Kaleidoscope

A Reference Architecture for Monitoring and Control Systems

Andrea Savigni, Francesco Tisato

DSI Università di Milano

Via Comelico 39/41 20153 Milano, Italy

emails: {savigni|tisato}@dsi.unimi.it

voice: +39 2 55006231

fax: +39 2 55006249

Key words: Software architecture, monitoring and control systems, connectors, strategy, object orientation

Abstract: Although monitoring and control systems can be applied to a great variety of application domains, they exhibit a number of common characteristics, particularly the extensive use of abstraction layers and information streams. This paper presents a reference architecture upon which a number of monitoring and control systems for a wide range of application domains can be designed. The architecture is described in terms of components and connectors, and the UML methodology is employed to specify class diagrams. The architecture is specifically conceived to be made of reusable components; to that aim, a clear separation is made between information components and strategic components, so that the former can be reused under different strategies. *Conceptual images* are information components that model concepts of the application domain, and are specialised in terms of *concrete images*, such as acquisition, processing, and presentation. The major task of the system is to align concrete images, which takes place via transfer of objects (facets) through particular connectors (projectors). This mechanism allows construction of systems where very little is hard-coded at compile time, and a lot is left to configuration, which can usually be performed by a domain expert rather than a software engineer.

1. INTRODUCTION

A software architecture should realise, in terms of architectural components, crucial concepts that are of use to application domain experts.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35563-4_35](https://doi.org/10.1007/978-0-387-35563-4_35)

P. Donohoe (ed.), *Software Architecture*

© IFIP International Federation for Information Processing 1999

First, the architecture must rely on concrete “generic” mechanisms that allow components to be defined and to be composed into a system. Second, both components and composition rules must be specialised to fit specific application domain models. This paper presents a software architecture for monitoring and control systems.

Monitoring and control systems cover a huge range of application areas, from classical process control, to environmental monitoring, urban traffic control, monitoring of historical buildings, and many more. They also share common features, from the point of view of both the domain model and of architectural requirements.

In terms of the application domain model there are two major issues:

1. Physical objects of the environment are represented at several *abstraction layers*, spanning from the field interface layer (sensors and actuators) to higher layers where abstract images of the physical objects live and control-related activities (computation, decision, and presentation) are performed;
2. There are two major *information flows*. The *observation flow* starts from the actual state of physical objects and updates more and more abstract images. The *control flow* starts from the expected state of abstract images and updates more and more concrete images – ultimately, it controls the state of the physical objects.

This scheme is quite general, and accommodates particular cases. For instance, in a pure monitoring system the observation flow only exists, whereas in an open-loop control system the control flow only exists.

In terms of concrete architecture, there are two major requirements:

1. modularity and configurability: the domain engineer must be capable of building, configuring and managing his or her system by exploiting reusable components that model domain-level concepts, without having to deal with implementation-related concerns
2. behaviour: the domain engineer must also be capable of defining the dynamic behaviour of the system – in particular, of specifying the timing of the observation and control flows – without knowing the internal structure of the components nor the idiosyncrasies of the technological platforms.

The Kaleidoscope architecture we present in this paper is a general framework (Gamma, 1995) that attempts to meet the above requirements of monitoring and control systems. The key issues, described in detail in the rest of the paper, are the following:

- Application domain entities are modeled by *conceptual images*.
- A conceptual image is realised by a set of *concrete images*, possibly hosted by different physical nodes.

- The alignment of information between concrete images is performed by *projectors*.
- The overall behaviour is defined by *strategic¹ components* that drive the projectors.

The major advantage of the Kaleidoscope architecture is that not only does it provide a sound basis for building a distributed system by composing domain-related components, but it also raises up to the programming-in-the-large level the definition of the strategies that drive the alignment of the concrete images and the execution of computational activities. In fact, individual components do not embed specific strategies, which can be easily defined by the domain engineer according to specific domain requirements.

A detailed description of the architecture is given in parts 2 to 6, and some implementation issues are set forth in part 8. The Unified Modeling Language (Fowler, 1997) (Penker, 1997) is employed to describe modeling and design issues, while the Java Programming Language (Gosling, 1997) has been chosen as the reference implementation language.

2. THE ARCHITECTURAL APPROACH

2.1 Static Architecture

According to (Shaw, 1997), the architecture will be described in terms of *computational components* and *connectors*. Adopting an optical metaphor, a computational component is the *image* of an object in the application domain. Images are defined at two levels.

1. *Conceptual images* model the entities that are meaningful from the point of view of the application domain. A conceptual image defines:
 - a set of *facets* i.e., attributes, that model the information associated with the entity (for example, current value, average, variance, etc.);
 - a set of *filters* i.e., methods, which are responsible for converting the information among different facets.
2. *Concrete images* are subclasses of the conceptual images that can be regarded as views of the latter. They provide concrete representations of an abstract image, according to both application requirements and physical deployment issues.

In general, a concrete image implements a subset of the facets and of the filters defined by the conceptual image; moreover, a conceptual image may not be associated with all the possible types of concrete images. Some

¹ To avoid misunderstandings, we shall use the term *control* in the sense of “process control”, and *strategy* to denote the policies which drive the dynamics of the system from the software architecture point of view.

standard types of concrete images, that will be treated in detail in the following, are:

- acquisition image
- peripheral processing image
- central processing image
- persistent image
- presentation image
- simulation image
- actuation image

In the remainder of this paper, unless explicitly stated, “image” will denote “concrete image.”

The connectors, on their part, play the role of *projectors*, which align the information contents (facets) of the images and encapsulate system-dependent communication issues.

For example, suppose a traffic engineer wants to control the access to a certain area of a town in order to avoid congestion. After installing the physical devices (e.g., photocells and traffic lights), he or she instantiates one acquisition image per photocell (and/or one actuation image per traffic light), plus one peripheral processing image for each device (which will be hosted by a peripheral node). Also instantiated are some images that will reside at the central control room, namely a central processing image, a persistent image that has the goal of permanently storing data, and a presentation image devoted to data visualisation. Moreover, the traffic engineer instantiates the projectors that will be in charge of aligning the proper facets of the images. The resulting static architecture is sketched in *Figure 1*.

Figure 1. The static architecture

2.2 Dynamic Behaviour

Information (in the example, the presence of a vehicle detected by a photocell) will be first acquired in the form of facets local to the acquisition component, then “projected” onto the different images, and “filtered” to obtain more abstract facets (counts, averages, statistics, graphics, and so on). This is the *observation flow*.

At the end of the observation flow, information will eventually reach the user, be it a traffic operator exploiting the presentation image, or an application component relying on the central processing image. The user can monitor data and exercise control by modifying the central image of the traffic lights (e.g., by changing the desired plan of “red” and “green” phases). The changes of the central images are projected and filtered, and eventually reach the actuation images that physically control the traffic lights. This is the *control flow*.

The traffic engineer is in charge of specifying the dynamics of the system. The heart of the dynamic behaviour lies in the updating of the images (which may imply filtering actions). Consequently, the projectors play a major role in the definition of the dynamic behaviour.

A key point of the whole architecture is that *neither images nor projectors embed any activation strategy*: they can be viewed as passive entities (Tisato, 1996). They define data (facets), operations upon them (filters), and alignment mechanisms (projectors), but they are not aware of all the issues regarding *when* to retrieve data, transfer them between images, and filter them. Such issues depend on specific application domain requirements and must be defined by the domain engineer without having to know the internal structure of the individual components (images and projectors). In other words, images and projectors are aware neither of the static structure nor of the dynamic behaviour of the system; this peculiarity can be summarised by saying that the components have “no implicit architecture” (Cazzola, *et al.*, 1998a).

The control of the overall dynamics is up to specific *strategic components* (also named *strategists*; see part 6), which can be selected and parameterised by the domain expert. This way, strategic components are kept strictly separate from images and projectors, and they can be easily reused under different strategies in a different context of the same domain, or in a completely new domain. To summarise:

- Conceptual images model concepts that are meaningful in the application domain.
- Concrete images model specific views of conceptual concepts, which are specialised based on the kind of processing they perform and on their deployment.

- Facets are the actual containers of data.
- Filters model manipulation of the data of a concrete image.
- Projectors align different concrete images of the same conceptual image.
- Strategic components drive the activities of projectors and filters.

3. INFORMATION COMPONENTS

The information components lie at the heart of the system's static architecture, as they are the actual managers of application data. There are three types of information components: conceptual images, concrete images, and facets.

3.1 Conceptual Images

Modeling complex domains requires a proper use of abstraction to make the system manageable. Conceptual images realise a high-level abstraction of reality, defining which aspects of the domain must be taken into consideration. A traffic control example is the concept of “gate”, which captures a flow of vehicles at different levels of abstraction – vehicle presence, vehicle count, average, and so on.

A conceptual image is an abstract class, defining data and methods that will be inherited and implemented by the subclasses (concrete images; see part 3.2); thus, a conceptual image defines the semantics of some monitored and/or controlled data.

3.2 Concrete Images

Conceptual images are abstract classes and, as such, represent abstract concepts; therefore they need to be further specified to be useful. For example, asking if vehicles pass through a gate at a given time makes little sense. What is sensible to ask is how many vehicles passed in the last minute, or how many passed yesterday between 2 and 3 p.m., or how many pass on the average. These different views of the same conceptual image are dealt with separately in the various subclasses called the *concrete images*, and in other classes, associated with these, called *facets* (see part 3.3).

Concrete images are the actualisation of conceptual images. They turn into practice the semantics of the abstract images. More technically, concrete images are subclasses of conceptual images, inheriting a subset of their attributes (i.e., facets; see part 3.3). Their instances are actual software objects deployed to different locations according to their functionality.

Special cases of images are *information sources* and *information sinks*. Sources retrieve data from the outside world and feed them to the system; they provide a “generate” method that, in a system-dependent way, acquires data from physical devices, or generates them itself, typically by simulation. Sinks feed back to the outside world information that was generated by the system; they provide a “use” method that transfers to the outside world the generated data. In this case, too, the way data are output is system-dependent (e.g., graphical report on a workstation, satellite transmission, Web-based diffusion, commands to a physical device, etc.).

Concrete images, albeit architecturally well characterised, do not constrain the designer’s choices as to which functionality each image is to integrate. As an extreme example, a designer might want to incorporate complicated statistical processing into an acquisition image; this would clearly be against the philosophy of the architecture (and good sense too!), but it is reported here just to give an idea of the flexibility of the system.

A basic set of concrete images has been identified, which, in the authors’ opinion, is enough to cover most concrete cases of monitoring and control systems, even if it is always possible for the designer to define new ones, or eliminate some of the existing ones.

- The *acquisition image* is a source image that collects physical data from the outside world. It can be thought of as a device driver of the physical sensor, and resides on the same node. For instance, it may represent the state of a photocell (vehicle passing/not passing). According to the type of sensor, different types of acquisition image may exist, and possibly some device-dependent code will have to be integrated.
- The *peripheral processing image* resides on a peripheral processing node and supports a first level of information abstraction. For instance, it might include a facet modelling a short-time count of vehicles passing in front of a photocell. In the symmetric case of a traffic light, it might include a facet modelling the desired duration of the next “red” or “green” phase.
- The *central processing image* provides an abstract view that supports user interfaces and major computational activities (statistics, optimisation, control decisions), and typically resides on a central processing node. It includes facets suitable for these tasks.
- The *persistent image* is a permanent view filed in a database. It typically includes facets that model timed sequences of primitive data. Once the time series have been stored, all sorts of computations can be performed on them, including statistics, data mining, etc.
- The *presentation image* is both a sink and a source image. As a sink image, it exports to the outer world the data acquired and processed by the system. We purposely use a vague term such as “exporting” because

it makes no assumption as to which medium will be used to communicate the information to the outside world, be it a graphical workstation, a satellite, or the Web. In this respect the proposed architecture is completely open; one may think of a simple Web server that, via CGI or servlets, delivers static snapshots of the system state, or of a complex, three-tier, full-fledged presentation server that in real time delivers the presentation image. Note that the projector-based approach (see part 6) allows the timings related to data transfer towards the presentation image to be decoupled from the timings related to inner data transfers. On the other hand, the presentation image may also be a source image, in that it exports methods that allow the user to exercise control. For example, the presentation image (i.e., the GUI) for a photocell will present the actual number of vehicles passing through a gate, whereas the presentation image for a traffic light will provide a slider to tune the duration of the “red” and “green” phases.

- The *simulation image* is a source image, situated below the peripheral processing image (i.e., at the same level as the acquisition image) or, possibly, below the central processing image. Its goal is to supply data that are not retrieved by direct acquisition. The rationale behind the existence of such an image is twofold. First there is *fault-tolerance*: in case of unavailability of the acquisition image (due to hardware or software failure), the simulator works “on-line” and can feed the system with reasonable data, thus ensuring the continuity of service. Second, there is *decision support*: it is possible to perform what-if analyses, based on the “off-line” execution of the simulator. As we shall explain later, the switch between acquisition mode and simulation mode can take place at run-time in an immediate and transparent manner, by simply activating the proper connector. This way, the system can work in a non-stopping mode.
- The *actuation image* is a sink image whose goal is to translate user commands and decisions taken by the processing components into real actions performed on the domain. Note that both the acquisition image and the actuation image are in fact nothing but drivers of physical devices: sensors and actuators respectively. This symmetry does not exist by chance; it is the expression of a substantial architectural equivalence between monitoring and control. In both cases, information output by the system (monitored data in the one case, commands in the other) is acquired, processed, and finally output by the system in the form of presentation (monitoring) or actuation (control).

3.3 Facets

Facets are the actual containers of data, and as such are the items of interest to the end user. Typical examples of facets are current value, average, and variance. They are the actual realisation, in concrete images, of the attributes found in conceptual images. In UML terms, an image is an aggregate of facets. The choice not to directly include these data into the concrete images may seem strange, but it stems from the intention to keep the architecture as general and flexible as possible. Incorporating the facets in the form of attributes of the concrete images would mean hard coding the semantics of the images. Using the proposed approach, however, the designer is free to include in each concrete image only the facets he or she is interested in. This approach allows insertion of new classes to meet particular requirements (for instance, advanced statistical processing) that obviously cannot be all provided *a priori*; in this case, the designer must simply define the facet classes he/she is interested in and associate them with the concrete images that will utilise them.

As facets are the actual data, it is these objects that are transferred through the projectors when images are aligned (as explained in part 6). Of course, the same facet may have different implementations for different images that may be hosted by different nodes and rely on different platforms. In this case, filters (see part 5) are in charge of performing marshalling. In order to ensure coherence of observations, every facet has a timestamp.

3.4 Aggregate images

What has been presented so far refers to monitoring and control of single, elementary entities. It is nonetheless clear that a general reference architecture must contemplate the possibility of monitoring and controlling images relative to complex entities that model significant domain concepts, but do not correspond to the direct abstraction of physical devices. For example, a traffic operator would like to reason in terms of “urban areas”, that can be observed (in terms of global number of vehicles in them) and controlled (in terms of incoming and outgoing vehicle flows).

In order to model such situations, the concept of *aggregate image* has been introduced, which models, at a higher abstraction level, a set of elementary images. Clearly, it is not possible to associate a physical sensor with a whole zone. Thus, aggregate data cannot be directly derived from information coming from the outside world, but need to be computed from elementary data that are themselves inside the system.

The structure of an aggregate image closely resembles that of elementary images, so there will exist a conceptual image that specialises into concrete

images, and a set of facets that are associated with the various concrete images. The main difference between aggregate and elementary images lies in data acquisition (and, symmetrically, actuation). While elementary images acquire data directly from the outside world through the acquisition image (or by simulation), aggregate images acquire data inside the system, namely from the elementary images the aggregate image refers to. This yields two main consequences:

1. An aggregate image is an aggregate of the elementary images it refers to.
2. An aggregate image is not associated with an acquisition image (nor with an actuation image), since acquisition (actuation) is performed by processing data associated with elementary images.

Figure 2. The relationships among classes

Aggregate images can be put at different levels of abstraction. Suppose that, in order to avoid network overloads, the designer chooses to perform aggregation at the peripheral processing level. The following scenario will occur:

- The aggregate image has no associated acquisition image.
- The peripheral processing aggregate image locally collects data from the peripheral images of the elementary entities it aggregates.
- From this point on there is no more actual difference between an aggregate image and an elementary one.

- It may be the case that elementary entities have no higher-level images (central processing, presentation, and so on).

The schema described so far, and illustrated in *Figure 2*, constitutes what may be considered a design pattern (Gamma, 1995).

4. PROJECTORS

Projectors are in charge of linking images, and aligning the information they host. According to this definition, a projector would seem to be nothing more than a communication channel; on the contrary it corresponds to the concept of a connector (Shaw, 1996), which has a much richer meaning and much deeper implications.

A connector is an abstract entity, completely independent from its actual implementation, that can be mapped onto the most diverse hardware and software communication architectures. Moreover, a connector, beyond its communication capability, can also perform information retrieval, i.e., extraction of information from the components it is connected to. Communication being the main task performed by a connector, a set of communication-related attributes and methods can be collected in a class, that will act as a superclass for all classes implementing connectors. This class is abstract, thus it only serves as a base class.

Projectors are in charge of communication among concrete images. Being specialised kinds of connectors, projectors are themselves abstract entities, independent of their actual implementation. At one extreme, there might be a projector implemented as an almost empty object; at the other extreme, a projector might implement a complex application protocol.

Despite their variety of possible implementations, projectors still retain a set of common characteristics that have to do with their main function i.e., alignment among concrete images. To this aim, they are provided with two standard methods (called respectively “alignUp” and “alignDown”) whose task is to realise the transfer of facets among concrete images, as we shall further explain in part 6. That transfer will take place upstream (alignUp) to realise monitoring and downstream (alignDown) to realise control.

Note that projectors perform an inter-image alignment of uniform facets, i.e., they transfer the state of the same facet from one image to another. In other words, if two images are connected by a projector, the same “projected” facet must be included in both images. No computation or conversion is ever performed by the projector (such conversions are performed by filters; see part 5). For example, both peripheral and central processing images of a photocell contain a “short-term-vehicle-count” image, which is aligned by a projector.

The symmetry between monitoring and control (see part 2.2) is now fully realised: each of these two activities is implemented exactly the same way, by alignment of concrete images. For monitoring (method “alignUp”) data are transferred from sensors to the central parts of the system, for control (method “alignDown”) commands from the user are transferred towards the periphery of the system, and finally to the actuators.

5. FILTERS

Filters are particular components associated and co-resident with, concrete images.

Figure 3. Facets, projectors, and filters

Their task is to transform information between different facets, perhaps performing data conversions if the same facet has different implementations in different images. Just as connectors are responsible for an inter-image alignment of uniform facets, filters are responsible for an intra-image conversion among different facets. For example, as shown in *Figure 3*, the peripheral processing image of a photocell has both a “vehicle-present” and a “short-term-vehicle-count” facet. The former one is aligned according to the corresponding facet of the acquisition image, whereas the latter one is updated by a local filter according to the changes of “vehicle-present”. Accordingly, the central processing image has both a “short-term-vehicle-count” and a “medium-term-average” facet.

In general, filters should be triggered by strategic components (see part 6). In many cases, filters are triggered as a consequence of information transfer performed by projectors, in which case an automatic triggering mechanism can be exploited.

6. STRATEGIC COMPONENTS

6.1 Goals

As pointed out in part 2.2, *strategic components* manage the execution of system activities. They perform the following tasks:

- Decide when to activate acquisition images (through the “generate” method discussed in part 3.2) to acquire data from the physical sensor.
- Decide when to align concrete images activating the appropriate projectors (through the “alignUp” and “alignDown” methods).
- Decide when to invoke filters. In fact, in case of very frequent updates, it may not be efficient to re-compute every time the statistical quantities, as this activity can be complex and time-consuming.
- Decide when to export information to the rest of the world (through the “use” method of the sinks), thus obtaining the visualisation or printing of data, in the case of the presentation image, or the activation of control devices, in the case of actuation images.
- Decide when to re-compute new plans (see part 6.2).
- React to asynchronous events generated by the concrete images (see part 6.3).

6.2 Architecture

The architecture of the strategic subsystem is somewhat orthogonal to the rest of the architecture. In the following we sketch the basic features of the

strategic subsystem of Kaleidoscope. It is organised into a set of *strategic components* linked by *strategic connectors*, according to a strongly hierarchical model in which each layer corresponds to a different abstraction layer, and higher-level strategic components control lower-level ones.

Strategic components are instances of a Strategist class (see (Cazzola, *et al.*, 1998b)) that can be specialised into subclasses. Instances of this class are linked by special kinds of connectors, called *strategic connectors*, which are also subclasses of the generic connector class.

All communication between a strategist and the non-strategic components it manages takes place locally; in other words, the two components reside on the same processing node. This reduces network traffic and allows hard-real-time problems to be managed locally.

Each strategist, in order to perform its task, needs an action plan, i.e., a list of the form <timeInterval, className:methodName>, that specifies the time interval in which a certain method of a given class is to be invoked. These plans can be passed on as parameters to the strategists; *this allows a change of plans at least at configuration-time, or even at run-time*, rather than at compile-time, thus rendering the system much more flexible.

Each strategist has the ability to execute various plans: this way, when faced with events or particular situations, the component can change plan at run-time without the need to stop and restart the system. To that aim, each component is equipped with a “changePlan” method that takes as a parameter the new plan identifier. Of course, a component cannot change plan before the current one has come to a “safe” situation. This yields the necessity of providing plans with breakpoints, i.e., points at which it is safe to interrupt plan execution. Given the hierarchical structure of strategists, the “changePlan” method is invoked by the upper-level strategist.

6.3 Strategies

Since Kaleidoscope wants to be a *general* architecture, it should accommodate both a time-driven and an event-driven model. The authors definitely do not want to be involved in the decade-long event-driven vs. time-driven debate (Tisato, 1995), so the basic architecture makes no assumption as to which triggering philosophy will be chosen. This is possible because *all the non-strategic components, be they images or projectors, are passive*, thus no strategy-related elements are embedded within them. The designer can freely decide to build the strategic subsystem on either a time-driven or an event-driven approach (an example will be provided in part 7).

The specific architecture we proposed in part 6.2 is basically time-driven, and relies on the execution of timed plans. In many cases, plans can be

defined at configuration time. For instance, photocells are sampled at fixed time intervals, the alignments and the decision activities are triggered at fixed intervals too, and the states of the traffic lights are controlled via pre-defined plans.

Asynchronous events can be taken into account either via a “plan-selection”-based policy or via a “plan-formation”-based one. In the former case the strategist reacts to an event by choosing a plan among the pre-defined ones. In the latter case it defines a new plan. Once again, the designer is completely free to choose whatever policy he or she prefers. As a guideline, the authors suggest that a default “emergency” plan be activated upon receipt of an alarm. Such a plan has the goal of governing the system in a consistent and safe, if not optimal, way. While the default plan is executing, it is always possible to re-compute, as a non-critical background process, a new optimal plan.

7. DYNAMIC BEHAVIOUR: SCENARIOS

Let us now look at a possible execution scenario, referred to as the traffic control example. The scenario is sketched in *Figure 4*, which highlights (as thin dotted arrows) the triggers of the actions, not the underlying strategy. For simplicity, the figure shows some generic “strategists” and does not consider the allocation of components to nodes, even if the strategist shown corresponds to a reasonable allocation.

The execution scenario depicted in the figure proceeds as follows:

- a) A strategist decides that it’s time to acquire data and calls the method “generate” of the photocells acquisition image, which gets information from the physical sensor and inserts them into the appropriate facet (“vehicle-present”).
- b) A strategist decides to align the “vehicle-present” facet of the photocells peripheral processing image, by calling the “alignUp” method of the appropriate projector. Note that this activity might have a timing that is different from the previous one – and might be controlled by a different strategist. This may be necessary to face the case of heterogeneous hardware and software devices, which are very likely to have very different updating intervals and modes.
- c) A strategist triggers the intra-image filtering that updates the “short-term-vehicle-count” facet of the peripheral processing image, as explained in part 5.
- d) A strategist decides to align the “short-term-vehicle-count” facet of the central processing image.

- e) A strategist triggers the filter that updates the “medium-term-average” facet of the central processing image.

Figure 4. Execution scenario

- f) A strategist decides to align the “medium-term-average” facet of the presentation image, which is presented to the user (be it a human being or a process control application).
- g) Eventually the user changes the “desired plan” facet of the traffic light presentation image, and a strategist decides to align the “desired plan” facet of the traffic light central processing image.
- h) A strategist triggers the filter that updates the “red/green duration” of the central processing image.
- i) After that, the control flow proceeds till the physical traffic light is changed.

As pointed out before, different triggers may have different timings, according to specific application requirements. Timings are defined by strategists, and neither images nor projectors are aware of them. Several scenarios can be devised.

- fully time-driven: all the actions of both connectors and filters are triggered by the strategists on the basis of timed action plans (see part 6.2). This is the scenario for a traffic control system oriented to medium-term optimisation, where sampling the system state and tasking decisions at a fixed rate is a major requirement.
- “upward” event-driven: the actions are triggered by the strategists as a consequence of events that asynchronously notify a state change of some acquisition image. This is the scenario for a system that must manage alarm conditions (incidents and the like).
- “downward” event-driven: the actions are triggered as a consequence of events generated by a presentation image – i.e., by a user. This is the scenario for a system that allows an operator to freely observe and control the traffic system.

Of course, mixed strategies can be conceived. We note that all the strategies can be implemented by changing the strategic components only, without any change to the re-usable components, i.e., images and projectors.

Simulation can play an important role in making the system reliable and in ensuring continuity of service. Should a physical sensor break, the strategic component can activate a simulation image and trigger a projector that gets information from the simulation image rather than from the acquisition one.

8. IMPLEMENTATION ISSUES

In part 2.1 we said that a concrete image inherits methods from its base class “conceptual image,” while attributes are rendered as separate objects, namely facets. Note that the concept of “inheritance” we are referring to in

this context is a very abstract one, meaning that, at a very high abstraction level, concrete images share the semantics introduced by a conceptual image. Obviously, this does not mean that concrete images must be implemented as actual subclasses of conceptual images; in fact, we already stated that, in order to implement attributes of the conceptual image, facets are used, which are not attributes in the usual sense.

Another issue relative to facets is that when the strategist commands an alignment, the projectors retrieve the appropriate facets from within images. In general, this implies marshalling, therefore a projector must invoke a filter. In order to make such an approach possible, each concrete image must maintain a data structure (presumably a hash table or something similar) containing the symbolic names and the addresses of the facets it includes. Moreover, each filter is equipped with a method, called “read”, which is called by the projector, retrieves from the concrete image the reference to the proper facet, instantiates a copy of the facet, and passes it on to the projector. A “write” method of the filter performs symmetric actions.

This procedure requires rather sophisticated mechanisms to avoid unpleasant situations, such as having to hard-code in the application code the mechanisms to recognise the kind of facet the projector is carrying. Such hard-coding would cause the resulting code to be non-extensible and not adaptable to new situations, which would be against the aims of the architecture. Vice-versa, one can think of using mechanisms such as computational reflection (Maes, 1987) or run-time type investigation in order to retrieve at run-time the type of the facet, and to be able to install a new one of the same type. (Such a mechanism is already implemented in the Java programming language (Gosling, 1997).)

Filters are an interesting example of the role abstraction plays in a software architecture such as this. In fact, at general design level, they are defined as stand-alone classes; however, proceeding further down to detailed design, one realises that a filter may actually be rendered as a distributed object that lies partly within the concrete image it is bound to, partly in the facets, and partly in the connector it is referred to.

As we said in part 2, images, strategic components, and projectors are strongly separate entities. This means that images and projectors never know who is controlling them, while the opposite always holds, i.e., each strategist knows the identity of all the images and projectors it is controlling, and thus knows exactly what kind of notifications can be sent to it. This way, in order to implement default plans, it is sufficient to provide an association table of the form `<notificationCode:planToExecute>`. Upon receipt of a notification, the strategist will adopt the associated plan.

To solve the opposite problem (how can the image know to whom it must issue the notification?), one can think of an “implicit invocation”

architectural style (Shaw, 1996), also known as the “Model-View-Controller” design pattern (Gamma, 1995). The strategist declares itself interested in a certain set of notifications, and these will be delivered to it. (Such mechanism is already implemented in Java (Gosling, 1997), under the name of “observer/observable”).

9. CONCLUSIONS AND FUTURE WORK

This paper presents a software architecture that can be used as a basis for the design of a wide range of monitoring and control systems. The architecture relies on a set of reusable components (*images*), connected by particular connectors (*projectors*), while all strategy-related activities take place in separate components (*strategists*). The proposed approach brings consistent benefits to both software engineers, who can build re-usable components according to the reference architecture, and domain experts, who can build a system without having to be software experts.

The current activity regarding the Kaleidoscope architecture is twofold.

1. A prototype is being implemented in Java to test implementation alternatives and check the soundness of the architecture.
2. A specialisation of the architecture to specific domains is in progress. In particular, the urban traffic domain is being treated; the system is essentially modelled as a set of intersections and arcs, represented as conceptual and then concrete images and by the relative facets. Other conceptual images model meaningful entities such as traffic lights, sensors, and in particular accumulation points, i.e., abstract containers of vehicles that can be recursively defined. Specific presentation images support a Web-based information export to citizens and patrols (police, fire brigade, etc.) deployed on the territory.

The preliminary tests confirm the soundness of the approach.

REFERENCES

- Cazzola, W.; Savigni A.; Sosio, A. and Tisato, F. (1998a), A Fresh Look at Programming-in-the-Large, *Proceedings of The Twenty Second Annual International Computer Software and Application Conference (COMPSAC '98)*, August 1998, Austria, Vienna.
- Cazzola, W.; Savigni, A.; Sosio, A. and Tisato, F. (1998b), Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification, *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering and 6th Reengineering Forum*, March 8-11 1998, Florence, Italy.
- Eriksson, Hans-E. and Penker, M. (1997), *UML Toolkit*, John Wiley & Sons, 1997.
- Fowler, M. and Scott, K. (1997), *UML Distilled: Applying the Standard Object Modeling Technique*, Addison Wesley Object Technology Series, 1997.

- Gamma, E.; Helm, R.; Johnson, R and Vlissides, J. (1995), *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.
- Arnold, K. and Gosling, J. (1997), *The Java Programming Language*, 2nd edition, Addison-Wesley, 1997.
- Maes, P. (1987), Concepts and Experiments in Computational Reflection, *Proceedings OOPSLA '87*, Sigplan Notices, ACM, October 1987.
- Shaw, M. and Garlan, D. (1996), *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- Tisato, F. and DePaoli, F. (1995), On the Duality between Event-Driven and Time-Driven Models, *Proceedings of the 13th IFAC Symposium on Distributed Computer Systems*, Toulouse, France, September 27-29, 1995.
Revised version also published as:
- DePaoli, F. and Tisato, F. (1996), On the complementary nature of event-driven and time-driven models, *Control Engineering Practice - An IFAC Journal*, Elsevier Science, June 1996.
- DePaoli, F.; Tisato, F. and Bellettini, C. (1996), HyperReal: a Modular Control Architecture for RealTime Systems. *Journal of System Architecture*, Vol. 42, N. 67. December 1996.