos satisfacen las restricciones especificadas. Las afirmaciones se definen en términos de declaraciones de especificaciones de datos, y deben colocarse donde los valores de los datos son referenciados o modificados. Las afirmaciones se pueden expresar como condiciones previas y posteriores para cada método y también como invariantes a nivel de clase. Esto da como resultado un aumento observabilidad, cuando una aserción se marca como fallida. Cada vez que un objeto de ese tipo es modificado, el código de comprobación se ejecuta automáticamente si se viola alguna condición. En la medida en que las afirmaciones cubren los casos de prueba, incorporan efectivamente el oráculo de prueba en el código— asumiendo que las afirmaciones son correctas y están codificadas correctamente.

Todas estas tácticas agregan capacidad o abstracción al software que (no estamos interesados en probar) de lo contrario no estaría allí. Se puede considerar que reemplacen el software básico por un software más elaborado que tiene campanas y silbidos para probar. Hay una serie de técnicas para realizar este reemplazo. Éstas no son tácticas de testeabilidad, per se, sino **técnicas para reemplazar un componente con una versión diferente de sí mismo.** Incluyen lo siguiente:

■**Reemplazo de componentes,** que simplemente intercambia la implementación de un componente con una implementación diferente que (en el caso de capacidad de prueba) tiene características que facilitan las pruebas. El reemplazo de componentes es a menudo logrado en los scripts de compilación de un sistema.

■ **Macros de preprocesador** que, cuando se activan, se expanden al código de informe de estado o activar declaraciones de sondeo que devuelven o muestran información, o devuelven el control a una consola de pruebas.

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

Limitar la complejidad

El software complejo es más difícil de probar. Esto se debe a que, según la definición de complejidad, su espacio de estados operativos es muy grande y (en igualdad de condiciones) es más difícil recrear un estado exacto en un espacio de estados grande que hacerlo en un espacio de estados pequeños. Porque probar no se trata solo de hacer que el software falle, sino de encontrar la falla que causó la falla para que pueda ser eliminada, a menudo nos preocupamos con hacer que el comportamiento sea repetible.

Esta categoría tiene tres tácticas:

■ **Limite la complejidad estructural.** Esta táctica incluye evitar o resolver dependencias cíclicas entre componentes, aislando y encapsulando dependencias del entorno externo y reducción de dependencias entre componentes en general .En sistemas orientados a objetos, puede simplificar la jerarquía de herencia: Limite el número de clases del cual se deriva una clase, o el número de clases derivadas de un clase. Limite la profundidad del árbol de herencia y el número de hijos de una clase. Limite 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 recuento del número de métodos de C más el número de métodos de otras clases que son invocados por el métodos de C. Mantener esta métrica baja puede aumentar la capacidad de prueba.

Tener alta cohesión, acoplamiento flexible y separación de preocupaciones, todos tácticas de modificabilidad (consulte el Capítulo 7): también pueden ayudar con la capacidad de prueba. La separación de preocupaciones puede ayudar a lograr controlabilidad y capacidad de observación (además de reducir el tamaño del espacio de estado del programa en general).

La controlabilidad es fundamental para hacer que las pruebas sean manejables.

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

Finalmente, algunos estilos arquitectónicos se prestan a la prueba en un estilo en capas, puede probar las capas inferiores primero, luego probar las capas superiores con confianza en las capas inferiores.

■ **Limite el no determinismo**. La contraparte para limitar la complejidad estructural está limitando 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 deterministas.

Esta táctica implica encontrar todas las fuentes del no determinismo, como paralelismo sin restricciones, y eliminándolos tanto como sea 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 grabación / reproducción) están disponibles.

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

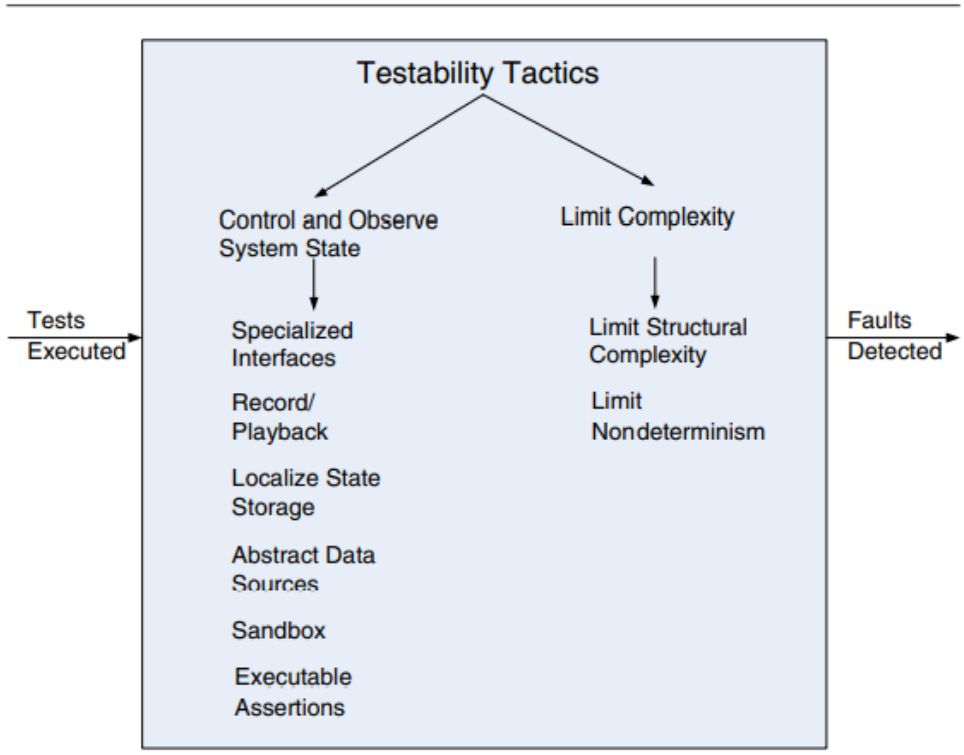

FIGURE 10.4 Testability tactics