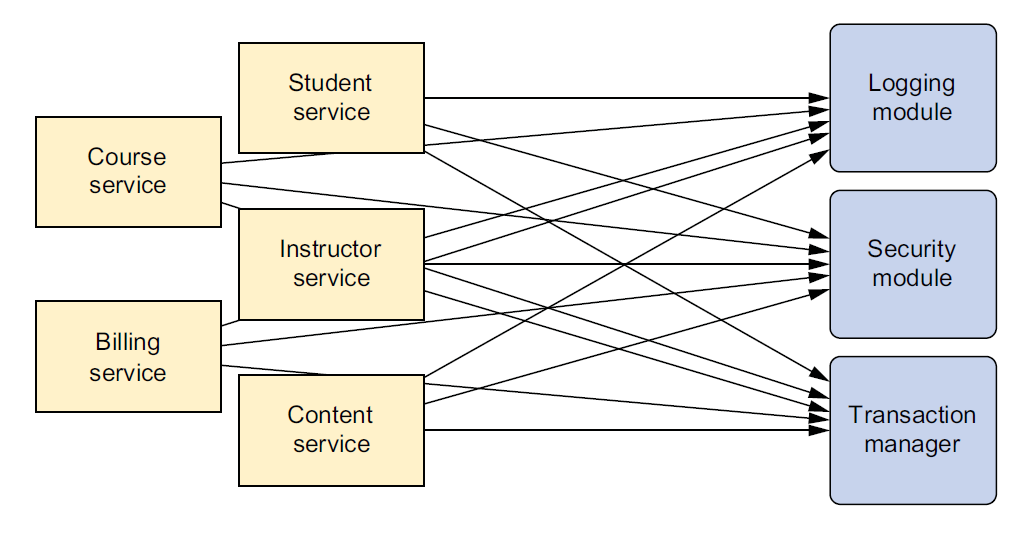
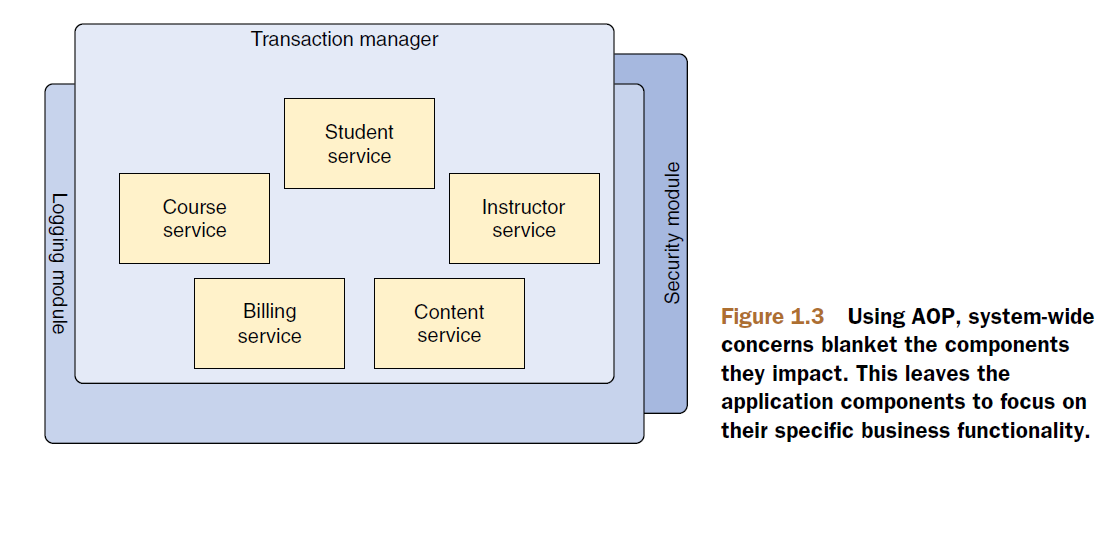
**Applying Aspects**

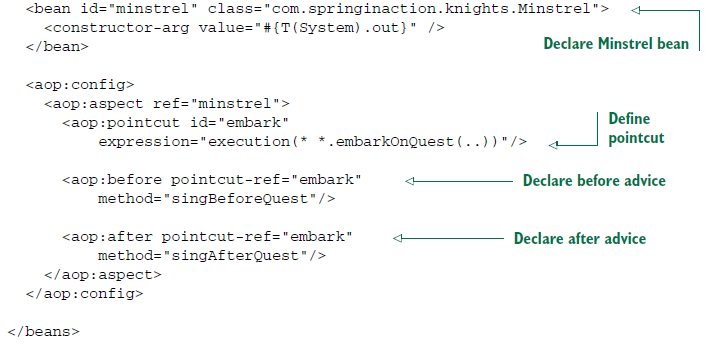
* Although DI makes it possible to tie software components together loosely, **aspect oriented programming (AOP)** enables you to capture functionality that’s used throughout your application in reusable components.
* AOP is often defined as a technique that promotes separation of concerns in a software system.
* Systems are composed of several components, each responsible for a specific piece of functionality.
* System services such as logging, transaction management, and security often find their way into components referred to as *cross-cutting concerns* because they tend to cut across multiple components in a system.
* By Spreading these concerns across multiple components, you introduce two levels of complexity to your code:
  + The code that implements the system-wide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you’ll need to visit multiple components. Even if you’ve abstracted the concern to a separate module so that the impact to your components is a single method call, that method call is duplicated in multiple places.
  + Your components are littered with code that isn’t aligned with their core functionality. A method that adds an entry to an address book should only be concerned with how to add the address and not with whether it’s secure or transactional.
* The below business objects on the left are too intimately involved with the system services on the right. Not only does each object know that it’s being logged, secured, and involved in a transactional context, but each object also is responsible for performing those services for itself.



* Call to system-wide concerns such as logging and security are often scattered about in modules where those tasks are not their primary concern.
* AOP makes it possible to modularize these services and then apply them declaratively to the components they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved. In short, aspects ensure that POJOs remain plain.
* It the below figure, an application consists of modules that implement business functionality. With AOP, you can then cover your core application in a flexible manner without your core application even knowing they exist. This is a powerful concept, because it keeps the security, transaction, and logging concerns from littering the application’s core business logic.







* Here you’re using Spring’s **aop** configuration namespace to declare that the Mistrel bean is an aspect. First you declare *Minstrel* as a bean.
* Then you refer to that bean in the <aop:aspect> element. Defining the aspect further, you declare (using *<aop:before> )* that before the *embarkOnQuest*() method is executed, the *Minstrel’s singBefore-Quest()* should be called. This is called before advice. And you (using *<aop:after>* ) declare that the *singAfterQuest*() method should be called after *embarkQuest()* has executed. This is known as *after advice*.
* In both cases, the *pointcut-ref* attribute refers to a poitcut named *embark.*  This pointcut is defined in the preceding <*pointcut*> element with an *expression* attribute set to select where the advice should be applied. The expression syntax is AspectJ’s pointcut expression language.
* There are two important points to take away from this example:
  + First, *Minstrel* is still a POJO – nothing about it indicates that it’s to be used as an aspect. Instead, *Minstrel* became an aspect when you declared it as such in the Spring context.
  + Second, and most important, Mistrel can be applied to BraveKnight Without BraveKnight needing to explicitly call on it. In fact, *BraveKnight* remains completely unaware of *Minstrel’s* existence.
* I should also point out that although you used some Spring magic to turn *Minstrel* into an aspect, it was declared as a Spring *<bean>* first. The point is that you can do anything with Spring aspects that you can do with other Spring beans, such as inject them with dependencies.
* The Spring’s AOP can be used for even more practical things. As you’ll see later, Spring AOP can be employed to provide services such as declarative transactions and security.