Paradigmas de Programación

Semana 7a

Agenda

- ¿Qué es un paradigma de programación?
- Paradigma orientado a aspectos
- Paradigma orientado a componentes

¿Qué es un paradigma de programación?

¿Qué es un paradigma de programación?

- El término **paradigma de programación se refiere a un estilo de programación**. No se refiere a un lenguaje específico, mas bien se refiere a la forma en que se programa.
- Para diseñar un sistema de software, se debe conocer información básica de los estilos y lenguajes de programación.
- Este conocimiento permite explorar diversos diseños de programas con un mejor entendimiento de sus principios de funcionamiento.

¿Qué es un paradigma de programación?

- Los paradigmas son formas de programar, relativamente sin relación con los lenguajes de programación.
- Un paradigma te dice qué estructuras de programación utilizar y cuándo usarlas.
- Existen varios paradigmas:
  - Paradigma estructurado
  - Paradigma orientado a objetos
  - Paradigma orientado a aspectos
  - Paradigma orientado a componentes
  - Paradigma funcional
  - Entre otros

¿Por qué aprender distintos paradigmas?

- Los programadores que conocen sólo un paradigma intentan resolver todo con él, incluso si es descabellado hacerlo.
- Tener la posibilidad de escoger los mejores paradigmas que se ajusten al tipo de problema a resolver.
- Para poder mantener sistemas grandes y complejos, compuestos de varios subsistemas y escritos en distintos y paradigmas de programación.

Tipos de Paradigmas

- **Paradigma imperativo.**
  - Se escriben instrucciones para cambiar el estado de un programa.
  - Se escriben algoritmos para resolver problemas.
  - Lenguajes: Fortran, C.
  - Ejemplos:
    - Paradigma estructurado
    - Paradigma orientado a objetos
- **Paradigma declarativo**
  - Se especifica el resultado esperado, pero no el cómo obtenerlo.
  - No se indica un orden en el cual ejecutar operaciones. En vez de ello, se provee un número de operaciones que están disponibles en el sistema con las respectivas condiciones bajo las cuales cada operación puede ejecutarse.
  - Lenguajes: LISP, Prolog, Haskell.
  - Ejemplos:
    - Paradigma funcional
    - Paradigma lógico

Paradigma Imperativo

- Ejemplo en C:

```c
#include <stdio.h>

int main()
{
    int sum = 0;
    sum += 1;
    sum += 2;
    sum += 3;
    sum += 4;
    sum += 5;
    sum += 6;
    sum += 7;
    sum += 8;
    sum += 9;
    sum += 10;

    printf("The sum is: %d\n", sum); //prints-> The sum is 55

    return 0;
}
```

Paradigma Declarativo

- Ejemplo en Prolog:

```
/*We're defining family tree facts*/
father(John, Bill).
father(John, Lisa).
mother(Mary, Bill).
mother(Mary, Lisa).

/*We'll ask questions to Prolog*/
?- mother(X, Bill).
X = Mary
```

- *father(John, Bill)* significa que John es el padre de Bill
- *?- mother(X, Bill)* le pregunta a Prolog qué valor de X hace a esta sentencia verdadera. Responderá *X = Mary*

En la práctica

- Se utilizan formas mixtas de paradigmas
- Los lenguajes declarativos se complementan con métodos imperativos.
- Este tipo de programación es más propenso a errores
- Dificulta la legibilidad del código.

Paradigma orientado a aspectos

Paradigma orientado a aspectos

- Programación Orientada a Aspectos (AOP) es un paradigma de programación que complementa la programación orientada a objetos por medio de la separación de ***concerns*** de un sistema de software para mejorar su modularización.
- La separación de concerns busca facilitar la mantenibilidad del software agrupando características y comportamiento en torno a un propósito específico.

Paradigma orientado a aspectos

- La orientación a objetos ya modulariza ***concerns*** a través de métodos, clases y paquetes.
- Sin embargo, algunos concerns son difíciles de ubicar ya que cruzan los límites de clases e incluso paquetes. Por ejemplo: un ***cross-cutting concern (concern transversal)*** es la seguridad.
  - Aún cuando el principal propósito de un paquete Foro es mostrar y manejar los posts de un foro, este tiene que implementar algún tipo de seguridad para validar que solo los moderadores puedan aprobar o borrar posts.
- AOP nos permite mover el aspecto de seguridad (y cualquier otro) en su propio paquete y dejar a los otros objetos con una responsabilidad clara, probablemente sin implementar ninguna seguridad por sí mismos.

Paradigma orientado a aspectos

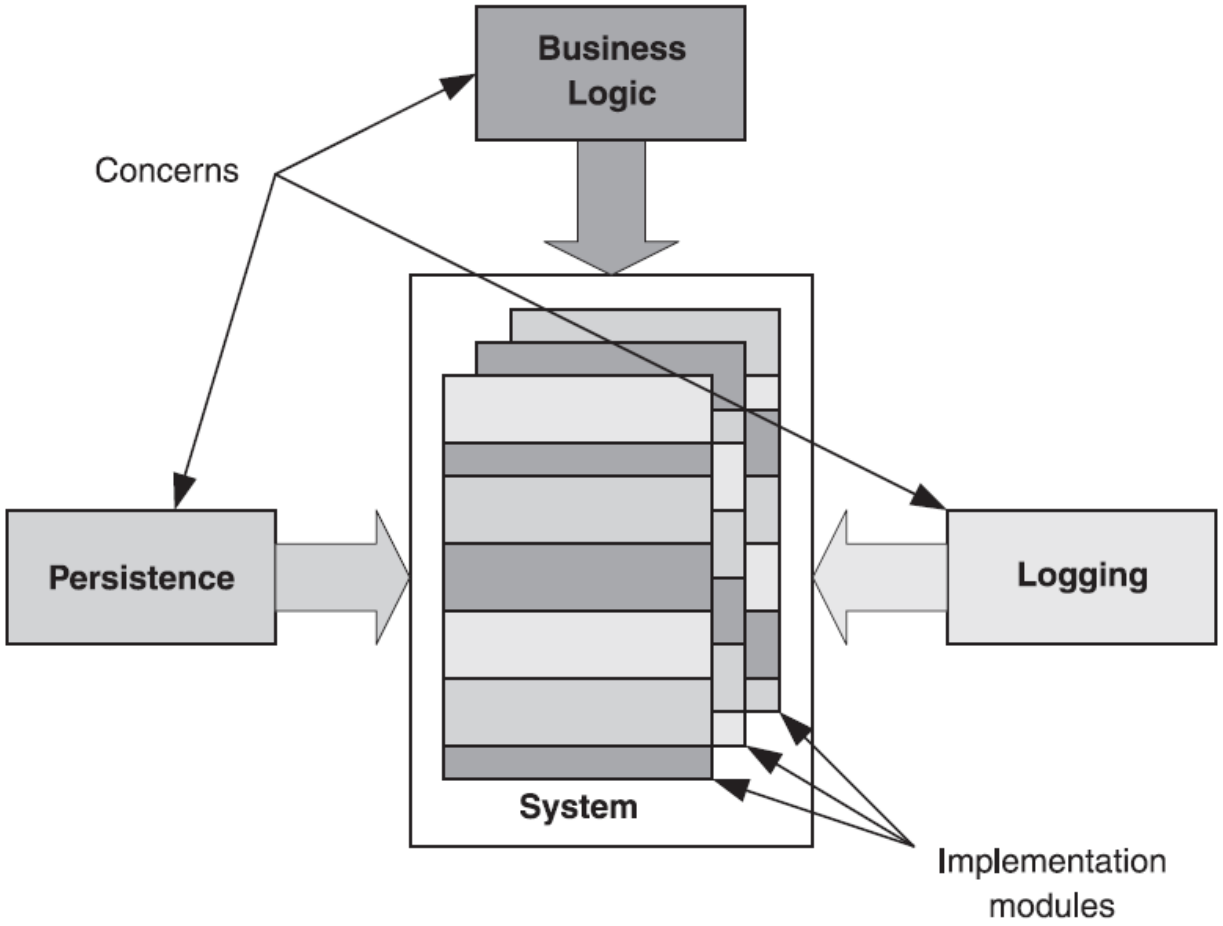


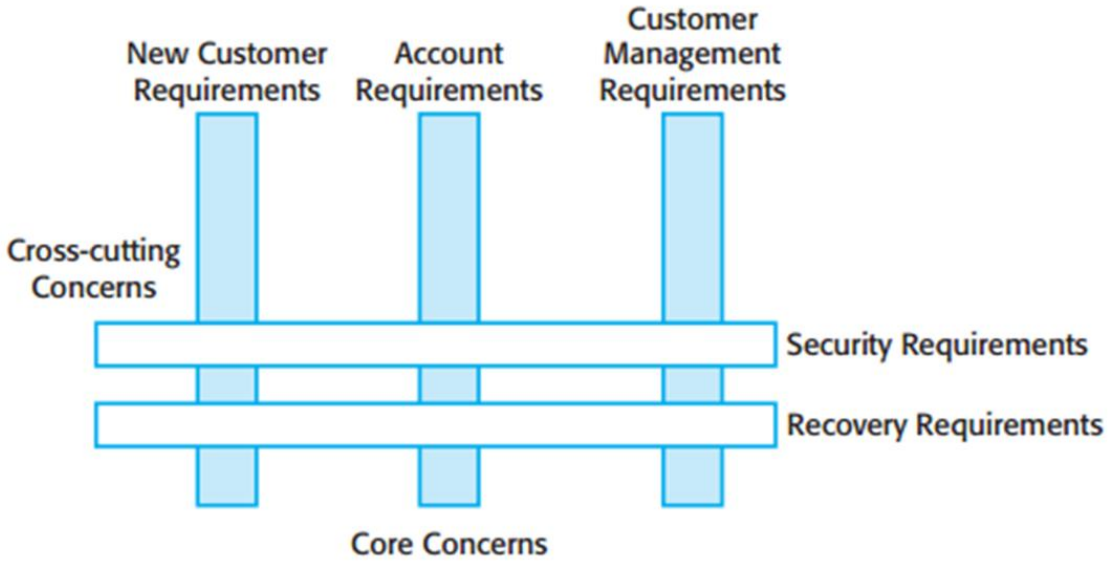
Figure 1.1 Viewing a system as a composition of multiple concerns. Each implementation module addresses some element from each of the concerns the system needs to address.

Paradigma orientado a aspectos

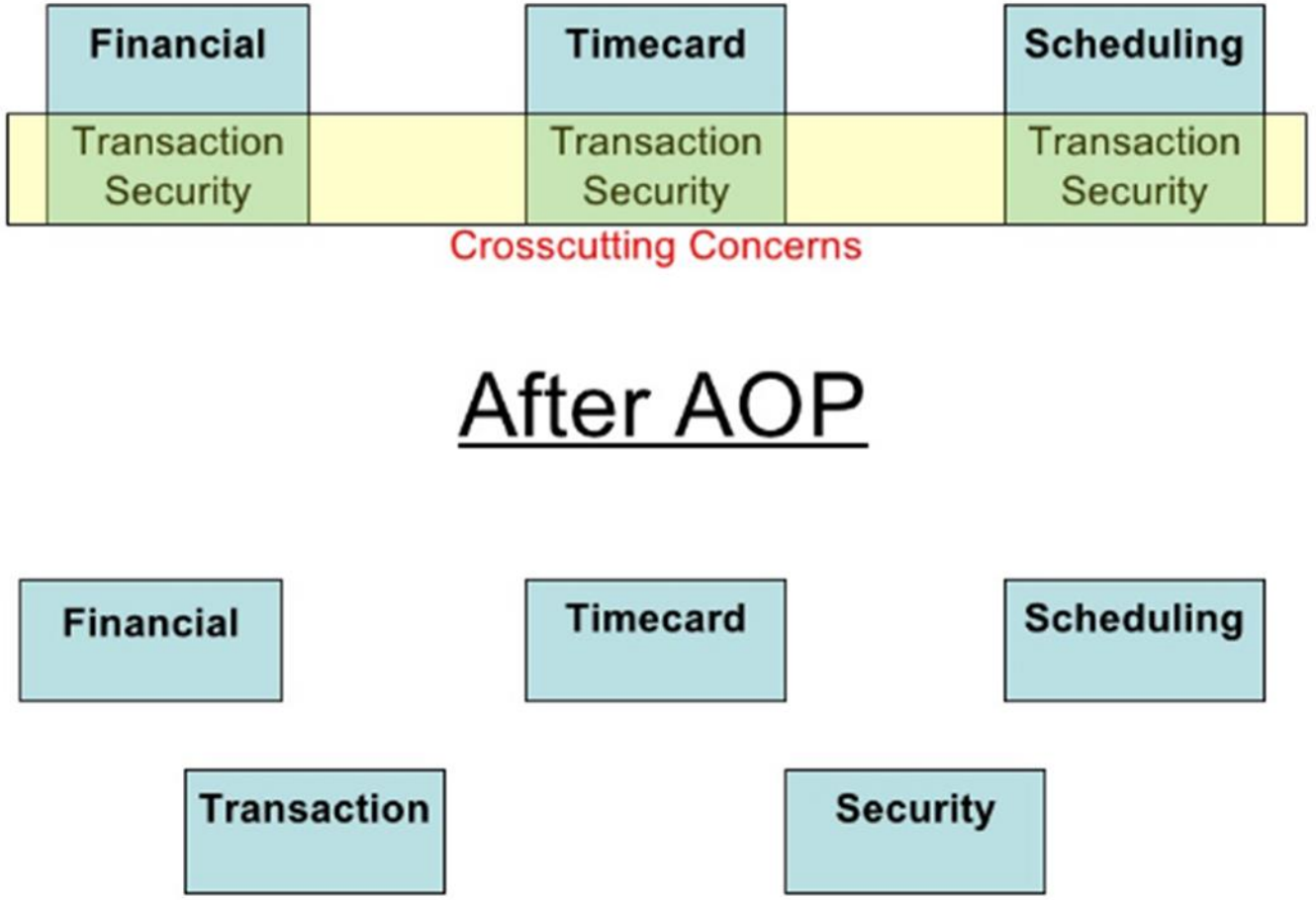
- Un concern puede ser clasificado en una de dos categorías:
  - **core concerns** que capturan la funcionalidad central de un módulo
  - **crosscutting concerns** que capturan requerimientos periféricos, a nivel de Sistema que Cruzan múltiples módulos.
- Ejemplos de crosscutting concerns:
  - Autenticación
  - Registros de Auditoría (Logging)
  - Pool de recursos
  - Desempeño
  - Persistencia de datos
  - Seguridad
  - Seguridad de multi-hilos
  - Integridad transaccional
  - Chequeo de errores

Paradigma orientado a aspectos

- Los Aspectos son las abstracciones que representan a los concerns transversales.
- Se puede decir que los Aspectos son los concerns transversales modelados.



Paradigma orientado a aspectos



Síntomas de mala modularidad

- **Code tangling**: se produce cuando un módulo manipula múltiples concerns simultáneamente.

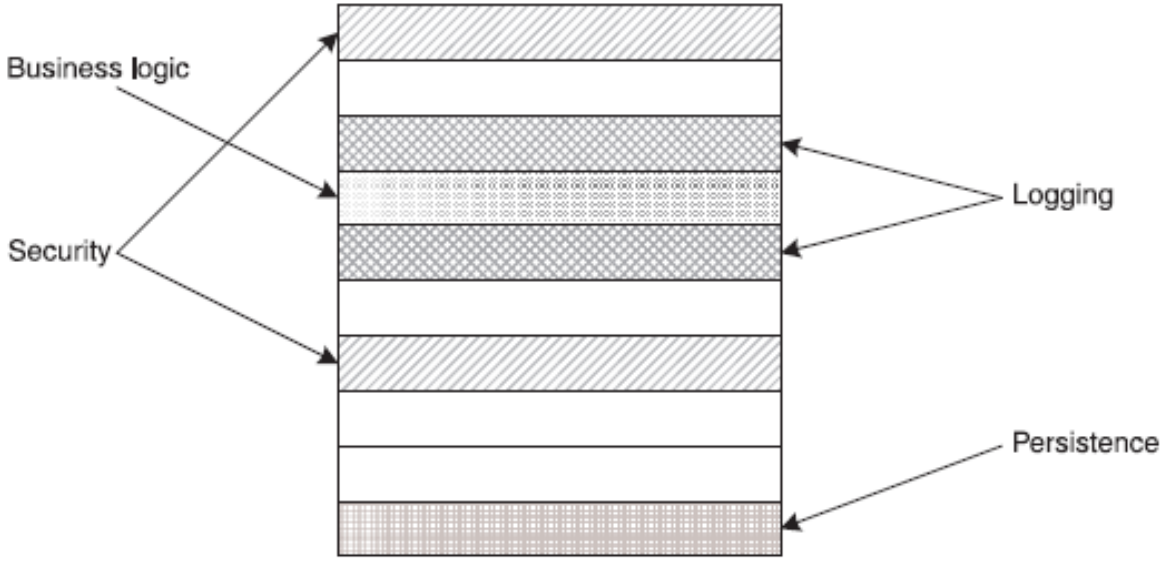


Figure 1.7 Code tangling caused by multiple simultaneous implementations of various concerns. The figure shows how one module manages a part of multiple concerns.

Síntomas de mala modularidad

Listing 1.1 Business logic implementation along with crosscutting concerns

```
public class SomeBusinessClass extends OtherBusinessClass {

    ... Core data members

    ... Log stream                                   ❶ Data to support
                                                       peripheral concerns
    ... Cache update status

    ... Override methods in the base class

    public void someOperation1(<operation parameters>,
                               <authenticated user>,
                               ...) {
        ... Ensure authorization

        ... Ensure info satisfies contracts

        ... Lock the object to ensure thread-safety

        ... Ensure cache is up-to-date

        ... Log the start of operation               ❷ Invocation of
                                                       peripheral
        ... Perform the core operation                 services

        ... Log the completion of operation

    }

    ... More operations similar to above addressing multiple concerns

    public void save(<persitance storage parameters>) {
        ...                                          ❸ Methods to
    }                                                  support
                                                       peripheral
    public void load(<persitance storage parameters>) {  services
        ...
    }
}
```

Síntomas de mala modularidad

- **Code scattering**: es causado cuando un único asunto es implementado en múltiples módulos. Hay dos tipos de scattering:
  - Bloques de código duplicados.
  - Bloques de código complementarios: varios módulos implementan partes complementarias de un concern. Por ejemplo: un sistema de control de accesos en el que existe un único módulo de autenticación, el cual pasa usuarios autenticados a módulos que requieren autorización, la cual es desarrollada por cada uno de estos módulos.

Síntomas de mala modularidad

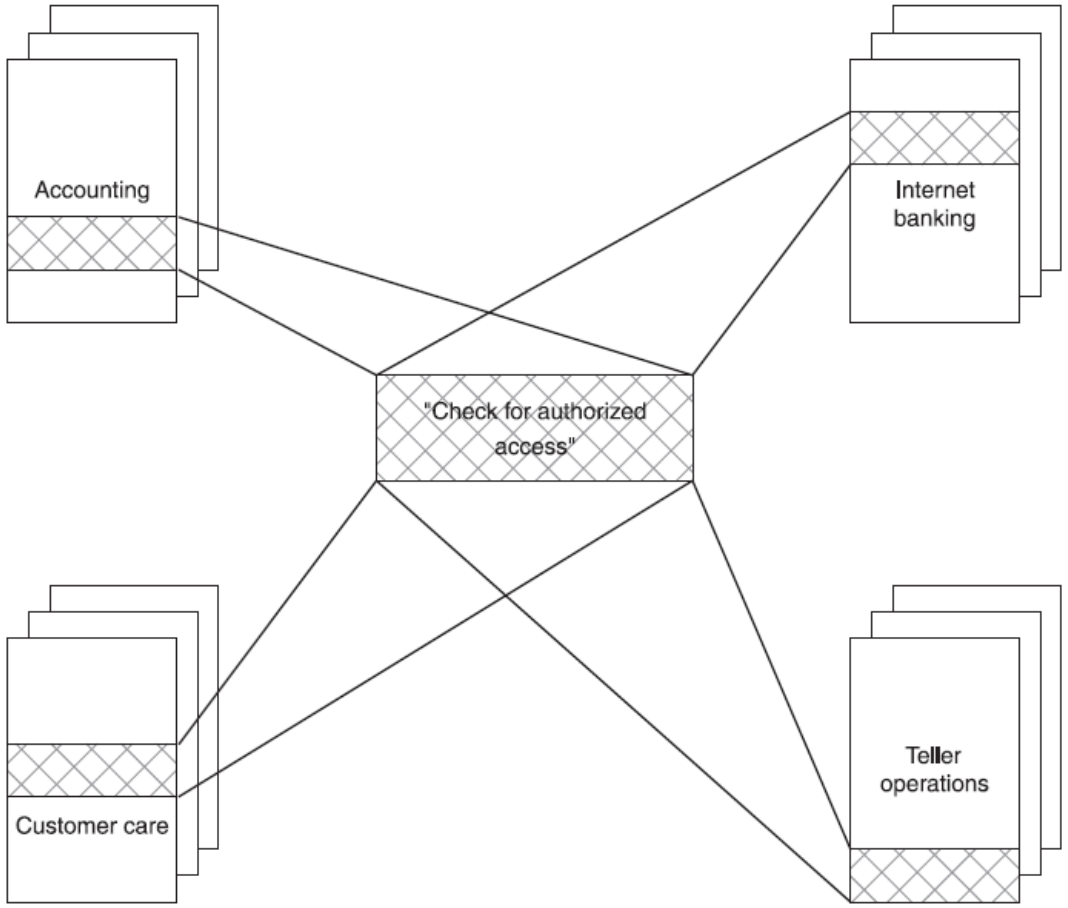


Figure 1.8 Code scattering caused by the need to place nearly identical code blocks in multiple modules to implement a functionality. In this banking system, many modules in the system must embed the code to ensure that only authorized users access the services.

Síntomas de mala modularidad

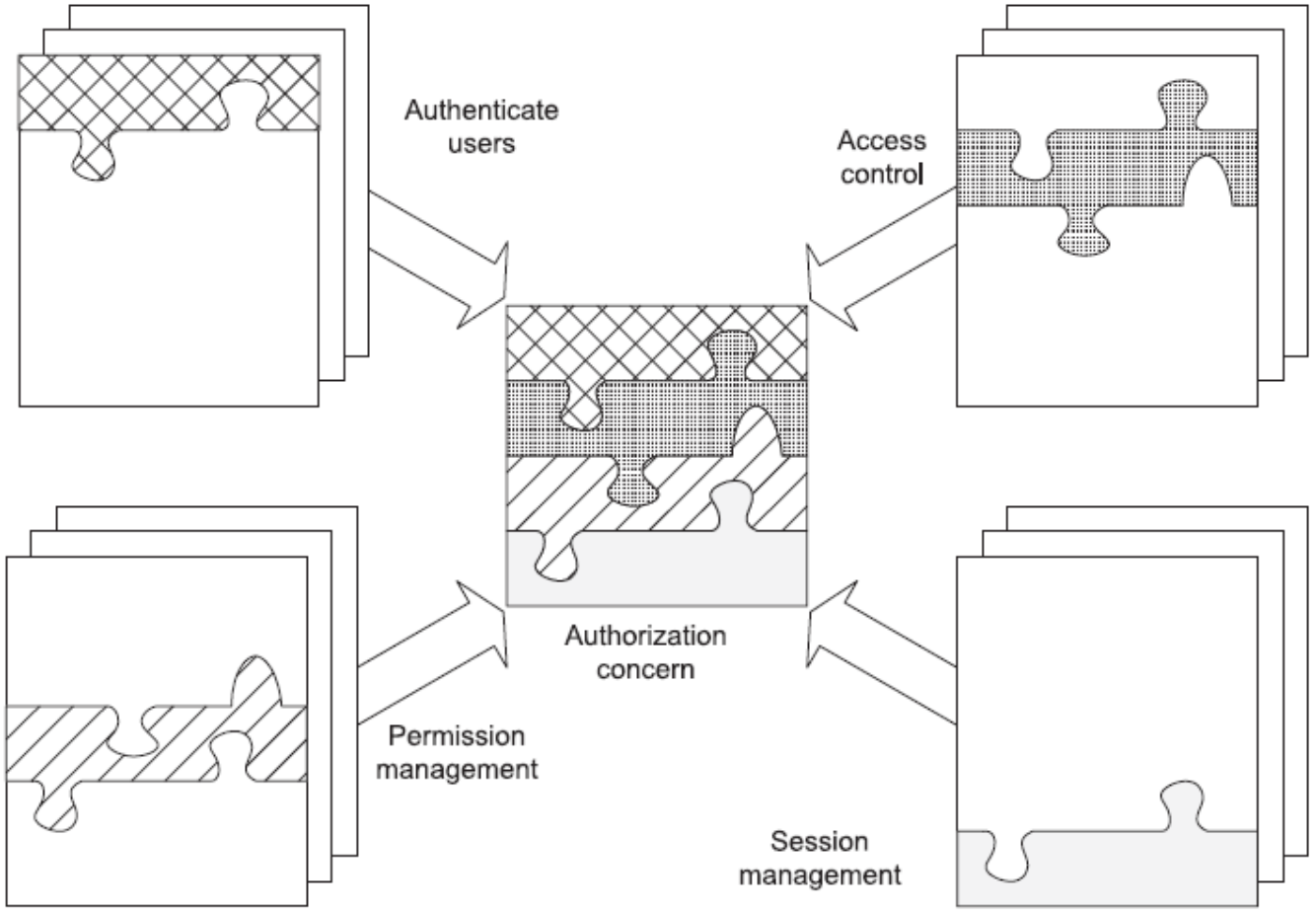


Figure 1.9 Code scattering caused by the need to place complementary code blocks in multiple modules to implement a functionality

La metodología AOP

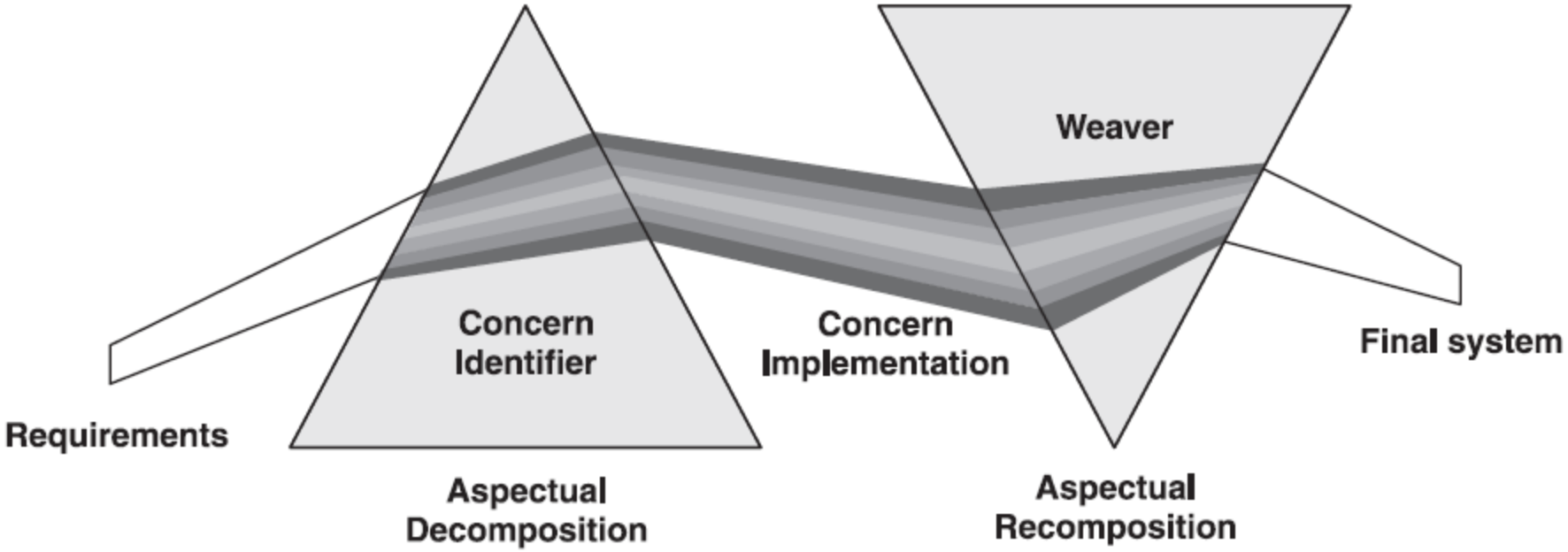


Figure 1.10 AOP development stages. In the first stage, you decompose the system requirements into individual concerns and implement them independently. The weaver takes these implementations and combines them together to form the final system.

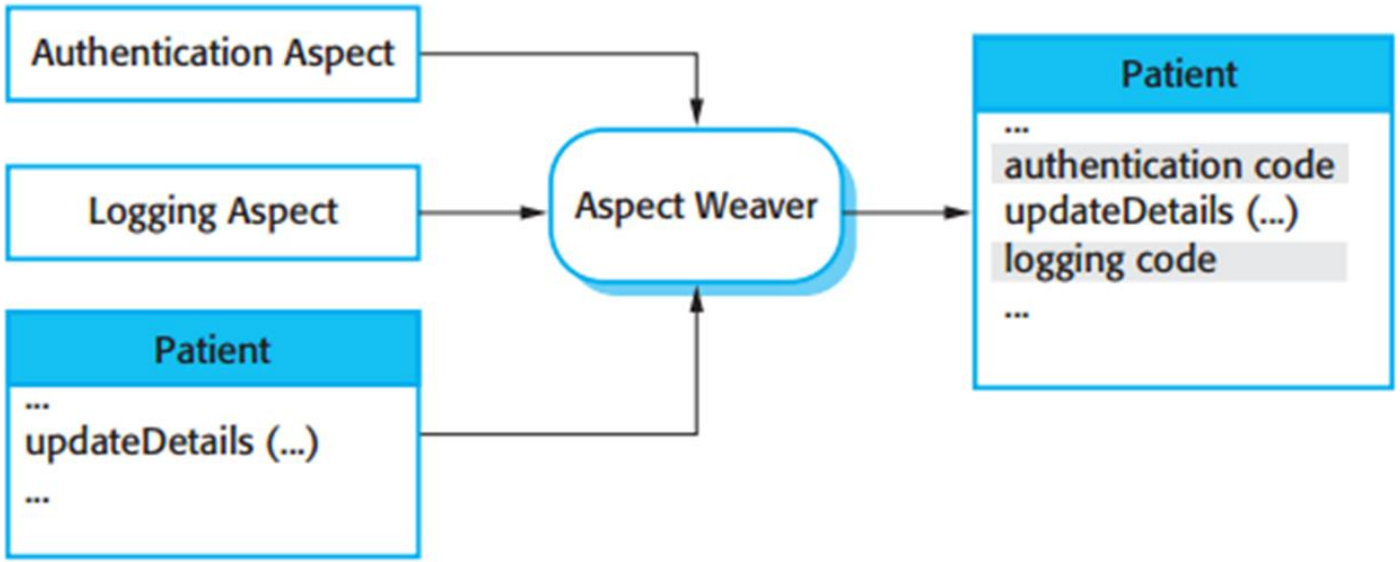
Weaving

Paradigma orientado a aspectos: Weaving

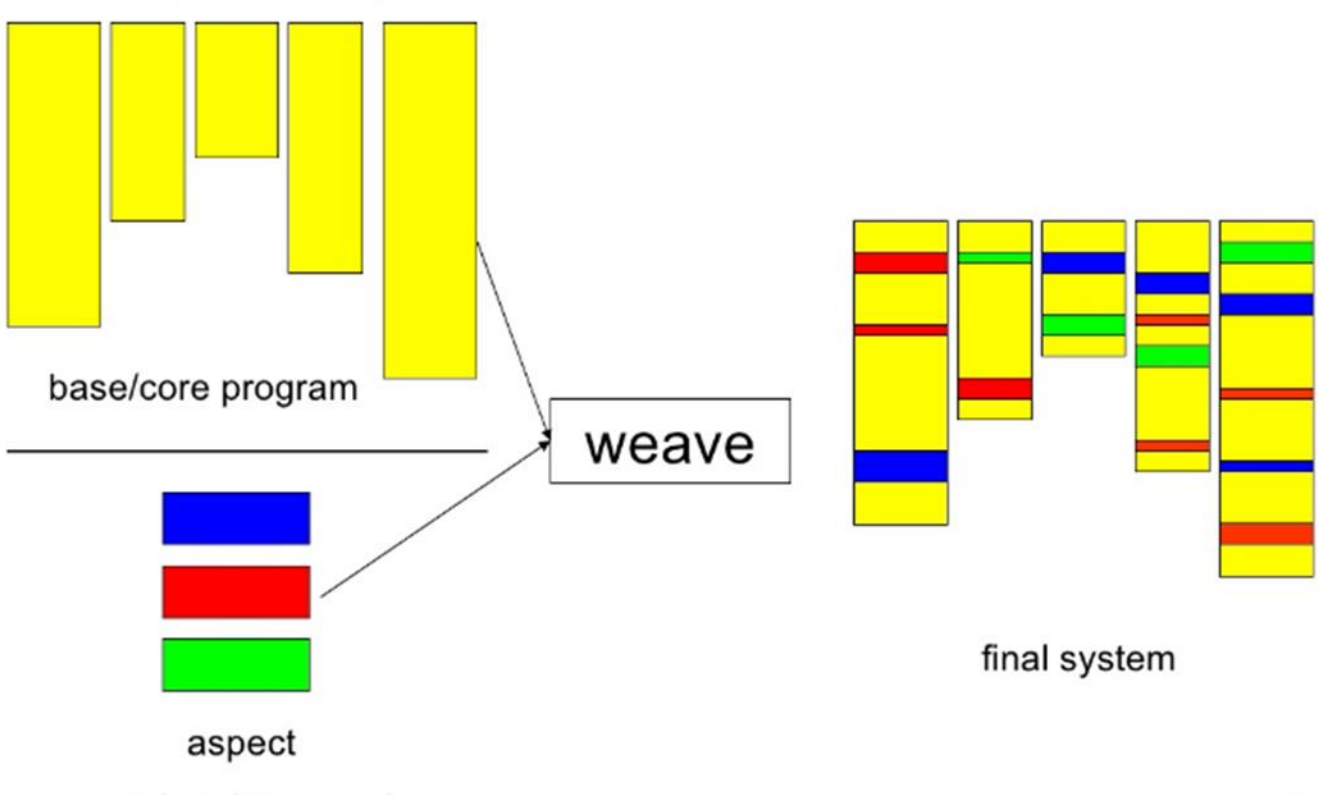
- **Weaving** es el proceso de componer un sistema con base en módulos individuales por medio de reglas de combinación. En esencia, las reglas de combinación determinan la forma final del sistema.

Paradigma orientado a aspectos: Weaving

- Un programa orientado a aspectos es creado combinando (**weaving**) objetos, métodos y aspectos, de acuerdo a los concerns del sistema.



Paradigma orientado a aspectos: Weaving



Paradigma orientado a aspectos: Weaving

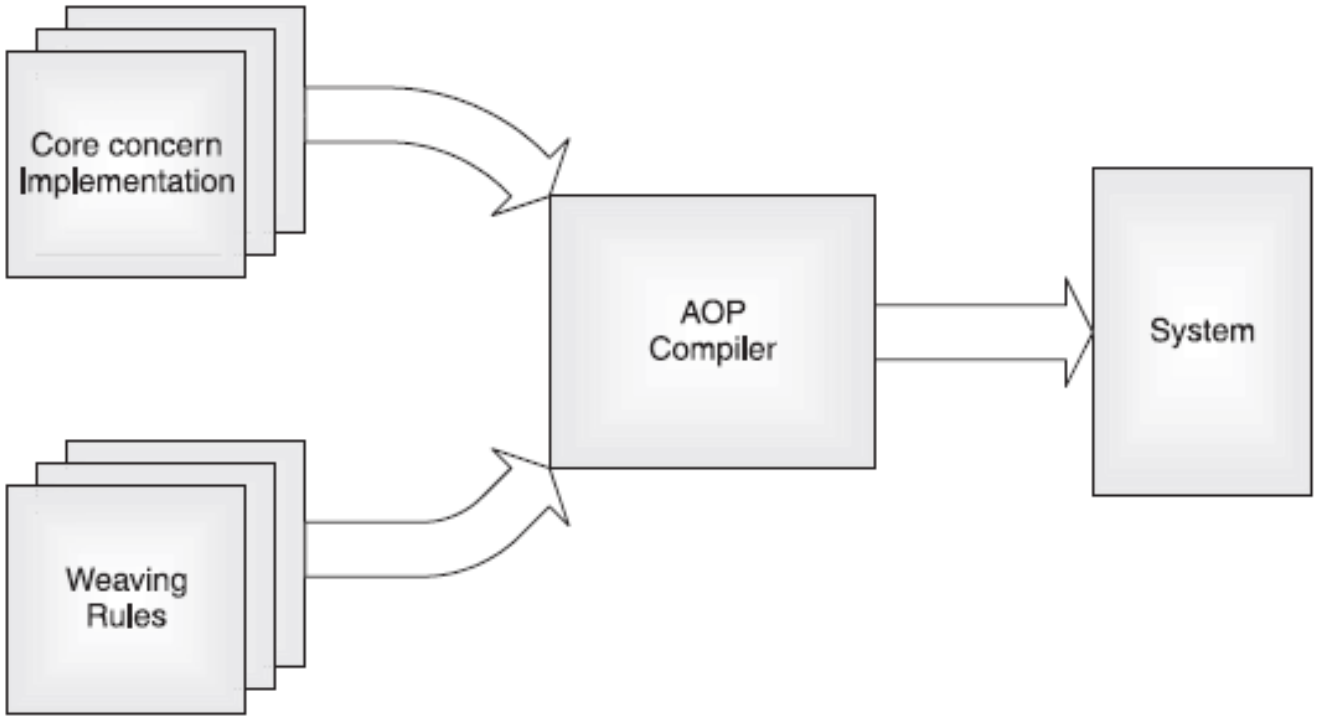


Figure 1.11 An AOP language implementation that provides a weaver in the form of a compiler. The compiler takes the implementation of the core and crosscutting concerns and weaves them together to form the final system. In a Java-based AOP implementation, the core concern implementation would be Java source files and class files and the system would be a set of class files.

Beneficios de AOP

Beneficios de AOP

- Responsabilidades más limpias para los módulos individuales
  - Un módulo solo se especializa en un concern
- Mayor Modularidad
  - Se reduce el acoplamiento y la duplicidad de código
- Facilita la evolución del sistema
  - Al agregar un nuevo módulo core, los aspectos crosscut ayudan a mantener una evolución coherente del sistema
- Retrasar decisiones de diseño
  - El arquitecto puede retrasar decisiones de diseño respecto a requerimientos futuros porque es posible colocarlos en aspectos separados.

Beneficios de AOP

- Mayor reutilización de código.
  - La clave para reutilizar gran cantidad de código está en una implementación con bajo acoplamiento
- Reducción de costos de implementación de requerimientos.
  - Se especializan roles de los desarrolladores al enfocarlos en su área de experticia.

Elementos de AOP

Elementos de AOP

- **Join point** es un **punto** identificable **en la ejecución** de un programa. Puede ser un método o una asignación a un miembro de un objeto.
- Son los lugares en donde se hace weaving de las acciones crosscut.
- Join points de la clase Account incluyen la ejecución del método credit() y el acceso al atributo _balance

```
public class Account {

    ...
    void credit(float amount) {
        _balance += amount;
    }
}
```

Elementos de AOP

- Un **pointcut** es un elemento de un programa que selecciona join points y captura el contexto en esos puntos.
- Por ejemplo, podemos escribir un pointcut que capture la ejecución del método credit() en la clase Account.

```
execution(void Account.credit(float))
```

Elementos de AOP

- **Advice** es el código que ha ser ejecutado en un join point y que ha sido seleccionado por un pointcut.
- Un advice puede ser antes, después o alrededor (around) del joinpoint. El advice around puede modificar, reemplazar o incluso saltarse la ejecución de un código.
- Por ejemplo, usando el anterior pointcut, podemos escribir un advice que escriba un mensaje antes de la ejecución del método credit() de la clase Account.

```
before() : execution(void Account.credit(float)) {
    System.out.println("About to perform credit operation");
}
```

Elementos de AOP

- **Dynamic crosscutting** es la combinación (weaving) de nuevo comportamiento en la ejecución de un programa. La mayor parte del crosscutting es dinámico.
- **Static crosscutting** es la combinación de modificaciones en estructuras estáticas del sistema: clases, interfaces y aspectos.
  - Por ejemplo, se puede agregar nuevos datos y métodos a clases e interfaces para definir específicos estados de la clase (**introduction)** `declare parents: Account implements BankingEntity;`
  - Por ejemplo, se puede agregar una advertencia y errores en tiempo de compilación (**compile-time declaration**)

Elementos de AOP

- El **aspecto** es la unidad central de AOP, en la misma forma que una clase lo es a OO.
  - Contiene el código que expresa las reglas de weaving para crosscutting dinámico y estático.
  - Combina pointcuts, advice, introductions y declarations. Contiene lo mismo que una clase normal en Java

```
public aspect ExampleAspect {
    before() : execution(void Account.credit(float)) {
        System.out.println("About to perform credit operation");
    }

    declare parents: Account implements BankingEntity;

    declare warning : call(void Persistence.save(Object))
        : "Consider using Persistence.saveOptimized()";
}
```

AspectJ - Eclipse

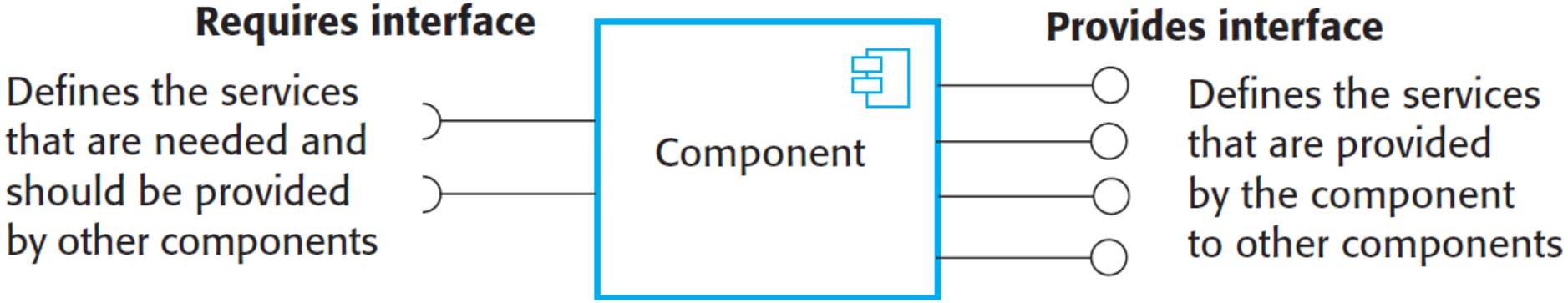
org.aspectj/IDE.md at master · eclipse/org.aspectj (github.com)

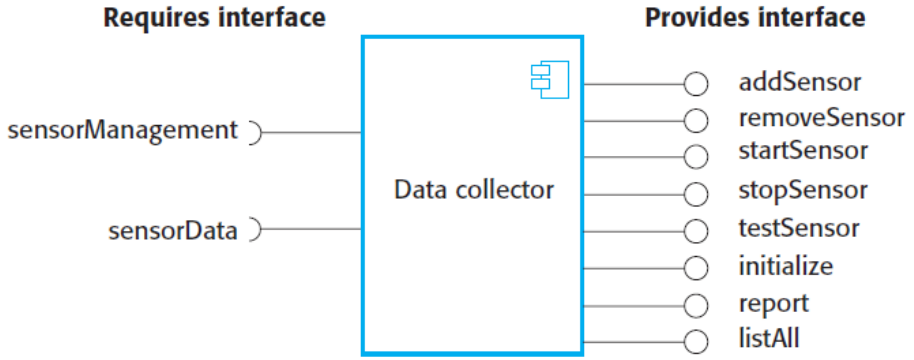
The AspectJTM Programming Guide (eclipse.org)

Paradigma orientado a componentes

¿Qué es un componente?

- Un **componente** es una unidad de software **independiente**, **desplegable** que está completamente definida y accedida por medio de un conjunto de interfaces.





Características de un componente

Component characteristic	Description
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of that component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a "requires" interface specification.
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

Composición de componentes

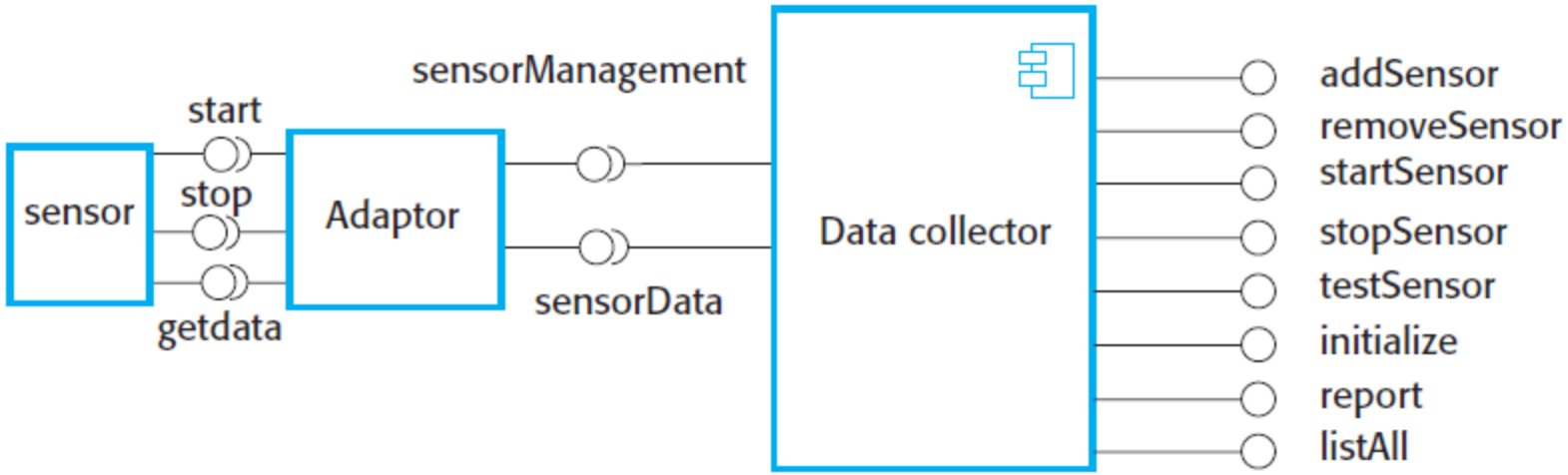


Figure 16.12 An adaptor linking a data collector and a sensor

Paradigma orientado a componentes

- En la mayoría de los proyectos de software, generalmente hay algún software a reutilizar.
- Este enfoque se fundamenta en el reuso de componentes de software y un framework de integración para la composición de estos componentes.
- Algunas veces estos componentes son por sí mismos sistemas independientes (COTS) que pueden proveer funcionalidades específicas.

Paradigma orientado a componentes

- Está basado en la existencia de un número significativo de **componentes reutilizables**. El desarrollo del sistema se centra en la **integración** de estos componentes en un sistema en vez de desarrollarlos desde cero.

Etapas del paradigma orientado a componentes

1. **Análisis de componentes**: dada una especificación de requerimientos, buscar el componente que mejor se ajuste.
2. **Modificación de requerimientos**: De acuerdo a la disponibilidad de componentes, se ajustan ciertos requerimientos.
3. **Diseño del sistema con reuso**: Los diseñadores toman en consideración los componentes a reutilizar y organizan el framework para ello. Podría haber algún desarrollo si no hay componentes disponibles.
4. **Desarrollo e integración**: Componentes y sistemas COTS son integrados para crear el nuevo sistema.

Paradigma orientado a componentes

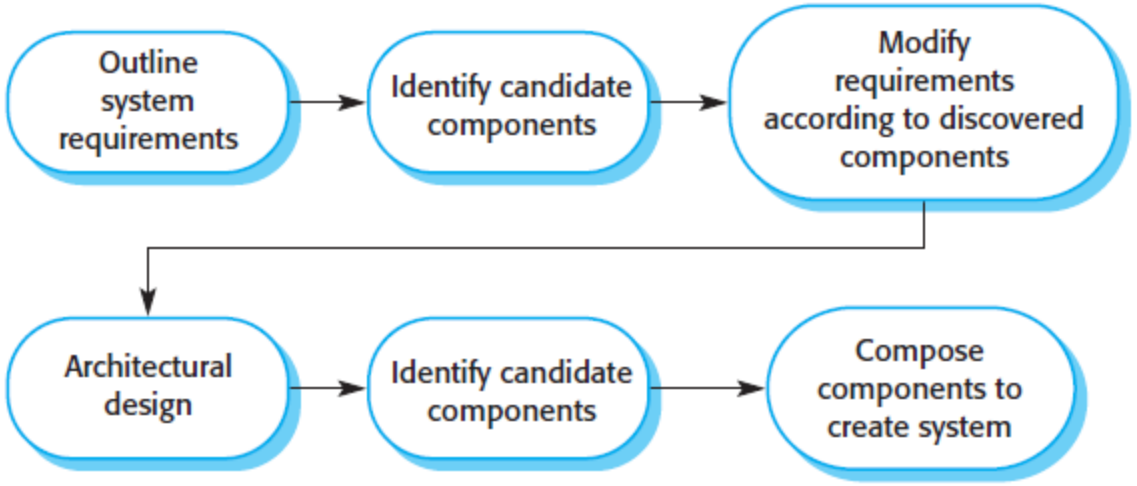


Figure 16.7 CBSE with reuse

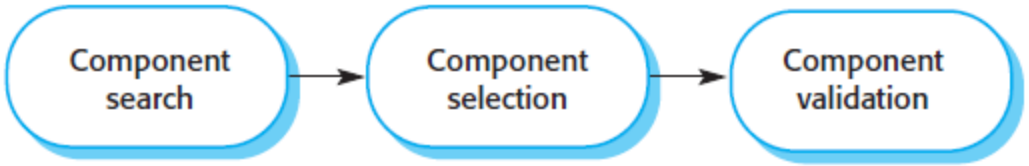


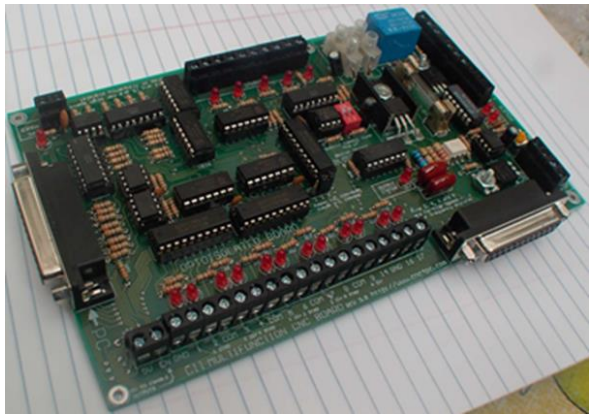
Figure 16.8 The component identification process

Paradigma orientado a componentes

- Tipos de componentes que pueden ser utilizados en este paradigma:
  - **Servicios Web**, los mismos que son desarrollados de acuerdo a los estándares de servicios y están disponibles por invocación remota.
  - **Colecciones de objetos** que son desarrollados como un **paquete** a fin de ser integrados en un framework de componentes tales como .Net o JEE.
  - Sistemas de **software stand-alone** que pueden ser configurados para su uso en un ambiente en particular.

Modelos de componentes

- Un modelo de componentes es una definición de estándares para implementación, documentación y despliegue de componentes.
- Los estándares aseguran que los componentes podrán interoperar.
- Los estándares además aseguran que el middleware para ejecución de componentes soporte la operación de los mismos.
- Ejemplos de modelos de componentes:
  - Enterprise Java Beans (EJB) model
  - Microsof's .NET model
  - Corba



Ventajas del paradigma orientado a componentes

- Reduce la cantidad de software que debe ser desarrollado, lo que reduce costos y riesgos.
- Usualmente se disminuyen los tiempos de entrega del software.

Limitaciones del paradigma orientado a componentes

- Se pierde el control sobre la evolución del software si las nuevas versiones de los componentes reusables no están bajo el control de la organización.
- El ajuste de requerimientos podría producir un sistema que no satisfaga las necesidades reales del usuario.

Antes de finalizar

Puntos para recordar

- ¿Qué es un paradigma de programación?
- ¿Qué es AOP?
  - Conceptos de AOP
  - Beneficios de AOP
- ¿Qué es el paradigma orientado a componentes?
  - ¿Qué es un componente?
  - Beneficios y limitaciones

Lectura adicional

- Ramnivas Laddad, “AspectJ In Action”
  - Chapter 1: Introduction to AOP
  - Chapter 2: Introducing AspectJ
- Thanoshan MV, “What exactly is a programming paradigm?”
  - https://www.freecodecamp.org/news/what-exactly-is-a-programming-paradigm/
- Ian Sommerville, “Software Engineering”
  - Chapter 16: Component-based Software Engineering

Próxima sesión

- Principios del diseño orientado a objetos: abstracción, encapsulación, herencia y polimorfismo.
- Lección 1. Capítulo1. 30 minutos al final de la clase.