Framework Composition: Problems, Causes and Solutions

Michael Mattsson & Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
E-mail: [Michael.Mattsson, Jan.Bosch]@ide.hk-r.se
WWW: http://www.ide.hk-r.se/[^michaelm,~bosch]

Abstract
Reuse of software has been one of the main goals of software engineering for decades. With the emergence of the object-oriented paradigm, an important enabling technology for reuse of larger components became available and resulted in the definition of object-oriented frameworks. Our and others experiences with frameworks have shown that frameworks indeed provide considerable reuse in framework-based application development. However, whereas framework-based application development initially included a single framework, increasingly often multiple frameworks are used in application development. These frameworks have to be composed, but the software engineer may experience a number of problems while doing this, related to (1) composition of framework control, composition with legacy components, (3) framework gap, (4) overlap of framework entities and (5) composition of entity functionality. The primary causes for these composition problems are related to (1) the cohesion between classes inside each framework, (2) the domain coverage of the frameworks, (3) the design intentions of the framework designers and (4) the potential lack of access to the source code of the frameworks. Based on the identified problems and causes, we analyse the existing solutions and their limitations.

1 Introduction

Reuse of software has been one of the main goals of software engineering for decades. From the early ages of computer science on, the approach of constructing system by putting together reusable components was a clear goal of software engineers. However, the early approaches, e.g. function and procedure libraries, only provided reuse of small, building-block components. With the emergence of the object-oriented paradigm, an important enabling technology for reuse of larger components became available and resulted in the definition of object-oriented frameworks. An object-oriented framework can be defined as a set of classes that embodies an abstract design for solutions to a family of related problems [Johnson & Foote 88]. In other words, a framework is an abstract design and implementation for an application in a given problem domain.

The presence of a framework influences the development process for the application and several authors, e.g. [Mattsson 96], have defined the following phases in framework-centred software development:

- The framework development phase, normally the most effort consuming phase, is aimed at producing a reusable design and related implementation in a domain. Major results of this phase are the domain analysis model and the object-oriented framework.
- The framework usage phase, sometimes also referred to as the framework instantiation phase or application development phase. The result of this phase is an application developed reusing a framework. Here the software engineer has to adapt the existing frameworks through inheritance of the abstract classes in the framework. For those parts of the application design not covered by reusable classes, new classes need to be developed to fulfil the application’s requirements.
- The framework evolution & maintenance phase. The framework, as all software, will be subject to change. Causes for these changes can be errors reported from shipped applications, identification of new abstractions due to changes in the problem domain, etc.

The advantage of framework-based development is the high level of reuse in the application development. Although this advantage has been claimed by proponents of the technology, recently also scientific evidence supporting this claim has started to appear, e.g. [Moser & Nierstrasz 96]. Although the development of an object-oriented framework
requires considerable effort, the corresponding benefits during application development generally justify the initial effort.

Traditional framework-based application development assumes that the application is based on a single framework that is extended with application-specific code. Recently, however, one can identify a development where software engineers make use of multiple frameworks that are composed to fulfil the application requirements. Experiences from the composition of two or more frameworks clearly show that frameworks generally are developed for reuse by extension with newly written application-specific code and not for composition with other software components. This focus on reuse through extension causes a number of problems when software engineers try to compose frameworks.

In this paper, we study the problems that the software engineer may experience while composing frameworks. The identified problems are based on our experiences from developing frameworks for, among others, fire alarm systems [Molin & Ohlsson 96], measurements systems [Bosch 96], gateway billing systems [Lundberg 96], resource allocation and process operation [Betlem et al. 95], as well as existing literature about frameworks, e.g. [Mattsson 96], [Bosch et al. 97], [Sparks et al. 96]. The composition problems that we identified are related to the composition of framework control, composition with legacy components, framework gap, overlap of framework entities and composition of entity functionality. Our investigations into these problems have lead us to the conclusion that the main causes for the problems are due to the cohesion between the classes in the framework, the domain coverage of the framework, the design intention of the framework designer and the potential lack of access to the source code of the framework. In addition, we discuss the available solution approaches for addressing the identified problems and their causes and the limitations of these solutions.

We believe the contribution of this paper to be the following. Firstly, it provides software engineers employing object-oriented frameworks with an understanding of the problems they may experience during application development, as well as an understanding of the primary causes and solution approaches. Secondly, it provides topics to researchers in object-oriented software engineering that need to be addressed by future research.

The remainder of this paper is organised as follows. In the next section, four example frameworks are introduced that are used to illustrate the problems later in the paper. In section 3, the framework composition problems that we identified are described. Section 4 discusses the primary causes underlying the identified composition problems. In section 5, the various solution approaches for the identified problems are discussed and their limitations. The paper is concluded in section 6.

2 Object-Oriented Framework Examples

In this section, we discuss four frameworks that, in the remainder of the paper, are used to exemplify and illustrate the discussion. These frameworks have been selected for their illustrative ability rather than for their industrial relevance. During the discussion, we assume that it is the intention to reuse two or more frameworks in one application, requiring the frameworks to be composed. The frameworks are a measurement systems framework, a graphical user interface (GUI) framework, a fire-alarm framework and a framework for statistical analysis. Below, the involved frameworks are described. However, since most readers are familiar with GUI frameworks, this framework is not described.

In cooperation with EC-Gruppen AB, we have been involved in the design of a framework for measurement systems. Measurement systems are systems that are located at the beginning or end of production lines to measure some selected features of production items. The hardware of a measurement system consist of a trigger, one or more sensors and one or more actuators. The framework for the software has software representations for these parts, the measurement item and an abstract factory for generating new measurement item objects. A typical measurement cycle for a product starts with the trigger detecting the item and notifying the abstract factory [Gamma et al. 95]. In response, the abstract factory creates a measurement item and activates it. The measurement item contains a measurement strategy and an actuation strategy and uses them to first read the relevant data from the sensors, then compare this data with the ideal values and subsequently activate the actuators when necessary to remove or mark the product if it did not fulfil the requirements. We refer to [Bosch 96] for a more detailed description of the framework.

The statistical analysis framework, in this context, will be used for the statistical analysis of the data gathered by the measurement system. Although measurement systems generally measure every production item, it is often beneficial to be able to collect measurement data and analyse it statistically to condense the amount of data and to increase the information value. Examples of statistical analysis on measurement item data are correlations between measurements on items and equipment having manufactured the items and the distribution of measured items in various quality cate-
categories. The results of the analysis may be used on-line to adjust the settings for manufacturing equipment. In the remainder of this paper, we will discuss two versions of the framework, one version only providing the core functionality for statistical analysis and a second version containing, next to the core functionality, functionality for the graphical representation of statistical data, e.g. histograms.

TeleLarm AB, a Swedish security company, develops and markets a family of fire alarm systems ranging from small office systems to large distributed systems for multi-building plants. An object-oriented framework was designed as a part of a major architectural redesign effort. The framework provides abstract classes for devices and communication drivers as well as application abstractions such as input points and output points representing sensors and actuators. System status is represented by a set of deviations that are available on all nodes in the distributed system. The notion of periodic objects is introduced giving the system large grain concurrency without the problems associated with asynchronous fine grain concurrency. Two production instantiations from the framework have been released so far. For more information, we refer to [Molin & Ohlsson 96].

3 Framework Composition Problems

Traditional framework-based application development is based on a single framework that is extended with newly developed code covering the application specific parts of the application. During recent years, we have seen that application development is increasingly often based on multiple frameworks that need to be composed each other, with class libraries and with existing legacy components. This composition process may lead to composition problems since frameworks generally are designed based on the traditional perspective, based on assumptions such as that the framework is in full control of the event loop, the framework is designed for extension (not composition) and no legacy components need to be incorporated. In this section, five composition problems are described that may occur when composing a framework with another software artifact. Although software engineers may encounter additional problems, we believe this list to cover the primary problems.

3.1 Composition of Framework Control

One of the most distinguishing features of a framework is its ability to make extensive use of dynamic binding. In traditional class or procedure libraries, the application code invokes routines in the library and it is the application code that is in control. For object-oriented frameworks, the situation is inverted and it is often the framework code that has the thread of control and calls the application code when appropriate. This inversion of control is often referred to as the Hollywood principle, i.e. “Don’t call us - we call you”. In figure 1, this inversion is graphically illustrated.

![Figure 1. The inversion of control](image)

In [Sparks et al. 96], a distinction is made between ‘calling’ and ‘called’ frameworks. Calling frameworks are the active entities in an application, controlling and invoking the other parts, whereas ‘called’ frameworks are passive entities that can be called by other parts of the application, i.e. more like class libraries. One of the problems when composing two calling frameworks is that both frameworks expect to be the controlling entity in the application and in control of the main event loop.

![Figure 2. Problem of framework control composition](image)
As an example, we use the composition of the measurement system framework and the GUI framework. The measurement system framework has, from the moment a trigger enters the system, a well defined control loop that has to be performed in real-time and that creates a measurement item, read sensors, computes, activate actuators and stores the necessary historical data. The GUI framework has a similar thread of control, though not real-time, that updates the screen whenever a value in the system changes, e.g. of a sensor, or performs some action when the user invokes a command. These two control loops can easily collide with each other, potentially causing the measurement part to miss its real-time deadlines and causing the GUI to present incorrect data due to race conditions between the activities.

Solving this problem is considerably easier if the frameworks are supplied with source code that makes it possible to adapt the framework to handle this problem. However, the control loop in the framework may not be localised in a single entity. Often it is distributed over the framework code, which causes changes to the control to affect considerable parts of the framework.

In addition to merging the framework control loops by adapting the framework code, each framework can be assigned its own thread of control, leading to two or more independently executing control loops. Although this might work in some situations, one has to be aware of, at least, two important drawbacks. First, all application objects that can be accessed by both frameworks need to be extended with synchronisation code. Since classes often cannot be modularly extended with synchronisation code, it requires editing all reused classes that potentially are accessed by both processes. A second drawback is that often one framework needs to be informed about an event that occurred in the other framework because the event has application-wide relevance. This requires that the control loops of the frameworks become much more integrated than two concurrent threads.

3.2 Composition with legacy components

A framework presents a design for an application in a particular domain. Based on this design, the software engineer may construct a concrete application through extension of the framework, e.g. subclassing the framework classes. When the framework class only contains behaviour for internal framework functionality, but not the domain specific behaviour required for the application, the software engineer may want to include existing, legacy, classes in the application. It is, however, far from trivial to integrate the legacy class in the application, since frameworks often rely heavily on the subclassing mechanism. Since the legacy component will not be a subclass of the framework class, one runs into typing conflicts.

In order to illustrate the problem, the measurement system framework is used as an example. The framework handles the measurement cycle and provides for attaching a sensor, a measurement algorithm and an actuator by specifying the interface classes Sensor, CalculationStrategy and Actuator. Figure 3 shows the framework.

![Figure 3. The measurement system framework and associated legacy classes.](image)

Assume that we have a library of legacy components with different types of sensors, actuators and strategies for measurement and calculation, and that we have found suitable classes that we want to use in the three places in the framework’s interface. Our library sensor class is called TempSensor, and it happens to be a subclass of the class Sensor, which is an abstract class defined in our library. However, since the framework also defines a class Sensor, the usage of the library class will lead to conflicts. Even if the class name for the library superclass matches with that of the
framework interface class, the class TempSensor cannot be used, because these names do not designate the same class. An alternative approach, which is used in figure 3, is to make use of the Adapter design pattern [Gamma et al. 95] for class adaptation. This approach normally solves the problem, provided name clashes between class names in the framework and the legacy components. However, as we identified in [Bosch 97], the Adapter design pattern has some disadvantages associated with it. One disadvantage is that for every element of the interface that needs to be adapted, the software engineer has to define a method that forwards the call to the corresponding method in the legacy class. Moreover, in case of object adaptation, also those requests that otherwise would not have required adaptation have to be forwarded as well, due to the intermediate adapter object. This leads to considerable implementation overhead for the software engineer. In addition, the pattern suffers from the self problem and lacks expressiveness. Finally, since behaviour of the Adapter pattern is mixed with the domain related behaviour of the class, traceability is reduced.

The problems related to integration of legacy components in framework-based applications is studied in more detail in [Lundberg & Mattsson 96].

3.3 Framework Gap

Often when thinking about composition problems of components, the first thing that comes to mind are different kinds of overlap between the components, but there may also exist problems due to non-overlap between the components. This occurs when two frameworks are composed and the resulting structure still does not cover the application’s requirements. This problem is generally referred to as framework gap (see e.g [Sparks et al. 96]).

If the framework is a called framework, the framework gap problem may be solved with an additional framework interface both including the existing and additionally required functionality. In figure 4, such a ‘wrapping’ approach is illustrated graphically.

![Figure 4. Called framework extended to fill the framework gap](image)

In the case where a calling framework is lacking functionality, mediating software is needed to alleviate the problem. This mediating software is often tricky to develop since framework A has to be informed by framework B what had happened in framework B in terms understandable for framework A. The mediating software may also need to cut out parts of the functionality offered by the frameworks and replace these parts with application-specific code that composes the functionality from framework A and B and the functionality required for the framework gap. An additional problem is that since the mediating software becomes dependent of the current framework versions, this may lead to rather complex maintenance problems for the application when new versions of the framework.

3.4 Composition overlap of framework entities

When developing an application based on reusable components, it may occur that two (or more) frameworks both contain a representation, i.e. class, of the same real-world entity, but modelled from their respective perspectives. When composing the two frameworks, the two different representations should be integrated or composed since they both represent the same real-world entity. If the represented properties are mutually exclusive and do not influence each other, then the integration can be solved by multiple inheritance. However, very often these preconditions do not hold and there exist shared or depending properties between the representations, causing the composition of the representations to be more complex. We can identify at least three cases where alternative composition techniques are necessary. Below, these cases are discussed.

First, consider the situation where both framework classes represent a state property of the real world entity but the entity is represented in different ways. For example, a sensor in the fire-alarm framework contains a boolean state for its value and when composed with a measurement framework which has modelled the sensor using more complex representations such as temperature and pressure. When these two representations of a sensor are composed into an
integrated sensor, this requires every state update in one framework sensor class to be extended with the conversion and update code for the state in the other framework sensor class.

In the second case, assume that both framework classes represent a property \( p \) of the real world entity, but one framework class represents it as a state and the other framework class as a method. The value of the particular property \( p \) is indirectly computed by the method. For instance, an actuator is available both in the fire-alarm and measurement system frameworks. In an application, the software engineer may want to compose both actuator representations into a single entity. One actuator class may store as a state whether it is currently active or not, whereas the other class may indirectly deduce this from its other state variables. In the application representation, the property representation has to be solved in such a way that reused behaviour from both classes can deal with it.

In the third case the situation is that the execution of an operation in one framework class requires state changes in the other framework class. Using the example of the actuator mentioned earlier, an activate message to one actuator implementation may require that the active state of the other actuator class needs to be updated accordingly. When the software engineer combines the actuator classes from the two frameworks, this aspect of the composition has to be explicitly implemented in the glue code.

### 3.5 Composition of entity functionality

Sometimes a real-world entity’s functionality has to be modelled through composition with parts of functionality from different frameworks. Consider the case of a typical software structure with three layers, each represented by a framework, at the top we have a user interface layer, in the middle a application domain specific layer and at the bottom a persistence layer. Our real world entity is now represented in the application domain specific framework but some aspects of the entity has to be presented in the user interface layer and the entity also has to be made persistent for some kind of transactions etc. Just composing the respective classes from the three frameworks or using multiple inheritance will not result in the desired behaviour. For example, changes of the state caused by messages to the application domain specific part of the resulting object will not automatically affect the user interface and persistence functionality of the objects.

Thus, the software engineer is required to extend the application domain class with behaviour for notifying the user-interface and database classes, e.g. using the Observer design pattern [Gamma et al. 95]. One could argue that the application domain class should have been extended with such behaviour during design, but, as we mentioned earlier, most frameworks are not designed to be composed with other frameworks but to be extended with application specific code written specifically for the application at hand.

This problem occurred in the fire-alarm framework, where several entities had to be persistent and were stored in non-volatile memory, i.e. an EEPROM. To deal with this, each entity was implemented by two objects, i.e. one application object and one persistence object. These two objects were obviously tightly coupled and had frequent interactions, due to the fact that they both represented parts of one entity.

### 4 Underlaying Causes

The framework composition problems identified in the previous section can be related to a number of causes that underlie these problems. In this section, we discuss what we believe to be the four primary causes to these problems, i.e. framework cohesion, domain coverage of the framework, the design intention of the framework designer and lack of access to the source code. In the following, each cause will be described.

#### 4.1 Framework cohesion

The main issue in the framework usage phase is to develop an application through reusing an existing framework. The existing framework has to be extended or composed with some other software artifact to realize the requirements of the application under development. Three different software artifacts can be used for the adaptation and composition are new application-specific source code, class libraries and other frameworks.

Virtually every application developed based on a framework will require application-specific code to be developed and, since this code is designed to be composed with the framework, composition problems rarely occur. However, as described earlier, if a class library or a framework has to be composed with our framework, several composition prob-
lems may occur. But why is it so hard to compose these software artifacts with our framework? One of the answers to this question is framework cohesion. The functionality of a class in the framework can be divided into various types. The two types relevant for the discussion here are the domain-specific behaviour corresponding to the real-world entity that the class represents and the interaction behaviour to communicate to other framework entities for mutual updating. The latter type of functionality we refer to as cohesive behaviour, i.e. it establishes the cohesion between the entities in the framework. So, for a class from a class library or another framework to replace a class in the framework, this class not only has to represent the appropriate domain behaviour but also correct cohesive behaviour. Since the cohesive behaviour is very specific for the framework, it is rather unlikely that a separately developed class will fulfil these requirements.

### 4.2 Domain Coverage

An object-oriented framework provides an abstract design for an application in some problem domain. However, the framework need not cover the complete domain but may only contain classes for a subset of the relevant entities in the domain. To determine whether a framework covers the complete domain is a rather subjective activity since application domains are not defined very precise. For instance, the measurement system framework that we developed we considered to cover a complete application domain. During discussions with other software engineers, we found that they considered the framework to provide partial domain coverage since it only provided discrete measurement of items and did not incorporate continuous measurement required for, among others, production of fluid products.

When composing a framework with another framework, one may experience no domain overlap, little overlap or considerable overlap. No overlap between the frameworks will, obviously, not cause problems with respect to composition of overlapping entities, but it may be the source of a framework gap problem. Little domain overlap is often relatively easy to deal with since it only requires the adaptation of a few classes in both frameworks. Considerable overlap, however, may cause the reuse of one of the frameworks to be more effort consuming than writing the code from scratch. This problem is especially problematic for long lived applications since the frameworks on which an application is based often evolve over time, just as the application itself. Considerable redesign of a framework are then required to be repeated for every consecutive version of the application, if the application is to benefit from the improvements of later framework versions.

The solution for this problem is primarily a matter of standards. If an accepted definition of application domains and the boundaries between the domains would exist, framework designers could take these definitions into account and composition of frameworks would probably require less effort since most gaps and overlaps could be avoided. However, the domain definition is far from trivial. For instance, the statistical analysis framework described in section 2 provides user-interface functionality specifically for statistical representations, such as histograms and normal distributions. A general purpose GUI framework will most likely not have support for these representations, but the statistical graphical representations may be very complex to integrate with the GUI framework. When defining standard application boundaries, one has to decide where such boundary functionality should be assigned, which is a non-trivial task.

### 4.3 Design Intention

Several authors have identified that a reusable software artifact has to be designed for reuse by adaptation and composition. Both frameworks and class libraries are intentionally designed for reuse by adaptation but generally not by composition. Since class libraries generally have relatively low cohesion between the classes, it is often feasible to compose class library elements with existing software. Object-oriented frameworks, on the other hand, are generally designed for reuse through extension of the framework and not through composition.

One can identify two variants of framework composition, i.e. horizontal and vertical composition. Horizontal integration is the situation where the two frameworks that are to be composed can be found on the same layer in a software system. Vertical integration indicates the situation where one framework depends on the services of another framework, i.e. a so-called layered system. For example, the measurement system framework and the statistical analysis framework are composed horizontally, whereas these two frameworks are composed vertically with the user interface framework. However, independent of whether vertical and horizontal composition is used, an additional issue is whether there is one-way or two-way communication between the composed frameworks. Two-way communication often complicates the composition of the frameworks considerably.
Concluding, the design intentions for the framework should be defined explicitly to make it easier for the software engineer to decide on the composability of a framework for the application at hand. It should be made clear whether the framework can be reused by extension only or that provisions for composition are available. In addition, it should be made clear whether the framework is intended for two-way communication or one-way communication. Since the design intention for most existing frameworks, generally implicitly, is for reuse by extension and one-way communication from the framework to the newly written application-specific code, this is an cause for the framework composition problems that we have identified.

4.4 Access to Source Code

Another, seemingly trivial, cause for framework composition problems is the lack of access to the source code of the framework. This lack of access can be a practical limitation where the framework has been delivered to the reusing software engineer in the form of a library of compiled code in combination with a set of header files. It may also be a ‘conceptual’ lack of source code access due to the complexity of the code of the framework. The internal workings of the framework often are so complex, that it will require considerable efforts from the software engineer before he or she is able to make the changes necessary for composition purposes. Since research in software reuse has identified that the perceived effort required for reusing a component should be less than the perceived effort of developing the component from scratch, also the conceptual lack of access to the source code may form an important obstacle for framework reuse.

Access to the source code is important since framework composition, as identified earlier in the paper, may require editing of the framework code to add behaviour necessary for other frameworks. If no access to the source code is available, the only way to achieve the additional behaviour required from the framework is through wrappers encapsulating the framework. However, as identified by [Hölzle 93], wrapper solutions may cause problems such as significant amounts of additional code and serious performance problems. In addition, wrappers are unable to extend behaviour in response to intra-framework communication. For example, one framework object invoking another object in the framework may require an object in another part of the application to be notified. However, since the wrapper is unable to intercept this communication, it is not possible to achieve this.

5 From Problems to Causes to Solutions

In the previous sections, several framework composition problems and their primary causes have been identified. In this section, we start with relating the causes to the identified problems. Subsequently, the existing solution approaches for each problem are discussed and their limitations are identified.

<table>
<thead>
<tr>
<th></th>
<th>composition of frame-</th>
<th>composition with legacy</th>
<th>framework gap</th>
<th>overlap of entities</th>
<th>composition of entity functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohesive behaviour</td>
<td>complicating factor</td>
<td>primary cause</td>
<td>-</td>
<td>complicating factor</td>
<td>complicating factor</td>
</tr>
<tr>
<td>domain coverage</td>
<td>-</td>
<td>-</td>
<td>primary cause</td>
<td>primary cause</td>
<td>-</td>
</tr>
<tr>
<td>design intention</td>
<td>primary cause</td>
<td>complicating factor</td>
<td>-</td>
<td>complicating factor</td>
<td>primary cause</td>
</tr>
<tr>
<td>no source code access</td>
<td>complicating factor</td>
<td>complicating factor</td>
<td>complicating factor</td>
<td>complicating factor</td>
<td>complicating factor</td>
</tr>
</tbody>
</table>

Table 1. Framework composition problems and their causes

Table 1 relates the framework composition problems to the identified causes. Each cell describes whether the cause is the primary source of the problem or complicates the solution of the problem, that is whether it is a secondary cause. In the section below, these relations are explained in more detail and the available solution approaches are described.
5.1 Composition of Framework Control

The context for this problem is a situation where two frameworks are to be composed that both assume ownership of the main event loop of the application. When composing two such frameworks, the threads of control of both frameworks have to be composed in some way. The main cause for this problem is that both frameworks are intentionally designed for adaptation and not for composition which makes neither of the frameworks able to give up the control. Since the control loop often is deeply embedded in the source code of the framework, lack of access to the source code complicates resolving the problem considerably. Some changes to the control loop may even be impossible to achieve without changing the actual source code. For instance, events internal to the application at hand, for example for notification of another framework, cannot be intercepted by framework wrappers or adapters but require actually changing the framework code. An additional factor complicating the composition of calling frameworks is the cohesive behaviour inside the framework. The cohesive behaviour, since it indicates the behaviour of framework classes for updating the other framework classes, mixes the event behaviour with the domain behaviour of classes. Since the event loop becomes implicit through the cohesive behaviour, making changes to the event loop is more difficult due to the, potentially many, locations where code changes are required.

Integrating the framework control from the composed frameworks requires changes to the event loops in the involved frameworks. This can, among others, be achieved by three solutions, as discussed in the points below.

- **Concurrency**: One possible solution is to give each framework its own thread of control rather than merging the control loops. This solution approach can only be used when there is no need for the frameworks or the application code to be notified about events from the other frameworks. A disadvantage of this approach is that the application specific objects that potentially are accessed by more than one framework need to be extended with synchronization code. An advantage is that no access to the source code is required.

- **Wrapping**: A second solution is to encapsulate each framework by a wrapper intercepting all messages sent to and by the framework. The wrapper, upon receipt of relevant messages, can notify interested entities in the framework. This allows some level of integration of event handling in the framework-based application, but only based on events external to the framework. Internal framework events are not intercepted and can consequently not be dealt with.

- **Remove and rewrite**: If external notification of internal framework events is necessary in the application, a solution is to remove the relevant parts of the control loops from the frameworks and to write an application specific control loop that satisfies the needs from the involved frameworks and the application. This requires access to the source code and it may be difficult since the control loop often is not centralised to one framework entity but spread out in the framework, i.e. the cohesive behaviour of the framework complicate the solution. This solution generally is more effort consuming approach than the previously indicated approaches.

5.2 Composition with Legacy Components

The software engineer may want to use legacy components in combination with the framework when a framework class does not contain the correct application-specific domain behaviour for the actual application. In addition, a legacy library of classes with the required application-specific behaviour is available that fulfils the application’s requirements. The integration between a framework and legacy components is not as easy as one may expect due to the cohesive behaviour of the framework which make it difficult to replace a framework class with a legacy class and due to potential typing conflicts in the programming language. A complicating factor in solving the composition is that the framework is designed for adaptation and extension and not for composition. One approach is to design the framework for composition by means of the role concept. Obviously, access to the source code of the framework simplifies the composition, since one can change the framework to refer to the legacy class rather than a framework class. Some solutions to the problem of composing legacy components with a framework are discussed below:

- **Adapter design pattern**: Composing a legacy component with a framework is almost a schoolbook example of a situation for which the Adapter design pattern has been designed. Provided that no name clashes occur, this solution is very suitable, but it results in some implementation overhead as for each operation in the class interface that has to be adapted there must be methods that forward the request to the equivalent method in the legacy class. Additional drawbacks are that the Adapter patterns suffers from the self problem and traceability is reduced since pattern-specific behaviour is mixed with domain-specific behaviour for the class [Bosch 97].
• **Change framework**: Provided the software engineer has access to the framework’s source code, one possible solution is change those parts of the framework that refer to the framework class that should be replaced by the legacy class.

• **Roles**: As described in [Lundberg & Mattsson 96], a solution dealing with the problem once and for all is to represent the framework’s reuse interface as a set of roles that has to be played by objects that fulfil the requirements for the roles. Using this solution, it is possible to fill the roles either with classes related to the framework or existing legacy classes, provided that the class fulfils the requirements. This, however, requires the framework to be designed based on this approach.

### 5.3 Framework Gap

The framework gap problem occurs when two (or more) frameworks have to be composed to satisfy the application’s requirements, but the composition does not cover the requirements completely. Thus, we have a gap of missing functionality. The main cause for this problem is that the frameworks do not cover their respective domains sufficiently. As described earlier in the paper, the definition of an application domain is still rather ad-hoc and this may lead to domain overlap or domain gaps when composing frameworks. Also for this problem, the lack of access to source code may complicate the solution to a framework considerably. Anyway, in either case, one can identify a number of solution approaches for closing the framework gap.

• **Wrapping**: As described earlier in the paper (see e.g. figure 4), one can extend the framework interface of a called framework with the missing functionality through an additional API. The wrapping entity, in this case, aggregates the original framework and the extension for closing the gap, but provides a uniform interface so that clients are unaware of the internal structure.

• **Mediating software**: For a calling framework, the software engineer has to develop some kind of mediating software that manages the interactions between the two frameworks. The mediating software not only manages the interaction, but also contains the functionality required for closing the framework gap. A potential disadvantages are that it has proven to be rather difficult to develop mediating components and that the software becomes vulnerable for future changes of framework versions and thus is expensive to maintain.

• **Redesign and extend**: If one has access to the source code of the framework and intends to reuse the framework also in future applications, the most fruitful approach may be to redesign the framework and extend the source code to achieve a better domain coverage and thus bridge the gap.

### 5.4 Composition overlap of Framework Entities

When two frameworks represent the same real-word entity, from their respective perspectives, with own representations and these frameworks need to be composed, the two representations of the real-world entity have to be composed as well. This constitutes the problem of overlap of framework entities. The main cause for this problem is that part of the application domain is covered by both frameworks, since the real-world entity has been modelled from different perspectives. The composition of the two framework classes is often complicated due to the cohesive behaviour of the classes since, after the composition, actions in one class may need to notify the other framework class. We present three solution approaches to the problem:

• **Multiple inheritance**: An obvious solution to this problem is to use multiple inheritance, but this requires that the properties of the classes do not affect each other and are mutually exclusive, which is seldom the case. The main issue is to provide a solution that takes care of necessary updates and conversions between different representations provided by the two frameworks.

• **Aggregation**: An alternative solution is to use aggregation instead of inheritance which will result in an aggregate class which has each framework representation as parts. The aggregate class is the application’s representation of the real-world entity. However, this approach requires that the source code is available for the frameworks since the involved frameworks have to be changed so that every location in the framework that references the framework-specific representation of the real-world entity is replaced by a reference to the aggregate class. In addition, it is necessary that the interface of the aggregate class contains the framework class interface for both frameworks and the necessary conversion and update operations. The implementation of the aggregate class has, in appropriate
methods and places, to include the necessary conversion and update behaviour. The major drawback of this solution is that both frameworks has to be changed and that a considerable amount of implementation overhead is required.

- **Subclassing & aggregation**: Another solution that does not require changes in the frameworks are to subclass each framework class that represent the real-world entity. Both subclasses are involved in a bidirectional association relationship to enable update and conversion behaviour. In addition, each subclass has to override the operations in the superclass. In the subclass implementation of relevant operations, there are calls to the corresponding subclass for conversion and updates and, in addition, the operations call the overridden operations in the superclass. A limitation of this solution is that the application's representation of the real-world entity is partitioned over two classes. To overcome this limitation we may introduce an additional aggregate class which has the two subclasses as parts. The aggregate class solution also implies that the application specific part of the code has to work with the aggregate class (to reach the part objects) and that there will be communication between the two subclass objects that will not pass the aggregate object.

### 5.5 Composition of Entity Functionality

This problem occurs when a real-world entity has to be modelled through composition of parts of functionality from different frameworks, e.g composing application domain-specific functionality with persistence functionality. This means that the real-world entity is represented in an application domain-specific framework and has to be composed with functionality from a persistence framework. The cause for this composition problem of functionality is that the frameworks are designed for extension and thus it is not easy to add additional functionality. The cohesive behaviour of a framework complicates the composition of functionality since it makes it difficult to break up the existing collaborations inside the framework and add the necessary functionality. Below, aggregation and multiple inheritance and the use of the observer design pattern are discussed as possible solutions.

- **Aggregation or multiple inheritance**: The simplest solution to this problem is to use multiple inheritance or aggregation to compose the framework classes with the required functionality into a single class. The problem with these solutions is that state changes caused by messages to the application domains specific part of the composed object will not automatically influence the persistence functionality part of the object. One way to solve this is to add update behaviour in appropriate places in the frameworks and the composed class but this is a rather complex action to perform.

- **Observer design pattern**: To overcome the problem with the previous solution the application domain specific class can be extended with notification behaviour, e.g through applying the observer design pattern [Gamma et al. 95]. This may seem as an obvious solution and thus it should have been included in the application domain specific framework from the beginning. But, as often is the case, the framework is originally designed for extension and not for composition.

### 6 Conclusion

Object-oriented frameworks provide an important step forward in software reuse since frameworks allow reuse of large components instead of the traditional small, building block like components. Initially, frameworks were primarily used as the only reused entity in application development and the application was constructed by extending the framework with application specific behaviour. During recent years, among others due to the increased availability of frameworks for various domains, one can identify a development towards the use of multiple frameworks in application development. These frameworks have to be composed to fulfil the application’s requirements which may lead to composition problems.

In this paper, we have studied the framework composition problems that may occur during application development, their main causes, the solutions available to the software engineer and the limitations of these solutions. The composition problems that we identified are related to the composition of framework control, the composition of the framework with legacy components, the framework gap, the composition overlap of framework entities and the composition of entity functionality. These problems are primarily caused by the cohesion of involved frameworks, the accuracy of the domain coverage of frameworks, the design intentions of the framework developer and the lack of access to the source code of the framework. The identified problems and their causes are, up to some extent, addressed by existing solutions, such as the adapter design pattern, wrapping or mediating software, but these solutions generally have limitations and either do not solve the problem completely or require considerable implementation efforts.
To the best of our knowledge, no work exists that covers framework composition problems as a primary topic of study, but several authors have identified some composition problems. For instance, [Pyarali et al. 96] suggest not to assume ownership of the event loop during framework design and to stress flexibility of the design. [Sparks et al. 96] identified the framework gap problem and suggest ways of dealing with them. Also, [Garlan et al. 95] discuss the notion of architectural mismatch that, although not specific to object-oriented frameworks, may also affect framework composition. Finally, [Hölzle 93] discusses the problems of using wrappers to adapt components for use in applications.

The contribution of this paper, we believe, is twofold. It provides software engineers using object-oriented frameworks for application development with an understanding of the problems they may experience during framework-based application development, as well as an understanding of the primary causes and solution approaches. Secondly, it provides researchers in object-oriented software engineering with topics that need to be addressed by future research.

References


[Gamma et al. 95]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.


