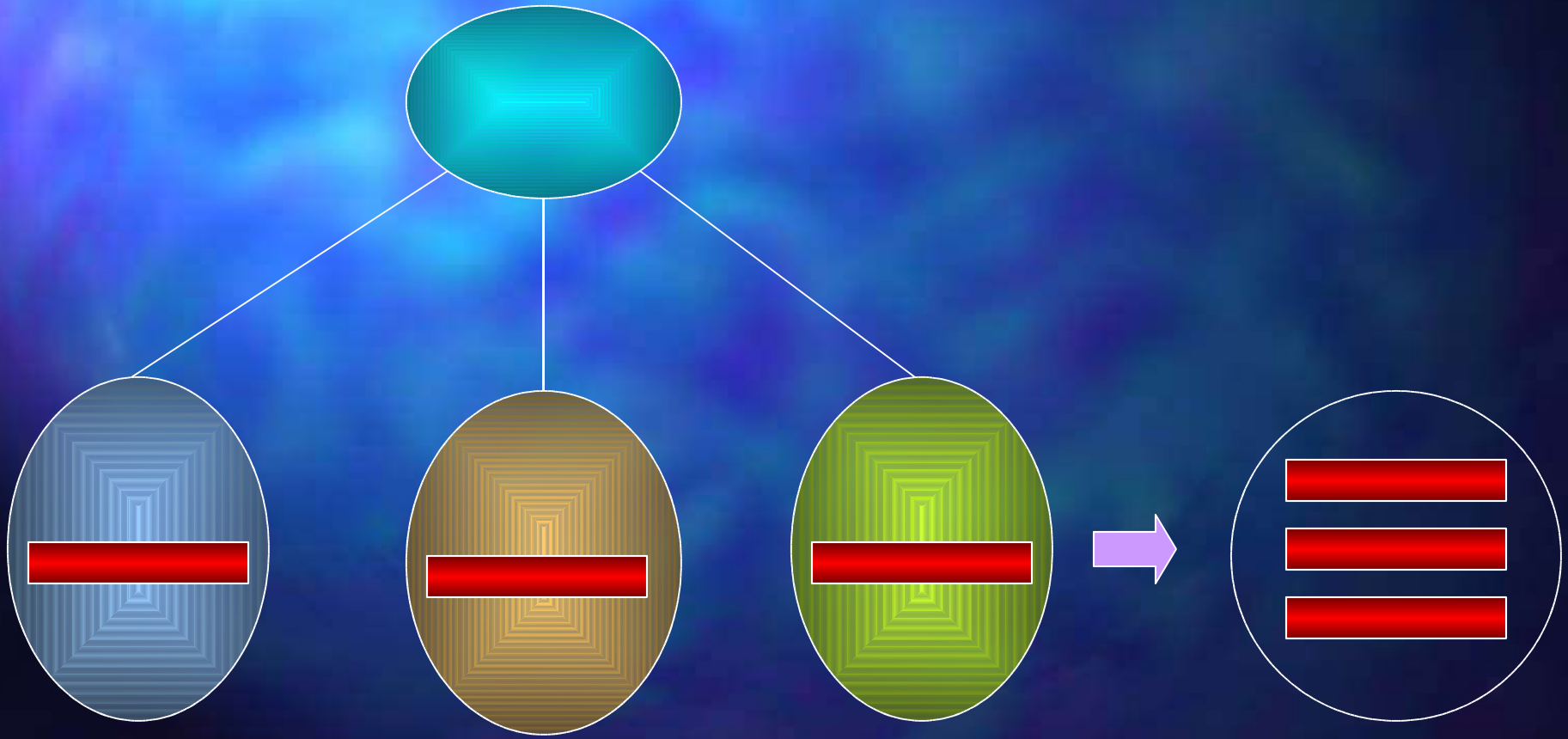
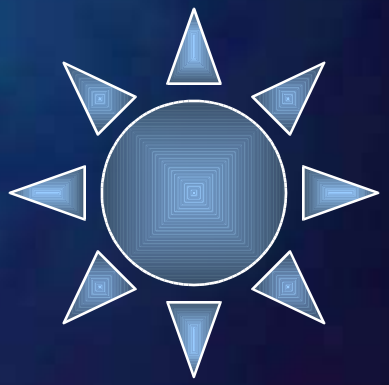
Aspect Oriented Programming (AOP)

Programación Orientada a Aspectos

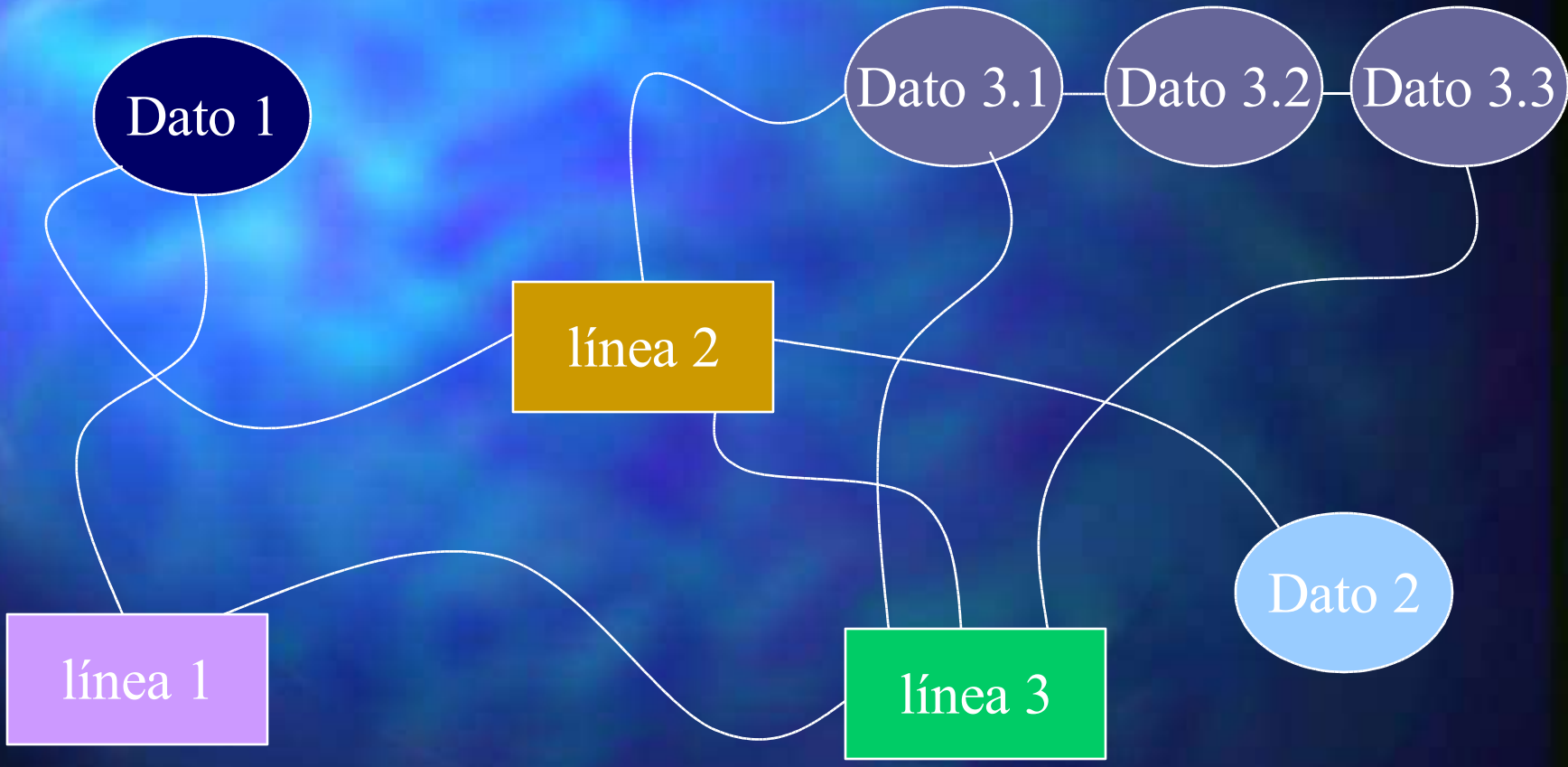


Evolución de las metodologías de programación

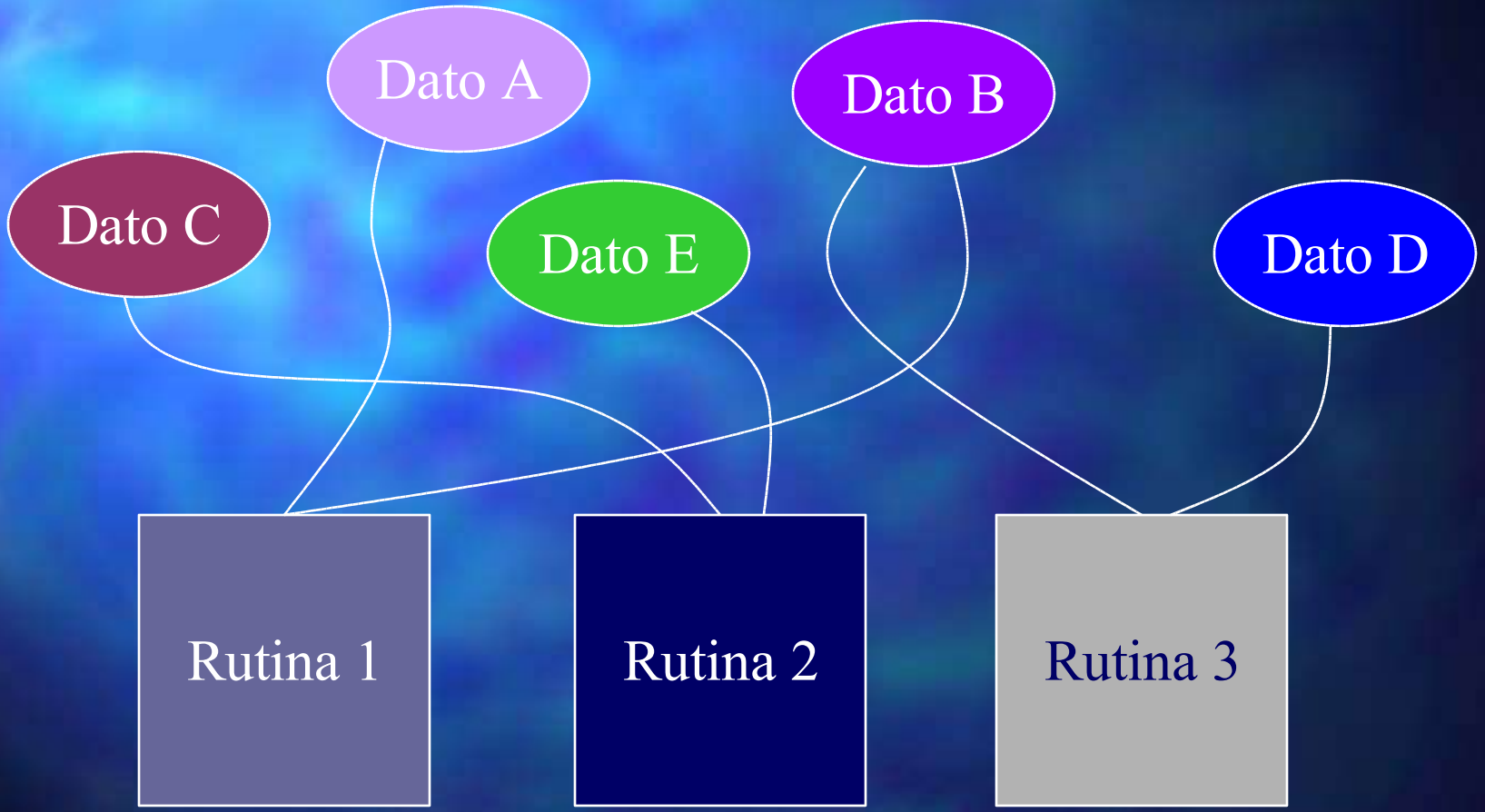
- **Mayor organización**
- **Separación de actividades**
- **Encapsulamiento**
- **Especialización en tareas**



Código Spaghetti



Programación procedural



Programación Orientada a Objetos (OOP)

Programación Orientada a Aspectos (AOP)

Nueva manera de ver a los sistemas de software

Es una evolución de la OOP

Se enfoca en la separación de responsabilidades

Ventajas:

Simplifica el desarrollo y mantenimiento

Aumenta el nivel de encapsulación

Historia

La programación orientada a Aspectos surgió a partir del trabajo de investigación sobre lenguajes de programación orientados a objetos desarrollado entre 1972 y 1980 por el equipo de **Gregor Kiczales** en el **Xerox Palo Alto Research Center**

Ideas de la AOP

Un problema involucra varios asuntos (concerns)

Algunos corresponden a la función específica que debe realizar el programa.

Existen asuntos secundarios que no están relacionados directamente con la función principal del programa.

Ejemplo

Un programa que busque en una base de datos todos los libros de matemáticas.

Asunto principal:

Encontrar todos los libros de matemáticas.

Asuntos secundarios:

Validación de permisos en la BD.

Realización de transacciones con la BD.

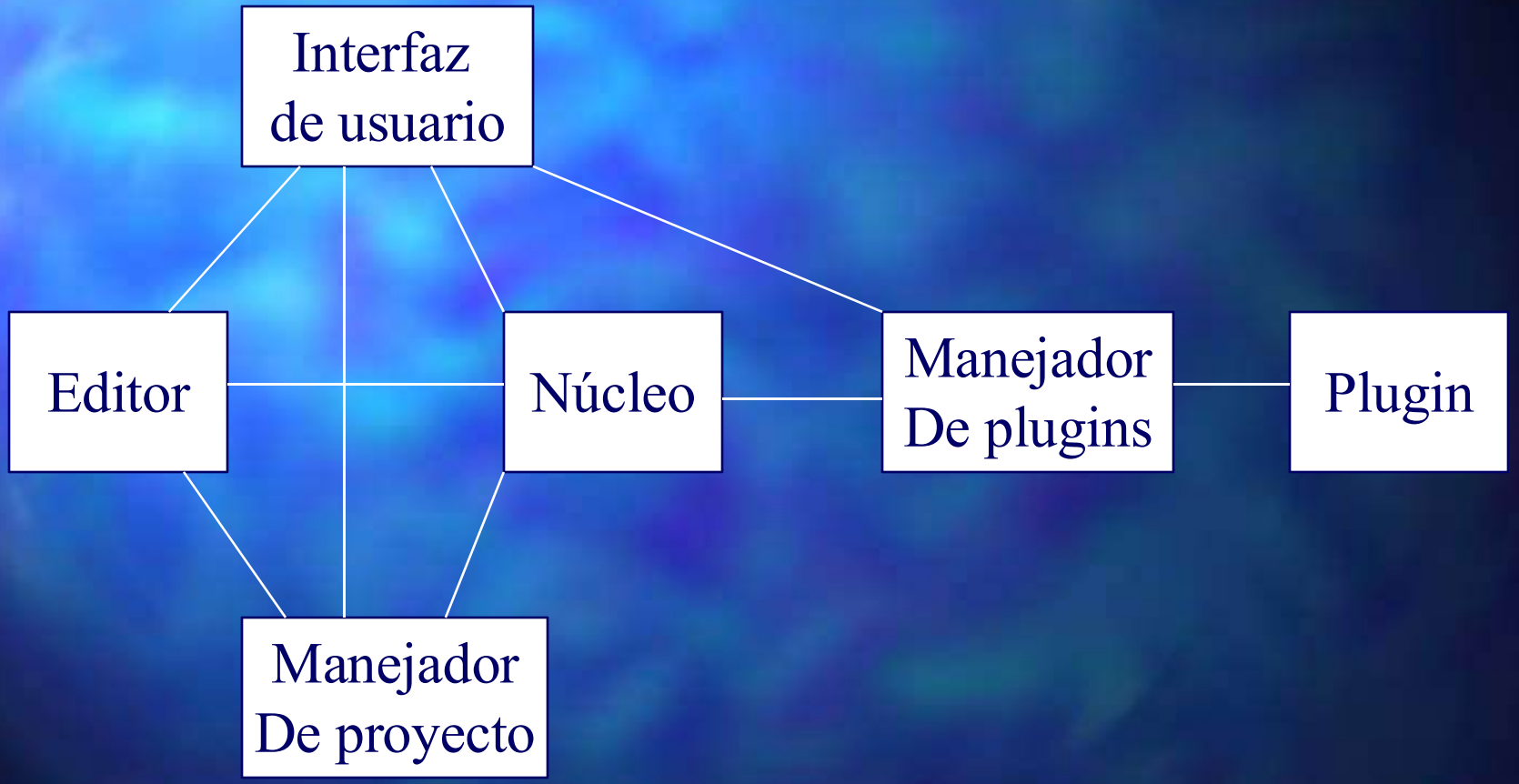
Manejo de excepciones de entrada/salida

Los asuntos que pertenecen a las funciones principales del programa son fáciles de separar:

Cada uno es realizado por un componente del sistema que aparece descrito en la arquitectura.

Corresponden a tareas bien definidas y planeadas desde que se diseñó el modelo del sistema.

Asuntos principales



Asuntos transversales (crosscutting concerns)

Actividades secundarias que no pertenecen a la funcionalidad principal del programa.

Estos asuntos no son fáciles de separar porque afectan a distintos componentes del sistema.

En estos casos se suele escribir código repetido que se encargue del asunto en todos los lugares donde se necesita.

Ejemplo de asuntos transversales: Registrar actividades en bitácora

- EditorImpl
- ManejadorProyectoImpl

El asunto: “escribir en la bitácora” aparece diseminado por todos lados.

Estos asuntos atraviesan la arquitectura (sin respetarla)

Consecuencias de los asuntos transversales

Código disperso “scattered code”

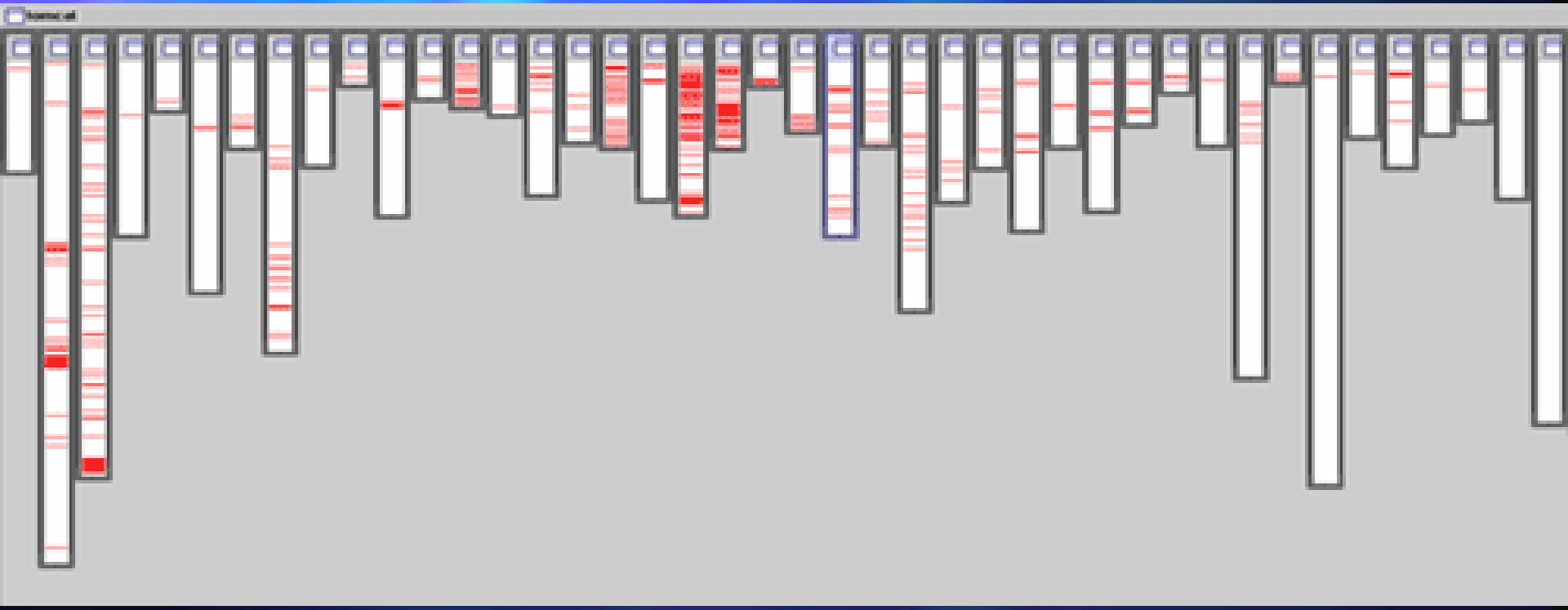
El mismo servicio se invoca desde muchas partes del programa.

Código enredado “tangled code”

Módulos que aparentemente se encargan sólo de su asunto, pero que en realidad tienen que tratar con muchos asuntos secundarios.

Ejemplo:

La aplicación Apache TomCat realiza escrituras en la bitácora.



Aspectos

La Programación Orientada a Aspectos pretende modularizar los asuntos transversales.

Definición:

Un aspecto es un asunto transversal que está modularizado

Lenguajes de programación de aspectos

Un aspecto se programa por separado usando un lenguaje de programación de aspectos.

Algunos ejemplos de estos lenguajes son:

- AspectJ
- Jboss
- AspectWerks
- Nanning

Aspectos y código base

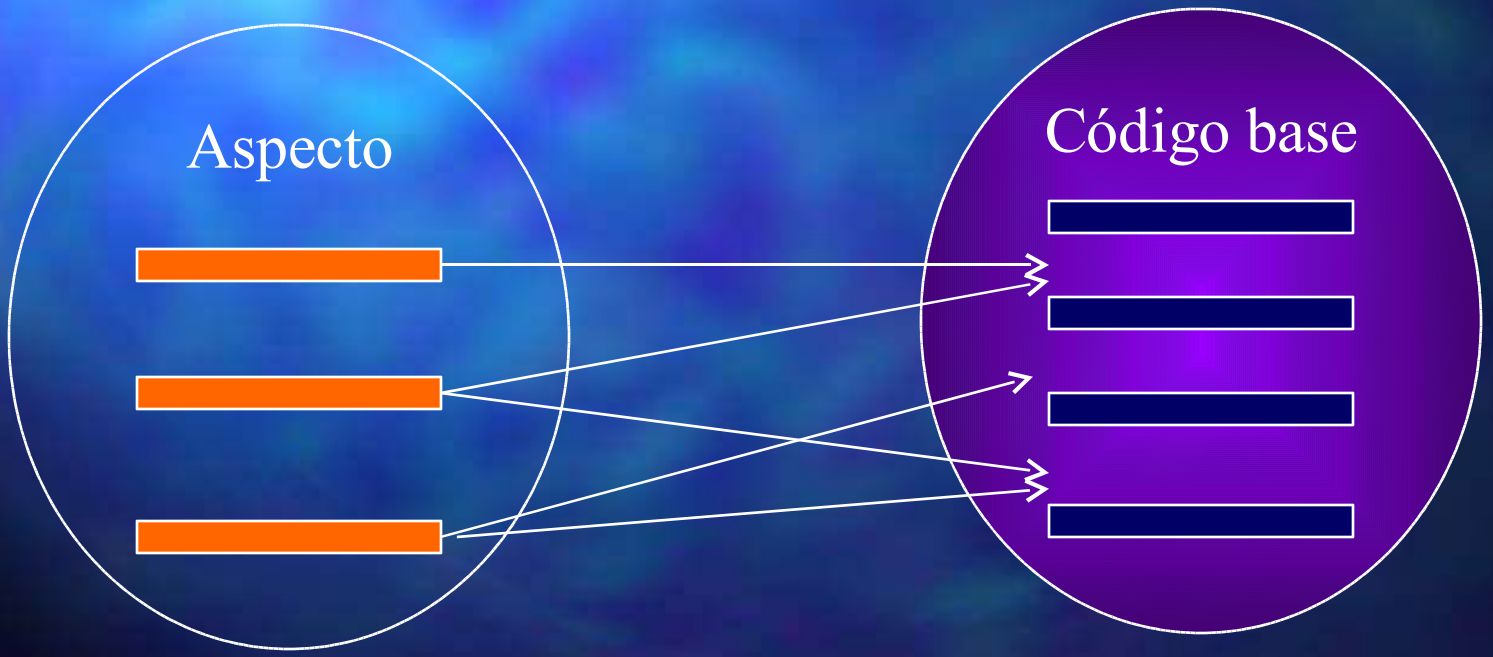
El código base sólo se dedica a sus propias actividades.

Los asuntos transversales se programan por separado

Aspectos y código base

Las acciones del aspecto son inyectadas en el código base (donde se necesiten).

A esto se le llama “entretejido” (weaving)



Aspectos y código base

Los aspectos son capaces de inyectar sus acciones en el código base.

Esto se logra interceptando:

- Llamadas a métodos
- Accesos a variables
- Excepciones
- Retornos de métodos

Elementos de la programación de aspectos

- Join Point (punto de unión)
- Pointcut (intersección)
- Advice (sugerencia)

Join Point (punto de unión)

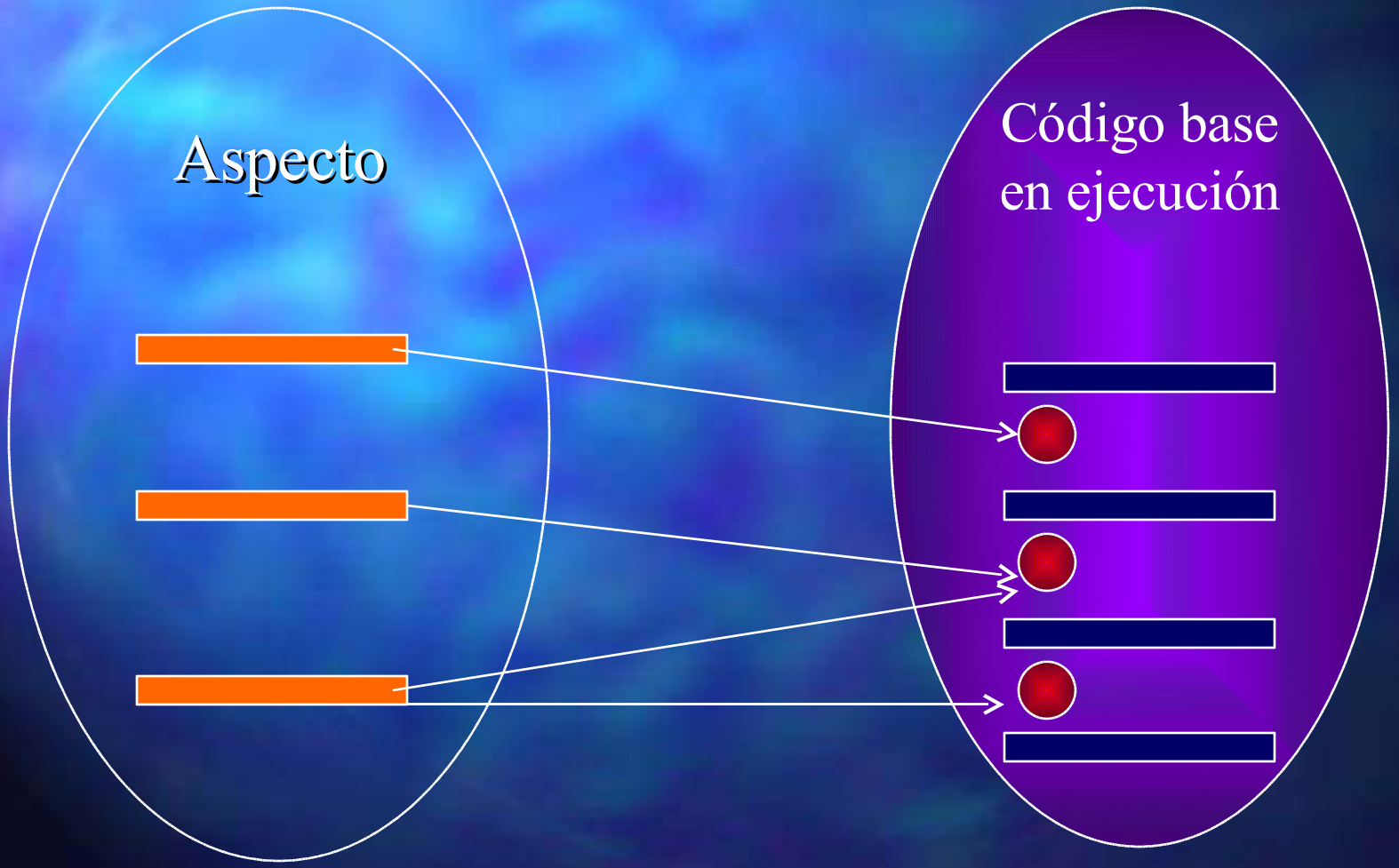
Son los puntos en la ejecución del programa en donde se van a inyectar las acciones del aspecto.

Por ejemplo:

"al invocar al método **editarArchivo** por cuarta vez".

No son puntos en el código fuente, sino situaciones en la ejecución.

Join Point (punto de unión)



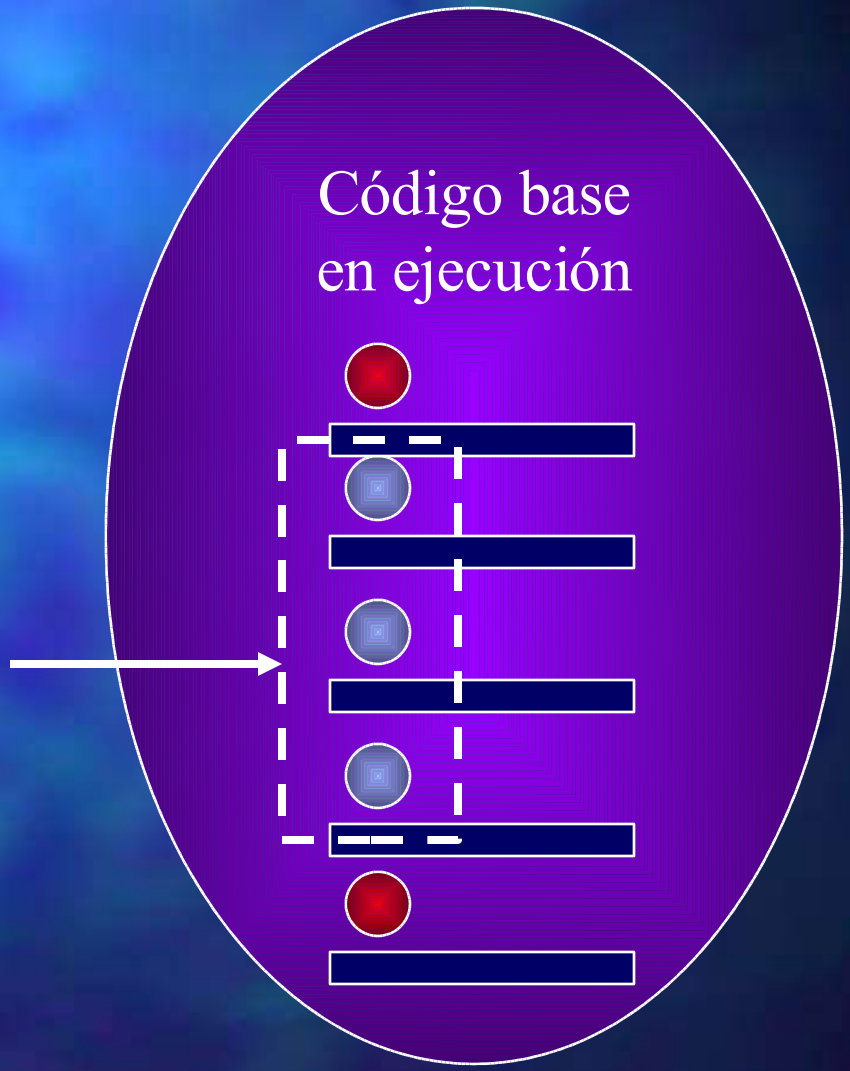
Pointcut (intersección)

Los puntos de unión son agrupados mediante descripciónes lógicas llamadas intersecciones (pointcuts).

Intersección: Es un predicado que especifica varios puntos de unión de manera descriptiva. Nunca por ennumeración.

Pointcut (intersección)

Las intersecciones agrupan a todos aquellos puntos de unión en donde se van a inyectar las mismas actividades.



Ejemplo:

Se necesita registrar en la bitácora cada vez que se modifica algún elemento del TreeModelProyecto.

Hay que definir una intersección así:

call (void TreeModelProyecto+.add*(E))

Significa: "al invocar (call) cualquier método de la clase TreeModelProyecto o de una sublclase (+), cuyo nombre comience con add (add*), que tenga void como tipo de retorno y con el parámetro E"

Ejemplo:

También se pueden crear pointcuts compuestos, por ejemplo para incluir aquellos métodos que comienzan con remove:

```
(   call ( void TreeModelProyecto+.add*(E) )  ||
    call ( void TreeModelProyecto+.remove*(E) )   )
```

Este pointcut especifica todos los puntos de unión dentro de la clase TreeModelProyecto en los que se necesitará escribir en la bitácora.

Ejemplo:

Además de **call** hay otros constructores de *pointcuts*:

get (al leer el valor de una variable)

set (al modificarlo)

cflowbelow (dentro del flujo de control)

initialization (al inicializar un objeto), etc.

target (clase que recibe el mensaje)

args (argumentos de una llamada), etc.

Advice (sugerencia)

Las sugerencias son las actividades que se van a inyectar en los puntos de unión.

En el ejemplo, hay que agregar la instrucción **Bitacora.escribe(...);** después de los puntos de unión indicados por el *pointcut*.

```
after( ): (  call ( void TreeModelProyecto+.add*(E) ) )
{
   Bitacora.escribe("Se agrega el elemento: "+E+" al TreeModel");
}
```

Advice (sugerencia)

En vez de **after**, se podría haber puesto:

before (antes),

after returning (después de ejecutar un método en forma normal),

after throwing (después de arrojar una excepción),

around (en lugar de ejecutar el punto de unión).

Aplicaciones

Trazas

Seguridad

Persistencia

Funcionalidad arbitraria

Monitoreo y administración

Cacheado de información