Inversion-of-Control Layer

Stefan Sobernig
Institute for Information Systems and New Media
Vienna University of Economics and Business
Vienna, Austria
stefan.sobernig@wu.ac.at

Uwe Zdun
Information Systems Institute
Vienna University of Technology
Vienna, Austria
zdun@infosys.tuwien.ac.at

Inverting the control is a common design practise that has been used in various application areas. It gained popularity in the context of object-oriented application frameworks and designs based on abstract classes or interfaces. Recently, dependency injection techniques, especially in the context of lightweight containers such as Spring, have raised the attention for inversion of control again. However, inversion of control has not yet been described in its architectural dimension with a focus on layering architectures, and the pros and cons of the design decision for control inversion. In this paper, we present the INVERSION-OF-CONTROL LAYER pattern that describes the design practise from an architectural point of view, rather than focusing on particular implementation techniques.

1 Introduction

An important and widely established architectural design practise for achieving reuse in complex software systems is to capture recurring behaviour (e.g., an activity flow in terms of instruction calls) in special-purpose software components. By integrating with these components and configuring them from the using components, it is possible to apply the components' predefined behaviour across various applications. In the past, this architectural design practise has been labelled in a number of ways. Most commonly, you find the notion of *inversion of control* [JF88, Fow04, Fow05]. This name has emerged out of the context of object-oriented (OO) application frameworks. Johnson and Foote [JF88], for example, discussed the inversion of control in terms of designs based on *abstract classes*. The use of the term *inversion* implies a distinction between the application framework, the framework-integrating application, and the typical direction of control between these two, i.e., the application controlling the framework. Inversion of control means in this context that this direction of control is inverted by some means, allowing the framework to take control over the behaviour described by an application, at least partially and in a well-defined manner. From a structural point of view, this is reflected in an inversion of responsibilities for managing call and use dependencies, also referred to as *dependency inversion* [Mar96].

From a slightly different point of view, an inverted direction of control was referred to as *Hollywood's Law* or later the *Hollywood Principle* [Swe85, Fow05]: "Don't call us, we'll call you." Sweet [Swe85] introduces the reader to the design of the Xerox Development Environment (XDE), an integrated development environment for the Mesa language and runtime platform. As an essential infrastructure element, XDE allowed extension developers to add their own development tools, and these add-ons were able to exchange notifications. This inter-tool notification scheme was realised by allowing tool developers to register callback procedures with the overall XDE windowing and execution system Tajo (e.g., sleep and wake-up procedures). Hence the saying: "Don't call us, we'll call you" [Swe85, p. 218]. Against a similar background, the idea made it into the GoF book [GHJV94], being heavily inspired by experiences on designing GUI frameworks itself.[1] From there, Hollywood's Law

[1] These experiences refer to the GUI framework and toolkit ET++ [WGM89].

got linked to certain GoF design patterns, in particular the GoF TEMPLATE METHOD pattern [Vli96] and generalisations of it (see e.g. [Pre96]).

However, despite a lacking of documented references and varying connotations, this scheme of behavioural reuse can be traced back to established practises in non-OO programming, in particular certain uses of *procedure call abstractions*. At the origins of OO programming, Simula-67 [BDMN75] introduced the strategy of *inner* method combinations along with the idea of class prefixes. Simplistically speaking, methods owned by superclasses were allowed to call upon their shadowing subclass methods. This style of method combination was continued by the Beta tradition of OO programming (i.e., Beta's inner mechanism [MMPN93]) to enable a particular specialisation scheme: The superclass method (i.e., the general action in Beta) defines sub-tasks (i.e., part-actions), ordered into a sequence by the superclass method. The inner mechanism allows for a stepwise specialisation of this general action by refining the part-actions along the inheritance path.

In recent years, so-called lightweight component containers such as Spring have raised a new attention for inversion of control as a design practise. Instead of being based on abstract classes or interfaces, the inversion of control is realised in these containers using a component wiring technique called *dependency injection*. The basic idea of dependency injection is to provide a special object for wiring the components by populating a field in a class with an appropriate implementation for the interface of the field [Fow04]. Hence, the concrete configuration of the field is concretised by the dependency injection component.

From complementary angles, some continued threads in the software pattern community document and discuss established architectural design practises for inverting responsibilities when managing variation points in layered systems (see for an overview [Hen05, Hen07b, Hen07c, Hen07a]). Primarily concerned with component configuration by parametrisation, these pattern works capture recurring architectural design challenges which call for parametrising a layered system from the top-level layer in a disciplined manner. These challenges reflect experiences in several distinct fields, i.e., software testing, plug-in and container infrastructures, parameter passing strategies, as well as programming language design (e.g., interpreters). This particular notion of inversion is also found in the context of variability management and variability implementation strategies for software product lines and product line architectures (e.g., component configuration interfaces [Bos00]).

This short and non-exhaustive overview of the history of inversion of control, limited to the context of OO languages, OO frameworks and lightweight containers, shows that this design practise reappears across different families of software systems, applied to software solutions in various technical domains (e.g., GUI application frameworks, remoting frameworks, event-driven I/O, OO programming language design, component-oriented programming, and so on), implemented by a variety design and implementation techniques. It has the potential to affect the architectural qualities of a software system and must be considered in the context of related architectural design decisions. Yet, current work on architectural patterns [BMR$^+$00, AZ05] does not document this architectural design practise in its own right. Capturing it as an architectural pattern description, however, puts us into the position to evaluate its interaction along with related architectural patterns, most notably the LAYERS, EXPLICIT INTERFACE, and PARAMETRIZE FROM ABOVE patterns.

Furthermore, an architectural pattern description of this design practise contributes to ...

- ... avoiding the frequent confusion of the inversion of control (or Hollywood's Law) with a set of implementation techniques, emerging out of interface-oriented programming (e.g., dependency injection) or class hierarchy usage (e.g., abstract class hierarchies);
- ... capturing pros and cons of applying the inversion of control in the context of complex architectural design decisions, with resulting evaluations being applicable in a wider range of applications and application domains.

- ...understanding the component interactions in a design based on the inversion of control. The notions of inverse control and Hollywood's Law imply a particular architectural view on a software system, namely the component interaction view. Current descriptions of this design practise do not reflect on the structuring capacity of this architectural design practise when compared e.g. to the LAYERS pattern.
- ...clarifying what kind of controls (note the plural!) are eventually inverted. Also, a convenience view only treats one direction of control inversion between components (e.g., inverting the control between a abstract base class and one of its concrete subclasses). In an architectural perspective, we will learn that often multiple and mutual control inversions between components are involved.

In short, this is what motivates us to propose the INVERSION-OF-CONTROL LAYER as an predominantly structural architectural pattern which integrates with existing architectural pattern languages, in particular [AZ05].

This paper leans itself primarily towards advanced students of and decision makers in software architecting. When studied and applied in the context of architectural pattern languages such as [AZ05], this pattern description helps creating, refactoring, and evaluating software architectures according to a well-established architectural design practise, i.e., the inversion of control. Also, by describing a variety of concrete design and implementation techniques (e.g., abstract classes, container frameworks, and dependency injection) as the variations of a more general principle, evaluating and communicating the consequences from applying concrete techniques in this family is facilitated in a work-sharing setting (e.g., between development managers, architects, and developers; or, between feature teams). We assume readers have or obtain some background on architectural styles and patterns, such as layering strategies, the notion of interface abstraction, and forms of component interactions (e.g., explicit vs. implicit invocations).

The remainder of the paper is structured as follows: In the next section we introduce the reader to the architectural pattern language presented in [AZ05]. We do so by giving an thumbnail overview of related architectural patterns and by summarising the INVERSION-OF-CONTROL LAYER in their context (see Section 2). Also, a motivating example will be presented (see Section 3). Section 4 is dedicated to the actual architectural pattern description. By applying selected techniques which use and so realise an INVERSION-OF-CONTROL LAYER, we then resolve the motivating example in Section 5. In Section 6, we elaborate on known uses of the INVERSION-OF-CONTROL LAYER pattern, taken from various application domains. Finally, in Section 7, we offer some concluding remarks on limitations of our current work and ideas for future refinements.

2 The Pattern in Context

One of the most important ways to structure your software architecture is to introduce layering. The LAYERS architectural pattern [BMR+00, AZ05] describes a decomposition of the underlying system with respect to vertical and horizontal responsibilities. Components residing at the same level of granularity are grouped into LAYERS, and each layer provides an interface to the next higher-level layer. Only invocations from one layer n into its next lower-level layer $n-1$ are possible, and usually bypassing layers is forbidden.

The interfaces between layers are usually realised as EXPLICIT INTERFACES [BH03, BHS07]. The EXPLICIT INTERFACE pattern deals with the problem how to avoid caller dependencies on the implementation details of a called component. EXPLICIT INTERFACES advise you to separate between the interface (specification) and the implementation of a called component.

The LAYERS pattern is concerned with a hierarchical decomposition of a system into strata of functional responsibilities and a particular structuring of use dependencies. These dependencies take a

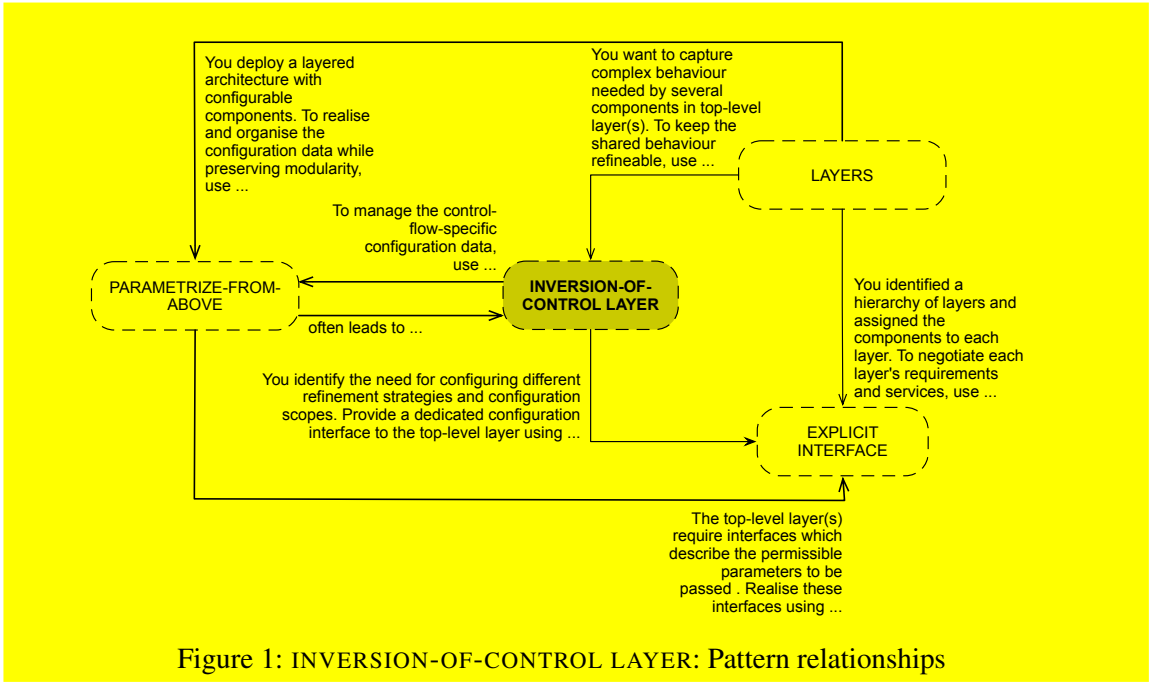

Figure 1: INVERSION-OF-CONTROL LAYER: Pattern relationships

strict top-down direction and, in a pure form, they are preserved by disallowing by-passing of intermediate LAYERS. Each single layer groups components and their connectors that reside at a comparable level of granularity. EXPLICIT INTERFACES negotiate between LAYERS. A lower-level layer offers an EXPLICIT INTERFACE and is responsible for realising it by assigning implementing components for various binding sites. Flipping sides, the higher-level layer requires behaviour as stipulated by the EXPLICIT INTERFACE (see Figure 2a). This dependency structure reflects an intended flow of control between components. Instructions, advertised and shielded by EXPLICIT INTERFACES, are called from the components of the higher-level LAYERS, going down the hierarchy.

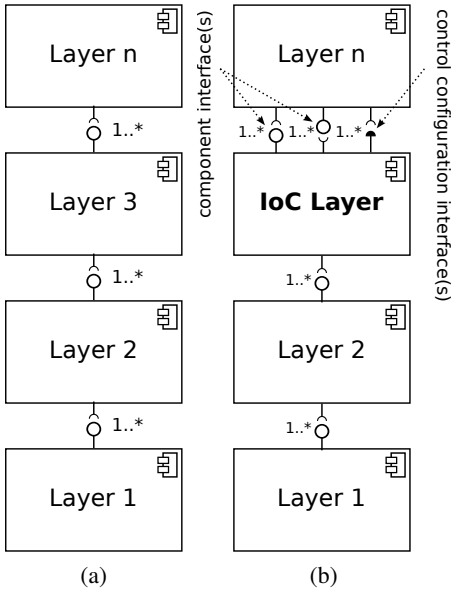

Figure 2: Inversion of control in a layered architecture

Capturing inter-layer dependencies by means of EXPLICIT INTERFACES allows for exchanging components at either end of the layer dependency relation. Dependent (i.e., higher-level) LAYERS may in principle build upon any component located at the independent (i.e., lower-level) layer as long as it provides the required EXPLICIT INTERFACE. The independent layer may be deployed in various configurations as long as the interface compliance holds.

To increase the range of application and the potential reuse, the component implementations commonly contain variation points, i.e., predefined points in the control and data flow which allow for modifying and extending a component's behaviour by, e.g., supplying configuration values. Many strategies for implementing these variation points (e.g., global constants or SINGLETON objects) cause implicit use dependencies from within the component implementations to the external storage of configuration values. Also, if configuration artifacts are located at lower-level layers, activating or deactivating an actual variation remains opaque to client components at higher-level layers. Even worse, a layered system is not necessarily developed or assembled layer by layer, let alone strictly bottom up contrary to the service dependency direction or in a centralised manner (i.e., by multiple developer teams). Hence, the variation data become distributed over the entire layered system due to an evolutionary (further-) development. To avoid or reduce the resulting coupling, variants of the PARAMETRIZE-FROM-ABOVE pattern [Hen05] can be applied. In a refactoring step, the various configuration data sources are so moved to the top-most layer. As a result, variation decisions are to be taken at a single architectural location only. From there, configuration values are then passed to the lower-layer components as variation parameters along the chain of service calls. This requires adapting the EXPLICIT INTERFACES between the LAYERS to stipulate the types of variation parameters (see also Figure 1).

In its essence, the INVERSION-OF-CONTROL LAYER pattern, introduced in the next section, captures a distinct and reusable piece of control flow and data flow in an intermediate layer that is expressed in terms of components and their interactions located at the next higher-level layer. Figures 2a and 2b contrast a pure LAYERS architecture and an INVERSION-OF-CONTROL LAYER architecture. Each layer is simplistically represented by a distinct component. Hence, layer interfaces map directly to component interfaces.

The intermediate (or, inversion-of-control) layer describes an abstracted, general design between interfaces, e.g., using abstract classes, dependency injection, or similar interface-oriented programming techniques. By implementing the interfaces in the higher-level layer, the abstracted behaviour becomes concrete while *reusing* the general design that describes the recurring control and data flow. Note that the data flow is also specified in terms of interfaces alone, denoting input requirements of and output expectations on single process steps. In realising this form of behavioural reuse, the intermediate layer exposes two kinds of EXPLICIT INTERFACES to its antecedent or the *integrator* layer (see Figure 2b). First, *interfaces for control configuration* are provided to the higher-level layer. The inversion-of-control (IoC) layer offers a configuration facility to adjust the piece of reusable control and data flow and to manage its execution. Second, the IoC layer foresees *component variation interfaces*. In doing so, the flow definition is kept refinable according to a variation protocol. This protocol is represented by a set of required variation interfaces, which are to be provided by the antecedent layer.

To sum up, the inverse control architecture is characterised by top-down required (i.e., bottom-up provided) component interfaces. An INVERSION-OF-CONTROL LAYER architecture, on the contrary, is determined by the presence of inverted or top-down provided (i.e., bottom-up required) component interfaces. In addition, at least one control configuration interface between the integrator and the inversion-of-control (IoC) layer must be present. Hence, applying the INVERSION-OF-CONTROL LAYER pattern implies particular uses of the EXPLICIT INTERFACE and the PARAMETRIZE FROM ABOVE patterns (see also Figure 1). The so-related patterns are sketched in Table 1 for later reference.

Pattern	Problem	Solution
LAYERS [BMR+00, AZ05]	You architect and design a system in which high-level components depend on low-level components to perform their functionality. You must obtain some decoupling between high- and low-level components to keep the overall component configuration modifiable, portable, and reusable. At the same time, high- and low-level components interact with other components at the same level of abstraction to realise a complex behaviour.	To balance between the simultaneous needs for vertical decoupling and for horizontal grouping while preserving the goals of modifiability and component reuse, you structure your system into LAYERS. Each layer provides a set of services to the layer above and consumes services from the layer below. Make sure that layers are not bypassed. Layer services are negotiated by providing and requiring interfaces between the layers.
EXPLICIT INTERFACE [BH03, BHS07]	You have identified and designed a piece of self-contained unit of functionality. You want to provide an implementation of this functionality as a freestanding component, along with a published usage protocol to be used by client components. Allowing direct and full access to your component implementation, however, makes the client components dependent on subtle implementation details and component-internal side effects.	To avoid this excessive coupling to your component's internals, provide a distinct interface effectively shielding client components from your component's implementation. By exporting this interface structure to client components and have them keep references to the interface entity rather than the component implementation itself, the coupling is limited to the interface which may evolve independently from any component implementation.
PARAMETRIZE FROM ABOVE [Hen05, Hen07b, Hen07c, Hen07a]	To apply configuration data at selected points in a control flow unfolding in a layered system, the configuration values must be accessible from within the configurable components. They are represented as *absolutely globally* accessible data structures (e.g., through global variables or constants, freestanding global functions, or SINGLETONS objects). As a consequence, you hardwire use dependencies into these components in a hidden manner (e.g., not elicited in a component's signature interface). Also, you risk scattering your configuration data across different code artifacts residing at different layers.	Organize your configuration data at the top layer of your system only. To provide it to the configurable components at the lower layers, pass it down to these layers and their components as parameters, negotiated through their parametrization interfaces (e.g., initialisation and constructor interfaces, operation and instruction interfaces). To turn configuration data of varying complexity into parameters, apply documented strategies of parameter passing (e.g., context objects).

Table 1: Thumbnail sketches of relevant architectural patterns

3 A Motivating Example

In this section, we illustrate the problem of the INVERSION-OF-CONTROL LAYER pattern using a concrete example from the Enterprise Resource Planning (ERP) domain.[2] Consider that you plan to create a small, object-oriented application framework for order management using Java. Order management itself includes a number of sub-tasks, such as placing, deleting, and tracking orders. Based on your development kit and its application frameworks, you set out to create three applications for Customer A and Customer B: Customer A commissioned two applications, one for integrating order management with a web shop environment (`OnlineOrderingA`), the other for linking up its call centre to the order management (`CallOrderingA`). Customer B also operates a web shop (`OnlineOrderingB`), but no call centre.

From a business process perspective, you find that the three target applications require fairly similar processing steps for incoming orders. Yet, the processing steps are not identical. Order processing involves a flow of several actions (see Figure 3). In a first step, the total order price is calculated based on the ordered items and the individual item prices. Then, a customer-specific spending limit is verified based on the computed order sum total. If this spending limit is violated by the order request under review, the violation is signalled, triggering a rollback and a cleanup of the order request handled up to this point. If the spending limit constraint is satisfied, the applications must check whether a non-cumulative quantity discount is to be granted to the customer. If the order request under review reaches the quantity threshold, the price reductions must become effective. Having completed these two preparatory tasks, the order request is actually placed and turned into a stored order. Based on this order and the calculated order price, an invoice is generated as the final action.

While this process definition is shared by the three target applications, there are also points of substantial variation. In this example, each application has to integrate with existing, but different storage components (i.e., database systems) to maintain orders and application states. Besides interacting with different, legacy storage backends, the data management involves incompatible data schemes and vendor-specific optimisations (e.g., strategies of caching and query optimisation).

To summarise the commonalities and variations in your framework, let us consider the various process assets:

- *Control and data flow*: The overall flow of control and data items between the order request actions (i.e., `Calculate sum total`, `Store order`, etc.) is a shared asset between the three target applications. As for the control flow, also major decision points are found in each of the three applications, such as the constraints regarding spending limits and price discounts. The input and output requirements expressed over the individual actions are identical for the three applications (e.g., the `Customer` input and `Limit` output of the `Retrieve spending limit` action).

- *Actions*: The order processing activity in Figure 3 describes a set of eight actions which are executed in three alternative configurations, depending on the decisions taken at the two control nodes (i.e., spending limit and discount threshold). However, each action remains widely *opaque* with respect to its internal behaviour. Each action might behave differently for each of the three applications due to the customer-specific database systems. In particular, the storage-critical actions are `Retrieve spending limit`, `Retrieve discount threshold`, `Store order`, and `Rollback order placement`. In contrast, the remaining actions appear independent of the storage system used, that is, `Calculate sum total`, `Apply discount`, and `Generate invoice`. Hence, these three actions (and their sealed behaviour) also represent shared assets between the three target applications.

[2] The following code examples are remotely inspired by the ones in [Joh02, Chapter 4].

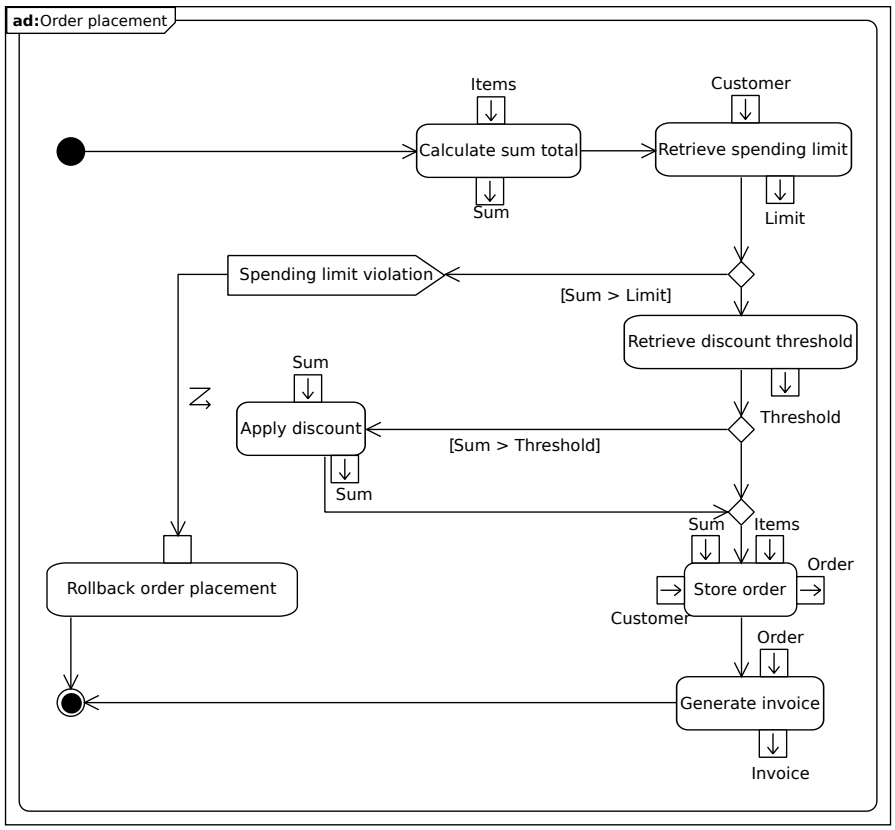

Figure 3: The order placement process

- *Input and output assets*: As already stated, the structure of input and output dependencies applies to all three target applications. As part of a shared domain model of order request handling and order placement, the object definitions such as order items, customer, and invoice certainly form shared assets (e.g., the signature interfaces). In particular, the input and output objects to storage-critical actions potentially carry storage-specific behaviour. For instance, customer objects might be responsible directly for retrieving the spending limit from the customer-specific database.

Let us now consider the structure of the application framework. As usual in such frameworks, you apply a structuring based on the LAYERS architectural pattern. An exemplary result is shown in Figure 4. The top-most layer is formed by the framework-integrating applications, i.e., `OnlineOrdering-A`, `OnlineOrderingB`, and `CallOrderingA`. In the following, we refer to it as the application layer. The application layer requires an interface for `OrderPlacement` to be provided by the first framework layer. The `OrderPlacement` interface contains instructions which realise some shared process assets, such as the `Apply discount` action. The framework layer contains the `GeneralOrdering` component which provides interface and implementation components realising the `OrderPlacement` interface.

Let us sketch out *one* possible Java implementation, outlined as a class diagram in Figure 5. At the first framework layer, the `GeneralOrdering` component is represented by an abstract Java class `erp.ordering.OrderRequest`. This abstract class specifies and implements the shared behaviour discussed above, i.e., calculating the total order price (`calculatePrice()`), applying a possible discount (`applyDiscount()`), and generating an invoice document (`create()`). Declaring `erp.ordering.OrderRequest` as an intermediate abstract class requires the application developer to devise and use concrete subclasses thereof in the applications. These subclasses allow developers to implement the variation points by providing the storage-specific behaviour. The order management sub-framework offers an Oracle-specific order handling component, to be accessed through the `erp.ordering.OracleOrderRequest` subclass. In doing so, your framework of-

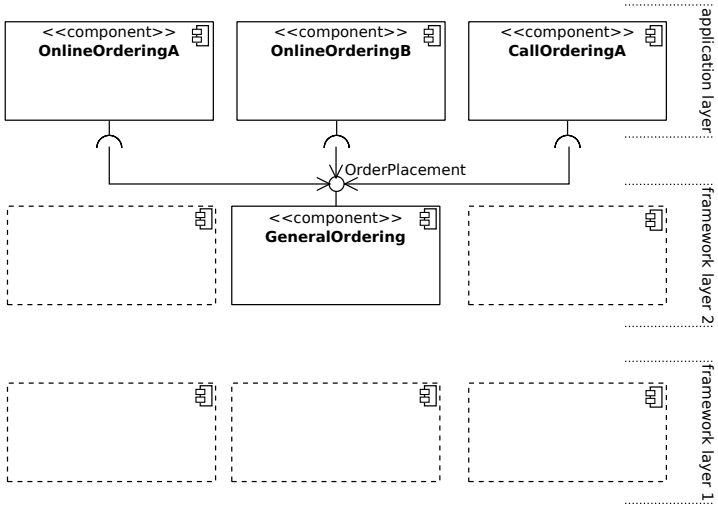

Figure 4: Layering of an order management framework

fers Oracle connectivity for integrating applications to retrieve the customer-specific spending limit and the configurable discount threshold from, as well as to store a processed order request with an Oracle backend.

Now, we turn to the application layer and the three application components to be developed, i.e., `OnlineOrderingA`, `OnlineOrderingB`, and `CallOrderingA`. The web shop system of Customer B operates against an Oracle backend. Therefore, you equip the `OnlineOrderingB` component with the `org.B.WebOrderRequest` class. Here, you take the decision to reuse the existing `erp.ordering.OracleOrderRequest` facility from within the `process()` method (see Figure 5). The `process()` method defines the overall order processing behaviour as captured by the UML2 activity model in Figure 3. To do so, it reuses the shared and storage-specific behavioural elements offered by instances of the `erp.ordering.OracleOrderRequest` class. By creating an instance of `erp.ordering.OracleOrderRequest` and sending messages to it, the `process()` method establishes a use (or call) dependency between the `OnlineOrderingB` component and the components provided by the framework layer. This is illustrated in Figure 5 by the dependency relationship arrow between `erp.ordering.OracleOrderRequest` as the supplier component and `org.B.WebOrderRequest` as the client component.

As for Customer A, the approach you adopt is different. This is due to the facts that Customer A wants to link two of his applications (i.e., the web shop and the call centre) to your order management infrastructure rather than a single one. Besides, the storage backend is provided by the RDBMS PostgreSQL which is not supported natively by your ERP framework, in contrast to Oracle. You create the `org.A.PgSqlOrderRequest` class, a concrete and final subclass of *`erp.ordering.OrderRequest`*. This new class serves for two purposes: On the one hand, it implements the storage-specific behaviour, i.e., the methods `getSpendingLimit()`, `getDiscount()`, and `store()` based on PostgreSQL connectors. On the other hand, it carries the overall business logic for processing order requests in its `process()` method (see also Figure 5). The method body combines the actions provided by the superclass and the ones owned by the subclass to form the activity shown in Figure 3. The web shop and call centre components then use instances of the `org.A.PgSqlOrderRequest` class to trigger the handling of order requests. The application components `OnlineOrderingA` and `CallOrderingA` hence become dependent upon the framework components.

While having omitted many details, the implementation realises the `GeneralOrdering` component as a class library. This library-based reuse strategy materialises a strict LAYERS structure with top-down use dependencies. These dependencies result in a a top-down, one-way coupling between the application components (e.g., `OnlineOrderingA`) and the `GeneralOrdering` component

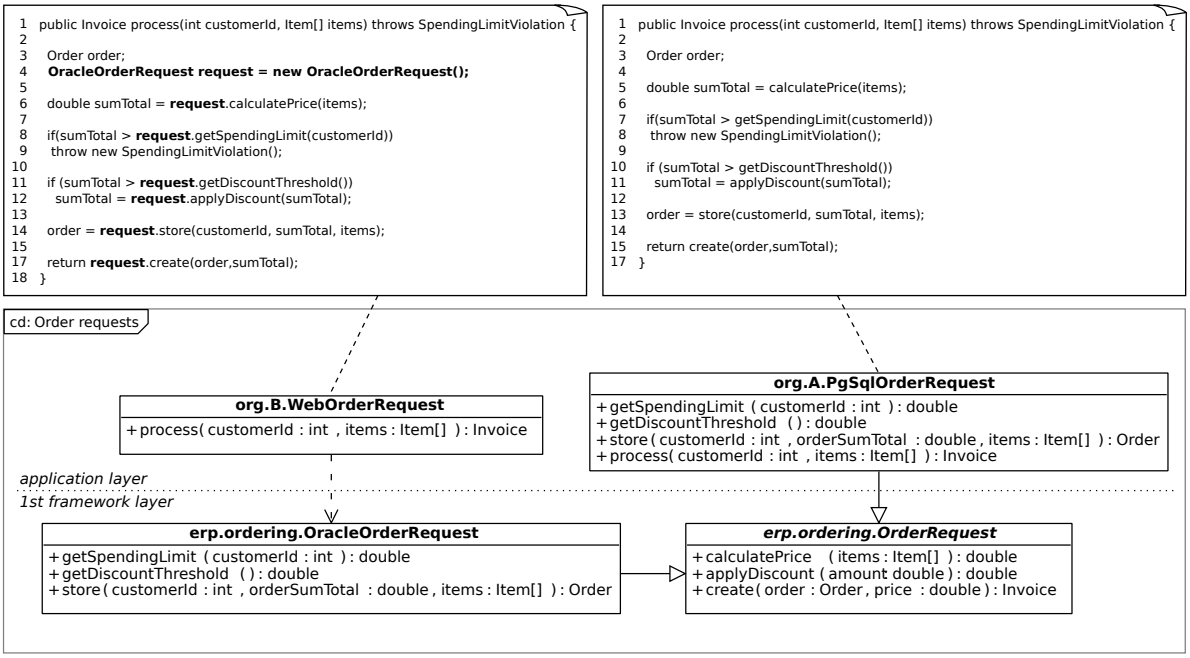

Figure 5: Layering dependencies

[BMR+00]. By one-way coupling, we mean that the called component (`GeneralOrdering`) is not aware of the calling component's identity (e.g., `OnlineOrderingA`) or any of its characteristics (e.g., signature interface details, announcements of error conditions etc.), leaving aside possible means of language-level introspection.

Let us take a closer look at the distribution of shared and non-shared process assets between the application and framework layers. First, you will find that the implementation strategy and the architectural configuration chosen cause code replication. It appears to you that the essential behaviour of `process()` method is implemented twice, once by the `org.B.WebOrderRequest` class, once by the `org.A.PgSqlOrderRequest` class. This implementation certainly realises a strict LAYERS structure and a clear separation of responsibilities between the application and framework layers. However, this implementation approach misses out opportunities for reusing a slice of implemented behaviour.

As a consequence, you consider a common refactoring step. By moving the `process()` method up the inheritance hierarchy (i.e., by applying a so-called PULL UP METHOD refactoring [Fow03]), you obtain this missing bit of implementation reuse. To obtain the maximum reuse, you make a `process()` method definition member of the abstract *`erp.ordering.OrderRequest`* class. With this, it is reusable both by inheritance and by call composition through the final and concrete subclasses of *`erp.ordering.OrderRequest`*.

However, at second sight, you realise that such a refactoring introduces a new and, this time, reversely directed use dependency between the components of the application and the framework layers. As the individual actions, which form the overall order placement activity, are effectively defined at the subclass level (due to the specificity of the legacy storage systems, data schemes, etc.), the now generalised `process()` method defers certain responsibilities the subclass-level method implementations realising the non-shared actions, e.g., `getSpendingLimit()`. As the concrete, storage-specific subclasses potentially reside at the application layer, this reversely directed use dependencies would mean a bottom-up dependency between the application and framework layer. Such a dependency configuration, however, contradicts the strict LAYERS structure while it appears necessary for achieving an important *implementation reuse*, here the reuse of a particular activity as control and data flow. You just encountered an occurrence of a major tension between implementation reuse and strict layering!

4 Pattern: Inversion-of-Control Layer

Your architecture is strictly layered. The LAYERS require and provide EXPLICIT INTERFACES from and to each other, in a top-down manner, from the higher to the lower LAYERS. Over time, your architecture becomes deployed in different requirements settings or product configurations. Against this accumulating experiences, you consider steps of perfective reengineering through refactoring. In particular, you plan to refacture towards a higher degree of reuse for implemented pieces of complex and composite behaviour. For instance, you identified recurring similarities in business process descriptions within a single or even across multiple target domains. Yet, now that your components shall offer implementations for shared behaviour (e.g., process descriptions) to arbitrary clients, you want to ascertain that the client components can refine selected and predefined steps within these reused units of behaviour following a precise protocol. Ideally, the needs for managing input and output requirements when executing the shared behaviour should be minimal and guided by an EXPLICIT INTERFACE.

However, if two layers (e.g., an application and a framework layer) are linked through a strictly one-way, top-down use relationship (e.g., top-down instruction calls), you as an architect will not be able to place *potentially* shared and reusable behaviour at the lower layer while preserving the original top-down, one-way call dependency structure. This is because reusing the behavioural descriptions while keeping certain steps adaptable through the client components requires calling from within the lower layer into the higher one, in some way or the other.

How do you realise advanced forms of behavioural reuse, in particular by implementing *composite behaviour* at lower layers, while preserving behavioural adaptability from higher layers in a LAYERS architecture?

When using LAYERS in a strict and unrefined manner, it appears unavoidable to limit a particular kind of reusability. A LAYERS structure expects us to place pieces of composite behaviour, i.e., behaviour that is expressed over other actions realised at the same or higher layers, at the most concrete or highest-possible layer. This is due to maintaining the premise of top-down call dependencies under a low coupling of the higher to the lower layer.

Hence, as a software architect, you must question yourself whether *top-down use dependency relationships between two layers (i.e., the application and the framework layer) resulting from applying a strict* LAYERS *structure allow for a maximum reuse of shared process assets (i.e., control and data flow, actions, and objects) between the potential client components?* To put it differently, to which degree are the potentially shared process assets placed at and available from the higher (e.g., framework) layer? The behaviour implementation in lower-level layers should – from the higher-level layer view – be modifiable, portable, and reusable.

It becomes clear that a strictly hierarchical decomposition, as proposed in the LAYERS pattern, is not always enough. Interactions between layers are one-way and constrained to a *protocol* provided by the EXPLICIT INTERFACES. In its purest and effect-free form, behaviour exposed through an EXPLICIT INTERFACE is considered stateless and, therefore, free of side-effects. Therefore, a purely layered structure can be cumbersome to use, if multiple configurations are of the shared behaviour implementation, along predefined points of variation, are needed by client components at a higher layer.

A solution must be viable under different design strategies available for layering, for instance, in class-based programming languages. On the one hand, layering can be achieved by means of method propagation along a class hierarchy, sometimes referred to as *layering by inheritance* [BMR+00]. On the other hand, layering can be achieved by establishing and managing call references to objects

within method bodies. This variant is often referred to as *layering by call composition*. Inheritance-based layering falls into the category of white-box reuse due to both the wide range of refinement possibilities as well as the unwanted dependencies on the internals of each level of the class hierarchy. Call compositions represent a black-box reuse as dependencies are limited to the signature interfaces of the objects referenced. At the same time, the internals remain entirely opaque to the calling components which prevents them from actually mangling the behaviour that is being reused.

In many architectures the general control flow and major components are given, but different applications contain different user-defined refinements (e.g., classes) that should extend base components, defined in lower layers of the architecture, with custom behaviour. Configuring this by passing configuration data using top-down parametrisation (e.g., passing configuration values through each layer successively) is rather cumbersome, details on how the configuration values turn into observable variations in the reused composite behaviour remain sealed from the application developer. Also, parameter passing means that we need to foresee in the lower-level layer which configuration options are possible. The permissiveness and flexibility of ordinary forms of parametrisation are too constraint. Configuring lower from higher layers through simple forms of PARAMETRIZE FROM ABOVE (e.g., extended operation parameter lists) alone is insufficient.

Therefore:
Loose some constraints present in a strict LAYERS architecture. Introduce an intermediate layer between the top-most (e.g., application) layer and the next-lower layer. Place an adaptable implementation of the identified piece of control and data flow at this layer. For offering this new service, allow the intermediate layer to impose usage requirements for the new service on client components at the top-most layer. Make sure that these requirements are expressed by EXPLICIT INTERFACES. For most scenarios, this means *adding* inverted call and use dependencies between the top-most and this intermediate INVERSION-OF-CONTROL LAYER.

Provide an INVERSION-OF-CONTROL LAYER between the top-most (e.g., an application) layer and the next-lower (e.g., a sub-system) layer. At this intermediate layer, place control inversion components which (a) embody definitions of higher-level, composite behaviour and which (b) expect concrete variation components to be provided by client components at the top-most layer to enact the outlined, yet abstracted behaviour. In addition, equip the INVERSION-OF-CONTROL LAYER with configuration components which allow client components to adapt the variation and execution behaviour of the control inversion components.

The INVERSION-OF-CONTROL LAYER offers regular service types according to the LAYERS patterns to its antecedent layers, i.e., concrete components through their component interfaces, while requiring services from its lower-level ones. In addition, the layer exposes two different kinds of EXPLICIT INTERFACES which deviate from ordinary component interfaces; i.e. *control configuration interfaces* and *component variation interfaces*:

- *Provided control configuration interfaces*: These configuration interfaces differ from ordinary component interfaces and may take various forms, ranging from language-level instructions (e.g., operators managing subclass-superclass relations) to freestanding and complex configuration and execution environment components (e.g., containers) hosted by this intermediate layer. The presence of control configuration interfaces is one unique property of INVERSION-OF-CONTROL LAYERS.

- *Required component variation interfaces*: These interfaces can resemble component interfaces in the sense of operation contracts, however, more commonly they expose entire component specifications (e.g., object-types, interface-only constructs, abstract classes) to the antecedent layer, describing predefined points of variation which are to be bound by components in the integration layer. Component variation interfaces are the building blocks to lay out the abstracted

behaviour (e.g., in terms of object-types cooperating in an object collaboration) and contract the variation requirements (e.g., superclass interfaces in abstract class hierarchies).

Therefore, the INVERSION-OF-CONTROL LAYER patterns establishes mutual dependencies between a top-most LAYER, the integration, and its descendant, the INVERSION-OF-CONTROL LAYER. However, these dependencies relate to different concerns, i.e., configuring the shared process execution, providing and activating the actual extensions and refinements, as well as triggering the shared process execution in terms of a layer service.

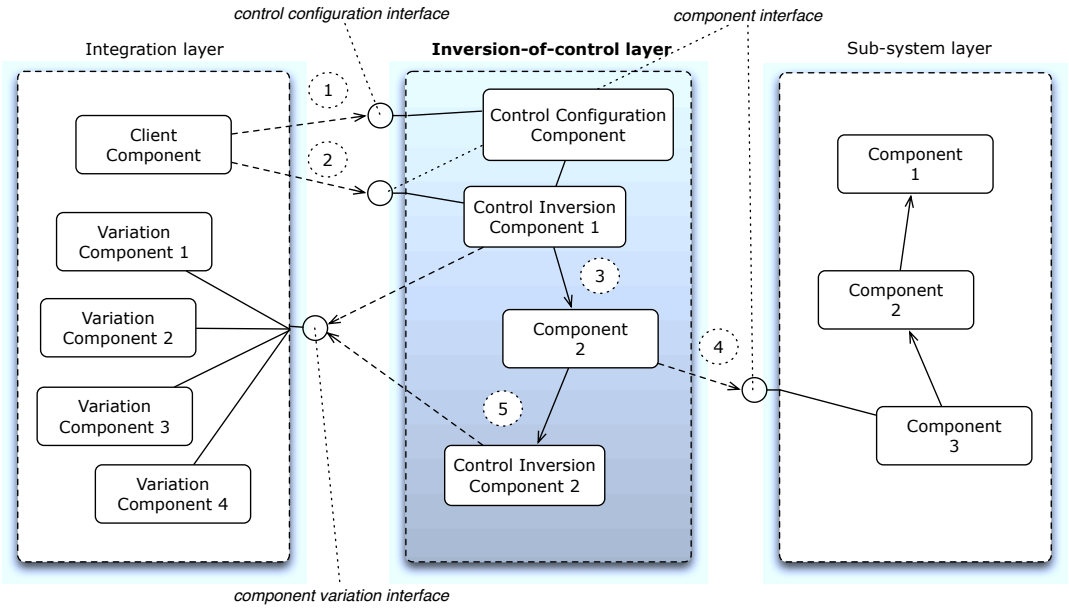

Figure 6: A structural view of the INVERSION-OF-CONTROL LAYER pattern

Figure 6 shows the main participants of a INVERSION-OF-CONTROL LAYER variant for a three-tier architecture including an application, the inverse control, and a sub-system layer. As the top-most layer, the application layer hosts two kinds of components. On the one hand, client components that realise the overall appearance and behaviour of a final concrete application. On the other hand, it contains variation components which are fed to the direct descendant layer, the actual INVERSION-OF-CONTROL LAYER. These variation components must comply with a contracted form (e.g., a signature interface), captured by a component variation interface. This variation interface is required by the actual INVERSION-OF-CONTROL LAYER. The intermediate layer contains three kinds of components: control inversion, control configuration, and conventional components. Control inversion components lay out pieces of composite behaviour in terms of object collaborations using the component variations interfaces as the collaborations roles. Hence, these components specify a control and data flow using abstracted components. At the application layer, these components are implemented and provided to the INVERSION-OF-CONTROL LAYER. This establishes a shared responsibility between the INVERSION-OF-CONTROL LAYER and the application layer. The INVERSION-OF-CONTROL LAYER defines the control and data flow, as well as the use of the components. At the application layer, it is decided which concrete instance of which kind of components is enacted. As a third kind of layer in Figure 6, there is the sub-system layer. This layer is unidirectionally used from its antecedent layer, containing only ordinary components which might participate in realising the composite and inverted behaviour expressed at the INVERSION-OF-CONTROL LAYER.

To illustrate and elaborate on this characteristic layering and dependency structure, consider the example of the GoF TEMPLATE METHOD pattern [GHJV94] in Figure 7 below. At the INVERSION-OF-CONTROL LAYER, the role of the *control inversion component* is taken by `AbstractClass`. This is because the abstract class defines a generic flow of control and data in

a template method, i.e., `TemplateMethod()`. The piece of composite behaviour appears generic and abstracted because, apart from declaring invocations upon `PrimitiveOperation1()` and `PrimitiveOperation2()`, *`AbstractClass`* lacks concrete implementations of these operations. They remain required, but unimplemented hook methods. The specified set of hook methods form the *component variation interface* which is to be implemented at the application layer. In the context of abstract class designs, the *component variation interface* is sometimes referred to as the standard protocol [JR91, Woo95]. In this example, the implementing *variation component* is `ConcreteClass`, offering two concrete and final implementations of the hook methods. As for the unfolded control flow, an application-layer component `Client` depends and calls upon the template method. This use dependency reflects a conventional bottom-up service offered by the INVERSION-OF-CONTROL LAYER to the application layer. Upon invoking on the template method, the control and data flow is actually inverted as the concrete and final operation implementations owned by `ConcreteClass` are executed. Finally, the *control configuration interface* is here provided at the language level, by the language-specific subclass-superclass relation operator and the variant of (abstract) method combination realised in the language model. For instance, the language model can foresee different strategies for realising abstract class designs. Examples are dedicated `abstract` modifiers, enforced by the language runtime, or the various flavours of subclass responsibility idioms in the Smalltalk tradition. Another control configuration aspect is determined by the availability of early or late binding of concrete superclass-subclass relations in a language model.

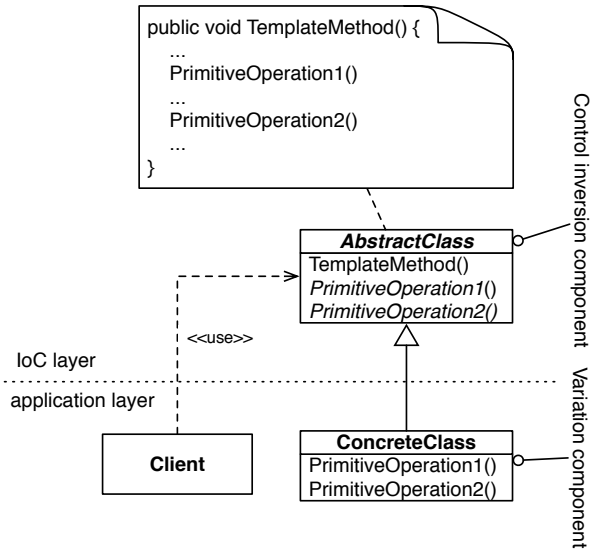

Figure 7: A known use of the INVERSION-OF-CONTROL LAYER pattern: A GoF TEMPLATE METHOD implementation variant (adapted from [GHJV94, p. 327])

From the perspective of the developer providing the variation components (e.g., a concrete subclass for an abstract superclass), her engineering tasks changes from being predominantly imperative (How to achieve a particular activity by reusing and sequencing certain instruction calls?) to being more declarative (What to provide in terms of behaviour and data to realise an already sketched activity?). The use of an INVERSION-OF-CONTROL LAYER helps to make such usage requirements more explicit. For instance, looking at the TEMPLATE METHOD example above, it is strictly stipulated which methods must be provided by a subclass rather than leaving this to the discretion of the application developer; [Vli96]).

When having a predominantly procedural background, this input-output-centric perspective introduced by the INVERSION-OF-CONTROL LAYER pushes you in a more object-based programming

model [Vli96]. Note that this is not necessarily related to using an object-based or an object-oriented programming language. Rather, you are tricked into thinking in terms of collaborations between input entities, without knowing the exact order of the resulting instruction calls or without being aware of the underlying web of structural relations (i.e., associations) between those input and output entities exchanged with the INVERSION-OF-CONTROL LAYER services.

When applying this architectural pattern and different developers or developer teams are responsible for realising different layers' tasks, additional coordination work between these developers or developer teams is necessary. This coordination effort needs to follow the direction of the use dependencies between the LAYERS [AA07]. In the case of the INVERSION-OF-CONTROL LAYER, the team in charge of creating and maintaining the control inversion components must actively communicate changes in the control and data flow to the teams responsible for integrating the INVERSION-OF-CONTROL LAYER services. The above communication requirement follows from the fact that an INVERSION-OF-CONTROL LAYER hides many details of the control and data flow finally exhibited by your application. As an application developer, you must investigate a framework's documentation, beyond studying mere signature interfaces and API documents, to anticipate this partially complete control and data flow when writing your application.

5 Motivating Example Resolved

So far, our motivating example of an order placement sub-framework (see Section 3) has assumed that a strict top-down call dependency structure following the LAYERS pattern must be maintained. The application layer components such as `org.B.WebOrderRequest` or `org.A.PgSqlOrderRequest` maintained one-way references to framework layer components, i.e., a dependency relationship to `erp.ordering.OracleOrderRequest` and a generalisation relationship to *`erp.ordering.OrderRequest`*, respectively. You obtained an implementation reuse of certain process assets (e.g., common process steps such as `calculatePrice()` as well as organising storage-specific behaviour such as `getSpendingLimit()`).

Yet, there is potential for further reuse. Namely, you can realise a sharing of the overall process definition among all client applications, i.e., the control and data flow described by `process()` methods). However, refactoring the ownership of this behavioural definition contests the LAYERS structure. Such a refactoring modifies the architectural configuration to an INVERSION-OF-CONTROL LAYER architecture. Figure 8 illustrates the distribution of shared and non-shared assets across the application and framework layer *before* such a refactoring, based on the exemplary implementation as given in Section 3. Now, consider a stepwise refactoring of the control and data flow definition towards the framework layer, adopting different, yet complementary design practises: ABSTRACT CLASS designs using TEMPLATE METHOD and hook methods (see Section 5.1), composition and delegation (see Section 5.2), as well as forms of dependency injection (see Section 5.3). These three different, yet architecturally related refactoring approaches introduce *inverse*, or bottom-up, interface requirements between the framework and the application layer.

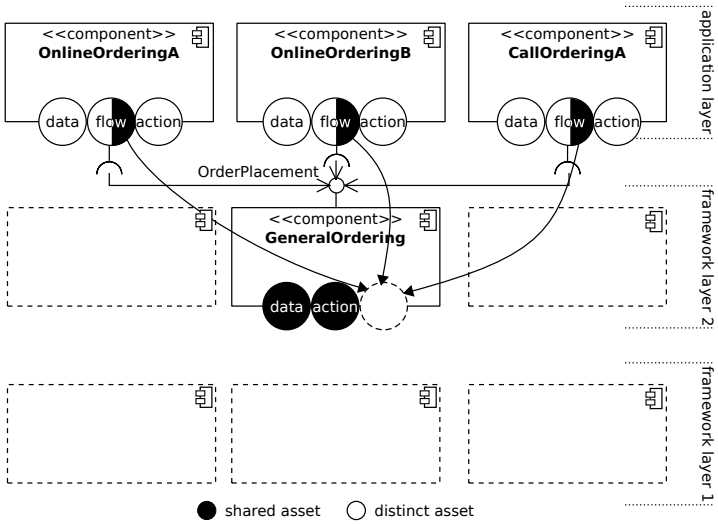

Figure 8: Refactoring towards an inversion-of-control architecture

5.1 Using Abstract Classes

Let us first consider how to realise an INVERSION-OF-CONTROL LAYER using an abstract class and template methods. In a first step, you can devise Java's class-based inheritance for refactoring towards activity reuse. You adopt a variant of the well-established ABSTRACT CLASS [JR91, Woo95] pattern and TEMPLATE METHODS [GHJV94]. This involves a couple of steps (see also Figure 9): First, you create a `process()` method owned by the abstract *`erp.ordering.OrderRequest`* class. The processing steps (i.e., method calls) and their overall ordering in the method body remain unchanged (see the bottom-right comment block in Figure 9). However, the messages produced at these call sites will be propagated and received differently. The method calls to the al-

ready shared method implementations, i.e., those already owned by *erp.ordering.OrderRequest* and inherited to its final, concrete subclasses (e.g., `create()`) are self-referentially resolved. However, the non-shared, storage-specific method implementations (e.g., `store()`) which are referenced in the `process()` body are only negotiated for being implemented by the final, concrete subclasses, on behalf of *erp.ordering.OrderRequest*. In a second step, you must provide method definitions. These abstract methods establish a contract between the abstract superclass and its concrete subclasses to provide the appropriate implementations for `getSpendingLimit()`, `getDiscountThreshold()`, and `store()`. The superclass is built around a specific interface which must be provided by its subclasses. The subclass interface in our example shown in Figure 9 consists of the signatures of the three non-shared methods and establishes an interface dependency of the lower upon the higher layer. The ABSTRACT CLASS and TEMPLATE METHOD approaches realise inversion-of-control layering based on standard inheritance mechanisms.

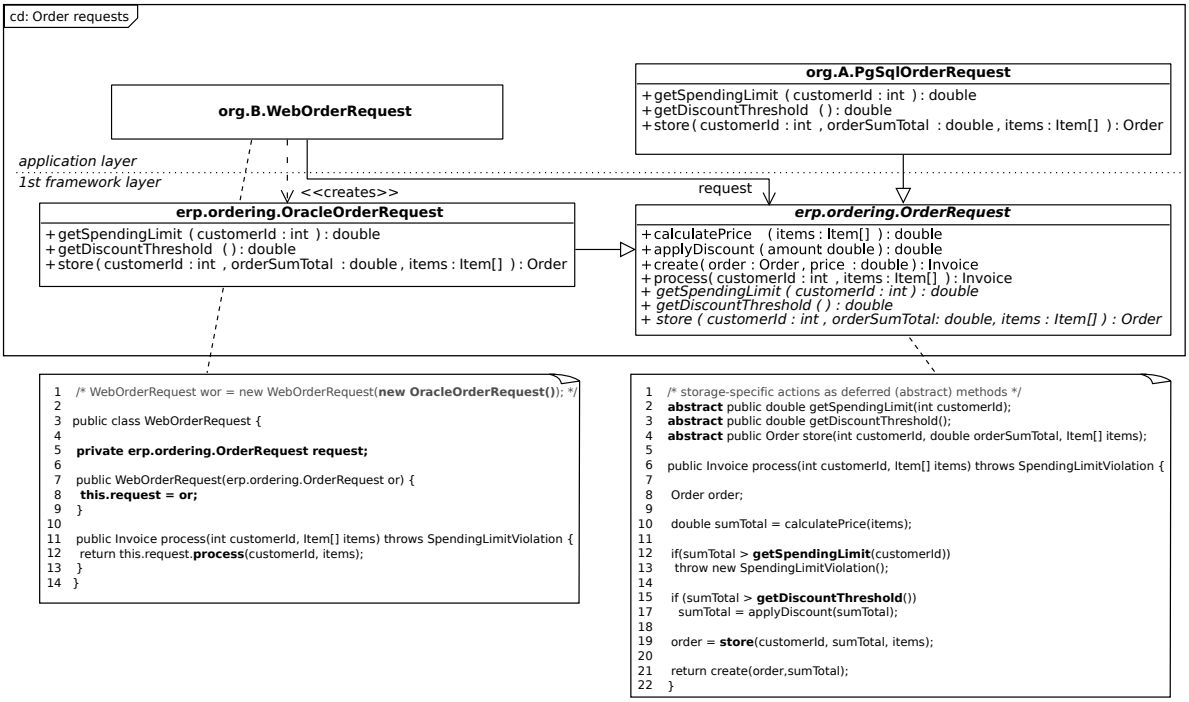

Figure 9: Inverting control for reusing shared control and data flow definitions

5.2 Using Delegation and Intrinsic Dependency Management

Layering by class-based inheritance, in general, bears advantages and disadvantages [BMR+00]. On the one hand, the higher or application layers preserve means to refine components used from the lower or framework layers, for instance by wrapping the `process()` method owned by *erp.ordering.OrderRequest* in a custom subclass. On the other hand, mere subclassing introduces dependencies beyond the top-down generalisation dependency and the aforestated bottom-up dependency on a subclass interface. For instance, the refining subclasses become dependent on the design decisions on the state layout of the super-objects . To boil down the top-down dependencies to an interface-only dependency, effectively sealing implementation details in the *erp.ordering.OrderRequest* class hierarchy from the actual call site, the integration of the now shared processing logic for order request can be achieved by referencing and instantiating a final and concrete subclass of the *erp.ordering.OrderRequest* hierarchy. This instance can then be sent `process` messages from within application layer components. Looking at the implementation sketch in Figure 9, the `org.B.WebOrderRequest` class is slightly rewritten to integrate the framework layer component `erp.ordering.OracleOrderRequest` by delegation, rather than inheritance. The `process()` method

specific to `org.B.WebOrderRequest` class forward-delegates to the actual `process()` method provided by the a subclass-instance of *`erp.ordering.OrderRequest`*. The serving *`erp.ordering.OrderRequest`* instance is to be provided to the `org.B.WebOrderRequest` instance upon construction time, by passing it as a constructor argument. With this, the dependency upon the *`erp.ordering.OrderRequest`* becomes explicit, turning it into an association relationship (see also Figure 9).

While this delegation-based wrapping allows for some sort of refinements , the dependency at this concrete call site is limited to the interface of the *`erp.ordering.OrderRequest`* abstract class. Yet, along this bottom-up dependency path between framework and application layer, an unwanted kind of nominal or referential coupling remains:

```
WebOrderRequest wor = new WebOrderRequest(new OracleOrderRequest());
```

Upon constructing a `org.B.WebOrderRequest` instance, the constructor-calling client must name a concrete instantiation target, i.e., a *`erp.ordering.OrderRequest`* class provided by the framework layer such as `erp.ordering.OracleOrderRequest`. Hence, the application layer must fulfil the construction requirement `org.B.WebOrderRequest` by selecting and instantiating a particular class explicitly. We refer to this as an *intrinsic* dependency management. This nominal dependency couples the application layer component to using one and only one specific order request processor, as well as makes the calling component dependent on the name of this processor component. This impedes certain quality attributes, for instance, testability and evolvability (e.g., imagine a later migration of customer A to another storage vendor). Note that this coupling along the top-down dependency path is not characteristic to this refactoring step. It was already present in the original implementation in Section 3:

```
public class WebOrderRequest {
    public Invoice process(int customerId, Item[] items) throws SpendingLimitViolation {
                /* ... */
                OracleOrderRequest request = new OracleOrderRequest();
                /* ... */
    }
}
```

5.3 Using Delegation and Extrinsic Dependency Management

So, how can you preserve the advantages of inverting the control between the application and framework layer when defining the processing activity for order requests, while avoiding the excessive use of subclassing and while reducing the dependency coupling of the application layer components to the framework layer? In a final refactoring step, you may apply inversion of control for a second purpose, namely by moving the responsibility of managing application layer components (in terms of their lifecycle) and populating them with their dependencies from the application to the framework layer. For this, consider a particular environment for *dependency injection* [Fow04] which is extrinsic to the application layer. We look at an implementation variant of extrinsic dependency management offered by the Spring/Java application framework [Spr10], i.e., the so-called constructor injection.

When integrating an application framework like Spring, the design of your framework is extended by some abstractions. Most prominently, you add controlled management and execution environments (i.e., so-called containers) for application components (e.g., `org.B.WebOrderRequest`) and framework components being part of the component library (e.g., `erp.ordering.OracleOrderRequest`). In the Spring flavour, the containers correspond to *application contexts* while their managed objects are referred to as *beans*.

Looking at the concern of dependency injection in isolation (see Figure 10), the `ApplicationContext` component is added to your framework layer. In the following, it takes to role of

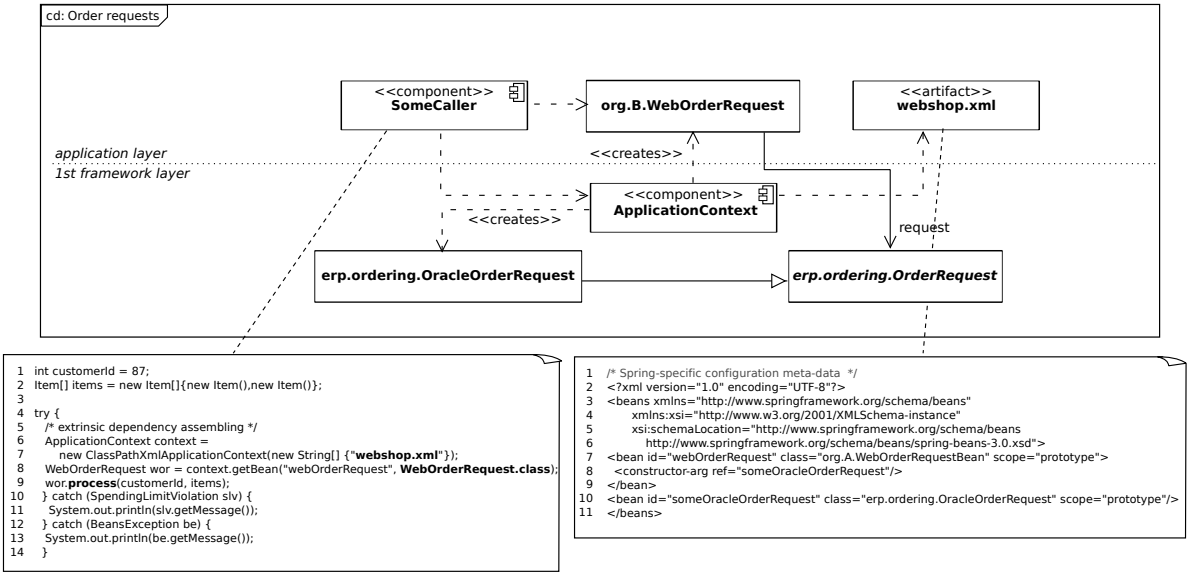

Figure 10: Inverting control for injecting dependencies

an dependency assembler and injector. Integrating your framework into the web shop application of Customer A now involves several steps. First, you provide a meta-data artifact which describes the dependency composition you want to achieve at the application layer, i.e., creating an instance of `org.B.WebOrderRequest` and equipping it with an instance of `erp.ordering.OracleOrderRequest`. A possible, quite verbose form of specifying the construction of a `org.B.WebOrderRequest` instance is an XML-based assembly document (i.e., `webshop.xml`; see Figure 10). In this Spring-specific XML notation, you commission the creation of two objects, referenced by the names `webOrderRequest` and `someOracleOrderRequest`, respectively. In addition, by noting a `<constructor-arg/>` element, you specify that the object `someOracleOrderRequest` shall be passed into the constructor call of the `webOrderRequest` object. This construction plan is then to be provided to the application context by the calling application component. The application context component then consumes the specification and proceeds by constructing an instance of `erp.ordering.OracleOrderRequest` which is then provided to the constructor of `org.B.WebOrderRequest`. In doing so, the `ApplicationContext` component assembles the object dependencies otherwise to be specified manually at the application layer. The dependencies exhibited at the application layer are limited to using a generic management interface provided by the `ApplicationContext` component to retrieve managed objects (i.e., beans; see Figure 10).

In this last refactoring step, the dependency managing is reduced to a problem of parametrising the component layer, in the above example through a document-like interface. Hence, we gain flexibility in assembling inter-object dependencies, for evolving an application by introducing new dependency implementations or test-enabling components (e.g., mock objects).

To sum up, INVERSION-OF-CONTROL LAYERS may be introduced in a variety of scenarios which usually fall into two categories: (a) inverting control for control and data flow reuse and (b) inverting control for managing object dependencies in framework-integrating applications from the outside of these applications. A refactoring towards extrinsic dependency injection certainly introduces new complexities, among others, an external dependency on a new sub-framework (i.e., Spring) and a reduced locality due to multiple source artifacts (i.e., the meta-data document `webshop.xml`).

6 Examples and Known Uses

The use of different variants of the INVERSION-OF-CONTROL LAYER pattern in object-oriented (OO) application frameworks distinguishes OO application frameworks from reuse based on mere class libraries [Mat00, pp. 8]. Class libraries incur the risk of duplicating development effort as well as lacking reuse possibilities. Duplicated effort and increased complexity results from assembling the behaviour offered in different ways, to obtain different solutions to comparable and related design problems, even within the same development project. Most importantly, class libraries do not permit you to capture a control flow design for reuse, which proofed applicable to different design and implementation problems. The application developer using the class library remains responsible for laying out a concrete control flow (e.g., object ordering and message paths) for reused entities provided by the library, and beyond. Designing and implementing control flows in an application-specific manner inhibits their reuse and risks being error-prone, depending on the proficiency of the developer team responsible.

From an architectural view point, an OO application framework exhibits a structure as described by LAYERS with framework-integrating applications residing at the top-most LAYER (applications or integrator layer) that knows at least one, but commonly several INVERSION-OF-CONTROL LAYERS (i.e., framework layers) as its direct descendants. Instruction calls issued at the integrator layer result in triggering and initialising calls to particular components residing at the INVERSION-OF-CONTROL LAYER. These, in turn, cause the framework-level control flow in the inversion-of-control components to be executed. That is, a collaboration of components living both in the INVERSION-OF-CONTROL LAYER and the higher-level layer is executed.

An important design technique for realising INVERSION-OF-CONTROL LAYERS, explored for implementing application frameworks in class-centric and class-only languages [JR91, p. 5], are variants of ABSTRACT CLASSES [Woo95], also sometimes referred to BASE CLASSES or TEMPLATE CLASSES [Pre96], though with slightly extended connotations. These works mainly assume that there is no language mechanism, construct or idiom available for helping you to implement the INVERSION-OF-CONTROL LAYER pattern. Many programming languages do provide explicit language constructs for this purpose, such as the interface constructs in Java and C#, which are discussed in the following. Central to this design practise is the use of a superclass for (a) specifying a signature interface and for (b) implementing reusable designs of object collaborations, control, and data flows.

To begin with, the ABSTRACT CLASS pattern instructs us how to implement EXPLICIT INTERFACES as essential parts of any LAYERS structure based on classes as language constructs alone. Classes as language constructs present you with the challenge of their double purpose which conflicts with the requirements expressed by EXPLICIT INTERFACE: Classes act both as generators for object behaviour (i.e., method implementations) and as generators of object specifications, i.e., the object-types with object-type referring to an object's signature interface [Sim95]. This historically challenged conceptualisation and usage of classes leaves you with an object generator and an object-type generator merged into a single modularisation unit. In spite of the widely available late binding of message receivers (i.e., a method signature) to method implementations under forms of class inheritance, for instance, the standard use of classes couples call-dependent objects more than necessary. The coupling is not restricted to a signature interface, but also to a default implementation strategy of both the behaviour and the representations of the object's state. Also, method implementations may produce side effects which are not reflected by the method's signature. To realise an EXPLICIT INTERFACE, you want a class to act as a complete object-type generator while not using its capacity as object generator, at least partially [Sim04]. Also, you want to restrict it usage to superclass-subclass relationships only.

- *Superclass-only for specification*: The LAYERS pattern foresees a clear call dependency structure between components residing at two adjacent layers. The INVERSION-OF-CONTROL LAYER specialisation describes two-way call dependencies between an integrating and a controlling layer. The ABSTRACT CLASS describes various practises of creating integrator and INVERSION-

OF-CONTROL LAYER components in terms of classes, class hierarchies, and inheritance-based behaviour refinement. The mutual call dependencies are realised as particular method combinations along superclass-subclass relationships. As a consequence, the top-level, abstract classes are often found restricted to act as superclass/subclass in an object system. That is, an abstract class conceived in such a role is sometimes prohibited from being instantiated directly.

- *Negotiating method signatures*: An object's signature interface is made up by the set of method signatures, each describing the kind and number parameter, their reference names, type specifications for parameters and return values, and possibly additional annotations such as pre- and post-conditions etc.). To realise an EXPLICIT INTERFACE which is binding both for the callers of its object offsprings and for its integrating subclasses, the special-purpose superclass only owns method specifications without implementations. This can either be achieved by means of first-class language features (e.g., the `abstract` modifier in Java and C#) or by providing *placebo* method implementations. A common placebo approach has originally been explored by the Smalltalk community and is referred to as *subclass responsibility* . For other languages, similar idioms have been proposed, e.g., the INTERFACE CLASS [Hen99, Hen00, Rad04, Rad05, Rad06] in C++.

- *Capturing a reusable control and data flow*: On the one hand, the ABSTRACT CLASS pattern devises the special-purpose superclass to own non-implemented operation specifications. The superclass is considered abstracted in this sense. On the other hand, a second piece of abstracted design is owned by the superclass, i.e., operations which lay out and enforce a certain, recurring order of calls (i.e., message sends) to the abstracted method declarations. This has also been referred to as variants of the TEMPLATE METHOD [GHJV94] and HOOK METHOD [Pre96] patterns.

More recent design practises for realising INVERSION-OF-CONTROL LAYERS are certain techniques subsumed under the label of *dependency injection* [Fow04]. While all forms of dependency injection preserve the idea of creating INVERSION-OF-CONTROL LAYER components (e.g., at the framework level) which require integrator components to fully or partially implement predefined interfaces, the distinguishing characteristic of the dependency injection variants is the presence of a dedicated dependency manager component, also referred to as an assembler object in [Fow04]. This dependency manager component is responsible for providing a programming model for selecting and serving integrator components to the actual INVERSION-OF-CONTROL LAYER components. This intermediary further decouples the integrator and INVERSION-OF-CONTROL LAYER components and permits you to foresee more complex dependency injection schemes (e.g., lifecycle management, conditional injection, injection by runtime configuration, etc.). For example, INVERSION-OF-CONTROL LAYER components do not have to be aware any concrete integrator component implementation in their own implementation. Also, the concrete strategy for selecting integrator components for injection does not need to be embedded into the IoC component's implementation.

7 Conclusion

As already hinted at in the introductory section, analytical and critical work on this architectural design practise is not new. Also in the pattern community and related architectural pattern work, the critical reader will find references to what has been described here as the INVERSION-OF-CONTROL LAYER. To give a prominent example, in [BMR$^+$00], a particular call dependency between between different LAYERS is discussed as *bottom-up communication* based on callbacks. However, here the emphasis is on preserving the top-down, one-way decoupling between the higher and lower layer while still inverting some of the control. The architectural need for such an inversion of control is not central to the discussion in [BMR$^+$00]. Besides, referring to the use of callbacks only reduces the notion of bottom-up communication to this form of implicit invocations. Given the number of characteristic

consequences of inverting the control between LAYERS, a dedicated treatment under an architectural pattern form appears justified to us.

8 Acknowledgements

From EuroPLoP 2010, we would like to thank Michael Weiss for his shepherding. Our work was partially supported by the European Union FP7 project COMPAS, grant no. 215175.

References

[AA07] Marwan Abi-Antoun. Making Frameworks Work: A Project Retrospective. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 1004–1018, 2007.

[AZ05] Paris Avgeriou and Uwe Zdun. Architectural Patterns Revisited – A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, pages 1 – 39, Irsee, Germany, July 2005.

[BDMN75] Graham M. Birtwisthle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur. Petrocelli/Charter, New York, NY, USA, 1975.

[BH03] Frank Buschmann and Kevlin Henney. Explicit Interface. In *Proceedings of EuroPLoP 2003*, Irsee, Germany, 2003.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*. Wiley Series on Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, April 2007.

[BMR+00] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, editors. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd., Chichester, England, 2000.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, USA, 1st edition, 1995.

[Fow03] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. The Addison-Wesley object technology series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://martinfowler.com/articles/injection.html`; last updated: January 23, 2004; last accessed: February 15, 2010, 2004.

[Fow05] Martin Fowler. Inversion of Control. In: *Martin Fowler's Bliki*. Updated June 26, 2005. Retrieved February 15, 2010, from `http://martinfowler.com/bliki/InversionOfControl.html`, 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, October 1994.

[Hen99] Kevlin Henney. Coping with Copying in C++. *Overload*, 33, August 1999.

[Hen00] Kevlin Henney. C++ Patterns Source Cohesion and Decoupling. *Overload*, 39, September 2000.

[Hen05] Kevlin Henney. Context Encapsulation – Three Stories, a Language, and Some Sequences. In *Proceedings of EuroPLoP 2005*, Irsee, Germany, 2005.

[Hen07a] Kevlin Henney. The PfA Papers: Context Matters. *Overload*, 82, December 2007.

[Hen07b] Kevlin Henney. The PfA Papers: From the Top. *Overload*, 80, August 2007.

[Hen07c] Kevlin Henney. The PfA Papers: The Clean Dozen. *Overload*, 81, October 2007.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June-July 1988.

[Joh02] Rod Johnson. *Expert One-on-One J2EE Design and Development*. John Wiley & Sons, Inc., 1st edition, October 2002.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, Department of Computer Science, University of Illinois, Urbana, IL, USA, May 1991.

[Mar96] R. C. Martin. The Dependency Inversion Principle. *C++ Report*, 8(6):61–66, June 1996.

[Mat00] Michael Mattsson. Evolution and Composition of Object-Oriented Frameworks. Phd thesis, Department of Sotware Engineering and Computer Science, University of Karlskrona/ Ronneby, Ronneby, Sweden, 2000.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[Pre96] Wolfgang Pree. *Framework Patterns*. SIGS Books & Multimedia, 1996.

[Rad04] Robert Radford. C++ Interface Classes — An Introduction. *Overload*, 62, December 2004.

[Rad05] Robert Radford. C++ Interface Classes — Noise Reduction. *Overload*, 68, August 2005.

[Rad06] Robert Radford. C++ Interface Classes — Strengthening Encapsulation. *Overload*, 76, December 2006.

[Sim95] Anthony J. H. Simons. A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language. Phd thesis, Department of Computer Science, University of Sheffield, 1995.

[Sim04] Anthony J. H. Simons. The Theory of Classification Part 12: Building the Class Hierarchy. *Journal of Object Technology*, 3(5):13–24, May-June 2004.

[Spr10] Spring Community. Spring Framework 3.0.2. `http://www.springsource.org/`, last accessed: April 7, 2010.

[Swe85] Richard E. Sweet. The Mesa Programming Environment. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments, Seattle, Washington, United States*, pages 216–229, New York, NY, USA, 1985. ACM.

[Vli96] John Vlissides. Protection, Part I: The Hollywood Principle. *C++ Report*, 8, February 1996.

[WGM89] André Weinand, Erich Gamma, and Rudolf Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2):63–87, 1989.

[Woo95] Bobby Woolf. Abstract Class. In Coplien and Schmidt [CS95].