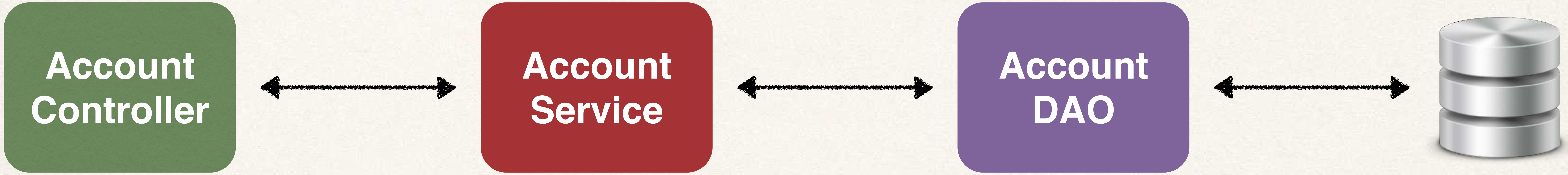
Aspect-Oriented Programming (AOP) Overview

Application Architecture



Code for Data Access Object (DAO)

```
public void addAccount(Account theAccount, String userId) {

        entityManager.persist(theAccount);
}
```

New Requirement - Logging

From: The Boss

- Need to add logging to our DAO methods
  - Add some logging statements before the start of the method
- Possibly more places … but get started on that ASAP!

DAO - Add Logging Code

```
public void addAccount(Account theAccount, String userId) {



    entityManager.persist(theAccount);


}
```

DAO - Add Logging Code

```
public void addAccount(Account theAccount, String userId) {



    entityManager.persist(theAccount);

}
```

New Requirement - Security

From: The Boss

- Need to add security code to our DAO
  - Make sure user is authorized before running DAO method

Add Security Code

```
public void addAccount(Account theAccount, String userId) {



    entityManager.persist(theAccount);


}
```

Add Security Code

```
public void addAccount(Account theAccount, String userId) {


    entityManager.persist(theAccount);


}
```

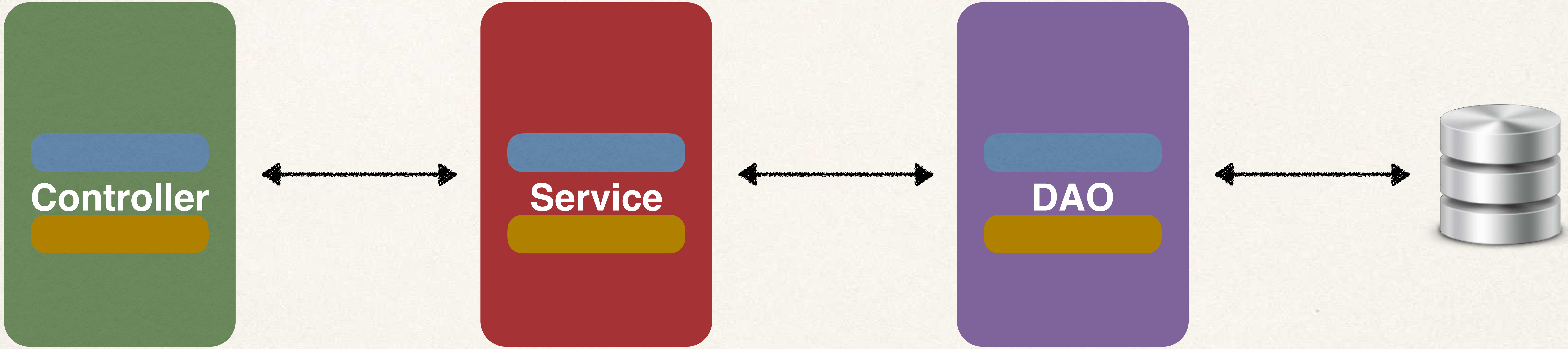
Add Security Code

```
public void addAccount(Account theAccount, String userId) {

    entityManager.persist(theAccount);

}
```

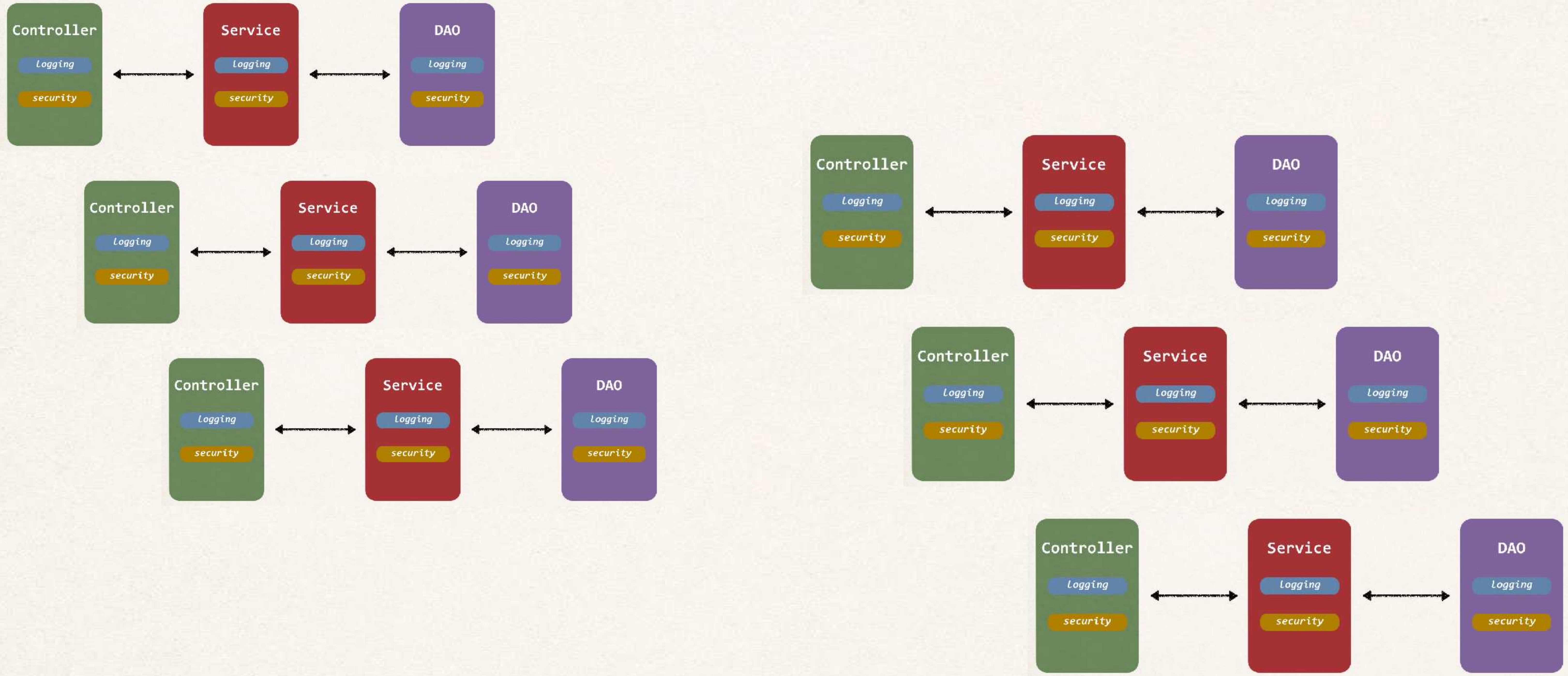
By the way

From: The Boss

- Let's add it to all of our layers…

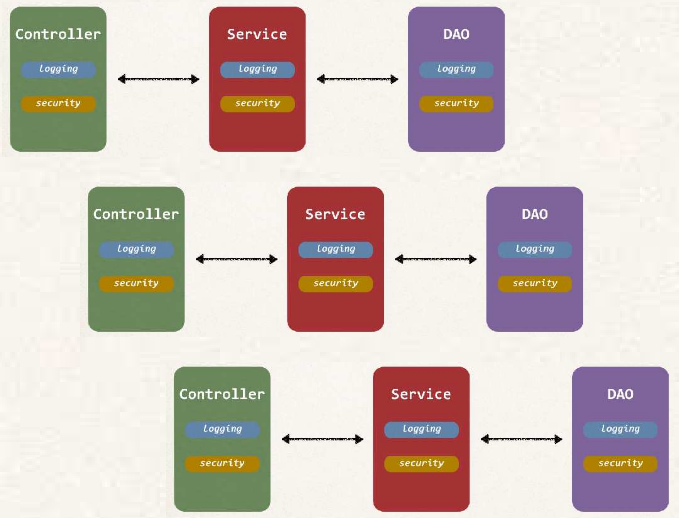


I'm Going Crazy Over Here

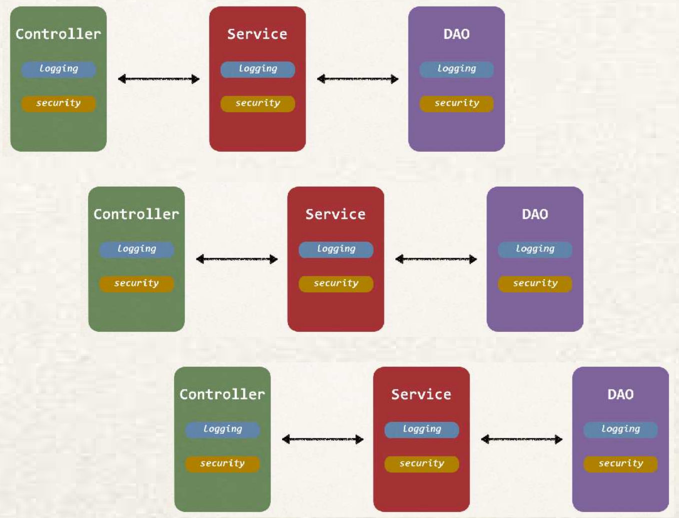


Two Main Problems

- **Code Tangling**
  - For a given method: addAccount(…)
  - We have logging and security code tangled in



- **Code Scattering**
  - If we need to change logging or security code
  - We have to update ALL classes



Other possible solutions?

- **Inheritance?**
  - Every class would need to inherit from a base class
  - Can all classes extends from your base class? … plus no multiple inheritance
- **Delegation?**
  - Classes would delegate logging, security calls
  - Still would need to update classes if we wanted to
    - add/remove logging or security
    - add new feature like auditing, API management, instrumentation

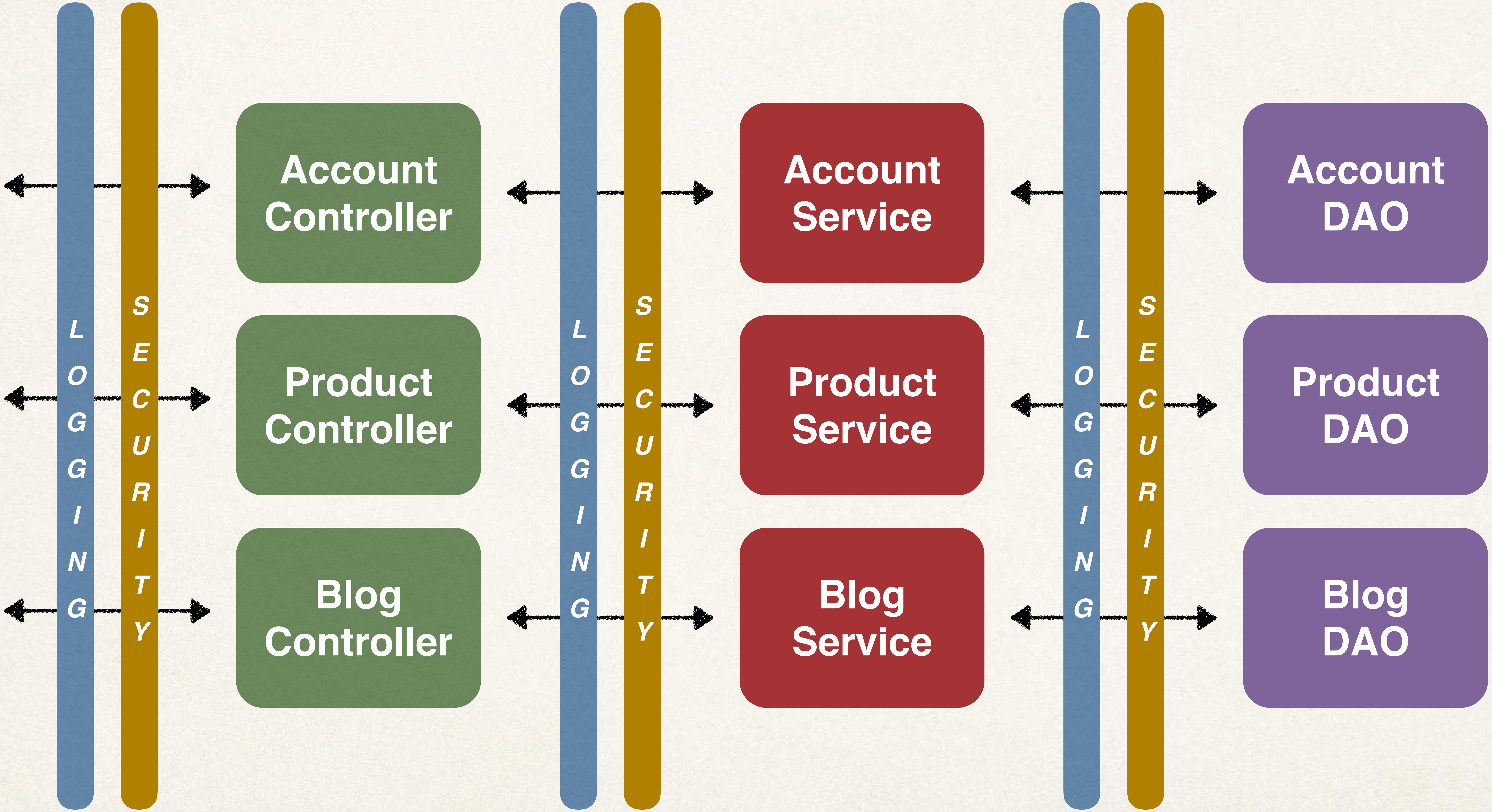
Aspect-Oriented Programming

- Programming technique based on concept of an Aspect
- Aspect encapsulates cross-cutting logic

Cross-Cutting Concerns

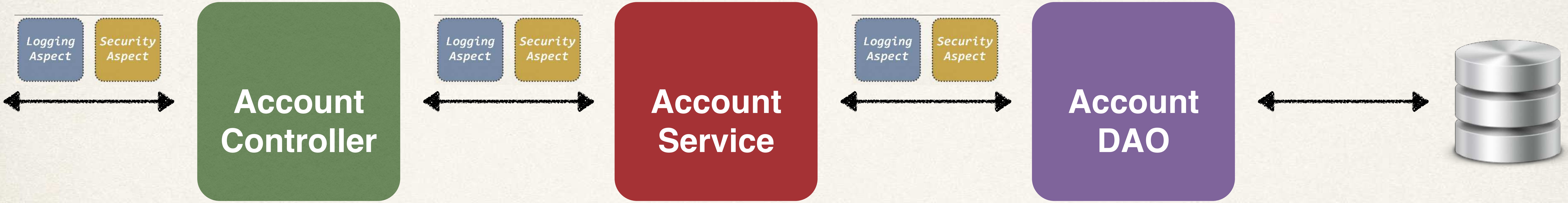
- "Concern" means logic / functionality

Cross-Cutting Concerns



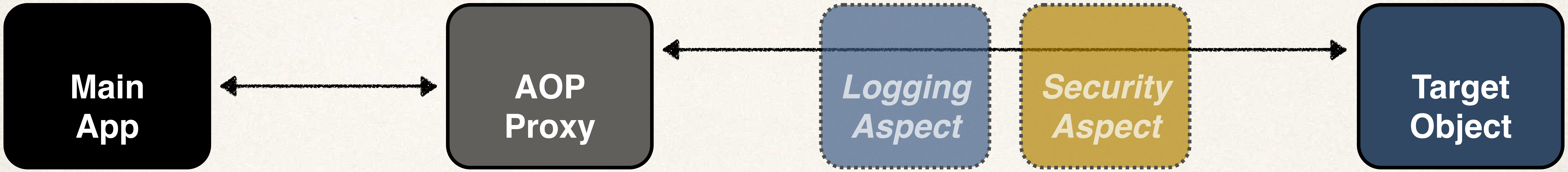
Aspects

- Aspect can be reused at multiple locations
- Same aspect/class … applied based on configuration



AOP Solution

- Apply the Proxy design pattern



Main App

AOP Proxy

Logging Aspect

Security Aspect

Target Object

MainApp

```
// call target object
targetObj.doSomeStuff();
```

TargetObject

```
public void doSomeStuff() {
    ...
}
```

Benefits of AOP

- **Code for Aspect is defined in a single class**
  - Much better than being scattered everywhere
  - Promotes code reuse and easier to change
- **Business code in your application is cleaner**
  - Only applies to business functionality: addAccount
  - Reduces code complexity
- **Configurable**
  - Based on configuration, apply Aspects selectively to different parts of app
  - No need to make changes to main application code … very important!

Additional AOP Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **Exception handling**
  - log exception and notify DevOps team via SMS/email
- **API Management**
  - how many times has a method been called user
  - analytics: what are peak times? what is average load? who is top user?

AOP: Advantages and Disadvantages

Advantages

- Reusable modules
- Resolve code tangling
- Resolve code scatter
- Applied selectively based on configuration

Disadvantages

- Too many aspects and app flow is hard to follow
- Minor performance cost for aspect execution (run-time weaving)

Aspect-Oriented Programming (AOP)
Spring AOP Support

AOP Terminology

- **Aspect**: module of code for a cross-cutting concern (logging, security, …)
- **Advice**: What action is taken and when it should be applied
- **Join Point**: When to apply code during program execution
- **Pointcut**: A predicate expression for where advice should be applied

Advice Types

- **Before advice**: run before the method
- **After finally advice**: run after the method (finally)
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **Around advice**: run before and after method

Weaving

- Connecting aspects to target objects to create an advised object
- Different types of weaving
  - Compile-time, load-time or run-time
- Regarding performance: run-time weaving is the slowest

AOP Frameworks

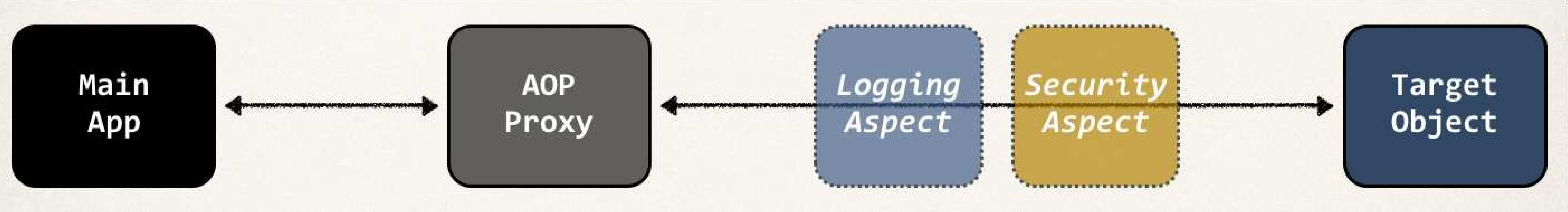
- Two leading AOP Frameworks for Java

Spring AOP

AspectJ

Spring AOP Support

- Spring provides AOP support
- Key component of Spring
  - Security, transactions, caching etc
- Uses run-time weaving of aspects



AspectJ

- Original AOP framework, released in 2001
  - www.eclipse.org/aspectj
- Provides complete support for AOP
- Rich support for
  - join points: method-level, constructor, field
  - code weaving: compile-time, post compile-time and load-time

Spring AOP Comparison

Advantages

- Simpler to use than AspectJ
- Uses Proxy pattern
- Can migrate to AspectJ when using @Aspect annotation

Disadvantages

- Only supports method-level join points
- Can only apply aspects to beans created by Spring app context
- Minor performance cost for aspect execution
  (run-time weaving)

AspectJ Comparison

Advantages

- Support all join points
- Works with any POJO, not just beans from app context
- Faster performance compared to Spring AOP
- Complete AOP support

Disadvantages

- Compile-time weaving requires extra compilation step
- AspectJ pointcut syntax can become complex

Comparing Spring AOP and AspectJ

- Spring AOP only supports
  - Method-level join points
  - Run-time code weaving (slower than AspectJ)

- AspectJ supports
  - join points: method-level, constructor, field
  - weaving: compile-time, post compile-time and load-time

Comparing Spring AOP and AspectJ

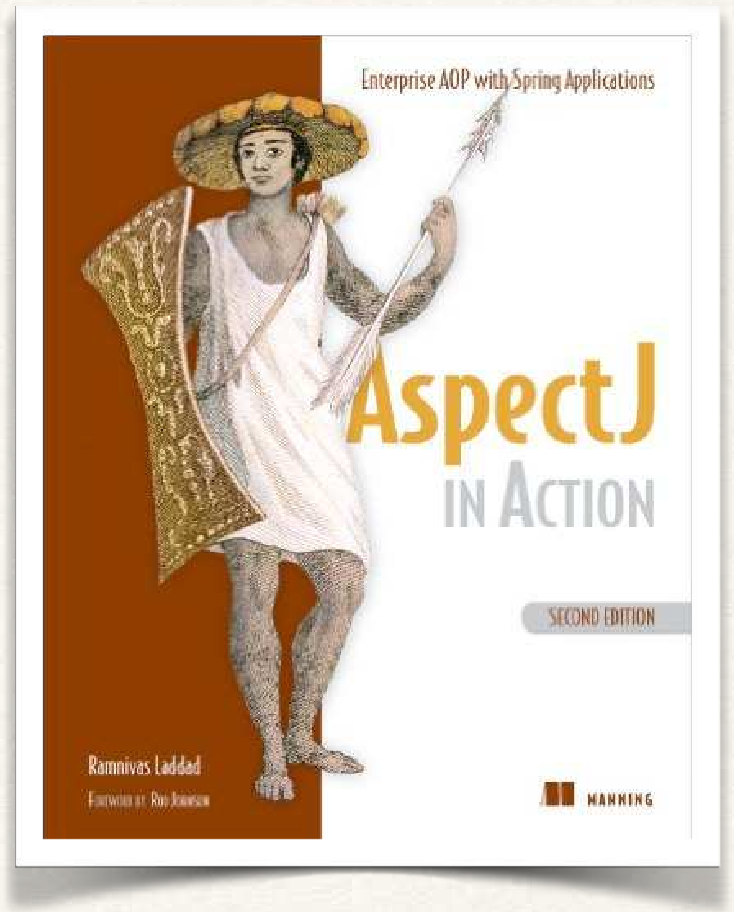
- Spring AOP is a light implementation of AOP
- Solves common problems in enterprise applications

- My recommendation
  - Start with Spring AOP … easy to get started with
  - If you have complex requirements then move to AspectJ

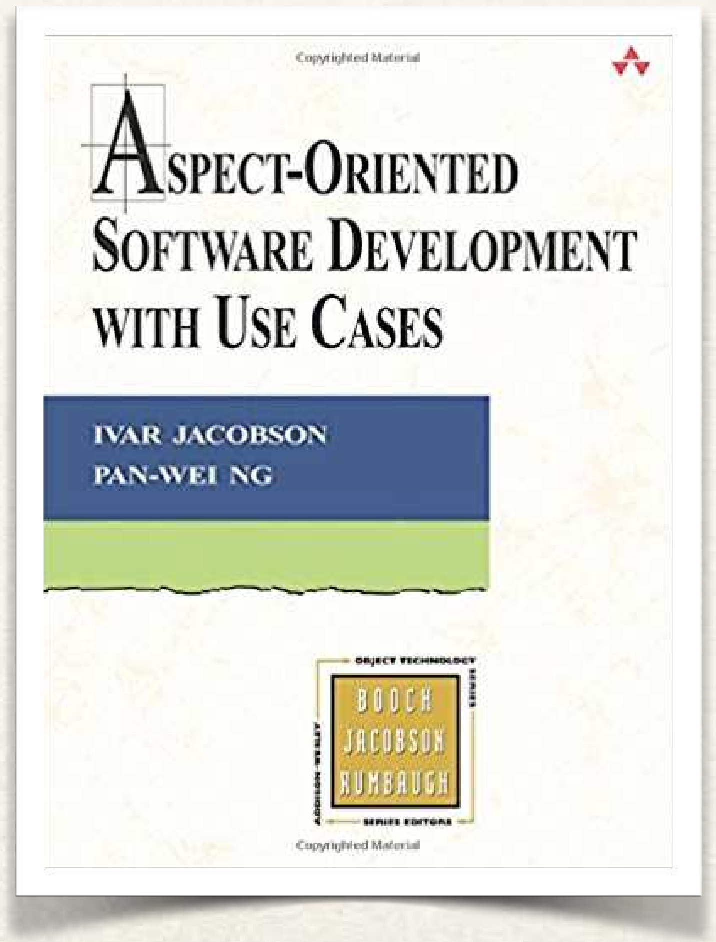
Additional Resources

- Spring Reference Manual: www.spring.io

- *AspectJ in Action*
    - by Raminvas Laddad



- *Aspect-Oriented Development with Use Cases*
    - by Ivar Jacobson and Pan-Wei Ng


Copyrighted Material

ASPECT-ORIENTED SOFTWARE DEVELOPMENT WITH USE CASES

IVAR JACOBSON
PAN-WEI NG

OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS

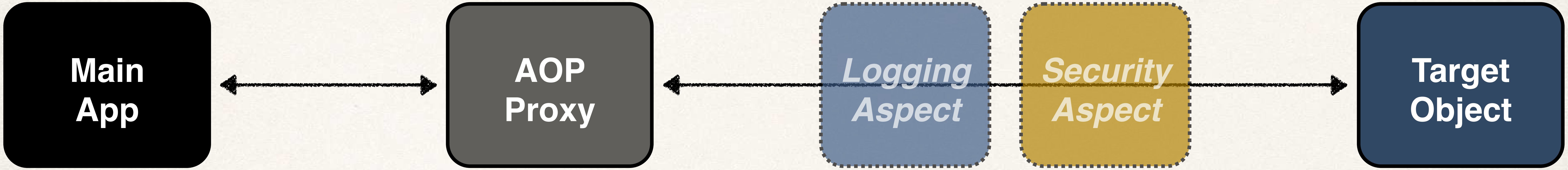
Copyrighted Material

Aspect-Oriented Programming (AOP)
@Before Advice

Advice Types

- **Before advice**: run before the method
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **After finally advice**: run after the method (finally)
- **Around advice**: run before and after method

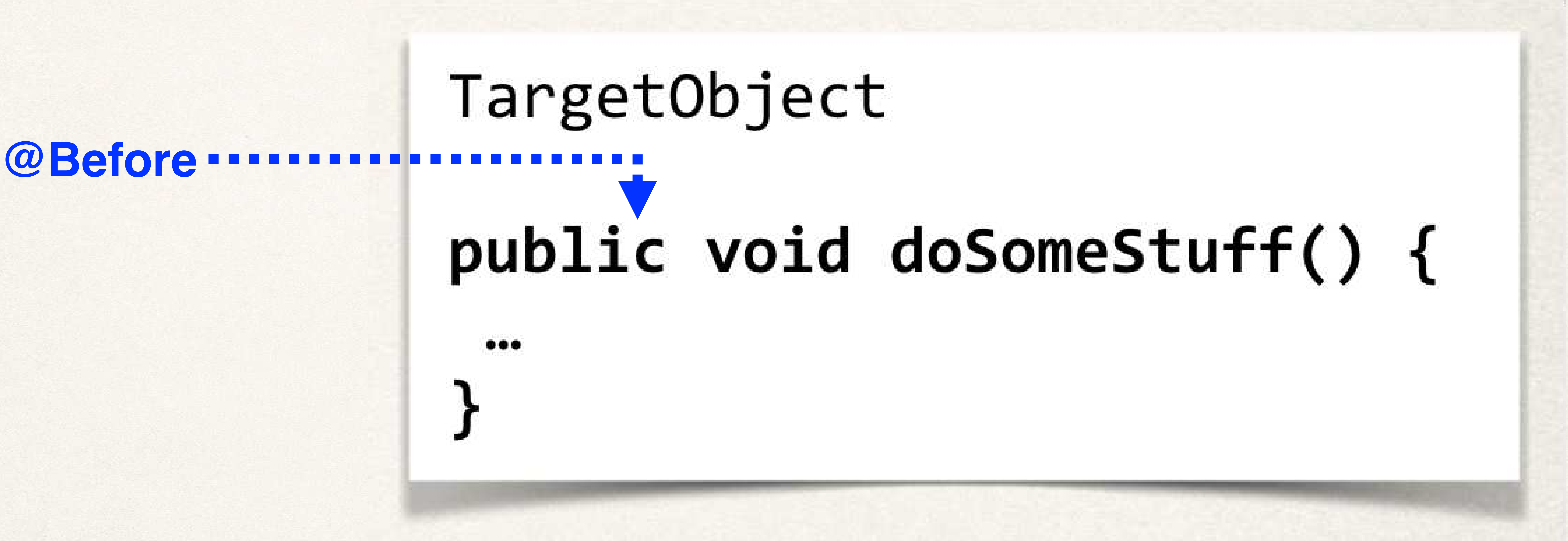
@Before Advice - Interaction



```
MainApp

// call target object
targetObj.doSomeStuff();
```

```
TargetObject

public void doSomeStuff() {
  …
}
```

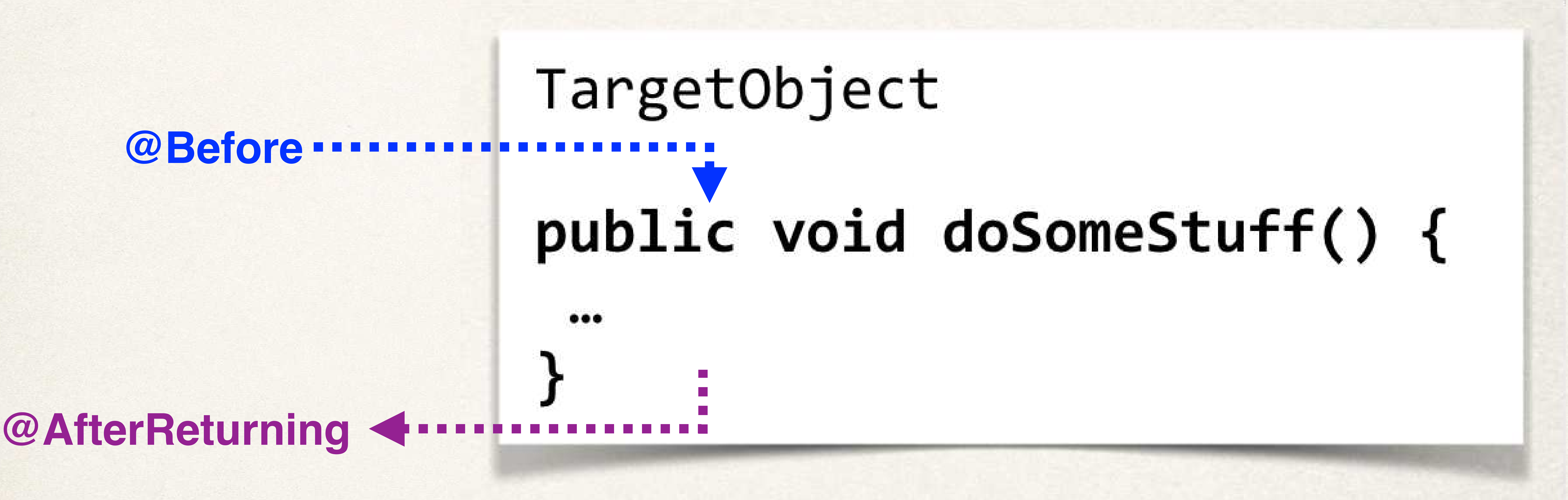
Advice - Interaction



@Before

```
TargetObject

public void doSomeStuff() {
  ...
}
```

Advice - Interaction



TargetObject

@Before

```
public void doSomeStuff() {
  ...
}
```

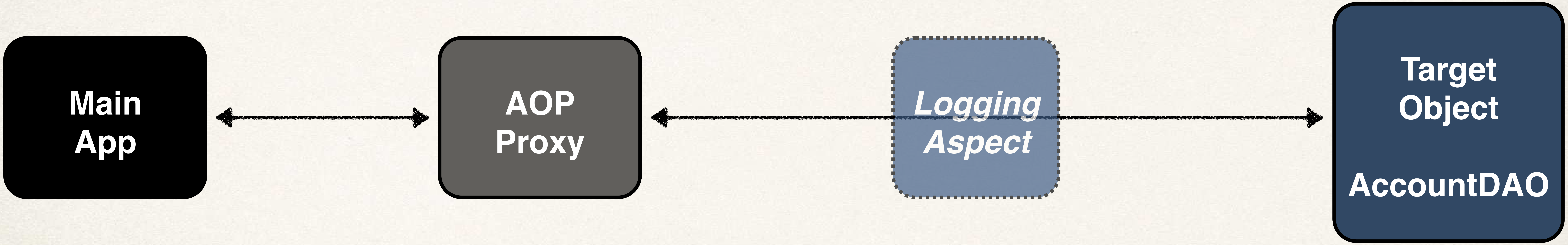
@AfterReturning

@Before Advice - Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **API Management**
  - how many times has a method been called user
  - analytics: what are peak times? what is average load? who is top user?

AOP Example - Overview

Step-By-Step



MainApp

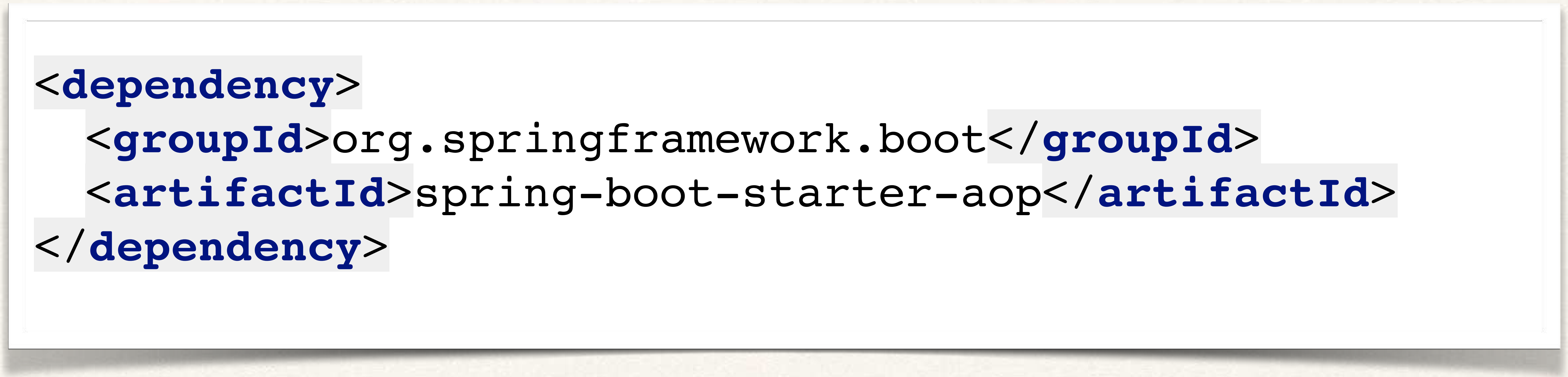
```
// call target object
theAccountDAO.addAccount();
```

TargetObject - AccountDAO

```
public void addAccount() {
  …
}
```

Spring Boot AOP Starter

- Add the dependency for Spring Boot AOP Starter



```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

- Since this dependency is part of our pom.xml
  - Spring Boot will **automatically** enable support for AOP
  - No need to explicitly use @EnableAspectJAutoProxy … we get it for free

Development Process - @Before

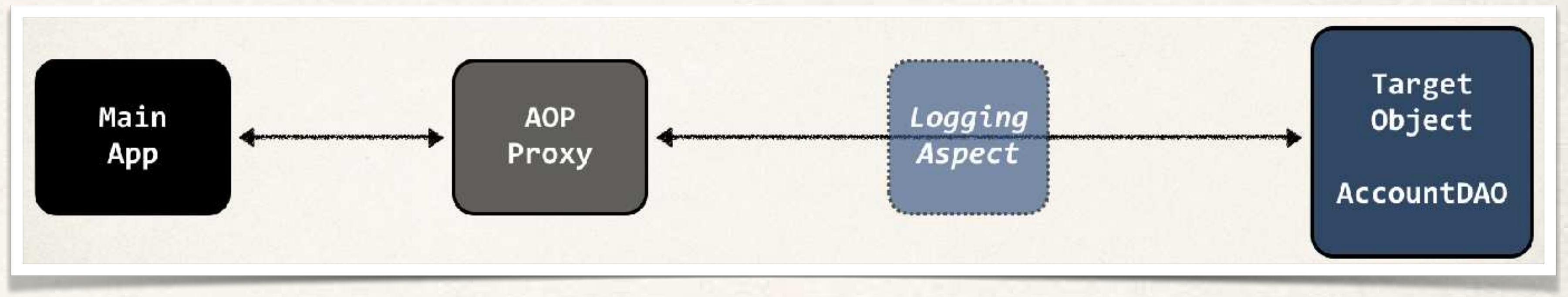
Step-By-Step

1. Create target object: AccountDAO

2. Create main app

3. Create an Aspect with @Before advice

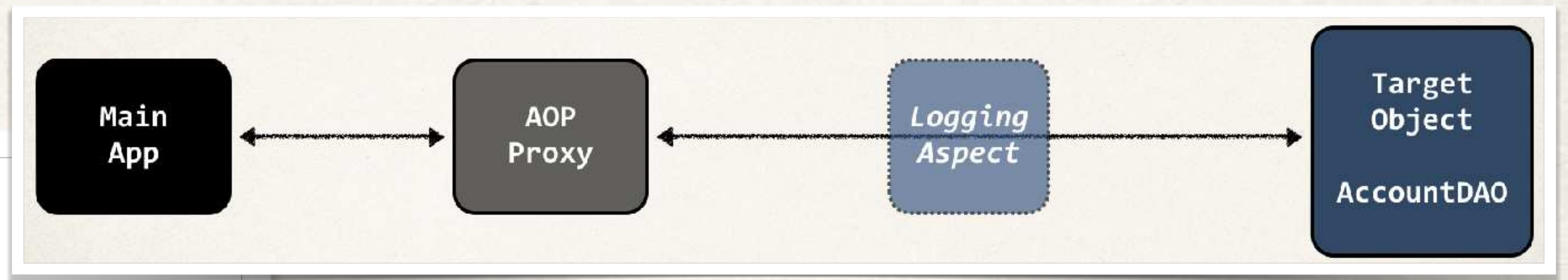
Step 1: Create Target Object: AccountDAO

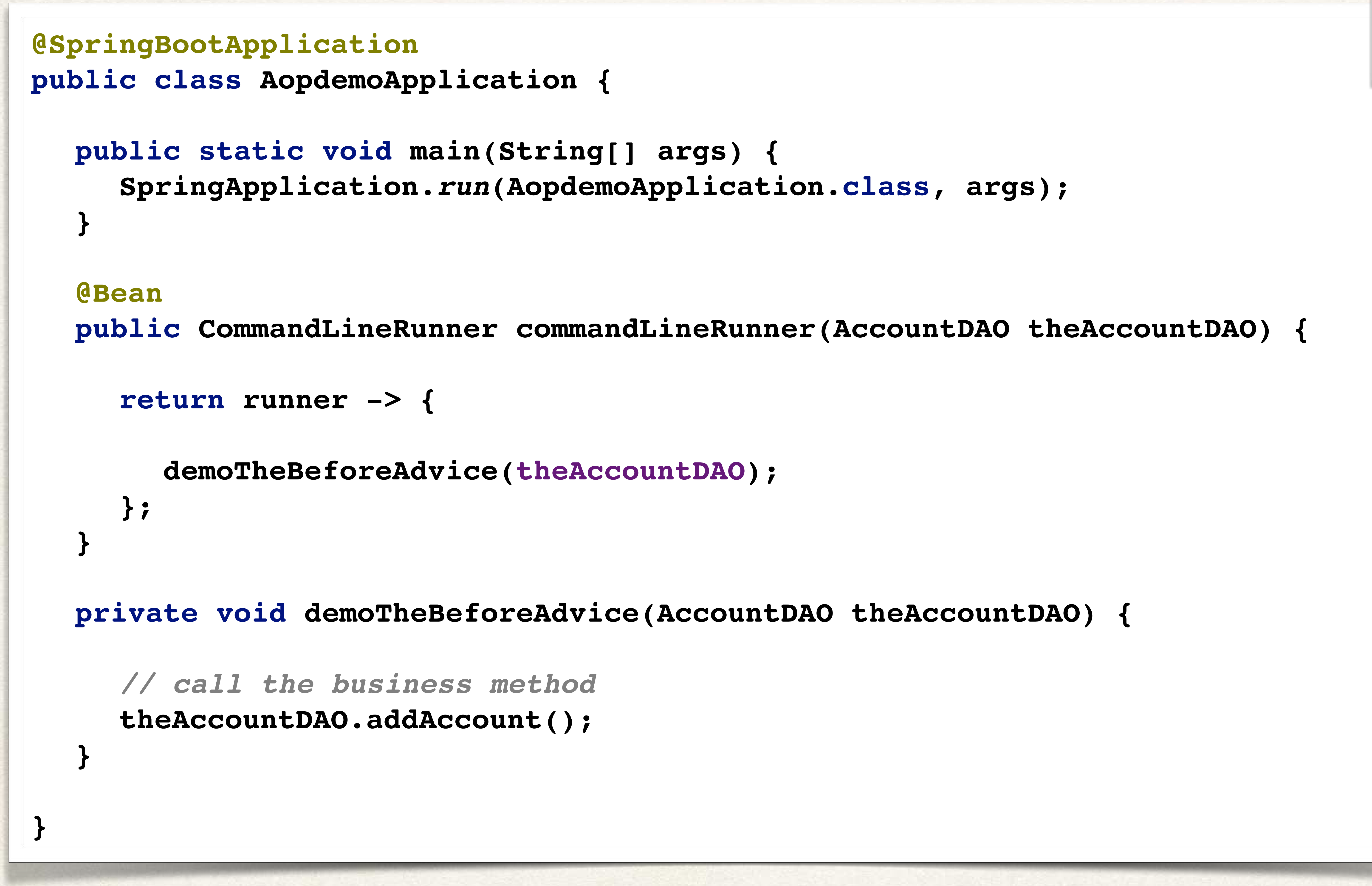


```
public interface AccountDAO {

    void addAccount() {
}
```

```
@Component
public class AccountDAOImpl implements AccountDAO {

    public void addAccount() {

        System.out.println("DOING MY DB WORK: ADDING AN ACCOUNT");

    }
}
```

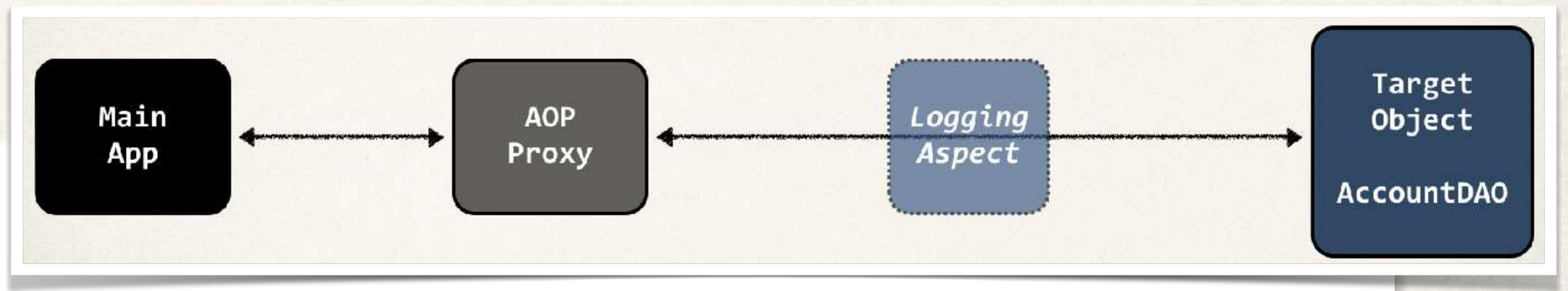
Step 2: Create main app





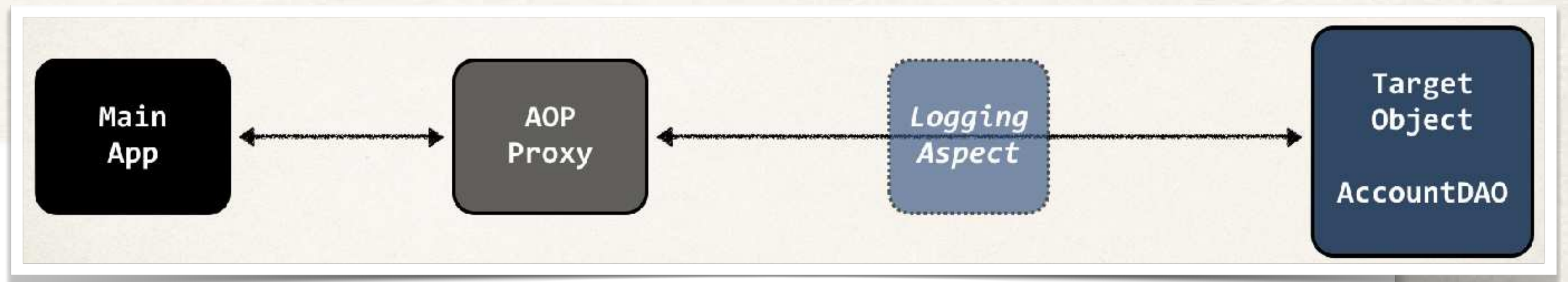
```
@SpringBootApplication
public class AopdemoApplication {

   public static void main(String[] args) {
      SpringApplication.run(AopdemoApplication.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(AccountDAO theAccountDAO) {

      return runner -> {

         demoTheBeforeAdvice(theAccountDAO);
      };
   }

   private void demoTheBeforeAdvice(AccountDAO theAccountDAO) {

      // call the business method
      theAccountDAO.addAccount();
   }

}
```

Step 3: Create an Aspect with @Before advice



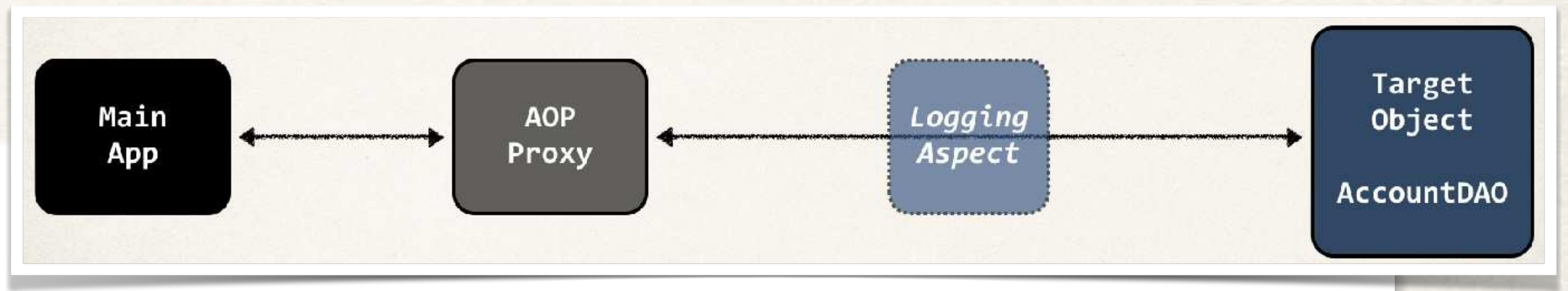
```
@Aspect
@Component
public class MyDemoLoggingAspect {



    ...



}
```

Step 3: Create an Aspect with @Before advice



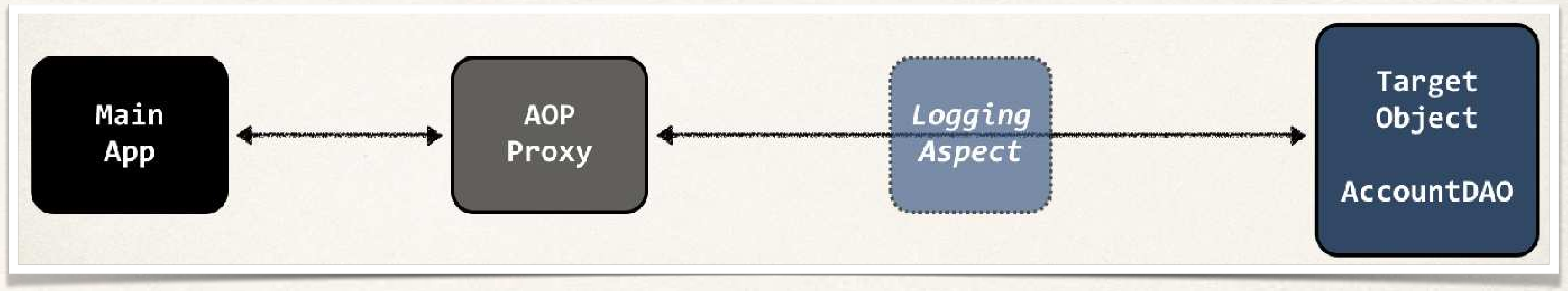
```
@Aspect
@Component
public class MyDemoLoggingAspect {

   @Before("execution(public void addAccount())")
   public void beforeAddAccountAdvice() {

      ...

   }

}
```

Step 3: Create an Aspect with @Before advice



```
@Aspect
@Component
public class MyDemoLoggingAspect {

  @Before("execution(public void addAccount())")
  public void beforeAddAccountAdvice() {

    System.out.println("Executing @Before advice on addAccount()");

  }

}
```

Best Practices: Aspect and Advices



- Keep the code small
- Keep the code fast
- Do not perform any expensive / slow operations
- Get in and out as QUICKLY as possible