



פיתוח מערכות תוכנה בשפת Java  
מבוא לארכיטקטורת תוכנה  
חתכי רוחב - Crosscutting Concerns

**אוהד ברזילי**

# מודולריות בתכנות מונחה עצמים

■ לתכנות מונחה עצמים יש החסרונות שלו וצריך להיות ערים להם

■ חסרון בולט קשור לניהול של חתכי רוחב ( crosscutting concerns ) במערכת תוכנה – הפוגע במודולריות של המערכת

■ ההבנה כי למודולריות נכונה של תוכנה יש השפעה על סיבוכיות התוכנה והבנתה מתוארת ב- **On the criteria to be used in decomposing systems into modules** מאת David Parnas משנת 1972

# Separation of Concerns

- המונח **Separation of Concerns** נתבע ע"י Edsger W. Dijkstra בשנת 1974 במאמר On the role of scientific thought

it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

# דוגמא

■ כחלק מהפתולוגיה של מערכת תוכנה מוכוונת עצמים אנו מגדירים מחלקות לייצוג Core Concerns (או Business Logic) אולם אנו מזניחים עניינים (הבטים) אחרים

■ נניח שכתבנו תוכנה שעושה משהו

■ במערכת התוכנה נמצא את המחלקה `SomeBusinessClass`

עם השרות `someOperation`

■ למשל המחלקה `BankAccount` עם השרות `withdraw` (רק

לצורך הדוגמא – הדבר תקף כמעט לכל תוכנה אמיתית)

# The wrong way

```
public class SomeBusinessClass extends OtherBusinessClass {  
    // Core data members  
    // Override methods in the base class  
    public void someOperation(OperationInformation info) {  
        // ==== Perform the core operation ====  
    }  
    ...  
}
```

# The wrong way(2)

- But what about logging capabilities ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
    }  
}
```

# The wrong way(3)

- **Actually, we want it multithreaded...**

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```

# The wrong way(4)

- Who enforces your contract ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...ensure info satisfies contract  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```



# The wrong way(5)

## ■ Authorization ? Authentication ?

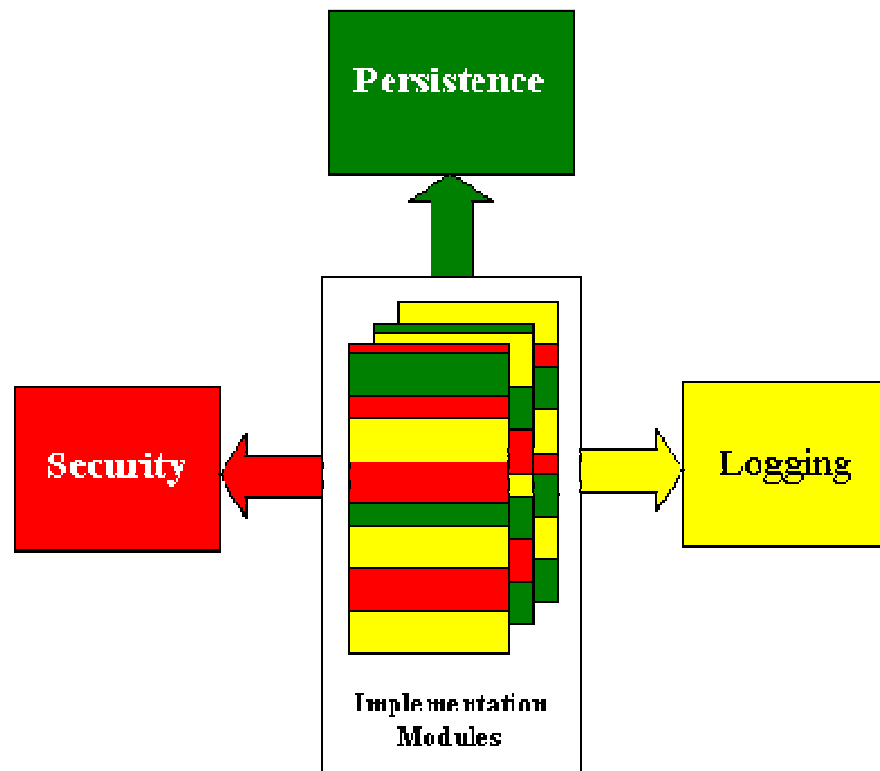
```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...ensure authorization  
        ...ensure info satisfies contract  
        ...lock the object - thread safety  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
}
```

# The wrong way(6)

- Persistence ? Cache consistency ?

```
public class SomeBusinessClass extends OtherBusinessClass {  
  
    // Core data members  
    ...Log stream ;  
    ...cache_update_status ;  
    // Override methods in the base class  
  
    public void someOperation(OperationInformation info) {  
        ...ensure authorization  
        ...ensure info satisfies contract  
        ...lock the object - thread safety  
        ...ensure cache is up to date  
        ...log the start of operation  
        // ==== Perform the core operation ====  
        ...log the completion of operation  
        ...unlock the object  
    }  
    public void save(PersitanceStorage ps) {...}  
    public void load(PersitanceStorage ps) {...}  
}
```

# מה קיבלנו?



בלאגן בשתי רמות:

■ ברמת המיקרו (השרות הבודד):

■ Code Tangling

■ הוא כבר לא עושה "רק משהו אחד" - לא מודולרי

■ ראו תרשים <=

■ ברמת המאקרו (מערכת התוכנה):

■ Code Scattering

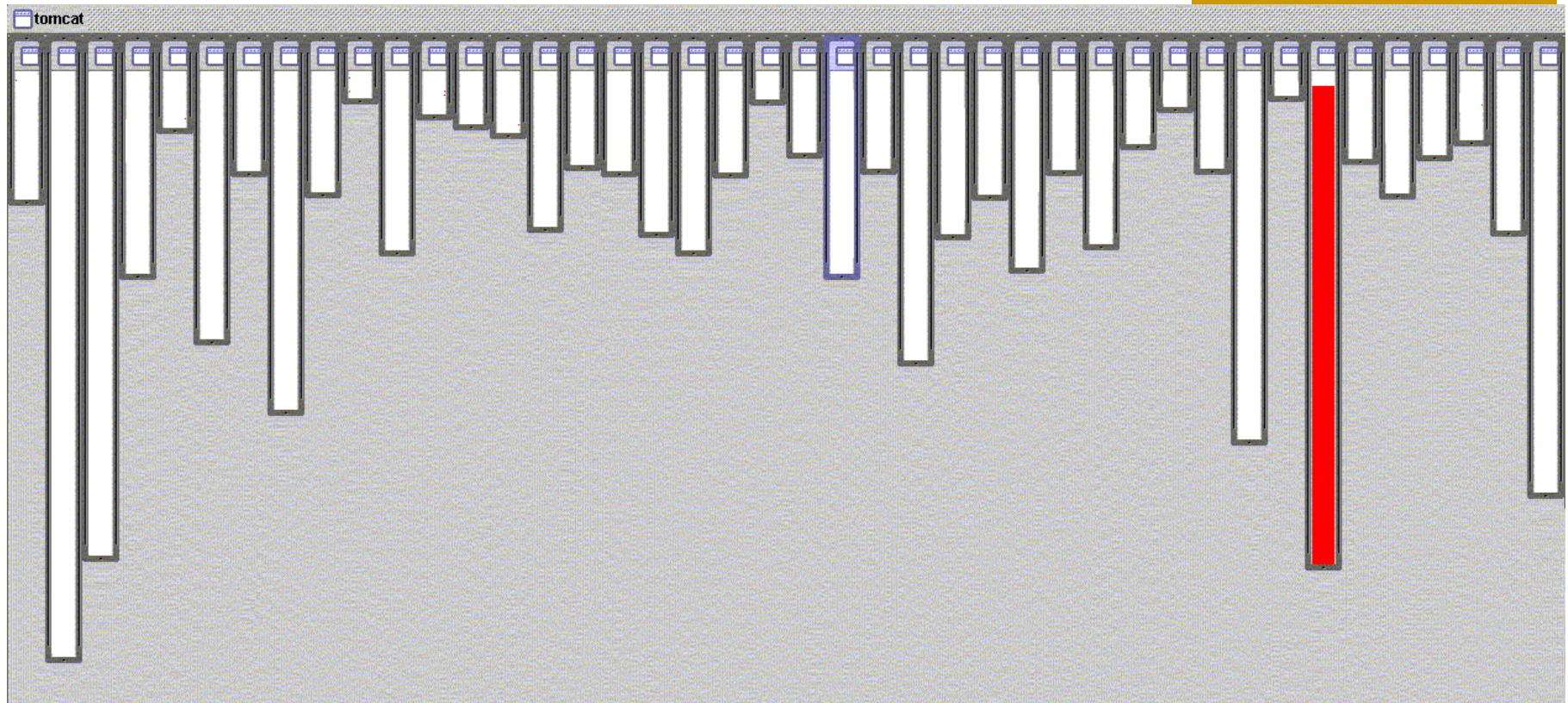
■ שכפול קוד, קטעי קוד קשורים אינם מופיעים יחד

■ ראו תרשימים גם בשקפים הבאים

■ שבירת המודולריות נוצרת בגלל אופי הספק-לקוח של תכנות מונחה עצמים

# good modularity

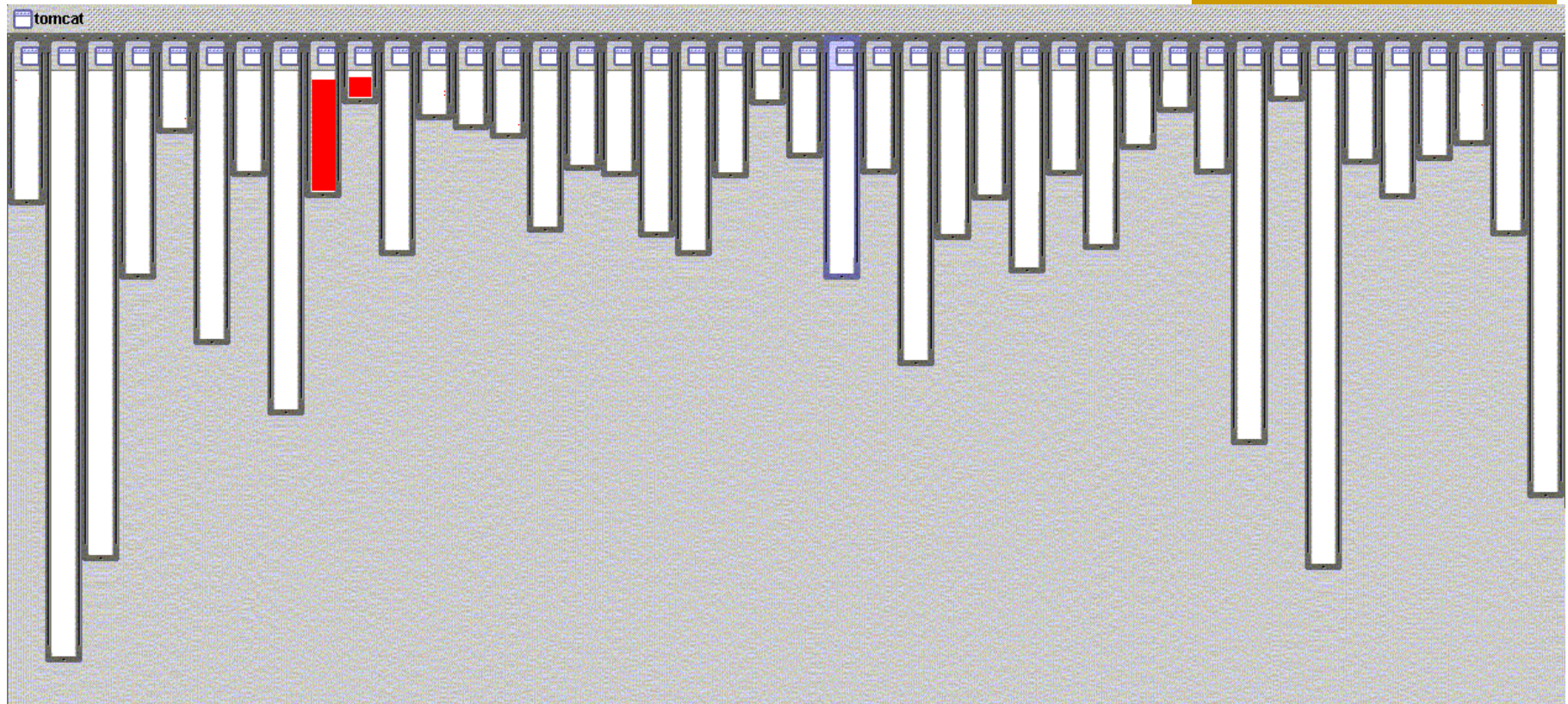
## XML parsing



- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

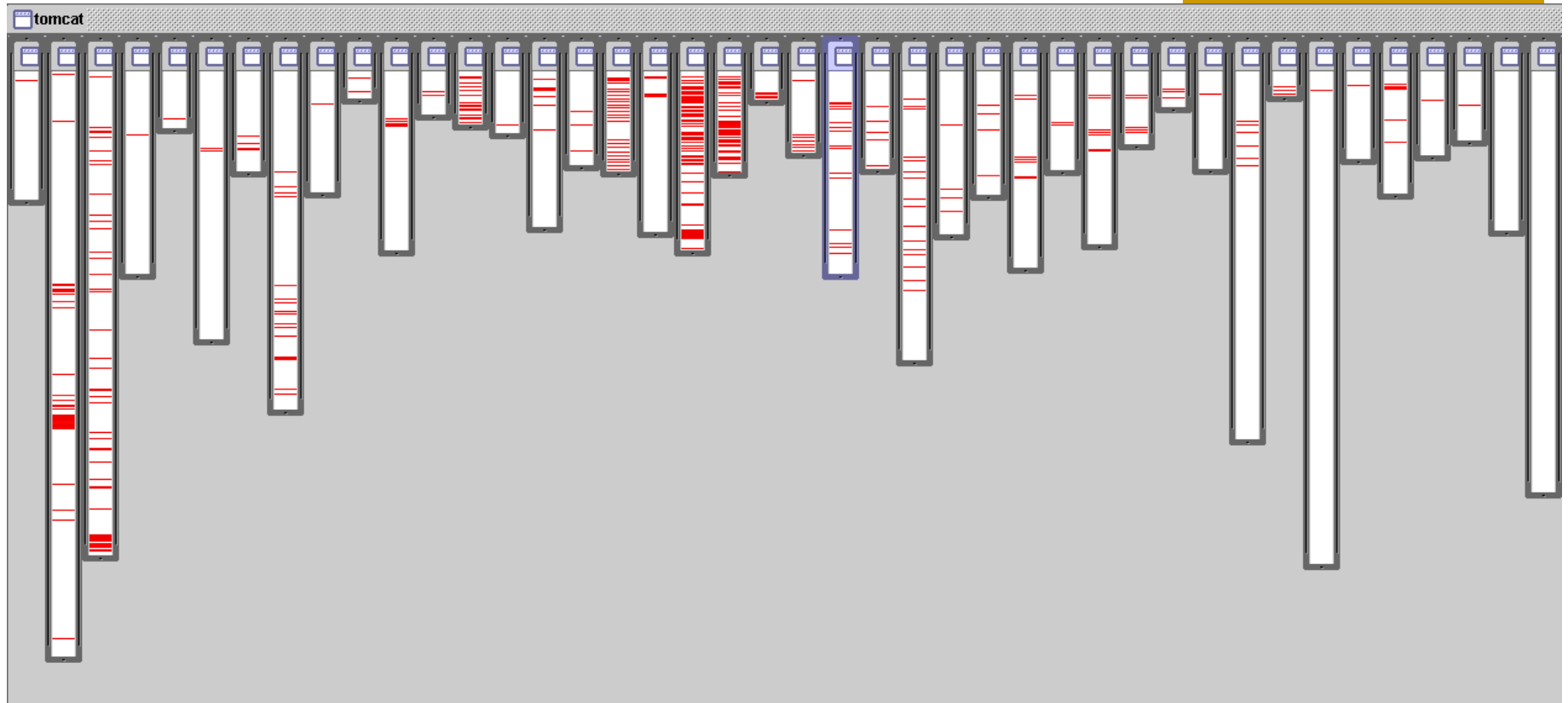
# good modularity

## URL pattern matching



- URL pattern matching in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)

# logging is not modularized...



- where is logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

# אילו רק יכולנו...

## ApplicationSession

```
public class ApplicationSession extends HttpSession {
    // ...
}
// ...

```

## StandardSession

```
public class StandardSession extends ApplicationSession {
    // ...
}
// ...

```

```
public class StandardSessionManager extends SessionManager {
    // ...
}
// ...

```

## SessionInterceptor

```
public class SessionInterceptor implements HttpSessionInterceptor {
    // ...
}
// ...

```

## StandardManager

```
public class StandardManager extends SessionManager {
    // ...
}
// ...

```

## StandardSessionManager

```
public class StandardSessionManager extends SessionManager {
    // ...
}
// ...

```

## ServerSession

```
public class ServerSession extends ApplicationSession {
    // ...
}
// ...

```

## ServerSessionManager

```
public class ServerSessionManager extends SessionManager {
    // ...
}
// ...

```

# שבירת המודולריות

■ נציג 4 גישות לפתרון הבעיה:

■ מעבר לשימוש ברכיבים (components) במקום עצמים

■ כגון: Servlets או EJB's

■ חסרון: Domain Specific Framework

■ פתרונות ברמת שפת התכנות ותבניות העיצוב:

■ כגון: Mixin או Dynamic Proxy

■ חסרון: דורש "תחזוקה ידנית" של העיצוב

■ מעבר לשפת תכנות בפרדיגמה התומכת ביחסים נוספים בין מחלקות

■ כגון: AspectJ או שפת E

■ חסרון: לימוד שפה חדשה

■ פתרון ברמת תצורת המערכת והעצמים (configuration)

■ כגון: Spring AOP

■ גישה פרגמטית פופולרית



# Components and Frameworks

■ כאשר אנו מזהים את הפעולות החוזרות בתחום מסוים (domain specific) אנו יכולים לכתוב מסגרת עבודה (framework) ייעודית לאותו תחום

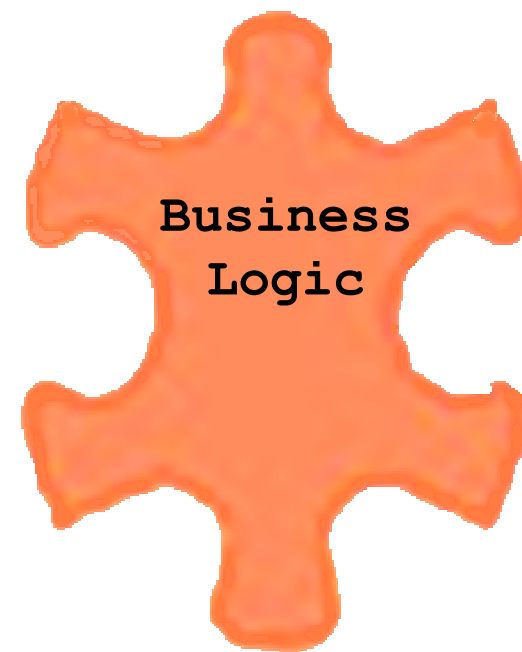
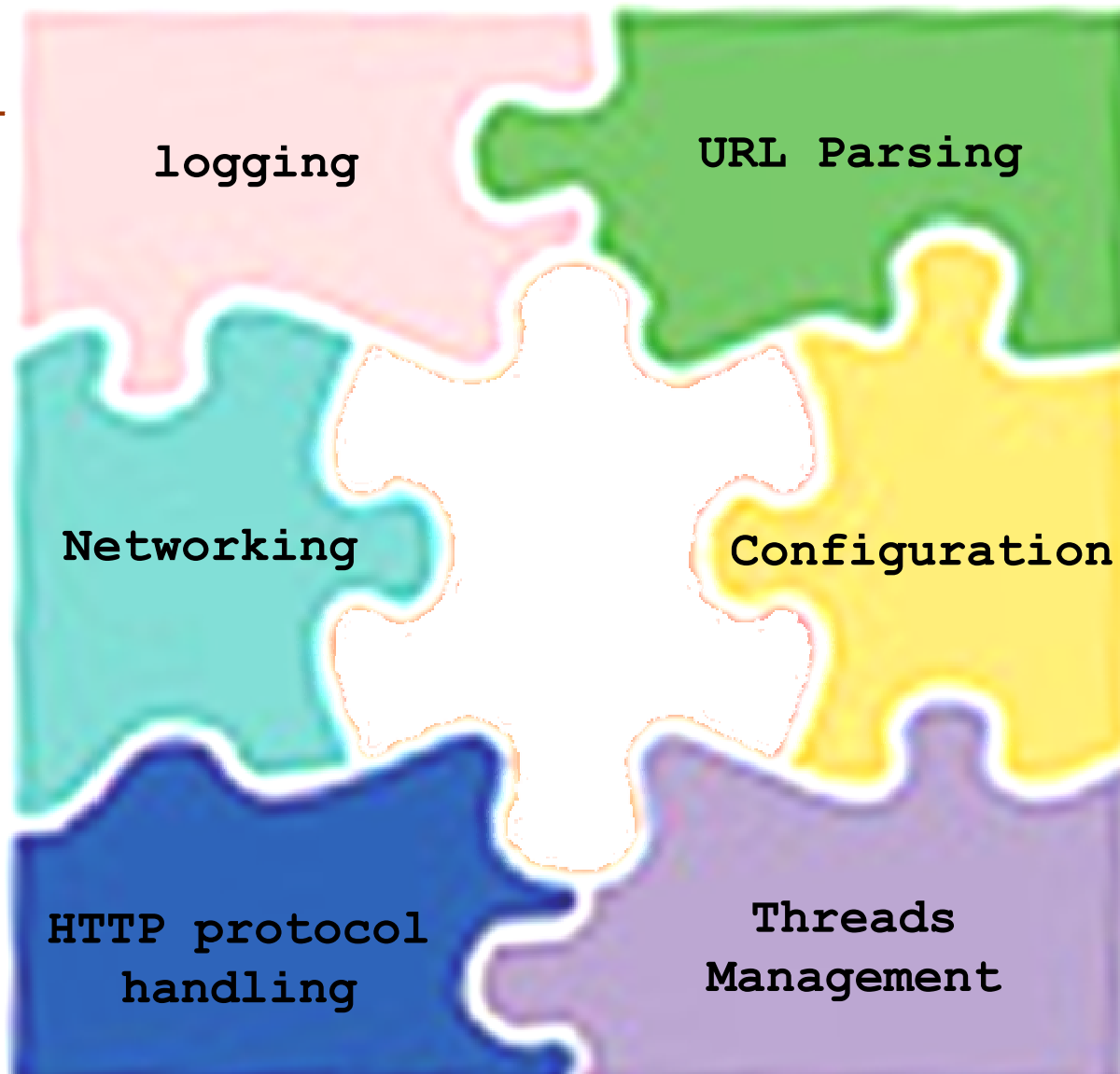
■ למשל:

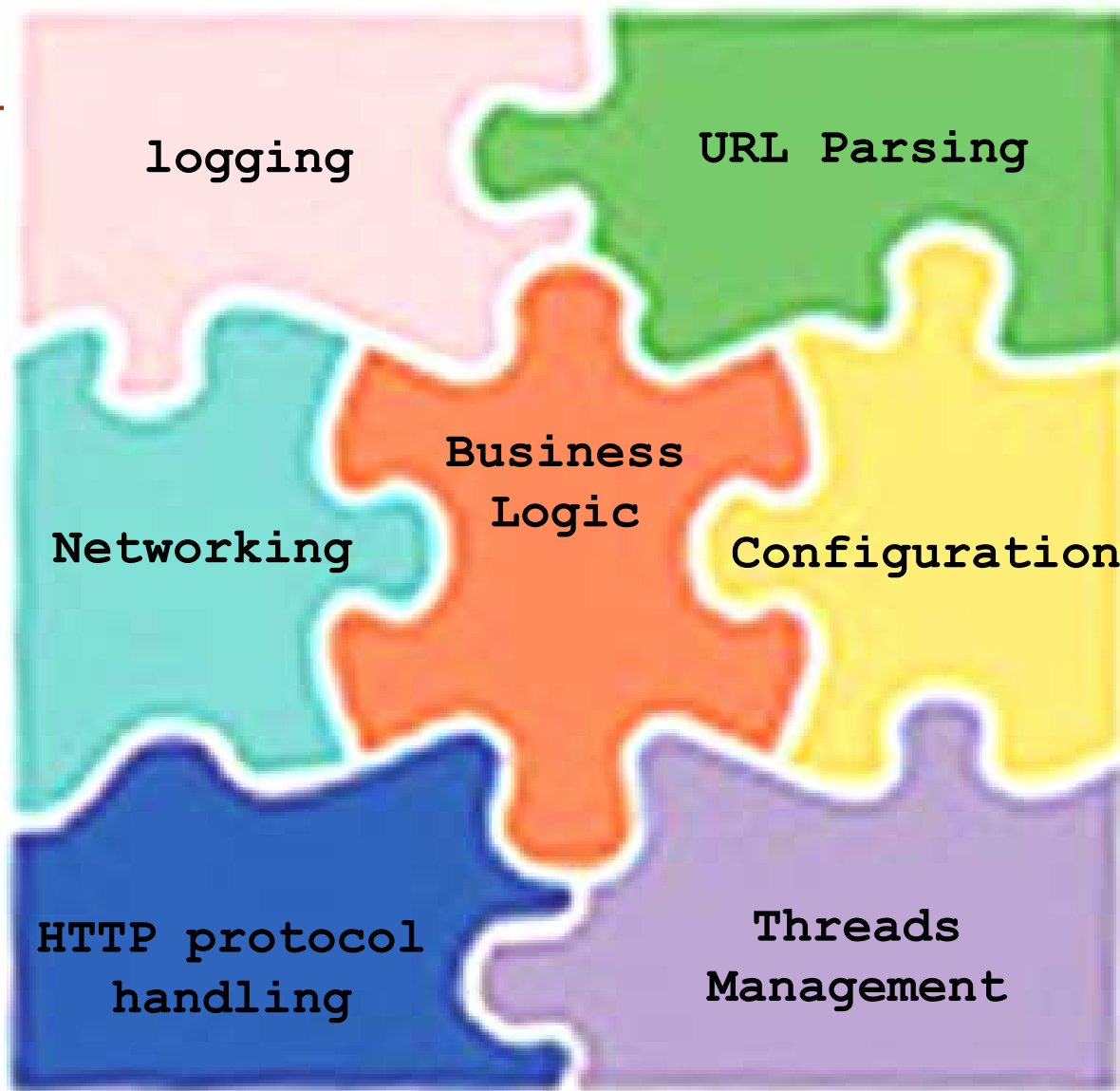
■ היינו רוצים שמסגרת עבודה ליישומי אינטרנט תטפל בניתוח שורת ה URL, הבטי התקשורת, קריאת וכתובת הודעות בפרוטוקול http, ניהול תזמון החוטים לטיפול בלקוחות מרובים ועוד...

■ הרכיבים שישתלו בתוך המסגרת יטפלו אך ורק בלוגיקה עסקית – במה שהאתר המסוים עושה ולא ידרשו לטיפול בהבטים החוזרים

■ כדי לעבוד עם מסגרת כזו, מפתחי הרכיבים חייבים לשמור על מבנה מסוים, בדרך כלל לרשת ממחלקה מסוימת או לממש ממשק מסוים

■ ביישומי אינטרנט אחד הרכיבים השימושיים ביותר הוא Servlet

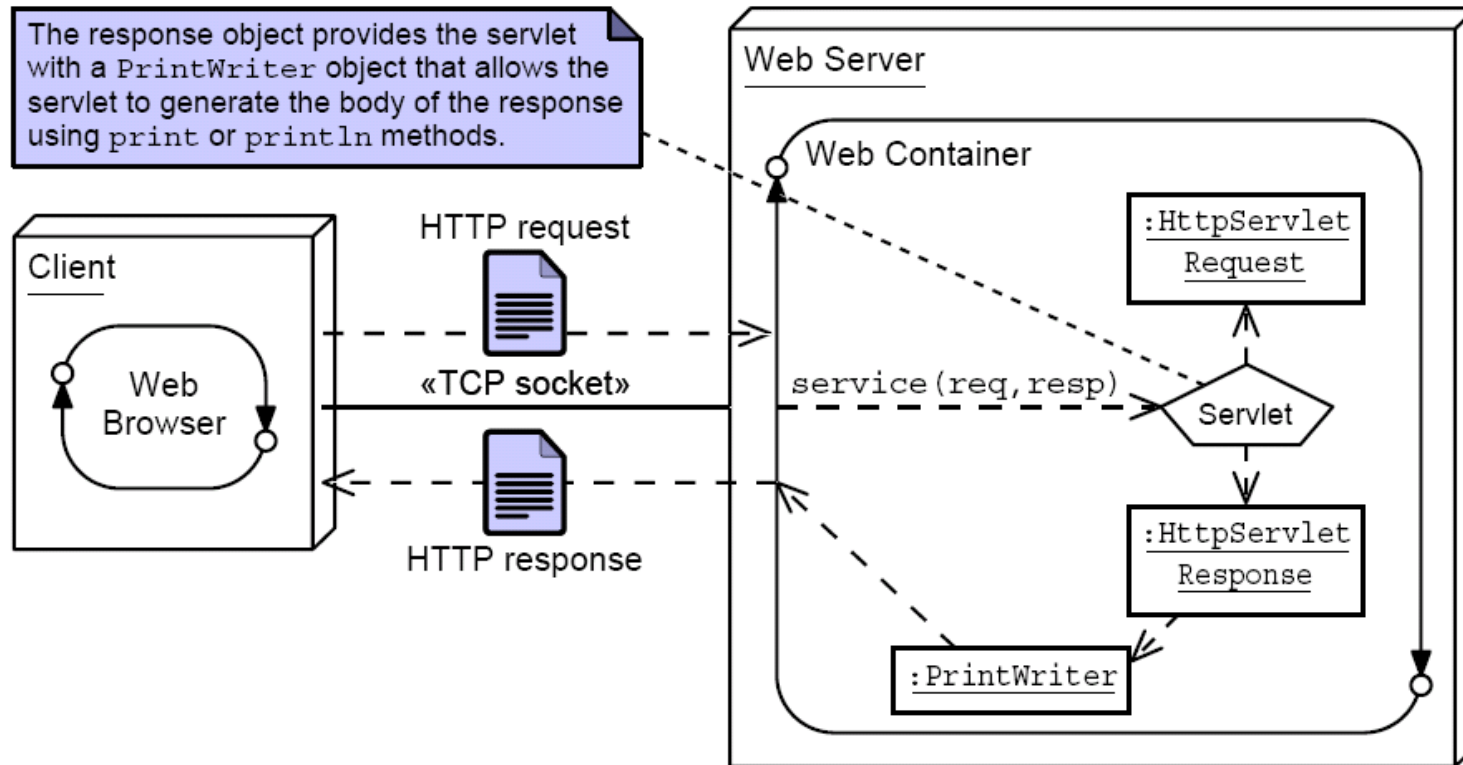




■ התקנת servlet  
בשרת, פירושה  
העתקת קובץ ה-  
class לתיקייה  
המתאימה

■ בפעם הבאה  
שיופעל השרת,  
ה-servlet  
יעבוד כחלק  
אינטגרלי ממנו

# ארכיטקטורת Web Container



על המחלקה הזו הוא יפעיל את השרות `doGet`, (במקרה הכללי `service`) וידאג להעביר לה כפרמטרים **מחלקות עזר** שבעזרתם תקרא את הפרמטרים אם הועברו כאלה בשורת הכתובת, ותייצר הודעת תשובה

# Hello World Servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

# Proxy Design Pattern

Proxy – יצירת פונדקאי או שומר מקום לעצם כדי לבצע הפשטה על הגישה אליו ■

■ לצורכי יעילות, פיקוח, מודולריות ועוד...

■ לדוגמא:

■ תמונות "כבדות" במסמך, מצביעים חכמים

■ Access Proxy

■ Firewall Proxy

■ Virtual Proxy (Lazy Proxy)

■ Remote Proxy

■ Synchronization Proxy

■ הרעיון ממומש במערכות תוכנה ובספריות רבות

■ Java מספקת את המחלקה `InvocationHandler` המאפשרת לנו להגדיר `Proxy` משלנו

```
/** A Proxy that intercepts String arguments & converts them to  
uppercase Then, as usual, it will forward method calls to the  
enclosed object */
```

```
import java.util.*;
```

```
import java.lang.reflect.*;
```

```
class UppercaseProxy implements InvocationHandler {  
    private Object obj;
```

```
    public UppercaseProxy(Object obj) {  
        this.obj=obj;  
    }
```

```
    public Object invoke(Object proxy, Method m, Object[] args)  
        throws Throwable {  
        if (args!=null){  
            for (int i = 0; i < args.length; i++) {  
                if ( args[i] instanceof String) {  
                    String s = (String)args[i];  
                    args[i] = s.toUpperCase();  
                }  
            }  
        }  
        return m.invoke(obj, args);  
    }  
}
```

```

/** You can now wrap this proxy around any object (e.g: List),
    provided you only work through interfaces */

public class ProxyTest {
    public static void main(String[] args) throws Exception {

        ArrayList myList=new ArrayList();

        // Create a proxy that wraps myList and implements
        // interface List:
        Object proxy = Proxy.newProxyInstance(
            java.util.List.class.getClassLoader(),
            new Class[] {java.util.List.class}, // interfaces
            new UppercaseProxy(myList)); // wrapped obj

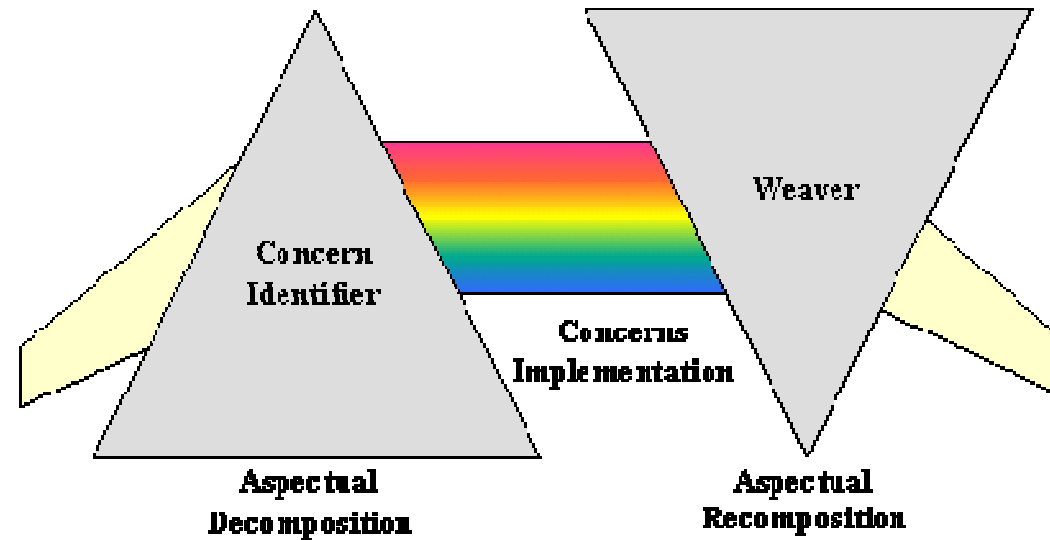
        // Add items to list, through the proxy:
        List pList= (List) proxy;
        pList.add("Aa");
        pList.add("bbb");
        System.out.println(pList);
    }
}

```

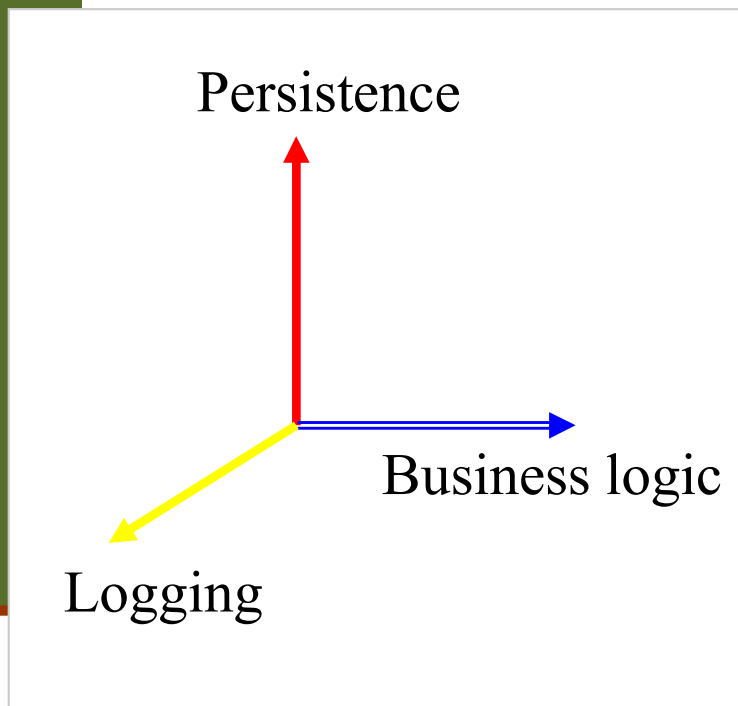


# AspectJ Terminology

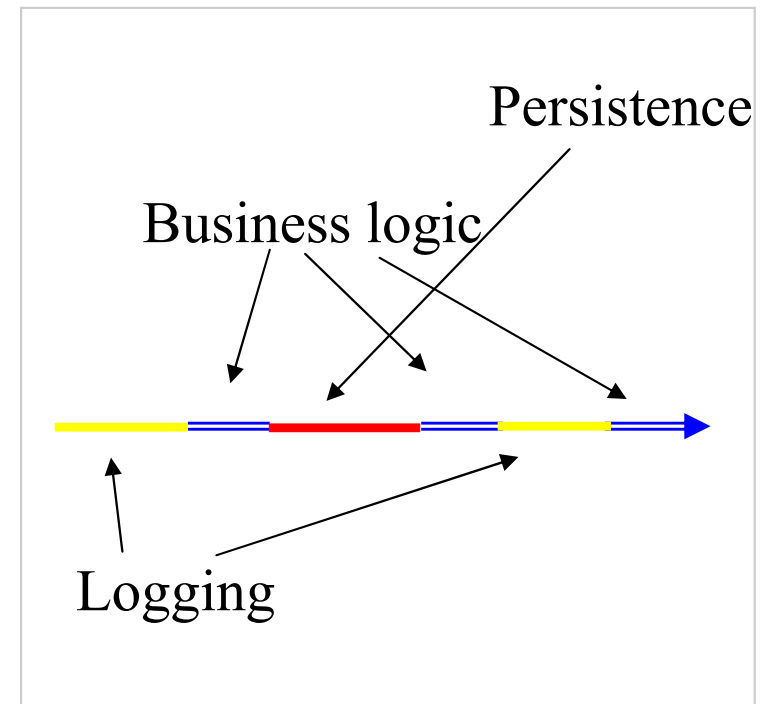
- A Weaver
- Join point
- Pointcut
- Advice
- Aspect



# Weaving orthogonal concerns



Implementation mapping



## ■ HelloWorld.java

```
public class HelloWorld {  
  
    public static void say(String message) {  
        System.out.println(message);  
    }  
}
```

## ■ Test.java

```
public class Test {  
  
    public static void main(String[] args) {  
        HelloWorld.say("Hello World");  
    }  
}
```

➤ `ajc HelloWorld.java Test.java`

➤ `java Test`

Hello World

## ■ MannersAspect.java

```
public aspect MannersAspect {  
  
    pointcut saying() :  
        call(public static void HelloWorld.say*(..));  
  
    before() : saying() {  
        System.out.print("Good day! ");  
    }  
  
    after() : saying() {  
        System.out.println("Thank you!");  
    }  
}
```

➤ `ajc HelloWorld.java MannersAspect.java Test.java`

➤ `java Test`

Good day! Hello World

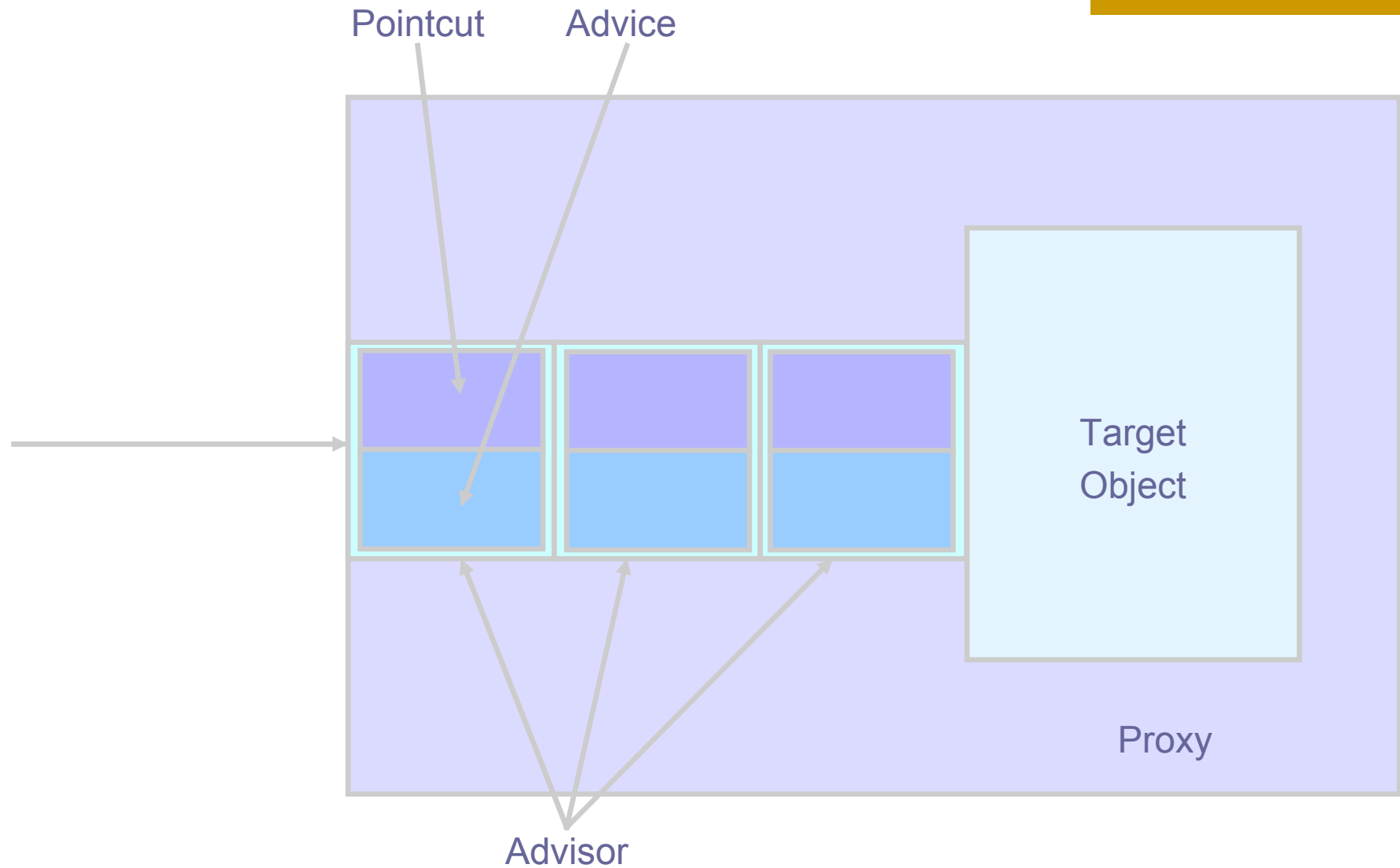
Thank you!

# Spring AOP basics

---

- A proxy-based AOP framework
  - JDK proxies
  - CGILIB proxies
- Targets many J2EE use cases
- Powerful integration with AspectJ
  - Allowing tapping into full AOP features

# Spring AOP schematic



# Configuring through XML: Defining advisor

```
<bean id="spamPreventionAdvisor"  
    class="o.s.aop.support.DefaultPointcutAdvisor">  
    <property name="advice">  
        <bean class="example.SpamPreventionInterceptor"/>  
    </property>  
    <property name="pointcut">  
        <bean class="o.s.aop.support.JdkRegexpMethodPointcut">  
            <property name="pattern" value=".*send.*"/>  
        </bean>  
    </property>  
</bean>
```

# Configuring through XML: Creating proxy

```
<bean id="emailerTarget" class="example.EmailerImpl">
</bean>
```

```
<bean id="emailer" class="o.s.aop.framework.ProxyFactoryBean">
  <property name="target" ref="emailerTarget"/>
  <property name="interceptorNames">
    <list>
      <value>spamPreventionAdvisor</value>
    </list>
  </property>
  <property name="proxyInterfaces">
    <list>
      <value>example.Emailer</value>
    </list>
  </property>
</bean>
```

} Target

} Advice/Advisor

} Proxy interface (optional)



# Bean client

```
package example;
...

public class Main {
    public static void main(String[] args) {

        ApplicationContext context
            = new ClassPathXmlApplicationContext("beans.xml");

        Mailer mailer = (Mailer)context.getBean("emailer");

        mailer.send("ramnivas@aspectivity.com",
            "Hi! Your paypal account...");
        mailer.send("ramnivas@aspectivity.com",
            "Hi, I have an AspectJ question...");
    }
}
```

# Advantages of proxy-based approach

---

- Requires no special compiler
- Allow per-object interceptors
  - Not per-class
- Less full-fledged
  - Only method-level interception
  - Allows easing into AOP
- Easy to modify applicable interceptors dynamically

# Disadvantage of proxy-based approach

- Limitation of method-only interception
  - No field-access or object creation
- Explicit creation of proxy required
  - Can't use 'new' to create objects
- Calls to 'self' don't go through interceptors
- Low performance
  - Typically not a huge concern, but beware
- Over-exposure of context

# Spring-AspectJ integration

---

- Provides power of AspectJ in Spring
- Core message
  - Spring AOP is good enough in many cases
  - But, you often need more power
- Multiple level of integration
  - Something for every one

# Do away with XML: @AspectJ

```
package example;

@Aspect
public class EmailLogger {
    @Before(execution(* send(String, String))
            and args(address, *))
    public void log(JoinPoint.StaticPart tjpsp,
                   String address) {
        System.err.println(
            "Invoking " + tjpsp.getSignature() +
            " for " + address);
    }
}
```

# Resources

## ■ Spring books

- *Pro Spring*, by Rob Harrop and Jan Machacek
- *Professional Java Development with the Spring Framework*, by Rod Johnson, Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu
- *Spring in Action*, by Craig Walls

## ■ AspectJ books

- *AspectJ in Action*, by Ramnivas Laddad
- *Eclipse AspectJ*, by Adrian Colyer, Andy Clement, George Harley, Matthew Webster