Wrapping a Real-time Operating System with an OSEK Compliant Interface — a Feasibility Study

Joachim Denil*,†, Serge Demeyer†, Paul De Meulenaere*, Kris Vanstechelman‡ and Kurt Maudens‡

*TERA-Labs
Karel de Grote University College, Antwerp, Belgium
Email: Joachim.Denil@kdg.be, Paul.DeMeulenaere@kdg.be
†Lab On REengineering
University of Antwerp, Antwerp, Belgium
Email: Serge.Demeyer@ua.ac.be
‡Spicer Off-highway Products Division Belgium
Dana Corporation, Brugge, Belgium
Email: Kris.Vanstechelman@dana.com, Kurt.Maudens@dana.com

Abstract—The drive towards standardization in the automotive sector puts a lot of pressure on software suppliers to comply with standards such as OSEK and AUTOSAR. However, many of these suppliers have a vested interest in proprietary software and are seeking ways to migrate their existing code-base to comply to these standards. This paper reports on a feasibility study to wrap a proprietary real-time operating system with an OSEK compliant interface. Besides investigating whether this is feasible, we also assess the performance impact in terms of computation time and memory consumption, as this is critically important for real-time systems. As such, we evaluate the typical trade-offs one has to make when adopting an incremental migration strategy towards a standard compliant interface.

I. INTRODUCTION

With the introduction of the AUTOSAR standard1, many automotive companies insist that their suppliers use standard components like an OSEK compliant operating system in their products. OSEK (Open Systems and their corresponding interfaces for automotive electronics) is a joint project of the German and French automotive industry that defines an open software architecture for automotive control units in vehicles. According to a study by Schoof and Wybo [1], the migration from OSEK to AUTOSAR is easier then directly adopting AUTOSAR. Gaining knowledge and expertise with OSEK helps understanding the philosophy behind AUTOSAR.

Despite the success of OSEK, a lot of suppliers are still using their own proprietary real-time operating system [2]. The “not invented here syndrome” certainly plays a role in this choice, however suppliers have often good reasons not to adopt 3rd party components. First of all because they have a lot more experience with their proprietary software; secondly because the internal design process and tool-chain is often highly dependent on the proprietary operating system and thirdly because the real-time operating system is tuned and optimized for the underlying hardware platform. This implies that switching to 3rd party components comes with a great risk and an considerable cost, hence suppliers are reluctant to introduce a new OSEK compliant operating system in their software.

A possible way out of such a catch-22 situation is to wrap the proprietary software with an interface that complies to the standard specification. Such a wrapper (also known as a wrapper-facade [3] or adapter [4]) is —when feasible— indeed a cheap way to integrate an existing component in a system that expects a different interface. Moreover, it allows for an incremental migration strategy (the wrapped software component can later be replaced with another component that adheres to the interface), which reduces risk. However, introducing a wrapper also has a performance impact as it triggers extra processor cycles and consumes more memory. Performance is a crucial aspect of a real-time operating system, thus must be assessed carefully.

To investigate this trade-off, we conducted a feasibility study adapting a proprietary real-time operating system from an off-highway company to the current OSEK OS specifications. Our study was guided by the following two research questions:

- RQ1 – Is it feasible? What are the technical implications when building such a wrapper?
- RQ2 – What is the performance impact? Can we quantify the extra computation time and memory consumption caused by introducing the wrapper?

As part of this feasibility study, we must show that the wrapped operating system is indeed a valid OSEK implementation. In our study, we have used the MODISTARC specifications [5], which are designed to test and assure that an implementation conforms to the OSEK specification.

The paper is organized as follows. First we list other approaches aiming to provide OSEK compliant operating systems (section II). Then our working method is explained

1http://www.autosar.org/ — AUTomotive Open System ARchitecture
how a real time operating system can be migrated to an OSEK compliant operating system (section III). Next, we demonstrate that it is indeed feasible to build an OSEK compliant wrapper interface (RQ1) with a proof by example in the form of a software prototype a single developer created in the span of less than 3 months (section IV). Afterwards, we assess the performance impact in terms of computation time and memory consumption (section V). Finally, we list some suggestions for improvement (section VI) before arriving at a conclusion (section VIII).

II. RELATED WORK

Since the release of the OSEK standard, a lot of commercial and academic implementations have been realized. One academic initiative is Trampoline [2]. It offers a full OSEK operating system, communication and network management layer for several hardware platforms. A configuration tool is provided to generate an optimal trampoline kernel. Another academic initiative is the EMARALDS-OSEK OS [6]. It uses several memory and performance optimizations for efficiency. Implementation details of both these operating systems were used during the case study.

Some vendors offer an OSEK compliant wrapper for their own commercial RTOS. An example of this is the Micrium C/OS-II OSEK extension layer [7] that provides a certified OSEK wrapper for the µC/OS-II kernel.

Other vendors offer an RTOS independent API to develop applications. When applications are developed with this 'neutral API', a wrapper has to be written to abstract the real-time operating system. Examples of this are the MapuSoft OS abstractor [8] and the Telelogic Rhapsody Object Execution Framework [9].

Though to the best of our knowledge, currently few examples of wrappers for proprietary operating systems and their impact on performance are documented in the literature.

With the arrival of AUTOSAR, OSEK is becoming more important since AUTOSAR embeds an OSEK compliant OS in its specifications. The AUTOSAR OS specification [10] defines an OSAL (Operating System Abstraction Layer) when an other operating system is used. The interface of the OSAL is exactly the same as the interface of AUTOSAR OS.

III. THE PROTOTYPE

OSEK provides a standard software architecture for distributed control units in vehicles. It meets two stringent automotive requirements: real-time support and small memory footprints. OSEK actually consists of a set of standards: (a) OSEK OS specifies the behavior and APIs for a real-time operating system; (b) OSEK COM describes an interface for transferring data between applications (through local communication or by use of a network); (c) OSEK NM provides system wide management functions; and (d) OSEK OIL proposes a language to configure the OSEK system. In the scope of this feasibility study, we restrict ourselves to the operating system because that was the primary focus for the off-highway company requesting the feasibility study.

The adaptation of the proprietary real-time operating system to OSEK is done by means of an OSEK-wrapper that is used to implement OSEK behavior on top of the RTOS. Figure 1 shows the software architecture.

![Software architecture of the control unit](image)

This architecture has a big advantage. The whole system behaves like an OSEK implementation without any significant changes to the proprietary operating system. This construction allows to create a valid OSEK implementation, as the MODISTARC specification says:

*All what behaves like OSEK is OSEK* [11]

But even within OSEK, some different options can be taken: OSEK OS defines four different conformance classes based on the required number of tasks per priority and on the required number of times that one task can be requested to activate. The option which conformance class to select, depends on the requirements that are imposed by the applications built on top of the OSEK OS. The selection of the conformance class is therefore a prerequisite that will influence the detailed migration strategy of the proprietary operating system.

OSEK OS defines following conformance classes:
- **BCC1** (Basic Conformance class 1): maximum one basic task can be assigned per priority, and there can be only one simultaneous request to activate the task;
- **BCC2** (Basic Conformance class 2): multiple basic tasks per priority are allowed, and there can be multiple simultaneous requests to activate the task;
- **ECC1** (Extended Conformance class 1): maximum one basic or extended task can be assigned per priority, and there can be only one simultaneous request to activate the task;
- **ECC2** (Extended Conformance class 2): multiple basic and extended tasks per priority are allowed, and there can be multiple simultaneous requests to activate the task;

The difference between basic tasks and extended tasks will be explained in section IV.

The off-highway company that requested for the current migration feasibility study, requires the ECC1 conformance class. Therefore, this paper is limited to the ECC1-migration only.
We split up the total OS-migration into following isolated subparts: (a) migration of the task model, (b) migration of the scheduler, (c) migration of the event management, (d) migration of the resource management, (e) migration of the interrupt services and (f) migration of the alarm functionality.

For each of these subparts of the migration, we compare the detailed mechanisms from the proprietary operating system to the OSEK-specifications. If the differences are big, the OSEK-wrapper often needs to keep state-diagrams or context variables to translate between both OS’s. If on the other hand the differences are small, the OSEK wrapper can often translate between the interfaces without the need to keep context variables.

Besides these mechanisms, other adjustments must be taken care of. The most important adjustments are the return values of the system calls. OSEK defines 2 types of error checking by means of these return values, the standard and extended. The extended version is used to support debugging of the systems, while the standard version fulfills the requirements of a debugged application. The easiest way is to implement the extended version and use a preprocessor macro to exclude the unnecessary code for the standard version.

IV. IS IT FEASIBLE?

A. Migration of the task model

Requirements: Depending on the chosen conformance class, an OSEK compliant operating system must allow for two sorts of tasks: the basic task and the extended task.

The basic task is the simplest type of task, switching between (a) the READY state (indicating that the task is ready to run); (b) the RUNNING state (indicating that the task is currently being executed) and (c) the SUSPENDED state (indicating that a task is terminated and can be restarted later from the beginning of the task).

The extended task adds an extra WAITING state, used when a task is paused until it is released to the READY state by means of a system call.

The state-machines specifying the legal sequences of state transitions for both the basic and extended tasks are shown in figure 2. The Start and Preemption transitions are the responsibility of the scheduler. The others are available as system calls in the OSEK operating system.

Wrapper implications: If the state machine of the proprietary operating system does not match with the state machine of the OSEK tasks, the OSEK-wrapper has to adapt the state-machine of the proprietary operating system to the OSEK state machine. The wrapper should: (a) introduce and/or exclude states that don’t match the OSEK model and implement the state transitions to and from these states. Though implementing a WAITING state in the wrapper, when no WAITING state is available in the proprietary operating system seems unlikely. If this is the case, the wrapper can still be targeted to a basic conformance class; (b) record and synchronize the state of the defined tasks in the wrapper; (c) implement and translate the interface for this new behavior.

If on the other hand a one-to-one match between the OSEK state machine and the state machine of the proprietary operating system is possible, only the interface for manipulating the task state has to be translated.

Prototype implementation: The task model of the proprietary operating system in our case study doesn’t match the model of tasks in the OSEK OS. On the one hand it has simple tasks that don’t have a WAITING state available. The proprietary operating system activates these tasks automatically in a time triggered way. On the other hand there are tasks that allow a WAITING state but run in a continuous loop; (they don’t have a SUSPENDED state), see figure 3.

It is clear that the task model of the proprietary operating system does not match the OSEK task model. Using the continuous running type of proprietary OS task (shown in figure 3), it is possible to create both the OSEK basic and extended task types. A task-structure is used to record and synchronize the states in the wrapper. The WAITING state in the proprietary operating system is used as the SUSPENDED and WAITING state in the OSEK-wrapped operating system. (a) When the call to terminate the task is called in the OSEK-wrapped operating system, the stack pointer and program counter of the task are reset to their initial values and the task in the proprietary operating system is put in a WAITING state. (b) Activating the task in the OSEK-wrapped operating system ...
system causes the transition from WAITING to READY in the proprietary operating system.

B. Migration of the scheduler

Requirements: The scheduler of an OSEK operating system must obey to a fixed priority scheme. Therefore, all tasks are assigned a fixed priority at system configuration. Tasks that are in the READY state are given time to execute on the processor. The higher priority tasks are processed before the lower priority tasks. Tasks of the same priority are allowed in the BCC2 and ECC2 conformance classes and are served on a first-come-first-serve basis.

Multiple activation requests can be allowed per task. This means that when a task is active (READY, RUNNING or WAITING) and an activation of the task occurs, this additional activation is recorded. When terminating a task, OSEK OS evaluates the number of activation requests and if necessary reactivates the task immediately.

Wrapper implications: If the scheduler of the proprietary operating system is not a static priority scheduler, it is unlikely that it can be adapted in the wrapper to schedule the tasks in a fixed priority scheme. If the proprietary operating system, allows tasks of the same priority, a conformance class of type 2 can be achieved.

Prototype implementation: The scheduler of the proprietary operating system is a fixed priority scheduler. No adaptations are required. Moreover, since in this study only conformance class ECC1 is required, it is not needed to implement to implement multiple activation requests per task.

C. Migration of event management

Requirements: The event-architecture is the primary synchronization mechanism in the OSEK operating system. Events are only available to extended tasks and are used to initiate the transitions to and from the WAITING state. Events are not independent objects but are assigned to extended tasks. If the required conformance class of the OSEK-wrapped operating system aims at an extended conformance class, the event mechanism must be available.

Wrapper implications: The proprietary operating system should also feature some kind of event-architecture, otherwise it is most unlikely that a wrapper interface can be constructed. If such an event mechanism is present, the wrapper should (a) map the event data structure from the wrapped operating system to the OSEK task records; (b) translate the interface manipulating the events. If there is no event-architecture available, the OSEK-wrapped operating system can be targeted to a basic conformance class.

Prototype implementation: The operating system in our case study has a mechanism to put tasks in and out of the WAITING state. By multiplexing and demultiplexing the calls, the OSEK event mechanism is created in the OSEK-wrapper. Therefore two variables were added to the task structure of the extended tasks in the wrapper. (a) One for the events that have been set for the task and (b) the other for the events that the task is waiting for. When in the OSEK-wrapped operating system a task is to be transferred to the READY state by means of an event, the correct event flag (a single bit in the variable) is set while the waiting flag is cleared. The system call to put the task in the READY state is called in the proprietary operating system. The opposite happens when waiting for an event.

D. Migration of resource management

Requirements: Protection of shared resources is done by the resource mechanism. This mechanism must work according to the OSEK-PCP protocol (priority ceiling protocol). The OSEK-PCP protocol is a variant of the original priority ceiling protocol presented in [12]. When a task acquires a resource, the priority of the task is temporarily raised to the priority of the highest priority task that will ever use the resource. This is called the ceiling priority of the resource. This mechanism ensures that, while a task occupies a certain resource, other tasks that share the same resource cannot get the CPU. This way, a deadlock can never occur.

Wrapper implications: If the proprietary operating system has a resource protection mechanism available with a priority ceiling protocol, the wrapper should translate the interface that manipulates these resources. Otherwise the resource mechanism must be constructed within the wrapper.

Prototype implementation: The operating system in our study has a priority ceiling protocol available on its resource mechanism. The interface was translated in the wrapper, so it complies to the OSEK interface.

However, if the resource service does not comply with the OSEK priority ceiling protocol or even if no resource service is available, the mechanism can be built into the OSEK-wrapper. The method is to delay the activation of tasks (i.e. the transition
of a task from SUSPENDED to READY, or from WAITING to READY) until the resource is released.

To illustrate the concrete mechanism, we consider a running task $T_1$ with a priority $Prio(T_1)$ and its resource $r_1$ with a ceiling priority of $CP(r_1)$ (figure 4). Task $T_2$ has priority $Prio(T_2)$ that is larger than $CP(r_1)$. At a certain moment in time, $T_2$ wants to activate the task $T_1$ that possibly also needs the resource $r_1$, and therefore has lower priority than $CP(r_1)$. This activation of $T_1$ may cause a concurrent access to $r_1$.

The naive solution is to delay tasks activations in the running task $T_1$ when a resource $r_1$ is taken, for all tasks $T_I$ with a priority $Prio(TA)$:

$$Prio(T_1) < Prio(TA) < CP(r_1)$$

until the resource $r_1$ is released.

Though this solutions works in most cases, it may run into trouble. In figure 4.a this is explained. The higher priority task $T_2$ preempts the task $T_1$ and activates the task $T_A$ with a priority complying to: $Prio(T_1) < Prio(TA) < CP(r_1)$. The activation of task $T_A$ by $T_2$ is in contradiction with the mandatory delay of $T_A$ imposed by $T_1$.

![Fig. 4. (a) Misbehavior in the naive solution. (b) Correct behavior of OSEK-PCP protocol](image)

To solve this problem, a priority-queue for the delayed tasks and a stack is used.

- When taking a resource $r_1$ in task $T_1$, a taskpointer to $T_1$ is pushed on the stack.
- The actual priority of the task is changed to the ceiling priority of the taken resource $CP(r_1)$.
- At Activate (TA), the priority $Prio(TA)$ of the task $T_A$ is tested. If $Prio(TA)$ is in the range:

$$\text{RealPrio}(T_1) < Prio(TA) < \text{CurrentPrio}(T_1)$$

with $\forall T_1 : T_1 \in stack$, then the task $T_A$ is put in the priority queue. In the above definition, $\text{RealPrio}(T_1)$ is the statically assigned priority of the task in the proprietary operating system and $\text{CurrentPrio}(T_1)$ is the actual priority of the task in the OSEK-wrapped operating system (that is lifted to a ceiling priority when a resource is taken.) If on the other hand $Prio(TA)$ is not within the specified range, the task is simply activated in the proprietary operating system.

- Releasing the resource $r_1$ causes the stack to be popped and the priority lowered to the normal priority of the task. Then all tasks with a priority greater then $Prio(T_1)$ are released from the priority-queue and activated.

This method works when only one resource can be taken by a task. The solution can be extended to include more resources for a task.

### E. Migration of Interrupt services

**Requirements:** OSEK defines that a separate stack must be used for interrupt service routines (ISR) that use OSEK system calls. The separate stack is introduced for reducing the memory requirements of the OSEK system. If this stack isn’t available, all run-time stacks of the tasks should be big enough to allow storage of multiple ISR-frames (context that is saved before executing an ISR). Besides the separate stack, an interface is defined to allow the manipulation of interrupt flags in the hardware.

**Wrapper implications:** When the proprietary operating system doesn’t provide a separate stack for handling interrupts, a stack in the OSEK-wrapped can be used. Implementation pointers for a stack in the wrapper can be found in [6]. The interface for manipulating the interrupt flags has to be provided if these aren’t available in the proprietary operating system. Else, the calls can be translated to comply with the OSEK interface.

**Prototype implementation:** Implementing these interrupt services is quite hardware dependent but does not impose any real difficulties, hence are not further discussed in this paper.

### F. Migration of the alarm functionality

Alarm functionality was excluded in this study. Though implementation pointers for data-structures and functionality of alarms can be found in [2].

### G. Validation

When part of the OSEK-wrapper has been written, it can be tested using the MODISTARC specifications [5]. The MODISTARC test suite is also used to assure that the whole system is an OSEK compliant system. Though some remarks have to be given. The test procedure for OSEK OS was written for version 2.0 of the OSEK specifications. Category 3 interrupts are removed in the current specification of OSEK OS but are still used within the test procedure. Interrupts of category 3 are comparable to interrupts of category 2, so they have to be replaced by an ISR of category 2.
Another remark that has to be made is on the nature of the test procedure. The test procedure is a black-box test. Because of this, no validation is done on the verification of the separate stack that is used by the interrupt service routines.

Some care has to be taken when using the specifications to test the resource mechanisms implemented in the OSEK-wrapper. No scenario as in figure 4 is described in the tests.

By means of the MODISTARC specifications the proprietary operating system with the OSEK-wrapper elaborated in this paper, was verified to be an ECC1 compliant OSEK implementation.

V. PERFORMANCE IMPACT

Before drawing any conclusions about the OSEK-wrapper in terms of functionality, some non-functional aspects have to be measured. OSEK is designed to run on small embedded systems, so speed and memory consumption are key issues in a real-time operating system. Introducing the OSEK-wrapper around the operating system causes overhead. In this section the overhead of the OSEK-wrapper is compared to the proprietary RTOS. Measurements were executed on both the extended (ext) and standard (std) error checking OSEK-wrapper.

A. Speed comparison

Task model overhead: An important factor in the performance overhead imposed by the wrapper is the extra time required to perform the context switching when a task finishes. To assess the relative influence of this switching overhead, we compare the execution time of a typical configuration of tasks executing on the bare proprietary operating system against the same configuration on the wrapped version. Therefore two tasks are scheduled to obtain the execution times: one, the highest priority task records the current time and then terminates. The other, the lowest priority task calculates the difference in time. Extra care has been taken so that the recording and output of the time does not influence the measurements. Tasks with intermediate priorities, that only terminate, are added to see the impact of the number of tasks.

In figure 5 the results are shown. The context switching with the OSEK-wrapper is 3.8 times slower than in the original case due to the reset of the program counter and stack pointer, putting the task in a WAITING state and keeping the OSEK-wrapper in a consistent state. Though the overhead introduced is linear.

This was the only overhead measured for manipulating the task model. Activation of tasks by means of a system call wasn’t available in the proprietary operating system and thus not measured.

Event mechanism overhead: A second important measurement, is the relative impact of the adapted event mechanism. Therefore two tasks are defined. The highest priority task waits for an event, while the lowest priority task sets this event. This functionality is executed in a loop that lasts for a certain amount of time. When a wait-release cycle is completed, a counter is incremented. To assess the influence of the wrapper, we compare the number of wait-release cycles in the proprietary operating system and the OSEK-wrapped operating system.

Figure 6 shows the comparison of the event mechanism. Speed of the event mechanism is decreased by 37% in the standard error checking case. This is the overhead caused by the added functionality.

Resource mechanism overhead: A final measurement is the impact of the wrapper on the resource mechanism. The measurement was based on the method proposed in [13].
a single task is used. This task gets and releases a resource in a loop that lasts for a certain amount of time. When a get-release cycle is completed, a counter is incremented. The difference in get-release cycles between the proprietary operating system and the OSEK-wrapped operating system is an indication of the performance loss introduced by the wrapper.

The measurements were done on the OSEK-wrapped operating system where the priority ceiling protocol was translated in the wrapper with both (a) an extended error checking mechanism, (b) a standard error check. And on the OSEK-wrapped operating system, where the priority ceiling protocol was implemented in the wrapper by delaying the task activations, also on both (c) extended error checking and (d) standard error checking.

Another important aspect of the OSEK-wrapper is the memory consumption. RAM in the wrapper and RTOS is removed. This is done for both the proprietary RTOS and the OSEK-wrapped RTOS. RAM consumption is quantified similar. The code is compiled with 0 to 5 defined tasks and 0 to 5 defined resources for both the proprietary RTOS and the OSEK-wrapped RTOS.

In the case study, the code size increased by 30% in the extended error checking case. For the standard error checking only an increase of 21% is measured.

The RAM-usage increased by less than 1% without the resources necessary for each task structure and resource structure needed for the bookkeeping in the OSEK-wrapper.

VI. FUTURE WORK

The presented work shows that it is feasible to design and implement an OSEK-wrapper for a proprietary operating system. The OSEK-wrapper can be used as is, but does have some drawbacks. First and foremost it’s not optimal. The OSEK-wrapper must be recoded for every new application. Tasks, events and resources must be set for the application. The solution to this issue is the automatic generation of the OSEK-wrapper. In the OSEK operating system this is done using the OSEK Implementation Language or OIL. Though, one could also use the AUTOSAR XML variant [15]. Because of the static configuration that OSEK requires, the code can be generated optimally.

Another problem is the redundant information in the OSEK-wrapper. Information about tasks, events and resources is stored in the OSEK-wrapper and the underlying RTOS. This can be solved by gradually moving specific OSEK code into the proprietary RTOS code. This could increase performance and decrease memory consumption.

VII. THREATS TO VALIDITY

This section discusses the threats to the validity of our approach and results. We consider internal and external validity.

A. Internal validity

The migration of the proprietary RTOS is highly dependent on the developer who conducted the case-study. Moreover, knowledge of the proprietary RTOS is key to the success of the migration. In this case study, the proprietary RTOS is treated as a black-box, the developer had no insights into the internals of the RTOS. This could affect the performance and outcome of the case-study since other constructs might have been possible with less effort and better performance characteristics.

To avoid any faults with the measurements executed for the performance impact, all compiler optimizations were turned off.

B. External validity

Though this study is based on one case study, the methods proposed can be generalized to other proprietary real-time operating systems, since most RTOSs have a similar functionality. The measured performance impacts are highly dependent on the case-study and more experiments are necessary to have a rough estimation of the impact when starting a migration project.
VIII. CONCLUSION

In this paper we have demonstrated that it is feasible to build an OSEK compliant wrapper for a proprietary real-time operating system. No modifications to the hardware, the original code-base nor the tools are necessary with this approach. We documented the steps and necessary implications a software designer must take into account in order to construct such an OSEK-compliant wrapper.

The OSEK-wrapper implies a significant increase in computation time and consumes a considerable amount of extra memory. This is caused by defining redundant information for tasks, events and resources in the wrapper, necessary to synchronize with the information in the proprietary real-time operating system. In this particular case, this overhead was acceptable, but this decision depends of course on the particular services built on top of the operating system. Doing the experiment however, we identified some opportunities for improvement, so we think one can reduce the performance penalties if necessary.

Consequently, we conclude that —when faced with the requirement of adopting an OSEK compliant operating system—the construction of a wrapper is a viable alternative compared to integrating 3rd party components. The wrapper indeed causes a certain performance penalty, but is quite cheap to implement and reduces the inherent risks and costs associated with adopting 3rd party components.

REFERENCES

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.