7. Modificabilidad

Adaptar o perecer, ahora como siempre, es el imperativo inexorable de la naturaleza.
—HG Wells

El cambio sucede.

Estudio tras estudio muestra que la mayor parte del costo del sistema de software típico ocurre después de su lanzamiento inicial. Si el cambio es la única constante en el universo, entonces el cambio de software no solo es constante sino omnipresente. Los cambios ocurren para agregar nuevas funciones, para cambiar o incluso retirar las antiguas. Los cambios suceden para corregir defectos, reforzar la seguridad o mejorar el rendimiento. Los cambios suceden para mejorar la experiencia del usuario. Los cambios ocurren para adoptar nuevas tecnologías, nuevas plataformas, nuevos protocolos, nuevos estándares. Los cambios suceden para hacer que los sistemas funcionen juntos, incluso si nunca fueron diseñados para hacerlo.

La modificabilidad tiene que ver con el cambio, y nuestro interés en él se centra en el costo y el riesgo de realizar cambios. Para planificar la modificabilidad, un arquitecto debe considerar cuatro preguntas:

• *¿Qué puede cambiar?* Se puede producir un cambio en cualquier aspecto de un sistema: las funciones que el sistema computa, la plataforma (el hardware, el sistema operativo, el middleware), el entorno en el que opera el sistema (los sistemas con los que debe interactuar, los protocolos que utiliza). para comunicarse con el resto del mundo), las cualidades que exhibe el sistema (su rendimiento, su confiabilidad e incluso sus futuras modificaciones) y su capacidad (número de usuarios admitidos, número de operaciones simultáneas).

• *¿Cuál es la probabilidad del cambio?* Uno no puede planificar un sistema para todos los cambios potenciales; el sistema nunca se haría, o si se hiciera, sería demasiado caro y probablemente sufriría problemas de atributos de calidad en otras dimensiones. Aunque cualquier cosa *puede* cambiar, el arquitecto tiene que tomar las decisiones difíciles sobre qué cambios son probables y, por lo tanto, qué cambios se deben respaldar y cuáles no.

• *¿Cuándo se hace el cambio y quién lo hace?*Más comúnmente en el pasado, se hizo un cambio al código fuente. Es decir, un desarrollador tuvo que hacer el cambio, que se probó y luego se implementó en una nueva versión. Ahora, sin embargo, la pregunta de cuándo se hace un cambio está entrelazada con la pregunta de quién lo hace. Un usuario final que cambia el protector de pantalla está haciendo un cambio en uno de los aspectos del sistema. Igualmente claro, no está en la misma categoría que cambiar el sistema para que se pueda utilizar a través de la web en lugar de hacerlo en una sola máquina. Se pueden realizar cambios en la implementación (modificando el código fuente), durante la compilación (utilizando conmutadores de tiempo de compilación), durante la compilación (mediante la selección de bibliotecas), durante la configuración de la configuración (mediante un rango de técnicas, incluida la configuración de parámetros), o Durante la ejecución (por parámetros de configuración, complementos, etc.). Un desarrollador, un usuario final, puede hacer un cambio.

• *¿Cuál es el costo del cambio?* Hacer un sistema más modificable implica dos tipos de costos:

• El costo de introducir el (los) mecanismo (s) para hacer que el sistema sea más modificable

• El costo de realizar la modificación utilizando el (los) mecanismo (s)

Por ejemplo, el mecanismo más simple para hacer un cambio es esperar a que llegue una solicitud de cambio, luego cambiar el código fuente para acomodar la solicitud. El costo de introducir el mecanismo es cero; El costo de ejercerlo es el costo de cambiar el código fuente y revalidar el sistema. En el otro extremo del espectro se encuentra un generador de aplicaciones, como un generador de interfaces de usuario. El constructor toma como entrada una descripción de la interfaz de usuario del diseñador producida a través de técnicas de manipulación directa y produce (generalmente) el código fuente. El costo de introducir el mecanismo es el costo de construir el generador de IU, que puede ser considerable. El costo de usar el mecanismo es el costo de producir la entrada para alimentar al constructor (el costo puede ser sustancial o insignificante), el costo de ejecutar el constructor (aproximadamente cero),

Para *N* modificaciones similares, una justificación simplificada para un mecanismo de cambio es que

N × Costo de realizar el cambio sin el mecanismo ≤ Costo de instalar el mecanismo + (N × Costo de realizar el cambio utilizando el mecanismo).

N es el número anticipado de modificaciones que utilizará el mecanismo de modificabilidad, pero N es una predicción. Si entran menos cambios de los esperados, es posible que no se justifique un mecanismo de modificación costoso. Además, el costo de crear el mecanismo de modificabilidad podría aplicarse en otro lugar: para agregar funcionalidad, mejorar el rendimiento o incluso en inversiones que no son de software, como la compra de acciones tecnológicas. Además, la ecuación no tiene en cuenta el tiempo. EsoPuede que a la larga sea más barato construir un sofisticado mecanismo de manejo de cambios, pero es posible que no pueda esperar por eso.

7.1. ESCENARIO GENERAL DE MODIFICABILIDAD

A partir de estas consideraciones, podemos ver las partes del escenario general de modificabilidad:

• *Fuente de estímulo*. Esta parte especifica quién realiza el cambio: el desarrollador, un administrador del sistema o un usuario final.

• *Estímulo* . Esta parte especifica el cambio a realizar. Un cambio puede ser la adición de una función, la modificación de una función existente o la eliminación de una función. (Para esta categorización, consideramos que corregir un defecto es cambiar una función, que presumiblemente no funcionó correctamente como resultado del defecto). También se puede hacer un cambio en las cualidades del sistema: hacerlo más sensible, aumentar su capacidad de respuesta. disponibilidad, y así sucesivamente. La capacidad del sistema también puede cambiar. Acomodar a un número creciente de usuarios simultáneos es un requisito frecuente. Finalmente, pueden ocurrir cambios para adaptarse a nuevas tecnologías de algún tipo, la más común de las cuales es la de trasladar el sistema a un tipo diferente de computadora o red de comunicación.

• *Artefacto* . Esta parte especifica qué se va a cambiar: componentes o módulos específicos, la plataforma del sistema, su interfaz de usuario, su entorno u otro sistema con el que interactúa.

• *Medio ambiente* . Esta parte especifica cuándo se puede realizar el cambio: tiempo de diseño, tiempo de compilación, tiempo de compilación, tiempo de inicio o tiempo de ejecución.

• *Respuesta* . Realice el cambio, pruébelo y despláce lo.

• *Medida de respuesta*. Todas las respuestas posibles toman tiempo y cuestan dinero; El tiempo y el dinero son las medidas de respuesta más comunes. Aunque ambos suenan simples de medir, no lo son. Puedes medir el tiempo del calendario o el tiempo del personal. ¿Pero mide el tiempo que tarda el cambio en abrirse camino a través de los paneles de control de configuración y las autoridades de aprobación (algunos de los cuales pueden estar fuera de su organización), o simplemente el tiempo que tardan los ingenieros en realizar el cambio? El costo generalmente implica un desembolso directo, pero también puede incluir el costo de oportunidad de que su personal trabaje en los cambios en lugar de otras tareas. Otras medidas incluyen la extensión del cambio (número de módulos u otros artefactos afectados) o el número de nuevos defectos introducidos por el cambio, o el efecto en otros atributos de calidad. Si el cambio está siendo realizado por un usuario,Capítulo 11).

La Figura 7.1 ilustra un escenario concreto de modificabilidad: el desarrollador desea cambiar la interfaz de usuario modificando el código en el momento del diseño. Las modificaciones se realizan sin efectos secundarios dentro de las tres horas.

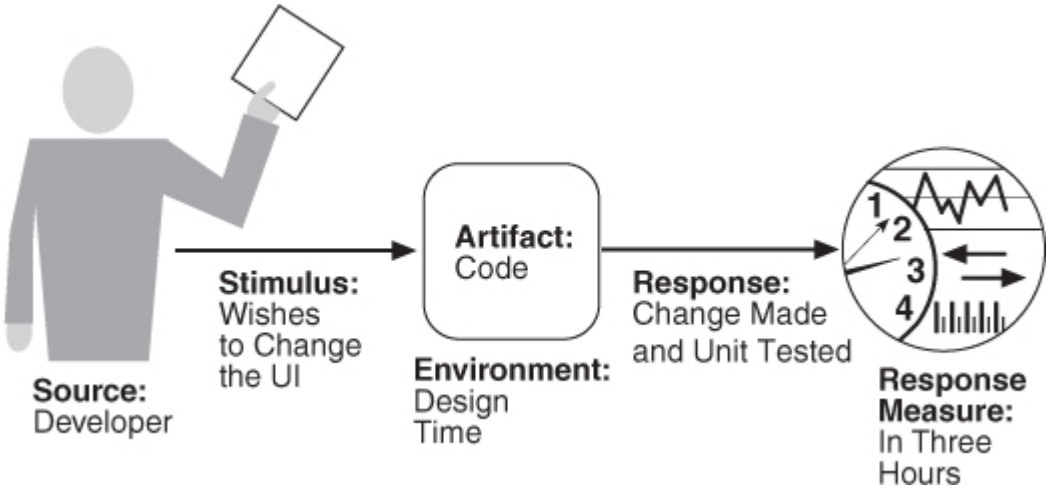

Figura 7.1. Ejemplo de escenario de modificabilidad concreto

La Tabla 7.1 enumera los elementos del escenario general que caracterizan la modificabilidad.

Tabla 7.1. Escenario general de modificabilidad

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

7.2. TÁCTICAS PARA LA MODIFICABILIDAD

Las tácticas para controlar la modificabilidad tienen como objetivo controlar la complejidad de los cambios, así como el tiempo y el costo de los cambios. La figura 7.2 muestra esta relación.

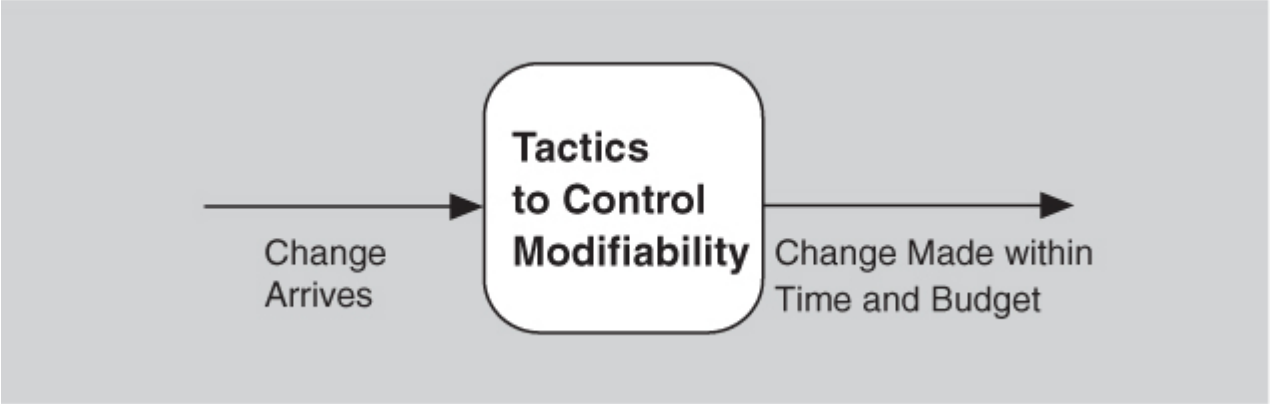

Figura 7.2. El objetivo de las tácticas de modificabilidad.

Para entender la modificabilidad, comenzamos con el acoplamiento y la cohesión.

Los módulos tienen responsabilidades. Cuando un cambio hace que se modifique un módulo, sus responsabilidades se modifican de alguna manera. En general, un cambio que afecta a un módulo es más fácil y menos costoso que si cambia más de un módulo. Sin embargo, si las responsabilidades de dos módulos se superponen de alguna manera, entonces un solo cambio puede afectar a ambos. Podemos medir esta superposición midiendo la probabilidad de que una modificación de un módulo se propague al otro. Esto se llama *acoplamiento* , y el acoplamiento alto es un enemigo de la modificabilidad.

La cohesión mide la fuerza con que se relacionan las responsabilidades de un módulo. Informalmente, mide la "unidad de propósito" del módulo. La unidad de propósito puede medirse por los escenarios de cambio que afectan a un módulo. La cohesión de un módulo es la probabilidad de que un escenario de cambio que afecte a una responsabilidad también afecte a otras responsabilidades (diferentes). Cuanto mayor sea la cohesión, menor será la probabilidad de que un cambio dado afecte a múltiples responsabilidades. La alta cohesión es buena; La baja cohesión es mala. La definición permite que dos módulos con propósitos similares sean cohesivos.

Dado este marco, ahora podemos identificar los parámetros que usaremos para motivar las tácticas de modificabilidad:

• *Tamaño de un módulo* . Las tácticas que dividen los módulos reducirán el costo de realizar una modificación en el módulo que se está dividiendo siempre que se elija la división para reflejar el tipo de cambio que es probable que se realice.

• *Acoplamiento* . La reducción de la resistencia del acoplamiento entre dos módulos A y B reducirá el costo esperado de cualquier modificación que afecte a A. Las tácticas que reducen el acoplamiento son aquellas que colocan intermediarios de diversos tipos entre los módulos A y B.

• *Cohesión* . Si el módulo A tiene una cohesión baja, la cohesión puede mejorarse eliminando las responsabilidades que no se ven afectadas por los cambios anticipados.

Finalmente, debemos preocuparnos por cuándo ocurre un cambio en el ciclo de vida del desarrollo del software. Si ignoramos el costo de preparar la arquitectura para la modificación, preferimos que un cambio se limite lo más tarde posible. Los cambios solo se pueden realizar con éxito (es decir, rápidamente y al menor costo) al final del ciclo de vida si la arquitectura está adecuadamente preparada para

adaptarse a ellos. Así, el cuarto y último parámetro en un modelo de modificabilidad es este:

• *Tiempo de modificación de la encuadernación* . Una arquitectura que esté adecuadamente equipada para adaptarse a las modificaciones tardías en el ciclo de vida costará, en promedio, menos que una arquitectura que obligue a realizar la misma modificación antes. La preparación del sistema significa que algunos costos serán cero, o muy bajos, para las modificaciones tardías del ciclo de vida. Esto, sin embargo, descuida el costo de preparar la arquitectura para el enlace tardío.

Ahora podemos entender que las tácticas y sus consecuencias afectan a uno o más de los parámetros anteriores: reducir el tamaño de un módulo, aumentar la cohesión, reducir el acoplamiento y aplazar el tiempo de enlace. Estas tácticas se muestran en la Figura 7.3 .

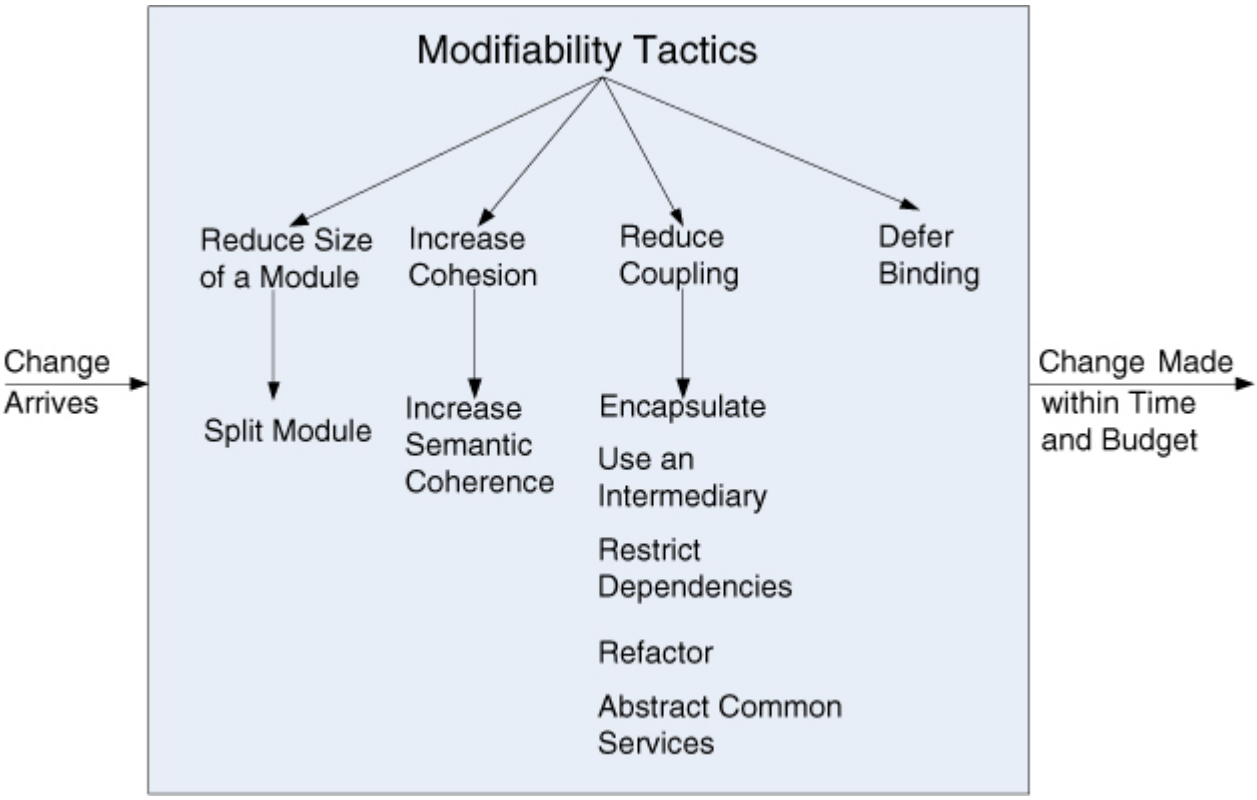

Figura 7.3. Tácticas de modificabilidad

Reducir el tamaño de un módulo

• *Split módulo* . Si el módulo que se está modificando incluye una gran capacidad, los costos de modificación probablemente serán altos. La

refinación del módulo en varios módulos más pequeños debería reducir el costo promedio de los cambios futuros.

Aumentar la cohesión

Varias tácticas involucran mover responsabilidades de un módulo a otro. El propósito de mover una responsabilidad de un módulo a otro es reducir la probabilidad de que los efectos secundarios afecten otras responsabilidades en el módulo original.

• *Incrementar la coherencia semántica* . Si las responsabilidades A y B en un módulo no tienen el mismo propósito, deben ubicarse en módulos diferentes. Esto puede implicar la creación de un nuevo módulo o la transferencia de una responsabilidad a un módulo existente. Un método para identificar las responsabilidades que se deben mover es formular hipótesis de posibles cambios que afectan a un módulo. Si algunas responsabilidades no se ven afectadas por estos cambios, entonces esas responsabilidades probablemente deberían eliminarse.

Reducir el acoplamiento

Ahora pasamos a las tácticas que reducen el acoplamiento entre los módulos.

• *Encapsular*.La encapsulación introduce una interfaz explícita a un módulo. Esta interfaz incluye una interfaz de programación de aplicaciones (API) y sus responsabilidades asociadas, como "realizar una transformación sintáctica en un parámetro de entrada a una representación interna". Tal vez la táctica de modificabilidad más común, la encapsulación reduce la probabilidad de que se propague un cambio en un módulo. a otros módulos. Las fortalezas de acoplamiento que anteriormente correspondían al módulo ahora van a la interfaz del módulo. Sin embargo, estas fortalezas se reducen porque la interfaz limita las formas en que las responsabilidades externas pueden interactuar con el módulo (quizás a través de una envoltura). Las responsabilidades externas ahora solo pueden interactuar directamente con el módulo a través de la interfaz expuesta (interacciones indirectas, sin embargo, como la dependencia de la calidad del servicio, probablemente permanecerá sin cambios). Las interfaces diseñadas para aumentar la modificabilidad deben ser abstractas con respecto a los detalles del módulo que pueden cambiar, es decir, deben ocultar esos detalles.

• *Utilizar un intermediario* rompe una dependencia. Dada una dependencia entre la responsabilidad A y la responsabilidad B (por ejemplo, llevar a cabo A primero requiere llevar a cabo B), la dependencia se puede romper utilizando un intermediario. El tipo de intermediario depende del tipo de dependencia. Por ejemplo, un intermediario de publicación / suscripción eliminará el conocimiento del productor de datosde sus consumidores. También lo hará un repositorio de datos compartido, que separa a los lectores de una parte de los datos de los escritores de esos datos. En una arquitectura orientada a servicios en la que los servicios se descubren entre sí mediante una búsqueda dinámica, el servicio de directorio es un intermediario.

• *Restringir dependencias* es una táctica que restringe los módulos con los que un módulo determinado interactúa o depende de ellos. En la práctica, esta táctica se logra al restringir la visibilidad de un módulo (cuando los desarrolladores no pueden ver una interfaz, no pueden emplearla) y por autorización (restringiendo el acceso solo a los módulos autorizados). Esta táctica se ve en arquitecturas en capas, en las que una capa solo puede usar capas inferiores (a veces solo la siguiente capa inferior) y en el uso de envoltorios, donde las entidades externas solo pueden ver (y, por lo tanto, depender de) la envoltura y no La funcionalidad interna que envuelve.

• *Refactor* es una táctica que se realiza cuando dos módulos se ven afectados por el mismo cambio porque son duplicados (al menos parciales) el uno del otro. La refactorización de códigos es una práctica fundamental de los proyectos de desarrollo Agile, como un paso de limpieza para asegurarse de que los equipos no hayan producido códigos duplicados o demasiado complejos; sin embargo, el concepto se aplica también a los elementos arquitectónicos. Las responsabilidades comunes (y el código que las implementa) se "excluyen" de los módulos donde existen y se les asigna un hogar propio apropiado. Al ubicar conjuntamente las responsabilidades comunes, es decir, al convertirlas en submódulos del mismo módulo principal, el arquitecto puede reducir el acoplamiento.

• *Resumen de servicios comunes* . En el caso de que dos módulos proporcionen servicios no iguales pero similares, puede ser rentable implementar los servicios solo una vez de forma más general (abstracta). Cualquier modificación al servicio (común) tendría que ocurrir en un solo lugar, reduciendo los costos de modificación. Una forma común de introducir una abstracción es parametrizar la

descripción (y la implementación) de las actividades de un módulo. Los parámetros pueden ser tan simples como los valores para variables clave o tan complejos como las declaraciones en un lenguaje especializado que se interpretan posteriormente.

Aplazamiento de encuadernación

Debido a que el trabajo de las personas casi siempre es más costoso que el trabajo de las computadoras, permitir que las computadoras manejen un cambio tanto como sea posible casi siempre reducirá el costo de hacer ese cambio. Si diseñamos artefactos con flexibilidad incorporada, ejercer esa flexibilidad suele ser más barato que codificar a mano un cambio específico.

Los parámetros son quizás el mecanismo más conocido para introducir flexibilidad, y eso recuerda la táctica abstracta de servicios comunes. Una función parametrizada $f(a, b)$ es más general que la función similar $f(a)$ que asume $b = 0$. Cuando vinculamos el valor de algunos parámetros en una fase diferente en el ciclo de vida que la que definimos Parámetros, estamos aplicando la táctica de vinculación diferida.

En general, cuanto más tarde en el ciclo de vida podamos vincular valores, mejor. Sin embargo, poner en marcha los mecanismos para facilitar esa vinculación tardía tiende a ser más costoso, otra compensación. Y así, la ecuación en la página 118 entra en juego. Queremos unirnos lo más tarde posible, siempre y cuando el mecanismo que lo permite sea rentable.

Las tácticas para vincular valores en tiempo de compilación o tiempo de construcción incluyen estas:

- Reemplazo de componentes (por ejemplo, en un script de compilación o makefile)
- Parametrización en tiempo de compilación
- Aspectos

Las tácticas para vincular valores en el momento del despliegue incluyen esto:

- Enlace de configuración en tiempo

Las tácticas para vincular valores en el momento de inicio o inicialización incluyen esto:

- Archivos de recursos

Las tácticas para vincular valores en tiempo de ejecución incluyen estas:

- Registro en tiempo de ejecución
- Búsqueda dinámica (por ejemplo, para servicios)
- Interpretar parámetros
- Tiempo de inicio vinculante
- Servidores de nombres
- Plug-ins
- Publicación-suscripción
- Repositorios compartidos
- polimorfismo

Separar la construcción de un mecanismo de modificabilidad del uso del mecanismo para realizar una modificación admite la posibilidad de que diferentes partes interesadas estén involucradas: una parte interesada (generalmente un desarrollador) para proporcionar el mecanismo y otra persona interesada (un instalador, por ejemplo, o un usuario) para ejercer Más tarde, posiblemente en una fase de ciclo de vida completamente diferente. Instalar un mecanismo para que alguien más pueda realizar un cambio en el sistema sin tener que cambiar ningún código a veces se denomina *externalizar* el cambio.

7.3. UNA LISTA DE VERIFICACIÓN DE DISEÑO PARA LA MODIFICABILIDAD

La Tabla 7.2 es una lista de verificación para respaldar el proceso de diseño y análisis para la modificabilidad.

Tabla 7.2. Lista de verificación para apoyar el proceso de diseño y análisis para la modificabilidad

Category	Checklist
Allocation of Responsibilities	Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes: ▪ Determine the responsibilities that would need to be added, modified, or deleted to make the change. ▪ Determine what responsibilities are impacted by the change. ▪ Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules.
Coordination Model	Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number set of modules. Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules. For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish-subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast.
Data Model	Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction.

	For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties. For each potential change or category of change: ▪ Determine which data abstractions would need to be added, modified, or deleted to make the change. ▪ Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions. ▪ Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, their creation, initialization, persistence, manipulation, translation, or destruction. ▪ Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes. Design your data model so that items allocated to each element of the data model are likely to change together.
Mapping among Architectural Elements	Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of the following, for example: ▪ Execution dependencies ▪ Assignment of data to databases ▪ Assignment of runtime elements to processes, threads, or processors

	Ensure that such changes are performed with mechanisms that utilize deferred binding of mapping decisions.
Resource Management	Determine how the addition, deletion, or modification of a responsibility or quality attribute will affect resource usage. This involves, for example: ▪ Determining what changes might introduce new resources or remove old ones or affect existing resource usage ▪ Determining what resource limits will change and how Ensure that the resources after the modification are sufficient to meet the system requirements. Encapsulate all resource managers and ensure that the policies implemented by those resource managers are themselves encapsulated and bindings are deferred to the extent possible.
Binding Time	For each change or category of change: ▪ Determine the latest time at which the change will need to be made. ▪ Choose a defer-binding mechanism (see Section 7.2) that delivers the appropriate capability at the time chosen. ▪ Determine the cost of introducing the mechanism and the cost of making changes using the chosen mechanism. Use the equation on page 118 to assess your choice of mechanism. ▪ Do not introduce so many binding choices that change is impeded because the dependencies among the choices are complex and unknown.
Choice of Technology	Determine what modifications are made easier or harder by your technology choices. ▪ Will your technology choices help to make, test, and deploy modifications? ▪ How easy is it to modify your choice of technologies (in case some of these technologies change or become obsolete)? Choose your technologies to support the most likely modifications. For example, an enterprise service bus makes it easier to change how elements are connected but may introduce vendor lock-in.

7.4. RESUMEN

La modificabilidad se relaciona con el cambio y el costo en tiempo o dinero de realizar un cambio, incluida la medida en que esta modificación afecta otras funciones o atributos de calidad.

Los desarrolladores, instaladores o usuarios finales pueden realizar cambios, y estos cambios deben prepararse para. Hay un costo de preparación para el cambio, así como un costo de hacer un cambio. Las tácticas de modificabilidad están diseñadas para prepararse para los cambios posteriores.

Las tácticas para reducir el costo de realizar un cambio incluyen hacer módulos más pequeños, aumentar la cohesión y reducir el acoplamiento. Aplazar la vinculación también reducirá el costo de hacer un cambio.

Reducir el acoplamiento es una categoría estándar de tácticas que incluye encapsular, usar un intermediario, restringir dependencias, ubicar responsabilidades relacionadas, refactorizar y abstraer servicios comunes.

El aumento de la cohesión es otra táctica estándar que implica la separación de responsabilidades que no sirven para el mismo propósito.

El enlace diferido es una categoría de tácticas que afecta el tiempo de construcción, el tiempo de carga, el tiempo de inicialización o el tiempo de ejecución.