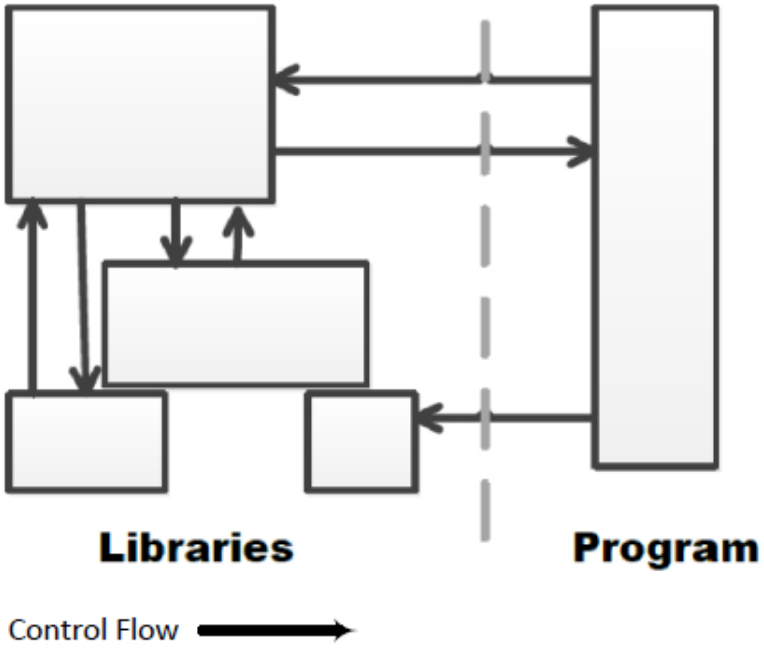

Figure 2.7.: An application making use of many software libraries

2.3. Reusable Software Component Models

In this part of the chapter, two software component models are provided: software library and object oriented framework. First, a *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Bos98]. A library is a component made available to develop software and an object oriented framework is a special case of software libraries, which is reusable abstraction of code wrapped in a well-defined Application Programming Interface (API) [Lic08].

2.3.1. Library

A library is a collection of reusable components, usually classes, used to develop software. The content of the library is independent to the application's context and provides with certain functionality. That is, it provides access to the code that performs a programming task.

The usage of the libraries can increase productivity by reusing code and Know-how and focus in the real problem; improve the quality since the libraries were developed by specialists; decrease maintenance effort because there exist *standard components* used in the different applications. A common usage of libraries is depicted in Figure 2.7.

The control flow relies entirely in the application from the user. In this scenario, the overall flow of control is dictated by the caller (program or user application) of the libraries.

2.3.2. Object Oriented Framework

An object oriented framework provides an abstract design and implementation for a particular domain. Applications are constructed by using the framework as a basis and extending it with specific functionality, proper from the applications. The probably most referenced definition of a framework is found in Johnson et at. [JF98]:

> *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.*

This means, that a framework is a design solution for an application in a given problem domain. This solution is provided in the form of common code and the user code is responsible to selectively override or specialize the framework thus providing the required functionality.

One of the most distinguishing features of a framework is the ability to make extensive use of *dynamic binding*, which is the practice of figuring out which method to invoke at runtime. In libraries or normal user applications, the overall flow control relies on the application code since the latter invokes the routines from the libraries that were made available. For the framework, the situation is inverted and its code has the control and calls the code application when appropriate. This inversion of control is often referred to as the *Hollywood principle*, i.e. *"Don't call us - we will call you"*. In Figure 2.8, this inversion is graphically illustrated.

A framework comprises a set of abstract and concrete classes. The concrete classes are intended to be non visible to the framework user, whereas the abstract classes are intended to be subclassed by the framework user. The abstract classes are referred to hot-spot [Pre97]. The quality of a framework is directly related to the flexibility required in a domain; hot-spots help to reach this quality. **Hot-spots** are points of predefined refinement where framework adaptation takes place. Framework can be categorized into white-box and black-box types [Pre97].

White-box framework. A white-box framework comprises incomplete classes, which means, classes that contain methods without meaningful implementations. The user is supposed to customize the framework behavior through subclassing. At the beginning of its life cycle, a framework is inheritance-based. The reason of this structure is because the application domain is not well understood to make it possible to parameterize the behavior.

Adaptation is accomplished through **inheritance**. Users from white-box framework modify the behavior by applying inheritance to override methods in subclasses of framework classes. In order to override methods, the framework user

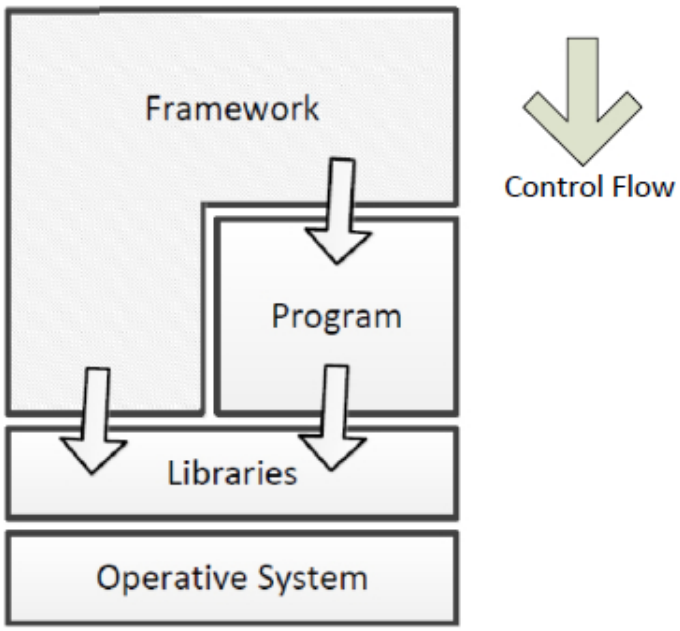

Figure 2.8.: The inversion of control

must understand the design and implementation of the framework, at least to a certain degree of detail. A white-box framework contains incomplete classes.

Figure 2.9 illustrates this property of hot-spots based on inheritance [Pre97]. Class A contains an abstract method (gray highlighted) without a meaningful default implementation. The abstract method forms the hot-spot in this case. Subclass A1 has to override this hot-spot. Class A could also be defined as an interface and the white-box characteristic of a framework does not change using interfaces.

Black-box framework. A black-box framework offers pre-fabricated components that are ready for adaptation and is based on composition. The behavior of this framework is combined by the usage of different combination of classes. Black-box framework requires a deep understanding of the flexible aspects of the domain. All this composition means, that a black-box framework provides with a pre-defined flexibility that is modeled through parameterization. From the previous premise follows that this type of framework is more rigid in the domain it supports.

Adaptation is accomplished through **composition**. Black-box framework offers pre-fabricated components ready for adaptation where modifications are performed by composition. Hot-spots also correspond to the overridden methods. The user of the framework doing the adaptation deals with the component as a whole.

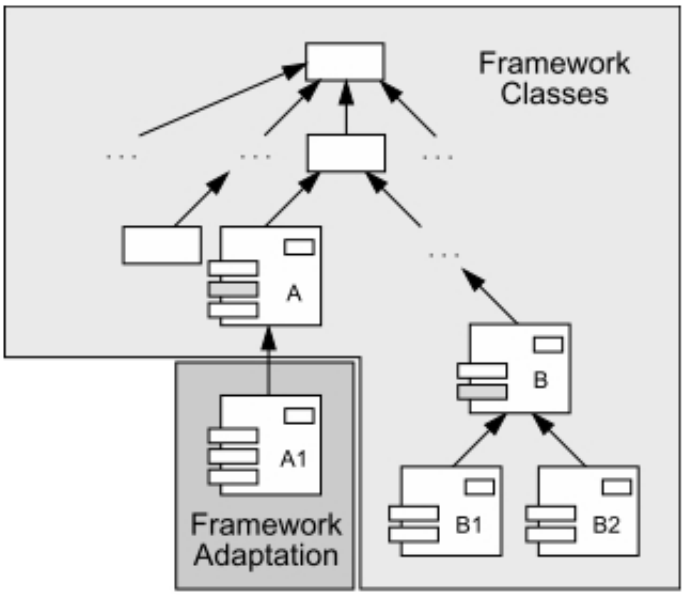

Figure 2.9.: Hot-spots based on inheritance (Class A) and composition (Class B)

This property of hot-spots based on composition is depicted in Figure 2.9. Class B contains subclasses B1 and B2. Both subclasses provide with default implementations of the abstract methods from class B.

The difference between hot-spots based on inheritance (white-box framework) and hot-spots based on composition (black-box framework) can be shown in Figure 2.9. In the case of Class A, the framework user has to subclass A first; in case of class B, the framework provides ready-to-use subclasses (B1 and B2).

Nevertheless, in practice, an object oriented framework hardly ever is only black-box or white-box. A framework has parts that can be parameterized and parts that need to be customized through subclassing [Pre97].

To conclude this section, a framework reduces the amount of development effort required for software development and it can be used as component in Software Product Lines (see Section 2.4). Commonalities are surrounded by the framework code, whereas variabilities must be provided by the user code. Thus, object-oriented framework proves to be an accurate model for reusable components in product line architecture. The framework manages variability by encapsulating commonalities and handling the differences, which are integrated by either inheritance or composition.