Research Report 8/97

An Object-Oriented Framework for Measurement Systems

by

Jan Bosch

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HKR-RES—97/8—SE

An Object-Oriented Framework for Measurement Systems
by Jan Bosch

ISSN 1103-1581
ISRN HKR-RES—97/8—SE

Copyright © 1997 by Jan Bosch
All rights reserved
Printed by Psilander Grafiska, Karlskrona 1997

An Object-Oriented Framework for Measurement Systems

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.pt.hk-r.se/~bosch

Abstract
Measurement systems are of increasing importance for manufacturing, due to high automation level of production processes. Although most measurement systems have much in common and are expensive to construct, these systems are often developed from scratch, hardly reusing the available designs and implementations. To address this, we have designed and implemented an object-oriented framework for the domain of measurement systems that can be used as the core of measurement systems. Evaluations of the framework show that it captures the main concepts in the domain and that the required extensions for individual applications are limited. In this paper, a number of example framework instantiations are presented. The lessons we learned during the framework design and an evaluation of the object-oriented modelling paradigm are presented.

1 Introduction

The increasing automation of the production process has begun to address processes beyond the primary production processes. During the last decade, one can recognise an increasing need for automated tools that support the quality control processes surrounding the actual production. The emergence of the ISO9000 quality standards, the quality thinking in general and the increased productivity of production technology requires the quality control systems to improve productivity as well and whereas many factories used manual quality control by personnel, nowadays the need for automated support is obvious. This development has dramatically increased the need for automated measurement systems. The advantages of measurement systems are generally improved performance/cost ratio and more consistent and accurate quality control. This development increased the needs for reusability of existing measurement system software. Although these systems, conceptually, have a rather similar structure, in practice, we found, the implementation of these systems to be rather diverse. This is due to the fact that real-time constraints, concurrency and requirements resulting from the underlying hardware strongly influence the actual implementation.

Despite these difficulties, we have, together with our industrial partner, EC-Gruppen[1], designed a framework for measurement systems that would decrease their software development cost by increasing reuse of existing software and, as an important second requirement, increase the flexibility of running applications. Operators of the measurement systems often need to make some adjustments in the way the measurement system evaluates a measurement item and the system should provide this flexibility. However, traditional systems constructed in C and assembly often have difficulty to provide this functionality.

The intention of this paper is to describe our experiences resulting from the design of the measurement systems framework that we were asked to construct. For this design, we used the conventional object-oriented paradigm as expressed in C++ and Smalltalk. A second intention of this paper is to evaluate the object-oriented paradigm in itself and to determine what expressiveness could be considered as lacking in the conventional OO paradigm.

The remainder of this paper is organised as follows. In the next section, measurement systems are described in more detail and the requirements on the object-oriented framework as put forward by the software engineers actually building these systems are described. Section 3 describes the actual architecture design using the conventional object-oriented paradigm. In the subsequent section, we describe the modelling problems that we identified during the

1. EC-Gruppen, Halda Utvecklingscentrum, S-376 23, Svängsta, Sweden. Tel. +46-454-370 00.

framework design and implementation. Section 7 discusses work related to the topics discussed in this paper and the paper is concluded in section 8.

2 Measurement Systems: Requirements

Measurement systems are a class of systems used to measure the relevant values of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e. as part of the same system, used to control the production process that creates the product or process that is measured. A measurement system is used for quality control on produced products that can then be used to separate acceptable from unacceptable products or to categorise the products in quality categories. In some systems, the results from the measurement are stored in case in the future the need arises to refer to this information, e.g. if customers complain about products that passed the measurement system.

Although a measurement system contains considerable amounts of software, a substantial part of these systems is hardware since it is connected to the real-world through a number of sensors and actuators. The sensors provide information about the real-world through the noticed impulses. However, whereas traditional sensors were primarily hardware and had a very low-level interface to the software system, new sensors provide increasing amounts of functionality that previously had to be implemented as part of the software. For instance, a conventional temperature sensor would only provide the A/D conversion and the software would need to convert this A/D value into the actual temperature in Celsius or Kelvin and, in addition, had to do the calibration of the sensor. Modern temperature sensors perform their own calibration and immediately provide the actual temperature in the required format. The interface between the sensor and the system is becoming more and more high-level, but also more complex since the amount of configurability of the sensors is increasing.

With respect to the actuators one can recognise a similar development. Whereas the software previously had to be concerned with the actuation through the actuators, modern actuators often only need a set value expressed in application domain concepts such as angular speed or force. For example, to control the open angle of a valve in a traditional measurement system, one would have to generate a ‘duty cycle’ in software. A duty cycle is the periodic process of sending out a ‘1’ for part of the cycle and a ‘0’ for the rest. The ratio between the time the output is ‘1’ and the time the output is ‘0’ represents the ‘force’ expressed through the actuator. When opening a valve for 70% requires that the system outputs a ‘1’ for 70% of the cyclic period and a ‘0’ for the remaining 30%. The implementation of this is often achieved through an interrupt routine that changes the output signal when required. Modern actuators contain considerably more functionality and will generate the duty cycle themselves, requiring only the set value from the software.

These developments in the domain of sensors and actuators changes measurement systems from small, single processor systems that are developed very close to the hardware to distributed computing systems since the more complex sensors and actuators often contain their own processors. However, although the increased functionality of the sensors and actuators reduces the complexity of constructing measurement systems, the increased demands on these systems and the resulting increase in size make that the construction of measurement systems is a complex activity. The languages and tools used to construct measurement systems ought to provide powerful means to deal with this complexity.

A measurement system, however, consists of more than sensors and actuators. A typical measurement cycle starts with a trigger indicating that a product, or measurement item, is entering the system. The first step after the trigger is the data-collection phase by the sensors. The sensors measure the various relevant variables of the measurement item. The second step is the analysis phase during which the data from the sensors is collected in a central representation and transformed until it has the form in which it can be compared to the ideal values. Based on this comparison, certain discrepancies can be deduced which, in turn, lead a classification of the measurement item. In the third, actuation phase, the classification of the measurement item is used to perform the actions that are associated with the classification of the measurement item. Example actions may be to reject the item, causing the actuators to remove the item from the conveyer belt and put it in a separate store, or to print the classification on the item so that it can be automatically recognised at a later stage. One of the requirements on the analysis phase is that the way the transformation takes place, the characteristics based on which the item is classified and the actions associated with each classification should be flexible and easily adaptable, both during system construction, but also, up to some extent, during the actual system operation.

Based on the above discussion, one can wonder whether the traditional view on measurement systems as a centralised system with one main control loop. During the project, we became convinced that the system should to be viewed as a collection of communicating, active entities that co-operate to achieve the required system behaviour. This improves decomposition of the system, decreases the dependencies between the various parts and increase system flexibility. However, decomposing the system into active entities requires processes to be available, or at least simulated, by the underlying operating system.

Another important aspect is the real-time behaviour of the measurement system. Different from many real-time systems, a measurement system is not a periodic system. The real-time constraints in the system are, directly or indirectly, related to the triggering point where a product to be measured enters the system. Although, when running at maximum performance, this becomes a periodic behaviour, the start is not determined by the clock, but by a physical event. In the ideal situation, the software engineer would specify the real-time constraints on the different activities in the system. Based on that specification, the system would schedule the activities such that the real-time constraints are met or, if it is not possible to schedule all activities, respond to the software engineer with a message. However, in the current situation, the software engineer implements the tasks that have to be performed and performs a test run of the system. Often, the system does not meet all deadlines at first and the software engineer has to adjust the system to fulfil the requirements by, e.g. changing the priorities of the different processes.

Finally, the requirements on modern measurement systems often result in systems that are no longer confined to a single processor. Distribution plays an increasingly important role in measurement systems. However, the presence of distribution should not require the software engineer to change the basic architecture of the system. The system should just be extended with behaviour for dealing with communication over address spaces.

2.1 Non-functional requirements

Based on the discussion above, we have distilled a number of requirements for measurement systems.

- **Intuitive**: As any type of system the designed framework should be based on concepts that have a direct correspondence in the application domain. The way these concepts are used and combined should be logically consistent with the view of a domain expert.
- **Reusable**: The framework should provide reusable components for the construction of measurement systems. This requires a delicate balance between generality and speciality. It also means that the components and decomposition dimensions have to be chosen such that relatively general components from different dimensions can be composed to form specific components that can be used in real system with minimal extensions.
- **Flexible**: Although flexibility would be considered to be a positive aspect of any system, the requirements on the flexibility of measurement systems are higher than average. As described, the actual composition of the system from its components, the analysis process and the reaction by the system based on the analysis results needs to be easily adaptable both during application development as well as during system operation.
- **Real-time constraints**: Although most traditional system construction approaches deal with real-time constraints by running tests on the system and measuring the system responses, we already discussed the advantages of expressing real-time constraints directly as part of the system. The difficulty with both real-time and concurrency is the platform dependence of the implementation of these techniques.

The version of the framework presented in this paper does not deal with the issues of concurrency and distribution. The reason for that is that, in the project, we were most concerned with achieving a stable application architecture and issues such as concurrency and distribution can be superimposed on top of the architecture. However, concurrency and distribution are relevant for measurement systems. As mentioned earlier in this section, considering the increasing complexity of measurement systems, it may easily become necessary to decompose a system into communicating, active components that are distributed among various processors. We intend to address these issues in a future version of the framework.

3 Measurement System Framework Design

3.1 Introduction

When basing a measurement system on object-oriented principles, one tries to model the real-world using objects that represent real-world entities. Both the sensors and the actuators are clearly existing in the real-world and are correspondingly modelled as objects. The measurement item, however, is a more complex question. The measurement item obviously exists in the real world, but it has to be instantiated every time when a physical item enters the measurement system. However, when it is instantiated, it is empty and contains no data whatsoever concerning its physical counterpart. This information has to be obtained from the sensors that measure the relevant variables of the physical entity. This is where an important design decision concerning the boundaries between the sensor objects and the measurement item has to be made. Only in the simplest systems, the values measured by sensor with an immediate corresponding hardware sensor can be used directly by the measurement item. Very often the data has to be transformed, e.g. condensed, cleaned from faults or converted into other domains. The question is whether this is the task of the measurement item or the task of the sensor measuring the data. The answer, as for all difficult questions, is ‘it depends’. Sometimes the data from one or more physical sensors needs to be transformed before it represents the data that can be used by the measurement item, whereas in other cases the data forms a logical part of the measurement, and needs to be transformed in the process intuitively associated with the measurement item. An example of the first is the use of redundancy. The designer of the system may make use of more than a single sensor to increase the reliability of a particular variable of the physical item. The measured values need to be compared and an average or most votes result will be used as the value measured by the abstract sensor. An example of the second type is the repeated measurement of a variable in time. The sensor may measure the value of a variable multiple times for a measurement item, e.g. the thickness of an item may be measured at several points to give a sufficient coverage of that aspect of the item.

3.2 The Measurement Process

In figure 1, the architecture of a simple measurement system is shown. It consists of five entities that communicate with each other to achieve the required functionality.

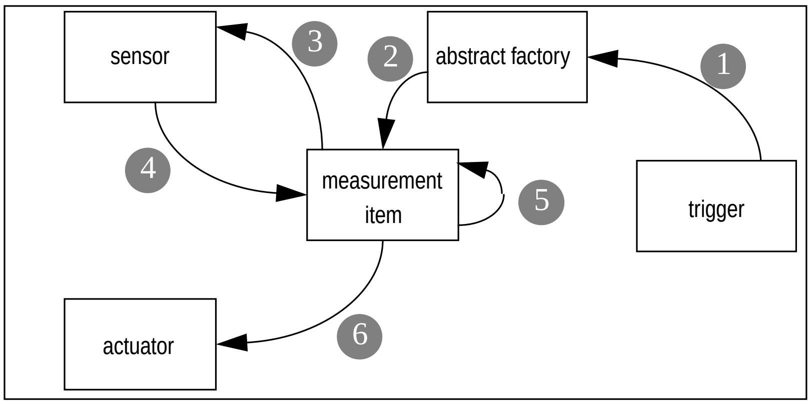

Figure 1. Architecture of a simple measurement system

1. The trigger triggers the abstract factory when a physical item enters the system.
2. The abstract factory creates a representation of the physical object in the software, i.e. the measurement item.
3. The measurement item requests the sensor to measure the physical object.
4. The sensor sends back the result to the measurement item which stores the results.
5. After collecting the required data, the measurement item compares the measured values with the ideal values.
6. The measurement item sends a message to the actuator requesting the actuation appropriate for the measured data.

3.3 Sensor

The sensor is the software representation of a hardware device that measures one particular variable of the item to be measured. The sensor is responsible for maintaining an accurate model of the hardware sensor. To achieve this the sensor communicates with the hardware sensor. The way the sensor updates itself with the data in the physical sensor can be different depending on the sensor type and the application. In the framework, the updating of the sensor has been modelled as a separate strategy. The update strategy is discussed in the next section.

Another aspect of the sensor is that the data read from the hardware sensor has to be converted into a value which has some meaning in the context of the software system. This conversion process can be very different depending on the application and the way the sensor is used. Therefore, also the conversion has been abstracted as a *calculation strategy*. The details of the calculation strategy are described in section 3.3.2.

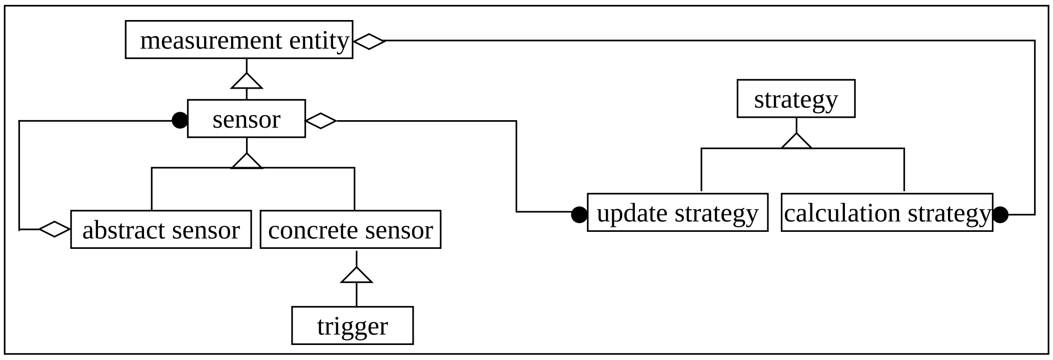

Figure 2. Class relations for Sensor

Figure 2 presents the class hierarchy of the sensor classes. The abstract superclass sensor has two subclasses, abstract sensor and concrete sensor. A concrete sensor has a one-to-one relation to a hardware sensor, i.e. for every hardware sensor a concrete sensor exists. The abstract sensor represents an abstraction of one or more sensors. For instance, two concrete sensors that both measure the same aspect of an item could be contained in an abstract sensor that calculates the average of the two and uses that as its value. The concrete sensor has a subclass trigger that is used for triggering the abstract factory when a physical item enters the measurement system.

The sensor has several methods, such as methods for reading and setting the update and calculation strategies. However, the main method used by the measurement item is the `getValue` method:

```
getValue
   "returns the stored measured value."

   (self updateStrategy) clientUpdate.
   ^self measuredValue
```

This method first calls the update strategy which, depending on the strategy type, might update the value stored in the sensor. Secondly, the object returns the measured value.

3.3.1 Update Strategy

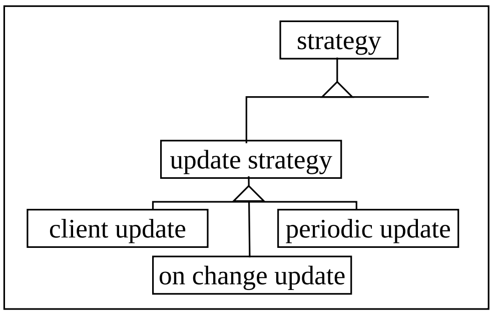

Figure 3. Update strategy class hierarchy

The correct way of updating the sensor depends on the application and the type of sensor. Recognising this caused us to abstract the updating behaviour using the strategy design pattern. So far, three strategies for updating have been defined:

- **Client update**: This update strategy is used when the amount of computation on a sensor should be minimized. The sensor never updates itself, until it is called by a client. Upon receipt of the request, the sensor first updates itself and subsequently return the (very recently) updated value.
- **Periodic update**: The periodic update strategy requires an time interval as an input. This time interval is used for the periodic update. At the beginning of each time interval, the sensor will update itself with fresh data from the physical sensor.

- **On change update**: This update strategy can be used when the physical sensor notifies, either through an interrupt or otherwise, the software system that its value has changed. Every time the sensor is notified, it will update itself.

In figure 3, the different update strategies are shown. In the current definition of the framework, these update strategies are mutually exclusive, i.e. only a single strategy can be active at any point in time. However, the update strategy of a sensor can be changed at run-time. Nevertheless, one could imagine that in certain applications two strategies should be combined, e.g. the periodic and client-based update strategies.

3.3.2 Calculation Strategy

The calculation strategy, as we will see later in this paper, is used by more than just sensors. Also measurement items and other entities make use of the calculation strategy which is why the aggregation relation is placed at the level of the measurement entity class. For the concrete sensor, the concrete calculation strategy class is defined. For the abstract sensor, no calculation strategies have been pre-defined, but these strategies are rather easy to define.

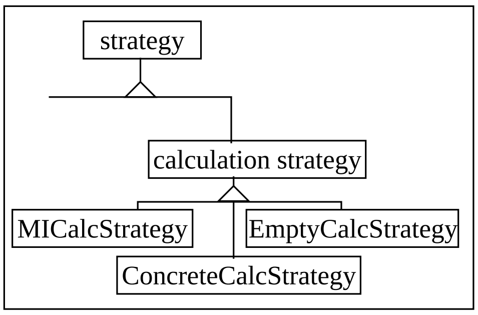

Figure 4. Calculation strategy class hierarchy

3.4 Measurement Item

The measurement item is the object which contains the data collected from the sensors concerning the physical measurement item. As shown in figure 5, the measurement item inherits from class measurement entity causing it to contain a calculation strategy. The calculation strategy has been discussed in section 3.3.2. The measurement item has its own calculation strategy, i.e. EmptyCalcStrategy, which just forwards the calculation process to the measurement item itself. Other parts of the measurement item are a collection of measurement values (discussed in section 3.4.1), a collection of actuators (discussed in section 3.6), an actuation strategy and a reference to the item factory that instantiated the measurement item.

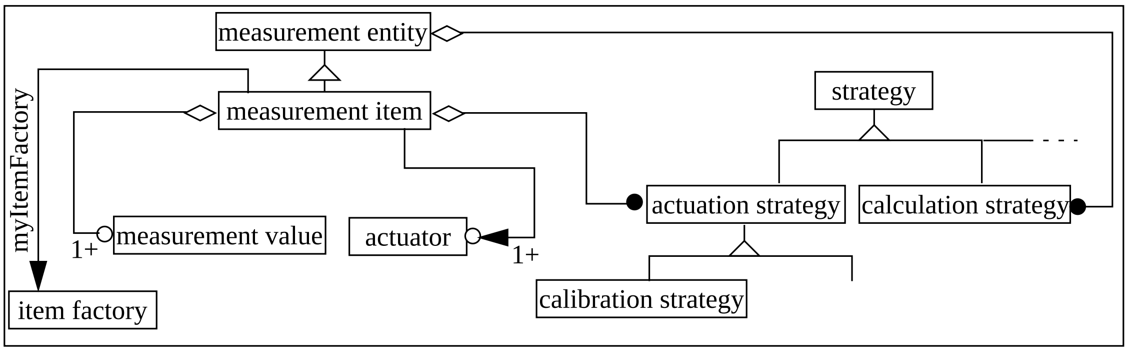

Figure 5. Relations for class Measurement Item

A measurement item is invoked via its start method by the item factory after it has instantiated the item. The start method contains the top-level behaviour for the measurement item with respect to its primary task, i.e. to collect data on the physical item it represents and to actuate the actuators appropriately based on the measured data and the comparison to the set data.

```
start

    (self calculationStrategy) performCalculation.
    (self actuationStrategy) actuate.
```

```
^self
```

The calculation phase of the measurement item is concerned with collecting the data and converting it to a value form that matches the requirements within the system. The only action performed by the measurement item is to invoke each measurement value that it contains with a request to collect the data from the sensor the measurement value is connected to and to process this data. The actuation phase is concerned with generating the necessary effects on the actuators, based on the values collected during the calculation phase. The actuation strategy uses the set of actuator references stored by the measurement item to activate the various actuators. After the measurement item has been sent the required actions to the actuators it is removed from the system.

3.4.1 Measurement Value

The measurement value class represents one aspect of the physical measurement item that is used by the system. A measurement value has 0, 1 or more dimensions and a domain. Examples of a measurement value are the presence of a part of the item, (0-dimensional, boolean domain), the temperature of an item (0-dimensional, real domain), the width of an item measured through a sequence of samples (1-dimensional, real domain) and a camera image (2-dimensional, 0-255 (gray scale) domain). Each measurement value contains a set value representing the correct value, a measured value representing the value measured at the current physical item and a compare method describing how to interpret differences between the measured and set values.

Multidimensional measurement values, i.e. one-dimensional and higher, are modelled using the *composite* pattern [Gamma et al. 95]. The class organisation is shown below. The measurement value inherits from measurement entity, which contains the behaviour common for all entities in the measurement system. Measurement value has two sub-classes, i.e. atomic and composed measurement value. The first represents measurement values directly obtained from a sensor. These values generally are zero-dimensional, although exceptions exist. The latter represents the (multi-)dimensional measurement values, i.e. one-dimensional and higher, which are composed of measurement value instances.

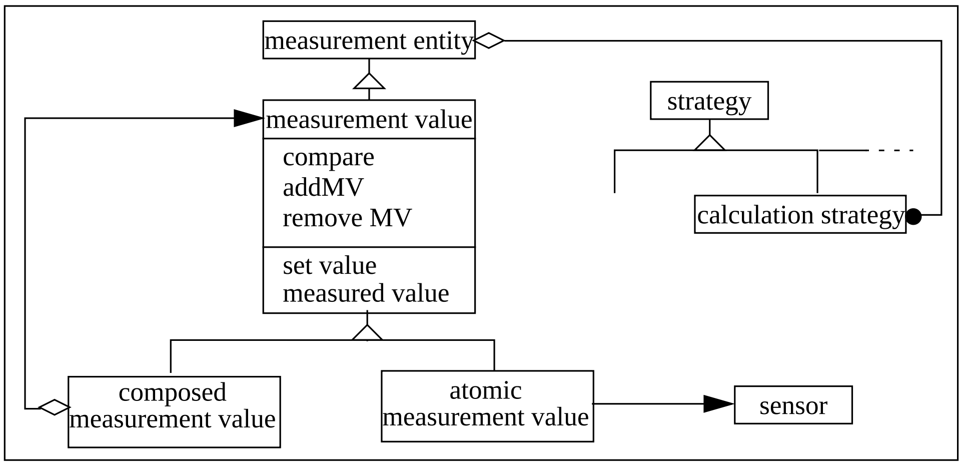

Figure 6. Measurement value class relations

As shown in figure 6, one can, using the composed measurement value, construct multi-dimensional measurement values that can contain several values from the same sensor, but measured as different time points, data values from various sensors that have to be composed in an integrated value or multi-dimensional data from one sensor, e.g. a camera.

3.4.2 Calibration Strategy

In measurement systems, calibration is the process of collecting the ideal values for the measurement item. This means that, opposite from the normal behaviour, the values measured during process are to be considered the correct values, rather than the set values stored in the measurement item. Calibration influences the actuation strategy of the measurement item rather than the calculation strategy. Therefore, the normal actuation strategy of the measurement item is replaced with a calibration strategy that performs the actions required during calibration.

In order to fully understand the calibration process, it is important to see that the item factory and the measurement item both play a role. The item factory (see also section 3.5) contains an instance of class measurement item that is used as a prototype for generating measurement items during normal system operation. When the user of the system decides to calibrate, the first step is to notify the item factory that the next measurement item is to be used for calibration. When the trigger notifies the item factory, the item factory instantiates a measurement item, but replaces the normal actuation strategy with a calibration strategy instance. The measurement item performs its measurements and calculations and then activates its actuation strategy. The calibration strategy used in this framework looks as shown below. It first requests its context, i.e. the measurement item to calibrate itself. This causes the measured values to be stored as set values in the measurement item. The next step is to call the item factory with the measurement item as an argument. This causes the item factory to replace its current prototype item with the measurement item passed as an argument. Now, all following trigger events will result in that the item factory instantiates a copy of the measurement item now stored as the prototype item, i.e. the prototype measurement item contains the new set values.

```
actuate
    "performs the required actuations for calibration after the measurement is finished"

    context calibrate.
    (context factory) calibrate: context.
    ^self
```

3.5 Item Factory

The item factory is an instance of the *abstract factory* design pattern [Gamma et al. 95] and it is responsible for instantiating instances of class measurement item whenever it receives a trigger event, to configure these instances and to activate each instance by providing it with a separate process or invoking its start method.

In figure 7, the class structure of the item factory is shown. It contains an instance of measurement item denoted as prototype item and a state variable inCalibration indicating whether the system is in calibration mode or in normal operation. The calibrate method is used to put the item factory in calibration mode, whereas the calibrate: method is used to replace the current prototype item with the passed argument.

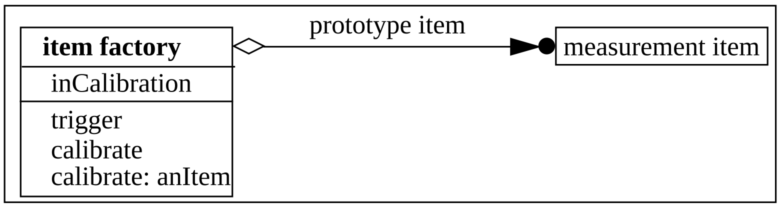

Figure 7. Item factory class

The trigger method is the main method of the item factory, since it is responsible for instantiating new measurement items. The code of the method is shown below. First, a new copy of the prototype item is created. Secondly, it is determined whether the system is in calibration, causing a calibration strategy to be used as the actuation strategy, or that the system is in normal operation, in which case the normal actuation strategy is used. Subsequently, the necessary references are bound and the measurement item is activated by invoking its start method. We refer to section 3.4.2 for a more detailed discussion of the calibration process.

```
trigger
    "called by the trigger to indicate that a measurement item has entered the system"

   | mi as|
   mi := prototypeItem copy.
   (inCalibration)
      ifTrue: [ as := CalibrationStrategy new. inCalibration := false]
      ifFalse: [as := ActuationStrategy new].
   as context: mi.
   mi actuationStrategy: as.
   mi factory: self.
   mi start.
   ^mi
```

3.6 Actuator

The actuator is used to perform actions in the real world based on the measurements taken by the measurement system on the measurement item at hand. Examples of actions performed by actuators are to remove a dirty beer can from a conveyer belt or to print a code on the physical measurement item. So far, only the superclass Actuator has been defined. More concrete actuator classes are often so application specific that they need to be defined for concrete applications.

The basic behaviour of an actuator is to perform actions in the real-world through an interface. These actions can vary widely, such as removing an item from a conveyer belt or printing a code on the item that later is used for classifying the item. The actuator is invoked by the actuation strategy stored in the measurement item during the actuation phase. After instantiation, the measurement item first enters the calculation phase during which it collects data from sensors which is stored in the measurement values. Subsequently, the measurement item enters the actuation phase, thereby invoking the actuation strategy to determine the appropriate actions. The actuation strategy will invoke the relevant actuators, causing the intended actions in the real-world.

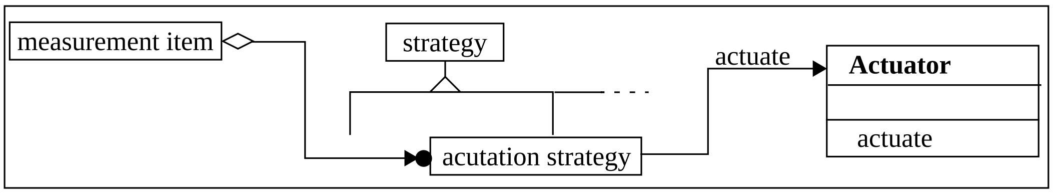

Figure 8. Context of class Actuator

3.7 Real-time aspects

Although time is not mentioned in the description of the classes in the framework, it does play an important role in the framework and in measurement systems in general. In a measurement system, an entity enters the system, e.g. on a conveyer belt, and is detected by a trigger sensor, causing the actions described earlier. However, all subsequent actions need to be performed at certain time points and not arbitrarily. A sensor needs to read the data at the time when the measurement item passes the sensor, just as the actuator has to perform its action at the right point in time. Although failure of real-time constraints is not catastrophic, the correct operation of the system depends on it and untimely behaviour may stop the production process and the system can therefore be viewed as relatively hard real-time. Different from most (hard) real-time systems, a measurement system is *not* a periodic system. Although there is a repeated behaviour in the system, this behaviour is not triggered by a clock but by a trigger sensor. Since the amount of time between two items may vary, so may the computation requested from the system.

In the framework design, all time points are considered relative to the time at which the trigger detected an item entering the system. The trigger sensor sends a trigger event to the item factory with a time stamp in milliseconds as an argument. This time stamp is passed on the measurement item object that is created in response to the event. The measurement item contains a calculation strategy and an actuation strategy, which both need to be executed in a time aware manner, for reasons described above. Both the calculation and actuation strategy use the time stamp to time their reading and actuation actions. Since the strategies are highly application dependent, it is no problem that these strategies hard code the necessary delays between their actions.

One aspect not taken care of by this approach, and consequently left out of the framework, is the situation where the time points at which actions need to take place are dependent on some external factor. For example, if the conveyer belt has a variable speed, the time points for reading and actuating will depend on the speed of the conveyer belt.

4 Simulating Framework Applications

Any framework requires evaluation of its design after it has been defined. This is a necessary activity to validate the generality of the design and the amount of reusability that can be achieved from it. Since the framework design often is based on a limited number of example systems, it is important to use with the framework for systems in the domain that are different from the example systems that were used for the framework design. However, for the domain of measurement systems constructing real applications only for testing the framework is not feasible due to the large cost involved, primarily due to the expensive equipment in a measurement system. Therefore, before constructing actual systems using the framework, it should be evaluated considerably. Based on this, it was decided to develop some framework applications in a simulated context. Rather than running the framework application in a real measurement system, it would initially run in a simulated environment.

The notion of a simulated environment requires that entities are available that act as place holders for the actual hardware that is part of the measurement system that is simulated. In case of measurement systems these entities are primarily the physical sensors and actuators since these are the entities that are connected to the sensor and actuator classes in the framework. A simulated framework application can visually be represented as shown in figure 9. One can identify three main parts, i.e. the framework, the simulated environment and, in between, the application code. The framework has been described in the previous section and the application code refers to the code necessary to adapt the contents of the framework for the application at hand and to functionality that is so specific for the application that it is unlikely that it can be reused in other contexts. The simulated environment is the subject of this section and is discussed in the following subsections.

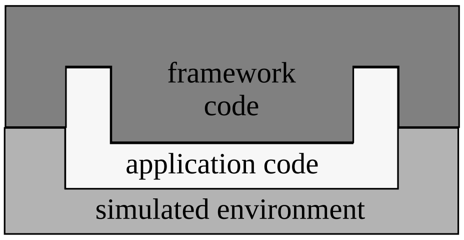

Figure 9. Simulated framework application

4.1 Physical Sensors

Instances of the sensor class in the framework are supposed to be connected to a physical (hardware) sensor that provides measurement data. In a simulated environment, the physical sensor has to be simulated by a physical sensor class. So far, as shown in figure 10, only a few classes have been defined, but this can easily be extended since the interactions between the framework sensor and the physical sensor have been defined.

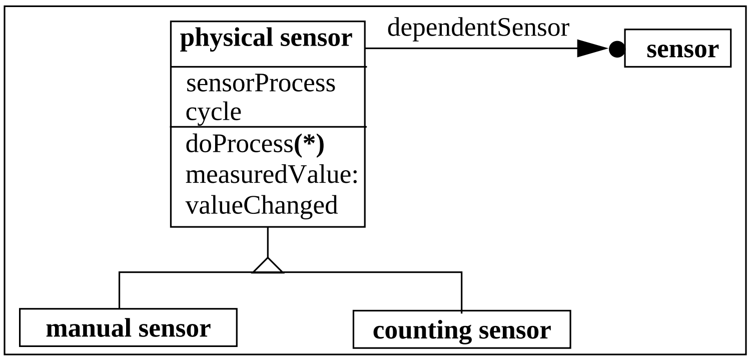

Figure 10. Subclasses of Physical Sensor

The physical sensor class contains a sensor process, a dependent sensor and a cycle. The process is used to represent the independent behaviour of some hardware sensors to generate different data over time. The dependent sensor refers to framework (software) sensor that is associated with the hardware sensor. As described in section 3.3.1, the hardware sensor may notify the software sensor that its value has changed and, in order to do that, the hardware sensor needs a reference to the it. The cycle instance variable indicates the speed of simulation process inside the hardware sensor. Since this process is iterative, the cycle indicates the amount of time to wait before starting the subsequent iteration.

The manual sensor, subclass of hardware sensor, implements a simple testing sensor that allows the user of the simulation environment to test parts of the application by incrementing the value stored in the manual sensor, potentially causing triggers and other computation to occur inside the application. The counting sensor automatically counts and can be used to test the iterative behaviour of the application. The counting sensor contains its own process, whereas the manual sensor is purely reactive and manipulated through the user interface.

4.2 Physical Actuator

Similar to the physical sensor, a physical actuator is connected to a framework (software) actuator, but the connection is only one directional, i.e. from the actuator to the physical actuator, since no communication is required in the opposite direction. The software actuator can actuate the physical actuator through a reference to it.

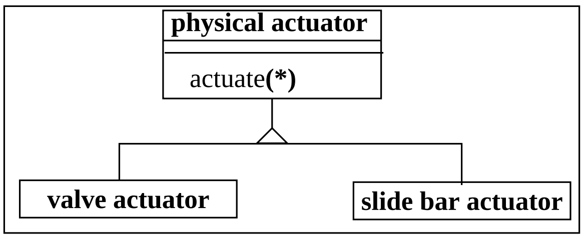

Figure 11. Subclasses of Physical Actuator

As shown in figure 11, also for the physical actuator class counts that only a few classes have been defined. Each physical actuator instance is accessed through the actuate method, defined as an abstract method at the physical actuator super class. Two concrete subclasses have been defined, i.e. valve actuator and slide bar actuator. The valve actuator class simulates a valve and can be in two states, i.e. open and closed, and two active states, i.e. opening and closing. Each actuation causes the valve to change state from closed to open or the opposite. The slide bar actuator simulates extender behaviour of some kind, i.e. on each actuation the extender goes from base position to full extended position and back to the base position.

4.3 Example

For the simulation, a class was defined that could contain a configuration of a measurement system (see figure 12). This configuration consists of the configurable parts of the measurement system, i.e. the sensors and the actuators, and the simulated context of the measurement systems, i.e. the hardware sensors and the hardware actuators. The user can add, delete and edit each of the entities. The link button is used to link a hardware sensor and a software sensor or an actuator to a hardware actuator. When all settings are made, the start button is pressed to instantiate the system, leading to situation as shown in the next picture, where the four shown entities and an instance of the trigger class are presented.

Figure 12. Measurement system configuration tool

For adding and editing entities in the measurement system, a window as shown in figure 13 is used. The shown example is used for adding and editing sensors. Each sensor has a class (selected from the set of subclasses of class Sensor) and two strategies, i.e. an update strategy and a calculation strategy. Each strategy can be selected from the available alternatives.

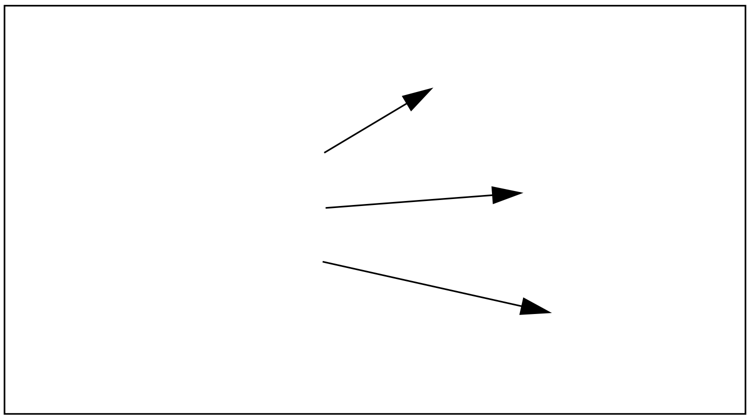

Figure 13. Sensor add and edit window

In figure 14, the result of an instantiation of the example shown above are shown. In this example, the counting sensor has a separate process that counts from 0 to 50. For every increment, the sensor is notified. Since the sensor has a OnChangeUpdate strategy, the sensor will update its own value representation by reading the value of the counting sensor. When the sensor updates itself, the trigger is notified. Depending on the update strategy of the trigger, the trigger either reacts on the notification or reads the sensor in a periodic manner. The trigger's reaction consists of reading the value of the sensor, deciding whether the value justifies a trigger and, if it does, trigger the item factory. The item factory, in response, creates a copy of the prototype measurement item that it contains and starts the measurement item. The measurement item reads the sensors it is connected to via its measurement value instances, performs a transformation of the acquired data and, depending on its actuation strategy, may actuate one or more of the actuators it has stored references to. Each actuator will, in reaction, actuate the hardware actuator it is connected to; in this case the valve actuator. The valve actuator will change state from closed to open or visa versa for each actuation. On each actuation, the actuator enters the active state, i.e. opening or closing, and goes through a iterative process 'opening' or 'closing' the slider bar in 10 steps.

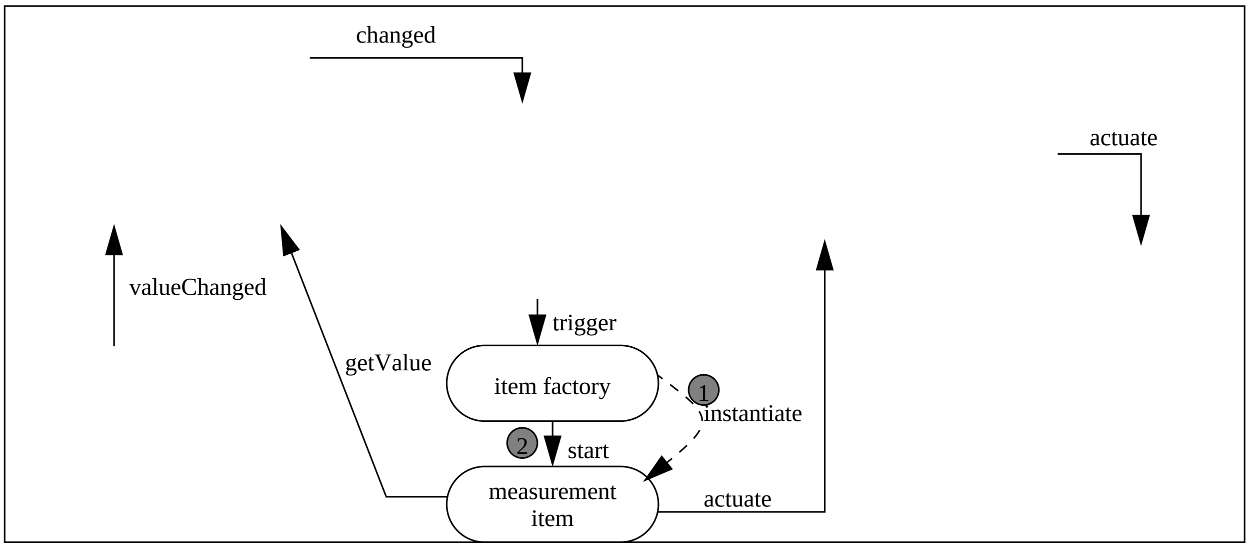

Figure 14. Example measurement system simulation

5 Example: Beer Can System

In this section, we develop a simple example measurement system that illustrates the use of the measurement system framework.

The beer can system is placed at the entrance of a beer can filling factory and its goal is to remove beer cans from the input stream that are not clean, i.e. cans that contain some dirt of some kind. Clean cans should just pass the system without any further action. To achieve this the measurement system consists of a triggering sensor, a camera and an actuator that can remove cans from the conveyer belt. When a can enters the measurement system the system receives a trigger event from the trigger in the hardware. After some amount of time, the camera will read a sample input from which only a single picture line is returned. This sampling is repeated a few times and subsequently are the measured values compared to the ideal values and a decision about removing or not removing the can is made. If the can should be removed, the actuator is invoked at a time point fixed relative to time the trigger event took place and the can is removed from the belt. In figure 15, the process is presented graphically.

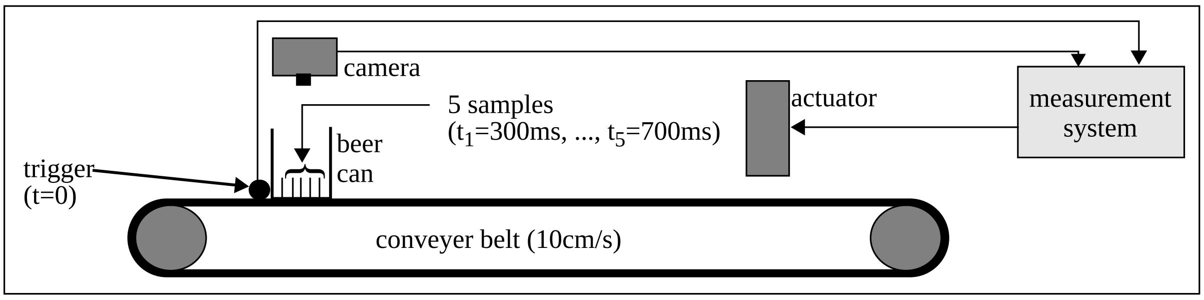

Figure 15. Example beer can measurement system

The camera reads a square area consisting of 256 x 256 pixels and 256 gray tones per pixel. The reason that the camera only returns a single picture line is an implementation issue: dealing with 5 line samples is much less computationally intensive than a complete matrix.

Since we intend to simulate this application, we also have to design the context of the measurement application. The context consists of the two sensors and the actuator. Since the trigger sensor and the camera are related to the same real-world process, these need to interact in order to generate realistic input for the measurement application. Thus when the trigger generates an event, not only the application, but also the camera is to be notified. We assume that the conveyer belt has a constant speed and that the beer cans pass the system with a minimal interval, but not necessarily a constant interval. For the simulation, we assume that the minimal interval between two beer cans is 1.5 seconds. During the first second the beer can passes the system and the subsequent 0.5 seconds cover the empty space until the next can. In order to be able to follow the simulation, the system runs at one tenth of the actual time, i.e. the 1.5 seconds between the cans take 15 seconds in the simulation.

To simulate the software of this measurement system, the first step is to define classes that represent the hardware entities of the measurement system, i.e. the trigger, the camera and the actuator for removing cans from the conveyer belt. Based on these classes, the necessary extensions to the framework can be defined that are needed to construct the application. In the next section, the hardware classes are defined, whereas in section 5.2, the application classes and the configuration of the beer can system are described.

5.1 Physical entity classes

A number of hardware simulating classes are required to construct a realistic simulation of the beer can system, i.e. the physical trigger, the physical camera and the physical actuator. For illustrative purposes, we also defined a clock class that allowed us to relate the occurrence of events to certain time points. In the following sections, the hardware simulating classes are described.

5.1.1 HWBCTrigger

The first class is the physical beer can trigger, HWBCTrigger, which is shown on the right. In reality, the trigger might be a light sensor circuit where the light beam is broken by a beer can entering the system. The physical trigger class shown here has two modes, i.e. manual and automatic. In the manual mode, the user of the simulation can cause a trigger by pressing trigger button. The trigger will pass the trigger event on to the software trigger, the hardware camera (discussed in the next section) and the clock. In the automatic mode, the trigger starts a process that creates a trigger event every 1.5 simulation seconds (i.e. every 15 seconds in reality).

5.1.2 HWBCCamera

The physical camera simulation class is based on a real camera used by EC-Gruppen. The physical camera reads an area, but the processor in the physical camera only takes a single image line from the read area. The image line covers the relevant part of the conveyer belt over the width. Since the camera reads an image line every 100 milliseconds (ms) simulation time, a beer can will be read 10 times since it takes 1 second to pass the camera. Both the camera user interface and the image line principle is shown in figure 16.

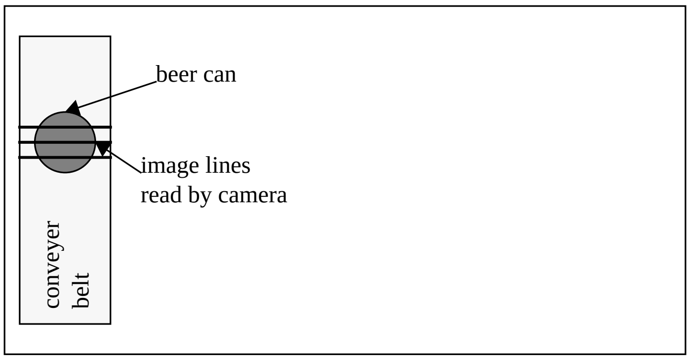

Figure 16. Image lines read by the camera

The physical camera can read and store data in many different ways, but for the use in this system the only relevant aspect is whether an image line is a ‘good’ line or a ‘bad’ line. The ‘good’ line indicates an image line that is read from a clean beer can, whereas a ‘bad’ line is an image line read from a dirty beer can. The camera, when receiving a trigger event (either from the trigger button or from an external source), starts a process that will generate 10 image lines, 100 ms apart. The reading radio button indicates when the camera is active. In reality, the camera would run constantly, but there is no reason to simulate that behaviour since the beer can will pass in one second. The camera can run in three modes, i.e. *only good*, *only bad* and *random*. In the ‘only good’ mode, only good image lines are generated, in the ‘only bad’ mode only bad image lines and in the ‘random’ mode both types of lines are generated using a random number generator. In reality the percentage of dirty beer cans is very little, typically less than one percent or one pro-mille, but the chances in the simulated camera are larger, e.g. 15%, to avoid having to wait for several hours before detecting a dirty can.

5.1.3 HWBCActuator

The physical actuator class simulates a physical device for removing dirty beer cans from the conveyer belt. The device uses a mechanical leg that ‘kicks’ the beer can from the belt into a container. The user interface of the actuator shows when the actuator is passive or active and the slide bar illustrates the mechanical leg.

5.2 Beer can system

When designing the beer can software system which uses the hardware simulating classes described in the previous section, we were very pleased to find that the extensions to the framework required to obtain the system functionality were very limited and located exactly where we intended them during framework design, i.e. the strategies. As we will show later, only two new strategy classes had to be defined, i.e. the calculation and actuation strategy for the measurement item. In addition, we had to write a configuration specification to describe the required objects and their relationships. The configuration specification is shown below.

```
configurationBeerCanSystem
   | mi mv clock |
   hwTrigger := HWBCTrigger new.
   hwCamera := HWBCCamera new.
   hwTrigger hwCamera: hwCamera.
   swTrigger := Trigger new.
   hwTrigger dependentSensor: swTrigger.
   itemFactory := ItemFactory new.
   swTrigger itemFactory: itemFactory.
   swCamera := ConcreteSensor new.
   swCamera calculationStrategy: (ConcreteCalcStrategy new context: swCamera).
   swCamera updateStrategy: (OnChangeUpdateStrategy new context: swCamera).
   swCamera hardwareSensor: hwCamera.
   hwCamera dependentSensor: swCamera.
   mi := itemFactory prototypeItem.
   mi calculationStrategy: (BCMICalcStrategy new context: mi).
   5 timesRepeat: [
      mv := (MeasurementValue new sensor: swCamera).
      mv calculationStrategy: (MICalcStrategy new context: mv).
      mi addMeasurementValue: mv.
   ].
   itemFactory prototypeItem: mi.
   itemFactory actuationStrategy: BCActuationStrategy new.
   hwbcActuator := BCHWActuator new.
   swActuator := Actuator new.
   swActuator hardwareActuator: hwbcActuator.
   mi addActuator: swActuator.
   clock := BCClock new.
   hwTrigger clock: clock.
   hwTrigger open. hwCamera open. swTrigger open. itemFactory open.
   swCamera open. hwbcActuator open. swActuator open. clock open.
```

Although the configuration consists of several lines, it is rather short if one bears in mind that it describes the complete beer can system, including the hardware simulation part and the software system. The running simulation system consists of several windows and several concurrent processes. In figure 17, a snapshot of the complete system is shown.

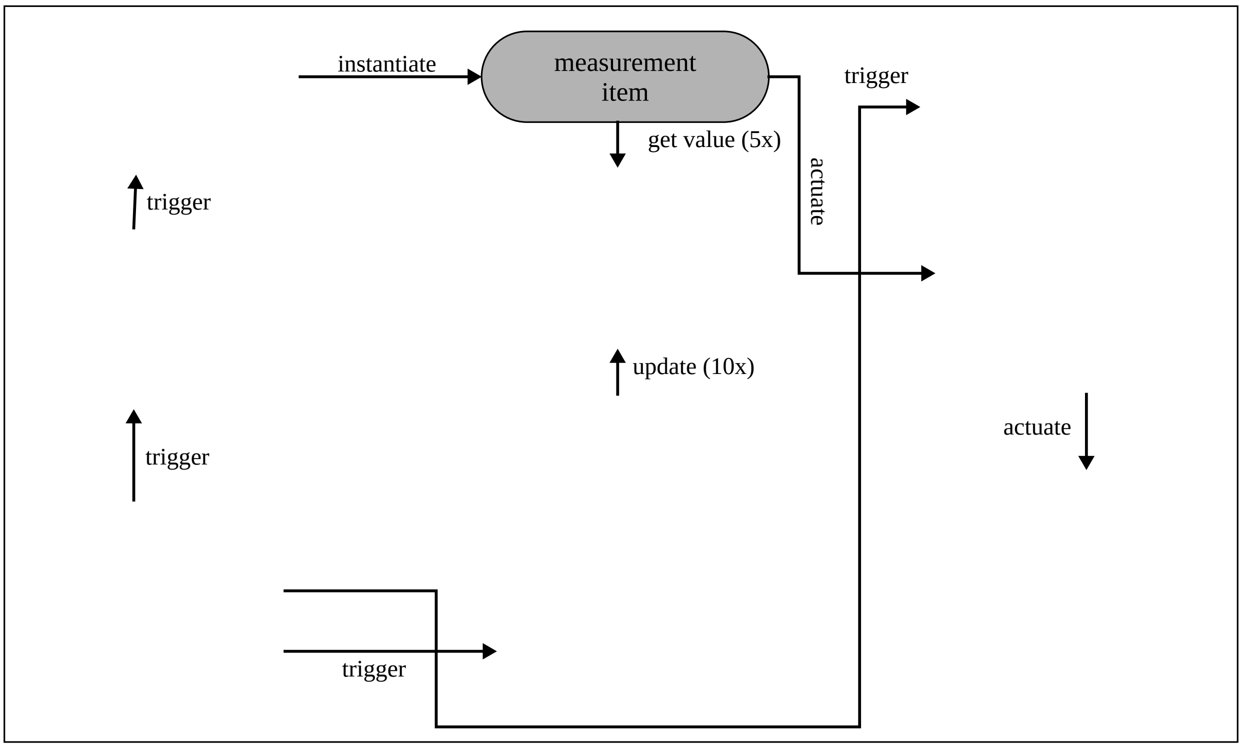

Figure 17. Graphical representation of example beer can measurement system

5.2.1 BCMICalcStrategy

The beer can measurement item calculation strategy (BCMICalcStrategy) contains one method, `performCalculation`, that describes how and when the measurement item should read the sensor. The measurement item contains 5 measurement values that are all connected to the camera sensor. The idea is to read the camera sensor at 5 different points in time, i.e. at 300, 400, ..., 700 milliseconds after the trigger event. The results from these read operations are stored in the measurement values and will subsequently be used by the actuation strategy described in the next section. The code of the `performCalculation` method is shown below.

```
performCalculation
   | st mvs |
   st := context startTime.
   mvs := context measurementValues.
   "MV - 1"
   (Delay untilMilliseconds: st+3000) wait.
   (mvs at: 1) performCalculation.
   "MV - 2"
   (Delay untilMilliseconds: st+4000) wait.
   (mvs at: 2) performCalculation.
   "MV - 3"
   (Delay untilMilliseconds: st+5000) wait.
   (mvs at: 3) performCalculation.
   "MV - 4"
   (Delay untilMilliseconds: st+6000) wait.
   (mvs at: 4) performCalculation.
   "MV - 5"
   (Delay untilMilliseconds: st+7000) wait.
   (mvs at: 5) performCalculation.
   ^self
```

5.2.2 BCActuationStrategy

The beer can actuation strategy class also contains a single method, i.e. `actuate`. The actuation strategy is responsible for determining the appropriate actions by the measurement system, i.e. whether or not to remove the beer can from the conveyer belt. The actuation strategy implemented in this system is that at most one of the read image lines may differ from the ideal image line. If more than one image line is different, the beer can will be removed. The time of actuation should be exactly 1 second after the trigger event, as is shown in the `actuate` method code below.

```
actuate
   | act mvs st counter|
   act := (context actuators) at: 1.
   st := context startTime.
   mvs := context measurementValues.
   "count bad lines"
   counter := 0.
   1 to: 5 do: [ :n | | mv | mv := mvs at: n.
            ((mv measuredValue value at:1) = (mv idealValue at: 1))
               ifFalse: [ counter := counter + 1]
         ].
   (counter > 1) ifTrue: [ act actuate: st + 10000].
   ^self
```

6 Evaluation

Constructing an object-oriented framework from a number of example applications is a challenging activity which is easy to underestimate in its complexity and in the required time. One important problem is that every issue that comes up during the design process has to be evaluated with respect to generality on the one hand and the applicability in real applications at the other hand. In this section, we perform an evaluation from two perspectives, i.e. the lessons learned during the design of this first version of the framework as presented in this paper and, secondly, an evaluation of the conventional object-oriented paradigm in itself. The latter, although perhaps unconventional, is highly relevant for our research since we use the object-oriented paradigm for modelling our applications and reusable components. If the object-oriented paradigm in itself lacks particular features or expressiveness for dealing with certain aspects of the domain, this should result in an adaptation of the object-oriented paradigm.

6.1 Lessons learned

The framework described in this paper is the result of a cooperation between the University of Karlskrona/Ronneby and an industrial organisation EC-Gruppen. From this cooperation and the framework design in itself, we experienced a number of issues that we believe are relevant for similar projects.

- **Research philosophy**: Especially in a research project that is as concrete as the project presented here, the active involvement of the industrial partner is of crucial importance. The traditional view of development of science is that knowledge is developed at universities and then is distributed to industry, i.e. knowledge transfer. The modern view on this process, which we have experienced to be much more accurate, is that knowledge develops in the process of meeting and discussing the topics. Knowledge is developed at both the industrial and the academic side using input from each other.

- **Project size**: Although this has been identified by other authors also, it was definitely valid for this project: Designing a framework is a very complex and time-consuming process that is easily underestimated. The amount of domain knowledge that is required to design a useful framework is much larger than for normal application development. Also a solid understanding of the design space of applications in the domain is very important. When entering the domain of interest as a novice, considerable amounts of time should be directed to studying the domain before a useful design can be made.
- **Boundary**: Designing a framework for a domain may seem like a well-defined task, but in practice we found that drawing the boundary for the framework is extremely difficult. In our early discussions, we constantly were extending the framework since we all agreed that the framework would be even better when it would include yet another feature. We soon realised that the size of the framework would be unmanageable for our project, but prioritizing features is a difficult process.
- **Hands on**: When designing the framework, we experienced it to be very important to experiment with the identified concepts and to prototype the framework. We used Smalltalk for this purpose since it allowed us to rapidly create applications from the framework that also had some graphics associated. In addition, Smalltalk provides processes and has a notion of time, resulting in a more realistic simulated prototype. Anyhow, prototyping the framework during design provides immediate feedback on the usefulness and flexibility of the design.

6.2 Evaluating the object-oriented paradigm

The goal of the project was to create a reusable structure, i.e. an object-oriented framework, that could be used to construct measurement system against less cost than when building them from scratch. The underlying paradigm used for the framework is the object-oriented approach. Although the object-oriented paradigm has several advantages over more traditional paradigms, one also has to critically evaluate the paradigm that one uses. Such evaluations reveal the weaknesses of a paradigm and allow one to deal with them by circumventing them during application development and, whenever possible, to solve the weaknesses by extensions to the paradigm. Although the latter approach is traditionally to be considered to be infeasible, our experience with the layered object model has shown that using an extensible language model with an extensible compiler that generates C++ code is well feasible and provides considerable extended expressiveness against a rather small implementation effort.

During the design of the measurement system framework, we have identified a number of problems. These problems are related to the use of strategies, the calling direction between two objects and the binding of the acquaintances of an object. Each of the problems is discussed in the sections below.

6.2.1 Strategies

In [Gamma et al. 95] the *strategy* design pattern is introduced as a solution to the problem where an application domain contains many similar classes that only differ in the algorithm used by the class. The strategy design pattern separates the algorithm from the class by modelling the algorithm as an independent class. The class that requires the algorithm creates an instance of an algorithm class, stores it as a part a part object and accesses the algorithm by calling its part object.

In the measurement system framework we made extensive use of the strategy pattern. For example, the sensor class contains an update strategy that determines when the sensor updates itself, i.e. when the hardware sensor indicates that something has changed, when a clients calls the sensor or as a periodic process. In addition to update strategies, we have used calculation strategies, describing how to collect data and how to transform it, and actuation strategies that describe how to interpret the accumulated data and how to actuate via the actuators.

The problem that we experienced with the use of strategies is that although a strategy dramatically increases the flexibility of a class, it also dramatically complicates the interactions within the object and between the strategy object and the containing object. The way the strategy pattern is presented in [Gamma et al. 95] is such that the object just calls the strategy object which, in response, just does its thing and returns. However, the strategies that were used in the measurement systems framework were much more interaction oriented in that the strategy had to call its context object to obtain data, references to acquaintances, change state and invoke methods. This created the need for a much more complex interface between the strategy and context object and, in addition, since much internal interaction within the context object takes place via the strategy object, is the understandability of the context object decreased since the functionality of many methods is not so obvious any more.

When analysing this problem, we identified that the strategy design pattern implements one or more methods of the context object in such a way that these methods can easily be replaced by other implementations. However, since the strategy pattern is implemented as a part object, it is not at the same level as the methods of the context object. This creates the complex interaction patterns within the context object. A solution to this problem would be an object model that allows the replacement of individual methods in an object or that manages to integrate the strategy implementation better into the object.

6.2.2 Calling direction

The traditional object communication semantics are that a sender object sends a message to a known receiver object. This is a very clear and simple approach to communication between computational entities such as objects. However, this simple message send may represent many different relations between objects. As we identified in [Bosch 96a], many types of relations can exist between two objects or classes and all these relations are implemented using message passing.

In the design of the measurement system framework, we ran into a problem related to message passing in a number of places. The problem was that, in the situation where two objects had some relation, it was not clear which of the two objects would call the other object. This was particularly clear in two locations in the framework. The first was between the hardware sensor and the sensor objects. In some applications, the hardware sensor will actively invoke the sensor with newly collected data, whereas, in other applications, the sensor will call the hardware sensor and retrieve the most current data from the hardware sensor. Both approaches are equally valid, but which is used depends on the particular application. In order to keep the flexibility to implement the required updating approach, it required that both objects were designed to contain update strategies such that each application could configure the update approach it required. Referring to the problems described in the previous section, this use of strategies is less than ideal.

The second location where the calling direction problem appeared was between the measurement item and the sensor. Depending on the application, either the measurement item invokes on the sensor to collect that data or the sensor pushes the data towards the measurement item. Since we were unwilling to complicate the interaction patterns between the entities in the framework further, we decided to hard code the first approach in the framework, i.e. the measurement item always collects the data from the various sensors rather than the other way around. However, ideally, one would have preferred to have flexibility in that respect also.

As a possible solution, it should be possible to define some type of relation between two objects that abstracts the calling direction between the two objects such that neither of the objects needs to be concerned with this. This issue is also related to the problem discussed in the next section.

6.2.3 Acquaintance handling

Virtually all objects use other objects in the course of their operation. These other objects are referred to as acquaintances of the object. These acquaintances are either bound to the object through the use of global object names (and classes) or through the use of references, of which the updating has to be programmed explicitly. Although this is an accepted way of dealing with acquaintance selection and binding, we found it to be rather troublesome in our design of the framework and, especially, during experimentation with it.

The problem is that all application specifications based on the framework lead to rather large configuration specifications where all objects need to be explicitly bound to other relevant objects; not seldom in both ways. This problem appeared in numerous places in the framework. One example are the strategies: for every strategy instantiated in an application, two bindings have to explicitly specified, i.e. one binding from the context object to the strategy object and one binding in the other direction. Thus, despite the fact that the strategy object is located inside the context object, these objects still need to be configured with references to each other. Another example is that the trigger has to be configured with a reference to the item factory despite the fact that only a single item factory exists within the system, or that the prototype measurement item has to be bound to each sensor and actuator, despite the fact that each sensor and actuator is of a particular type and could be identified very easily. As a result, in the specification of an application, up to two third of the code may consist of binding objects to their acquaintances. The acquaintance handling increases complexity, reduces reusability and requires considerable resources from the programmer.

We study this problem in more detail in [Bosch 96b] where we also propose a solution in the context of the layered object model, but one can deduce that it is important to separate between the specification of the requirements an object has on its acquaintance and the actual selection and binding process. Then, in more advanced systems, the selection and binding process can then be automated, relieving the software engineer from the configuration task.

7 Related Work

The domain of object-oriented frameworks has been studied by many authors, e.g. [Johnson & Foote 88], [Opdyke & Johnson 90], [Pree 94] and [Mattsson 96]. Although some discussion remains, most authors agree that a framework consists of a set of classes that embodies an abstract design for an application or subsystem domain. Several frameworks have been designed for various domains ranging from operation systems [Yokote 92] to user-interfaces [Weinand & Gamma 94]. In this paper, we have been using the advances made in object-oriented frameworks as an input. For example, design patterns [Gamma et al. 95] are becoming increasingly popular as a means to design and describe frameworks [Beck & Johnson 94]. We have used several design patterns throughout the paper to model and explain parts of the framework.

To the best of our knowledge, no other object-oriented frameworks specifically for measurement systems have been defined. [Schmid 95] reports on a framework for manufacturing which deals with much higher level aspects than individual sensors and actuators. [Lea 95] describes a framework for avionics that contains elements as sensors and actuators also present in our framework. However, the processing of data from the sensors and the activation of actuators is clearly different in the two frameworks due to difference in application-domain requirements.

8 Conclusion

The increasing automation of the production process has started to address processes beyond the primary production processes. During the last decade, one can recognise an increasing need for automated tools that support the quality control processes surrounding the actual production. The emergence of the ISO9000 quality standards, the quality thinking in general and the increased productivity of production technology requires the quality control systems to improve productivity as well and whereas many factories used manual quality control by personnel, nowadays the need for automated support is obvious. This development has dramatically increased the need for automated measurement systems. The advantages of measurement systems are generally improved performance/cost ratio and more consistent and accurate quality control. This development increased the needs for reusability of existing measurement system software. As a solution to the lack of reusability, we have introduced and discussed a design for an architecture and a framework for the domain of measurement systems. The presented version of the framework primarily focuses on the main elements of a measurement system, i.e. its architecture, and more concrete classes for e.g. types of hardware sensors and actuators are lacking. However, adding these classes is relatively simple since the interface of the classes and the interactions to the other classes have been defined. Aspects such as real-time behaviour, concurrency and distribution have not been the primary focus in this presentation of the framework, although we acknowledge the importance of these aspects. Instead these will be addressed separately.

Several examples of framework instantiations have been presented in the paper and an extended evaluation of the lessons we learned during the design of the framework was presented. In addition, we analysed the expressiveness of the object-oriented paradigm and identified three situations where the traditional object-oriented paradigm lacks expressiveness, i.e. dealing with strategies, the calling direction between objects and the selection and binding of acquaintances. In our future research on extended object-oriented language models, we intend to address these issues.

Acknowledgements

Many thanks to Anders Kambrin, Anders Dackehed and Sven-Ove Olsson from EC-Gruppen and to Martin Walfisz, Jonas Matton and Magnus Robertsson, master students at the University of Karlskrona/Ronneby, for the many hours of discussion during the design of the framework.

References

[Beck & Johnson 94]. K. Beck, R.E. Johnson, 'Patterns Generate Architecture,' *Proceedings ECOOP'94,* 1994.

[Bosch 96a]. J. Bosch, 'Relations as Object Model Components,' *Journal of Programming Languages,* Vol. 4, pp. 39-61, 1996.

[Bosch 96b]. J. Bosch, 'Object Acquaintance Selection and Binding,' *research report ISRN HKR-RES--96/13--SE,* University of Karlskrona/Ronneby, 1996.

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

[Johnson & Foote 88]. R.E. Johnson, B. Foote, 'Designing Reusable Classes,' *Journal of Object-Oriented Programming,* Vol. 1, No. 2, June 1988.

[Lea 95]. D. Lea, Design Patterns for Avionics Control Systems, *DSSA Adage Project ADAGE-OSW 94-01,* 1995.

[Mattsson 96]. M. Mattsson, 'Object-Oriented Frameworks - A survey of methodological issues,' LU-CS-TR:96-167, *Licentiate thesis,* Lund University, 1996.

[Opdyke & Johnson 90]. W.F. Opdyke, R.E. Johnson, 'Refactoring: An aid in designing object-oriented application frameworks,' *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications,* 1990.

[Pree 94]. W. Pree, 'Meta Patterns - A means for capturing the essential of reusable object-oriented design,' *Proceedings ECOOP'94*, 1994.

[Schmid 95]. H.A. Schmid, 'Creating the Architecture of a Manufacturing Framework by Design Pattern,' *OOPSLA '95,* pp. 370-384, 1995.

[Yokote 92]. Y. Yokote, 'The Apertos Reflective Operating System: The Concept and Its Implementation,' *Proceedings OOPSLA '92,* 1992.

[Weinand & Gamma 94]. A. Weinand, E. Gamma, ET++ - a Portable, Homogenous Class Library and Application Framework, *Proceedings of the UBILAB '94 Conference*, Zurich, September 1994.