INFORME
UNIDAD 1

Capítulo 27: "Ingeniería del Software Basada en Componentes"
Libro: "Ingeniería de Software: Un Enfoque Práctico" (5ta edición)
Autor: Roger S. Pressman

Asignatura: Ingeniería de Software II

Estudiante: Franthony Sánchez

Profesor/a: Natanael Geronimo Mena

Fecha: 01/02/2026

ÍNDICE

• Introducción

• 1. Concepto de Ingeniería de Software Basada en Componentes (ISBC)

• 2. ¿Qué es un componente de software? (estructura, interfaces y encapsulamiento)

• 3. Reutilización y componentes comerciales (COTS/CYD): selección y evaluación

• 4. Proceso de desarrollo basado en componentes

• 5. Diseño arquitectónico e integración: adaptación, ensamblaje y middleware

• 6. Calidad, riesgos y gestión del ciclo de vida de componentes

• 7. Ejemplo aplicado: cómo se implementan requerimientos con componentes

• Conclusión

• Bibliografía

Introducción

En el desarrollo de software moderno, una de las decisiones más importantes es determinar cuánto se construirá desde cero y cuánto se integrará a partir de piezas ya existentes. El capítulo 27 del libro de Pressman aborda la Ingeniería de Software Basada en Componentes (ISBC), un enfoque que propone construir sistemas a partir de componentes reutilizables, ya sean internos (desarrollados por la organización) o comerciales (COTS/CYD). Este paradigma cobra relevancia en proyectos donde el tiempo de entrega, la calidad, la escalabilidad y el mantenimiento son factores críticos.

El presente informe resume los conceptos centrales del capítulo y los organiza de manera práctica: definición de componente, rol de las interfaces, criterios para evaluar componentes de terceros, actividades del proceso basado en componentes, y aspectos de calidad y riesgo durante la integración. Se busca mostrar cómo este enfoque ayuda a mejorar productividad y a disminuir retrabajo, pero también qué desafíos introduce (compatibilidad, dependencia de proveedores, adaptación, licenciamiento y seguridad).

1. Concepto de Ingeniería de Software Basada en Componentes (ISBC)

La Ingeniería de Software Basada en Componentes es un enfoque de construcción de sistemas donde el producto final se obtiene ensamblando unidades de software preexistentes. En lugar de diseñar e implementar cada funcionalidad desde cero, el equipo identifica componentes que ya implementan parte de los requerimientos y los integra dentro de una arquitectura.

Pressman resalta que la motivación principal es lograr mejoras en costo, tiempo y calidad. Si una parte del sistema ya existe y ha sido utilizada en otros contextos, reutilizarla puede reducir el número de defectos y acelerar el desarrollo. Sin embargo, este beneficio no es automático:

depende de la capacidad de evaluar componentes, comprender sus interfaces y gestionar la integración de manera controlada.

En términos prácticos, ISBC se apoya en tres ideas clave: (1) componentes con comportamiento bien definido, (2) interfaces explícitas que actúan como contrato de uso, y (3) una arquitectura que permita incorporar piezas heterogéneas sin perder coherencia del sistema.

2. ¿Qué es un componente de software? (estructura, interfaces y encapsulamiento)

Un componente es una unidad de software que encapsula una funcionalidad y puede desplegarse o integrarse como una pieza relativamente independiente. Su rasgo más importante es que expone una o varias interfaces públicas (lo que ofrece) y, en ocasiones, declara también interfaces requeridas (lo que necesita de otros) para funcionar.

La interfaz es el punto de contacto: define operaciones, datos de entrada/salida, formatos, precondiciones y postcondiciones. Idealmente, la implementación interna permanece oculta, lo cual favorece la mantenibilidad: el componente puede evolucionar internamente sin romper a los consumidores, siempre que mantenga el contrato.

En ISBC se busca bajo acoplamiento y alta cohesión. Bajo acoplamiento significa que un componente depende lo menos posible de detalles internos de otros; alta cohesión implica que el componente mantiene un propósito claro y no mezcla responsabilidades. Cuando estas propiedades se cumplen, el sistema se vuelve más fácil de extender, probar y reemplazar por partes.

Otro concepto relacionado es la sustituibilidad. Si dos componentes cumplen la misma interfaz, el sistema puede cambiar de uno a otro con un impacto mínimo. Esta capacidad es

particularmente útil cuando un proveedor cambia condiciones, cuando se requiere una mejora de rendimiento o cuando se detectan vulnerabilidades.

3. Reutilización y componentes comerciales (COTS/CYD): selección y evaluación

Una fuente común de componentes son los productos comerciales listos para usar (COTS o CYD). Estos componentes suelen ofrecer funcionalidades estándar de alto valor, tales como: autenticación, pagos, notificaciones, almacenamiento, mensajería, monitoreo, generación de reportes o motores de búsqueda.

El uso de COTS requiere un proceso de selección. Pressman enfatiza que no basta con que un componente 'haga lo que se necesita'; también debe evaluarse la compatibilidad con la arquitectura, el costo total (licencias, consumo, soporte), la calidad del proveedor, el historial de actualizaciones, la documentación, la estabilidad de la interfaz y la comunidad o ecosistema alrededor.

Un criterio crítico es el ajuste funcional: ¿cubre el componente el requerimiento tal como se definió? Si la respuesta es parcial, el equipo debe decidir entre adaptar el componente (mediante configuraciones o extensiones), construir un adaptador (wrapper) o replantear el requerimiento. Además, se debe evaluar el impacto en seguridad y cumplimiento: componentes de terceros pueden introducir riesgos si no se controlan permisos, dependencias y versiones.

En general, la selección se fortalece con evidencia: pruebas de concepto (PoC), análisis de rendimiento, revisión de documentación, y validación de escenarios críticos del negocio. El resultado de esta etapa debe quedar documentado para justificar por qué se eligió un componente específico y bajo qué condiciones se utilizará.

4. Proceso de desarrollo basado en componentes

El proceso de ISBC modifica varias actividades tradicionales del ciclo de vida. En vez de pasar directamente de requisitos a diseño e implementación desde cero, el equipo agrega actividades orientadas a descubrimiento y evaluación de componentes.

De manera resumida, el proceso incluye: (1) análisis de requerimientos, (2) búsqueda de componentes candidatos, (3) evaluación y selección, (4) diseño arquitectónico considerando los componentes elegidos, (5) adaptación y composición (ensamblaje), (6) verificación y validación de la integración, y (7) mantenimiento y gestión de versiones.

Durante el análisis, algunos requerimientos pueden reformularse en términos de capacidades existentes. Por ejemplo, si se adopta un servicio de pagos, ciertos detalles de seguridad y cumplimiento se derivan del proveedor, mientras que el sistema debe enfocarse en orquestar la transacción, registrar eventos y manejar fallos.

La integración suele ser el núcleo del trabajo: conectar componentes, transformar datos, gestionar errores y mantener consistencia. Para lograrlo, se usan patrones como adaptadores, fachadas y capas de servicios. La calidad final depende de pruebas de integración robustas, métricas de confiabilidad y un control riguroso de dependencias.

5. Diseño arquitectónico e integración: adaptación, ensamblaje y middleware

Una arquitectura orientada a componentes define cómo se relacionan las partes del sistema. En muchos casos, se emplean estilos por capas (presentación, servicios, dominio, datos) o arquitecturas de servicios (microservicios o servicios modulares). El objetivo es que los componentes se conecten mediante contratos claros y puntos de integración controlados.

La adaptación aparece cuando un componente no coincide exactamente con el modelo interno del sistema. Por ejemplo, un servicio externo puede devolver datos con nombres distintos, o usar formatos de fecha diferentes. Un adaptador o wrapper traduce estos formatos para que el resto del sistema permanezca estable y no quede acoplado a decisiones del proveedor.

El middleware (o infraestructura de integración) puede incluir: gateways de API, buses de mensajes, colas, y mecanismos de descubrimiento de servicios. Estas piezas reducen complejidad en el frontend y centralizan temas transversales como autenticación, límites de uso, logging y observabilidad.

Un aspecto importante del diseño es decidir el nivel de confianza en el componente. Cuando la funcionalidad es crítica (por ejemplo, pagos o autenticación), el sistema debe implementar mecanismos de resiliencia: reintentos con backoff, circuit breakers, fallbacks, y manejo de degradación controlada. Así, una falla externa no necesariamente detiene toda la operación.

6. Calidad, riesgos y gestión del ciclo de vida de componentes

ISBC promete calidad por reutilización, pero introduce riesgos particulares. Pressman señala que la calidad no depende solo del componente individual, sino de la composición. Un sistema puede fallar por incompatibilidades, por supuestos diferentes entre componentes o por falta de pruebas de integración.

Entre los riesgos más comunes se encuentran: dependencia de proveedores (cambios de precio o términos), cambios de versiones con rupturas de compatibilidad, vulnerabilidades en librerías o servicios externos, y limitaciones funcionales que obligan a rediseñar procesos. También aparecen riesgos legales: licencias restrictivas, requisitos de atribución, o uso de datos en jurisdicciones específicas.

Para gestionar estos riesgos se recomienda: (1) mantener inventario de componentes y versiones, (2) aplicar políticas de actualización y parches, (3) automatizar pruebas de integración en CI/CD, (4) monitorear dependencias vulnerables, y (5) documentar contratos de interfaz.

Desde la perspectiva de calidad, se deben validar atributos no funcionales: rendimiento, seguridad, disponibilidad y mantenibilidad. En ambientes reales, la observabilidad (logs, métricas y trazas) es esencial para diagnosticar fallas entre componentes y medir acuerdos de nivel de servicio (SLA).

7. Ejemplo aplicado: cómo se implementan requerimientos con componentes

Para ilustrar la aplicación de ISBC, consideremos un sistema de reservas (por ejemplo, una plataforma de gestión de aerolínea o de renta de vehículos). El sistema requiere autenticación, pagos, notificaciones, generación de comprobantes, almacenamiento de documentos y búsqueda.

En un enfoque basado en componentes, el equipo puede integrar: (1) un componente de autenticación (OAuth/JWT) para registro e inicio de sesión; (2) un componente de pagos para procesar tarjetas y confirmaciones; (3) un servicio de notificaciones para enviar correos y SMS; (4) una librería para generar PDF con códigos QR; y (5) un motor de búsqueda para consultas rápidas.

El trabajo propio del sistema se concentra en lo específico del dominio: reglas de disponibilidad, gestión de inventario (vuelos/vehículos), políticas de tarifas, estados de la reserva, y auditoría. De esta manera, los componentes resuelven funciones transversales y repetitivas, mientras que el equipo desarrolla lo que verdaderamente diferencia el producto.

Para que la integración sea exitosa, se diseñan interfaces internas estables. Por ejemplo, el sistema define una interfaz 'PaymentService' con métodos como 'createPayment' y

'confirmPayment'. Internamente, se implementa un adaptador hacia el proveedor seleccionado. Si mañana se cambia de proveedor, el resto del sistema se mantiene igual, modificando solo el adaptador.

Este ejemplo refleja el principio de sustitución: la arquitectura debe permitir cambiar componentes sin reescribir toda la aplicación. La clave está en separar el dominio del proveedor, y en asegurar pruebas de integración que cubran los escenarios críticos (éxito, rechazo, reintentos y fallos).

8. Ventajas y limitaciones del enfoque basado en componentes

Entre las ventajas más destacadas se encuentran: rapidez de entrega, reducción de retrabajo y aprovechamiento de soluciones maduras. También mejora la mantenibilidad al dividir el sistema en partes reemplazables, y fomenta el trabajo paralelo (equipos distintos pueden integrar o mejorar componentes diferentes).

No obstante, ISBC tiene limitaciones. En ciertos dominios, la disponibilidad de componentes adecuados puede ser baja o la integración puede introducir una complejidad mayor que la construcción propia. Además, el componente puede imponer un modelo de datos o flujo que obligue a adaptar procesos del negocio. En consecuencia, se requiere disciplina de arquitectura y gestión de dependencias.

Pressman remarca que la decisión no es absoluta: en muchos proyectos se adopta un enfoque híbrido. Se reutilizan componentes donde agregan valor (infraestructura y servicios comunes) y se construye desde cero lo que es estratégico o altamente específico del negocio.

Conclusión

La Ingeniería de Software Basada en Componentes, tal como se discute en el capítulo 27 de Pressman, propone un cambio de enfoque: construir sistemas ensamblando piezas reutilizables con interfaces bien definidas. Cuando se aplica correctamente, permite acortar tiempos de desarrollo, reducir costos y aprovechar componentes probados, elevando la calidad del producto final.

Sin embargo, el enfoque introduce desafíos propios: evaluación de componentes, compatibilidad de interfaces, adaptación, dependencia de proveedores y riesgos de seguridad o licenciamiento. Por ello, su éxito depende de una arquitectura sólida, políticas de gestión de versiones, pruebas de integración rigurosas y documentación clara de contratos.

En conclusión, ISBC es especialmente valioso en sistemas empresariales y aplicaciones modernas, donde muchas funcionalidades transversales ya existen como servicios o componentes maduros. Adoptar este paradigma de manera estratégica (incluso en forma híbrida) mejora la capacidad de entrega del equipo y facilita la evolución del software a lo largo del tiempo.

Bibliografía

• Pressman, R. S. (2002). Ingeniería de Software: Un enfoque práctico (5ta ed.). McGraw-Hill.

• Sommerville, I. (2011). Ingeniería de software (9na ed.). Pearson.

• Szyperski, C. (2002). Component Software: Beyond Object-Oriented Programming (2nd ed.). Addison-Wesley.

• ISO/IEC. (2017). ISO/IEC 12207: Systems and software engineering — Software life cycle processes.