

In real-time systems, the transformation from a set of requirements to the implementation is a difficult process that is not covered well by existing methods. All too often, guesswork is involved when determining system constraints. The author proposes an approach based on extensions to the Unified Modelling Language, and provides examples of how they can be used.

# Systems development – from conception to delivery

**FRANÇAIS**  
 Développement de système - de la conception à la livraison  
 Dans les systèmes en temps réel, la transformation d'un ensemble d'exigences jusqu'à la mise en oeuvre est un processus difficile qui n'est pas très bien couvert par les méthodes existantes. Trop souvent, la détermination des contraintes du système est un jeu de devinette. L'auteur propose une approche basée sur des extensions du Unified Modelling Language, et donne quelques exemples sur la façon dont elles peuvent être utilisées.

**DEUTSCH**  
 Systementwicklung – vom Konzept bis zur Auslieferung  
 Bei Echtzeitsystemen ist der Werdegang von einem Satz Anforderungen zur Implementation häufig ein schwieriger Prozeß, der von bisherigen Methoden nicht besonders gut abgedeckt wird. Zu häufig muß bei der Festlegung der Systemgrenzen geraten werden. Der Autor suggeriert einen Ansatz, der auf einer Erweiterung der Unified Modelling Language [Einheitliche Modellierungssprache] beruht. Der Autor bietet ebenfalls Anwendungsbeispiele an.

System development is a tricky business. Systems are implemented to meet the needs of humans and humans, often those who need to use it rather than those who have to build it, conceive the original description of what is required. Computer systems do not just build themselves based on a page of A4, or even a 25cm stack of pages. Skilled engineers have to take these requirements and figure out how to build a system that meets all the requirements including levels of service such as maintainability and reusability in a cost-effective, time-efficient manner.

They need tools and techniques to help them and in this article we revisit the well-known concepts of analysis and design, in the context of Unified Modelling Language (UML)<sup>1</sup>, object-orientation and real-time.

System development is the process of taking a set of system requirements and delivering a system that meets them. For

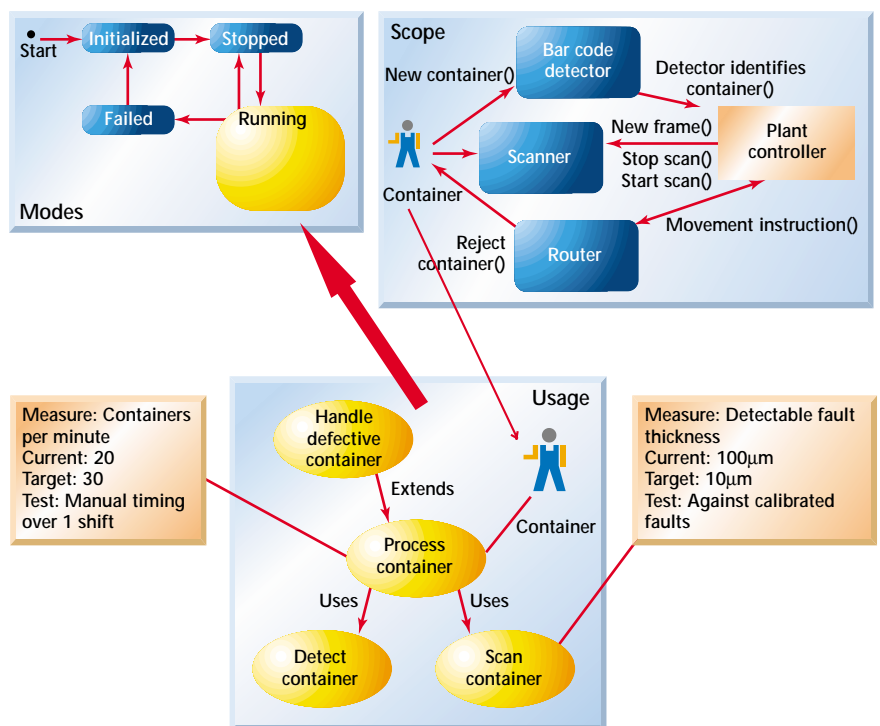


Figure 1: Requirement architecture fragment

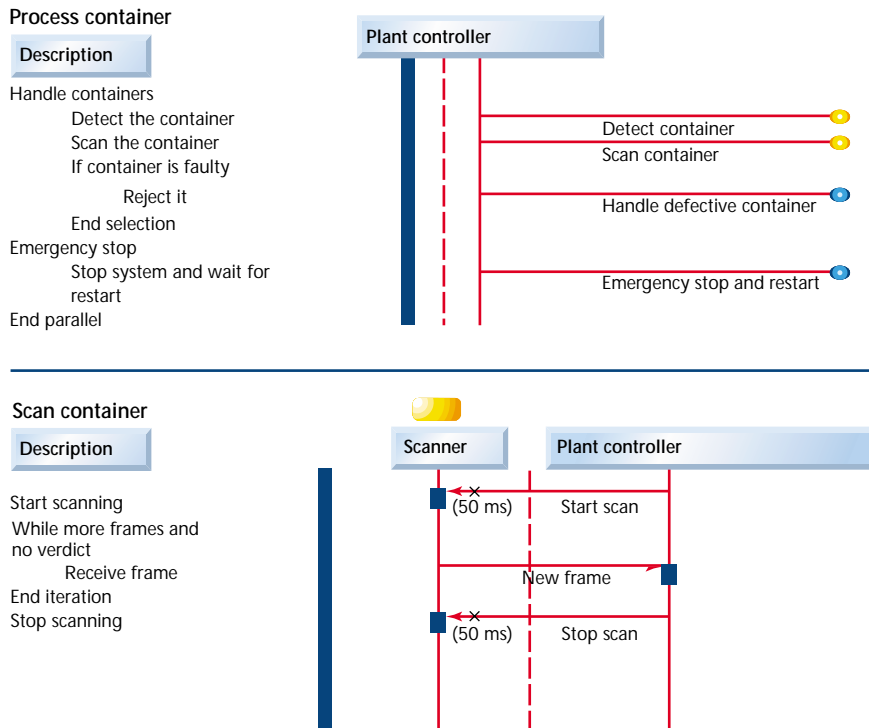


Figure 2: Sequences for process containers and scan containers use-cases

the purposes of this article I won't dwell on the mystical source of these requirements, nor taking care of the system once it is delivered.

So, how should we go about this awesome task? At the start of the process, a set of requirements is provided (we don't need to know how exactly). The project team then uses its skill and judgement to write source code and draft hardware layouts that describe a system able to meet these requirements. Finally, a real physical system is built from the code and hardware layouts, which actually does meet the requirements ... hopefully.

There are systems that are developed in this way. But they tend to be small and are often based on similar existing systems and components. There are two main reasons why this does not work for many systems. First, as an approach it doesn't scale. There comes a point at which it is no longer feasible to understand the requirements and visualize the structure of the solution in one's head. Second, systems are becoming more dependent on software. As there are very few existing software components around, more of the system has to be created afresh.

The most interesting bit of this process is the middle part. In some way, a textual

and maybe even verbal description of the requirements is realized in a set of source code and hardware specifications. In the rest of this article, I will address the issue of what we might put in the middle to help achieve this miraculous transformation. Briefly, if a system is on a grander scale than of old, and with a larger proportion of software, then the system's builders should consider adding intermediate processes. Detailed requirements, coming from the results of systems analysis, and a solution design, need to be added to the development process.

I will use an example to help illustrate the various ideas discussed in the paper. This example is a simple process-control problem. Our system contains a conveyor belt transport subsystem that moves containers, in this case cans, through a detector and scanner before moving them off the line for storage. Each can is identified by a bar code that is detected at the beginning of the transport process. The scanner unit scans the cans as they travel on the conveyor. A robot routing unit is told the ID of a defective unit so that the can be removed from the conveyor belt and discarded. An operator can monitor and operate the system from a remote computer terminal.

## Requirements specification

It is difficult to say exactly what form a specification of requirement takes. Traditionally, it has been a textual list of statements about what the system must do and any operational constraints on the way in which it has to do it. The most important thing is that there is some specification somewhere that we as engineers can work from.

Here's a small piece of the requirement, after some restructuring, that affects the scanning of containers:

- R3. The system must be capable of scanning each can and detecting faults in the cap-weld.

- N8. Cans must be scanned for cracks in the weld as small as  $10\mu\text{m}$

- N3. The system needs to process an average of 30 cans per minute

One of the biggest causes of rework is a poorly understood requirement or one that has been omitted completely. Often, these errors are not spotted until customer acceptance and so can be very expensive to fix. This problem has led to the idea of analyzing the requirements. During analysis, the requirements document is subjected to a pitiless examination of its flaws, holes and misrepresentations in order to reconstruct a newer, better specification that is self-consistent, coherent and complete. This section discusses an architecture for capturing the results of analysis.

## The requirements architecture

Given that initial statements of requirement come in many different forms, having a single representation of requirements has enormous benefits. The system's sponsors can expect a consistent look to what the engineering teams present for validation. Engineers downstream can use a more formal specification of requirements as the basis for their designs.

Many of the models in the requirements architecture will be familiar as UML<sup>1</sup> notations, particularly use cases and statecharts. However, the concept of system scope is not well addressed by the UML, nor are system constraints, as described here. The requirements architecture allows the analyst to capture the details of scope, function and operational constraints in a set of interdepend-

dent models, with a shared vocabulary. The four models are:

- **System Scope.** This defines the context in which the system will operate. It defines a vocabulary of actors and events that represent the external environment and communication with it. The system is represented as a rectangle, which is the source and sink of the various events. Interface devices may be added to show interactions with the environment in greater detail.

- **System Modes** identify any special modes that the 'whole' system may be in, and the external events, if any, that trigger mode changes.

- **System Usage** defines the functions or services that the system provides to the actors, in terms of use cases. Use cases are often valid only in certain modes, such as maintenance. They may supervise the transition between modes, such as an emergency shutdown, or exhibit mode-dependent behaviours, such as training.

- **System Constraints** define the levels of service required. These levels of service may be related to performance, robustness, accuracy or other criteria. They are related to use cases, and may be measured with respect to a specific operational scenario for a use case, using a specific sequence of input events and system responses.

The fragment of the requirements architecture in figure 1 shows how the requirements for the Waste System are represented.

Figure 4: An object collaboration diagram - part of the object architecture

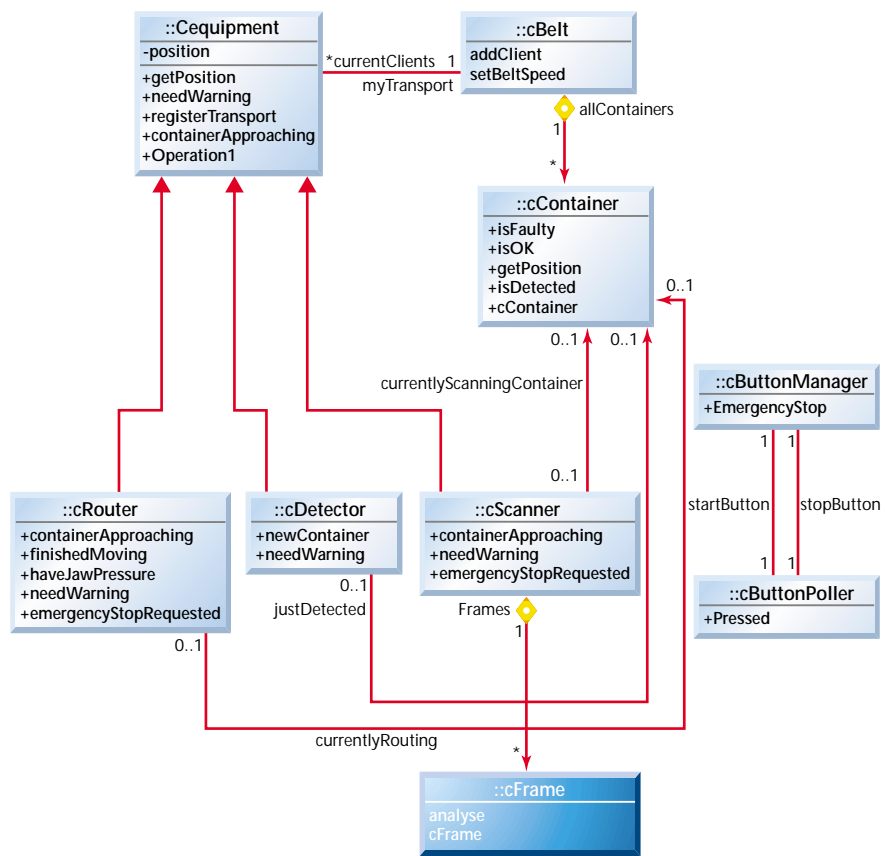
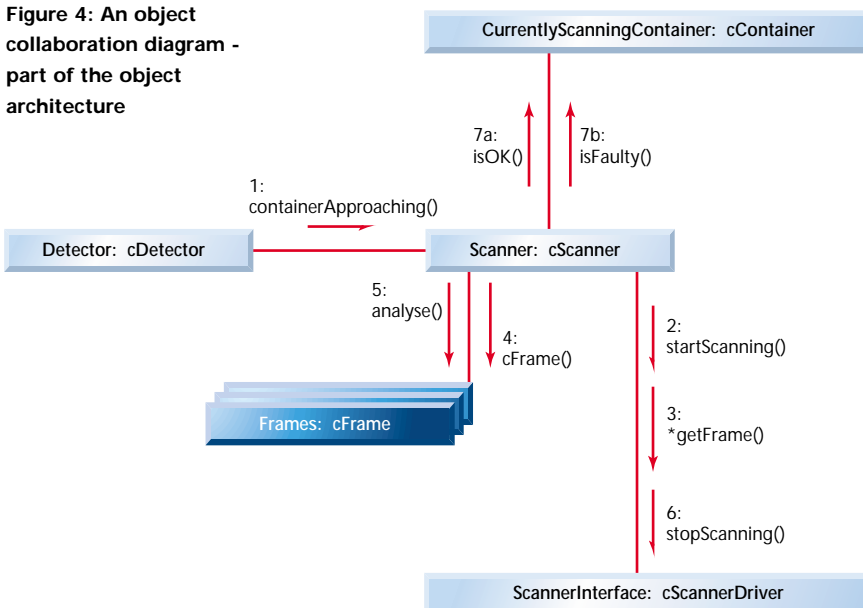


Figure 3: Example class diagram for the waste processing application

As we can see from the scope diagram, the Container is an actor. It is an external agent that is serviced by the system and is, at an abstract level, a source and sink of events. This may be all that is required, but often it is useful to

identify 'interface devices' that the system uses in order to sense and manipulate the external environment.

The Process Container use case provides the overall service of dealing with a container from start to finish. Process Container is associated with the overall performance constraint governing the number of containers to be processed every minute. We note that the entire Process Container use case is only valid in the Running mode. Defective containers are deemed to be an exceptional circumstance.

As a result, the Handle Defective Container use case extends the basic use case. The Scan Container use case is 'used' by Process Container as part of its processing and has, in turn, an associated constraint about the accuracy of fault detection.

### The role of the sequence diagram

Documenting use cases with a simple description is often sufficient, but if more clarity is required, especially about the order in which events occur, then a sequence diagram can be used to show

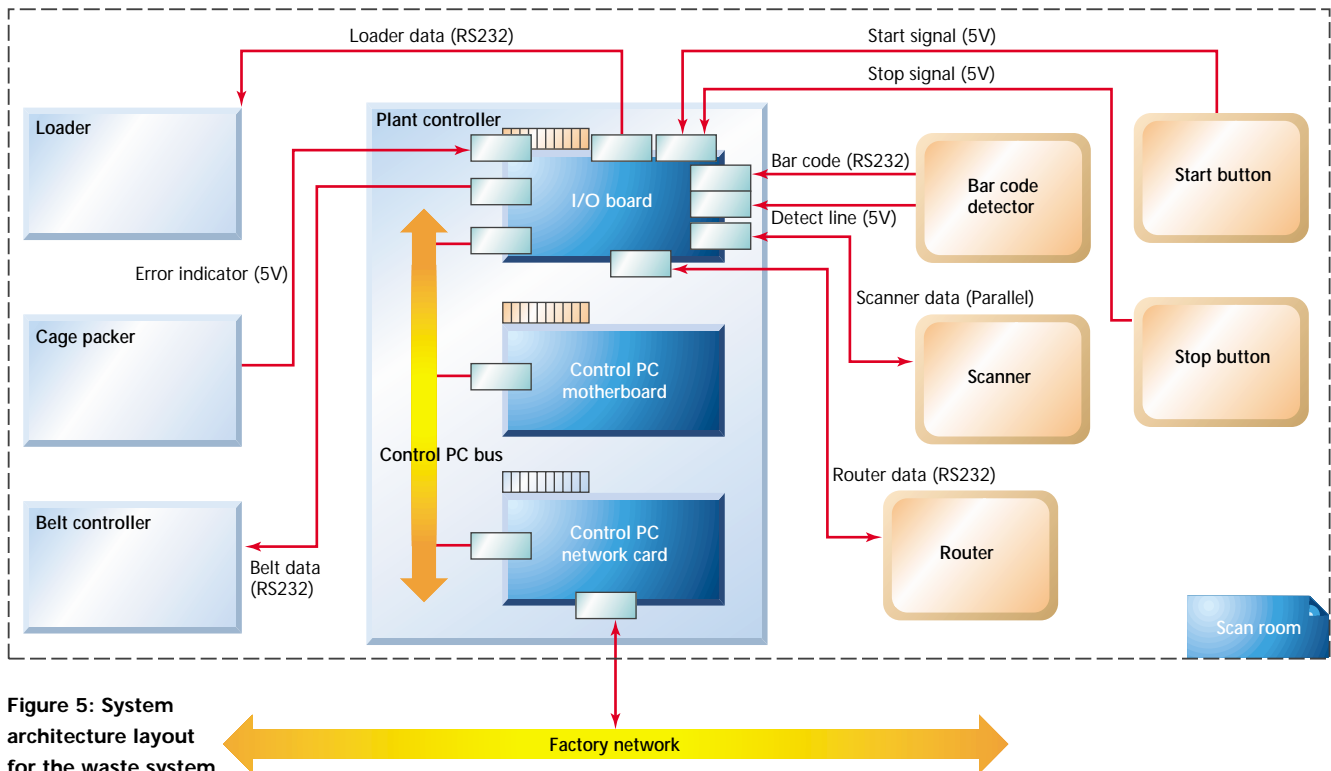


Figure 5: System architecture layout for the waste system

more detail. The sequence diagram treats the 'system' as a black box and shows its interactions with the external environment rather than any internal interactions between components of the system.

Sequence diagrams describe typical usage scenarios. They are valuable analysis tools that help to check requirements with the sponsor and to specify acceptance tests for the finished system.

In figure 2, we see the main sequence for Process Containers. It shows when the subordinate use cases are active. In addition, the use case we are following, Scan Containers, is shown in more detail with the specific sequence of events and responses shown. As we shall see, sequence diagrams form an important link between the analyst and designer. The designer will extend the sequences inside the system interface, showing how the components inside the system boundary collaborate to provide the required service.

### Solution design

Having analyzed the requirements, constructed a set of requirements models, and checked these with the user, we then need to design the solution: the system that meets the requirements. The most important characteristic of an architec-

ture that captures the solution is that it should be capable of describing the physical structure of the system. It needs to be detailed to trace through to the actual system description languages, whatever they may be. On top of all this is the need to support the visualization, communication and analysis of the system design, which lead to the requirement for a rich meta-model with a common graphical representation. UML provides much of this today and the extensions proposed in this article extend it for real-time systems design.

### Types and instances

Systems are built from things, or objects. During the design process, engineers look to reuse the same types of things that they have used in the past, or create new types of things that they can use in this system, and potentially future systems. There is a whole industry dedicated to producing different types of hardware components and, in the future, software components will be offered in a similar fashion.

A class diagram, such as figure 3, is a good example of the use of software components. It defines the types of objects that we can use in the system. We

could equally well have produced a diagram that shows the hardware devices that we might use.

In this example we can see that we have types, or classes, of objects that can be used when looking for objects to fulfil system requirements. An object of type Belt will convey information on containers to various pieces of Equipment on the conveyor belt.

One type of equipment, a Scanner, will manage a set of Frames that it will use to analyze the containers for faults. This will form a central role in implementing our use case,

Scan Containers. Container objects will track the containers being currently scanned or routed.

### Solution architecture

The Solution Architecture is tailored towards the implementation of software in object-oriented programming languages. However, taking an object-oriented approach irrespective of the eventual software description language can provide a better-structured, more maintainable design, with little overhead.

First, the Object Architecture describes a set of collaborating objects that together assume the responsibility for

# OO DESIGN

meeting the system's functional requirements. The objects are linked together and use these links to send messages to one another.

The example in figure 4 shows how the requirement to scan containers is met by the objects in the system. The scanner object is told about an approaching container and starts to scan it, building up a set of frames as it goes. Once it has analyzed the frames, it updates the current container based on its findings and waits for the next.

The System Architecture describes the physical elements of the system and how they interact. Types are very important in this architecture because many systems are built from existing types of physical components, such as integrated circuits, sensors, actuators and communications devices. A specific system model is then built from these types of component. The extract in figure 5 shows the Plant Controller subsystem, inside which are three boards that handle I/O, access to the factory network and container processing. They are connected to an inter-

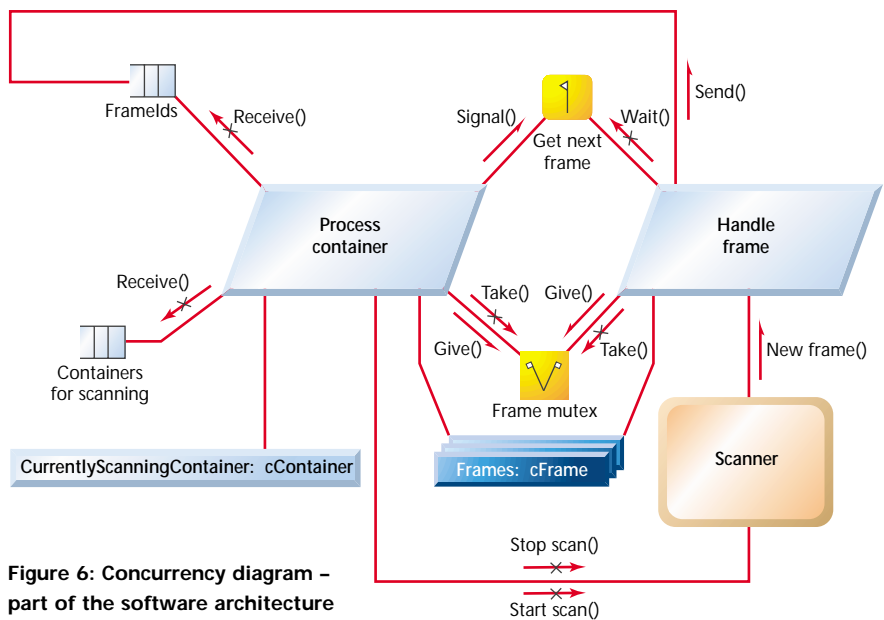


Figure 6: Concurrency diagram - part of the software architecture

nal bus and the whole subsystem is connected to the factory network.

The specific I/O connection to the interface devices is shown to the level of board interfaces. These interfaces have features such as interrupt request (IRQ)

numbers and I/O addresses that are needed by the software engineers. The devices and processing hardware used will be determined by constraints on the system. In the case of a scanner, those constraints will be the accuracy and through-

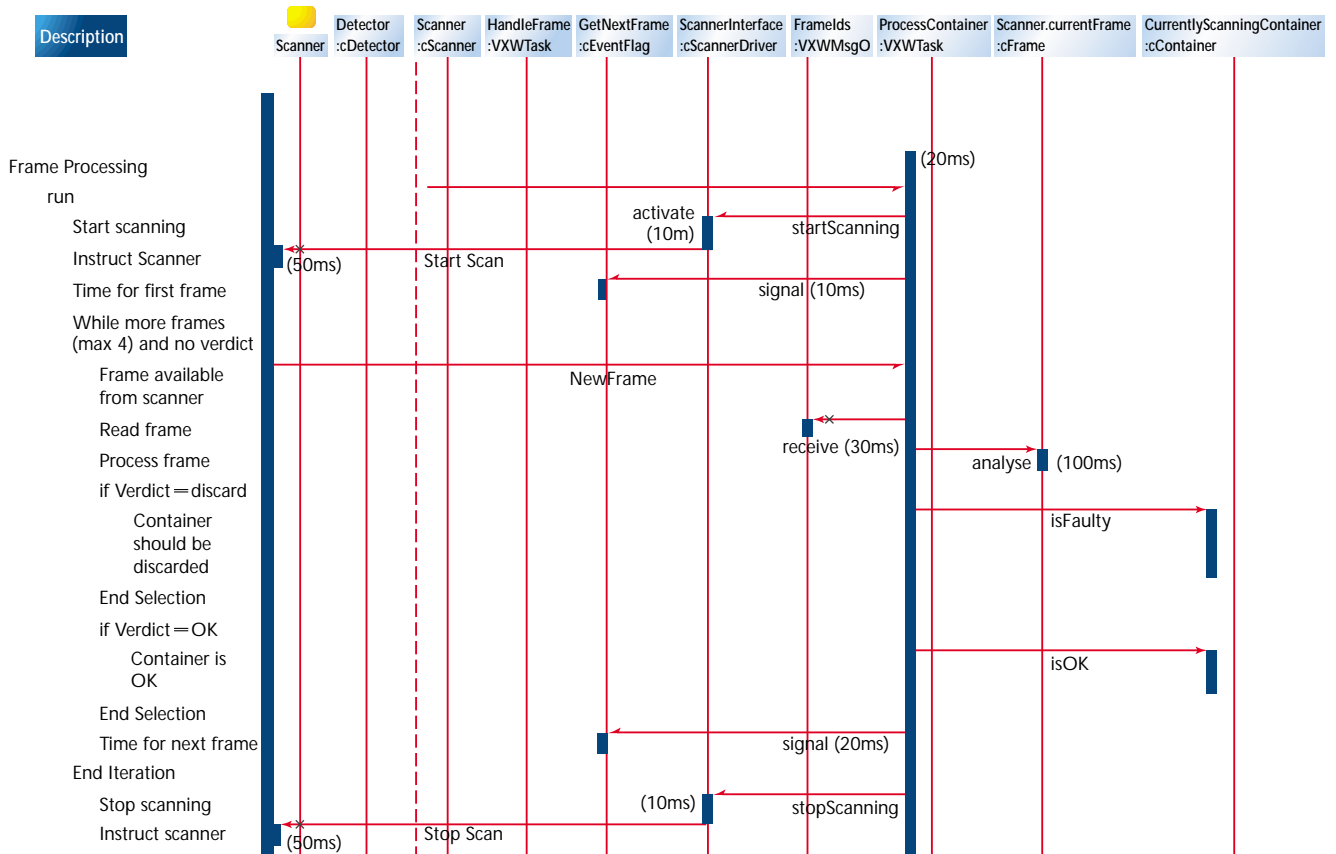


Figure 7: How the system's objects handle the task of scanning a container

put requirements specified earlier.

The Software Architecture is used to ensure that the functionality as defined in the object architecture runs on the system architecture with the required levels of service for persistence, responsiveness, throughput and other criteria. The most important aspect of the software architecture is a concurrency model that describes the threads or tasks: the units of concurrent execution. It also identifies the communications required between the threads in order to share data and pass control. Note that the threads and communications primitives have types, or classes, just like any other object. However, they are special in that they represent separate execution threads and operating system resources. As a result, they need specialized treatment in diagrams.

In the example fragment of figure 6, we see the tasks that implement the scanning of containers. One handles the reception of frames from the scanner de-

vice; the other processes the frames to identify any faults. Details of approaching containers are sent using a queue. The two tasks communicate using an event flag (Get Next Frame) and a shared object set (Frames), protected by a semaphore to ensure mutual exclusion. The results of the scan cause an update to another shared object, CurrentlyScannedContainer. Two tasks have been introduced because incoming frames need to be handled promptly or some data may be lost. The processing of that information may proceed at a slightly more leisurely pace, subject to the overall throughput requirements on the original use case.

### From requirements to the solution architecture

As we complete more and more of the solution design, we can introduce more and more detail into the sequence diagram we started during requirements analysis. This provides excellent trace-

ability between the initial requirement and the solution.

Many performance constraints are specified for specific scenarios. By using sequence diagrams to describe these scenarios and allocating time budgets to the individual messages and operations we can start to show how a constraint will be met. Figure 7 looks at a particular scenario of the Scan Container use case in more depth. This fragment of a sequence diagram shows a lot more detail about how the use case described in figure 2 is implemented by the system. We can see that the Scanner object is told of approaching containers and analyzes them to assess faults.

As well as satisfying ourselves that we know how the scanning function will be implemented, we can analyze the predicted end-to-end time and compare it to the constraint on Scan Container. In this case, with a maximum of four frames, the requirement to scan a container in 2s can just be met.

## The system

From the solution design it should be a straightforward transformation into the human-readable source that can be turned into the system. Note that if the fidelity of the design is great enough and the design is complete enough, it should not be necessary to have human-readable source for the system. However, as things currently stand, software designs are rarely this complete. Often, it is faster to write certain parts of the source, particularly code bodies, in the source code language.

## Conclusion

So, here we are at the end with a seemingly less streamlined development lifecycle but which eases the transformation from requirements to implementation. We have introduced a couple of additional stages, analysis and design. They help us to bridge the semantic gap between what is required of a system and the physical incarnation of the system itself.

Analysis is the process of examining the requirements for consistency, coherence and completeness. The aim is to replace those parts of the requirements specification that govern scope, function and constraints with a set of models that describe these areas more accurately, fully and clearly.

Design is the process of taking the requirements and producing a system that not only meets the requirements, but also can be delivered on time and within budget. Introducing these two extra activities into the development process eases the step from requirements to system implementation. The notion of having two separate architectures allows us to fulfil two major aims:

- Clearly communicate our understanding, as engineers, of the requirements to the user.
- Describing the system design in a way that can be translated readily into a system implementation.

The challenge is to trace from the

analysis to the design in a valid fashion. Fortunately, the use of sequence diagrams and scenarios, provides us with an excellent traceability mechanism, helping to assure a much smoother path from user requirements to system implementation.

■ *Alan Moore is Vice President, Product Strategy at Artisan Software Tools. He has 15 years of experience in the development of real-time and object-oriented methodologies, and their application in a variety of problem domains. He has been actively involved in product development, training and consulting related to OOAD and structured development tools during that time. Alan has co-authored a book on GUI design and published several papers and has lectured on a wide variety of analysis and design issues.*

## References

1. Unified Modeling Language v1.1, OMG, September 1997.