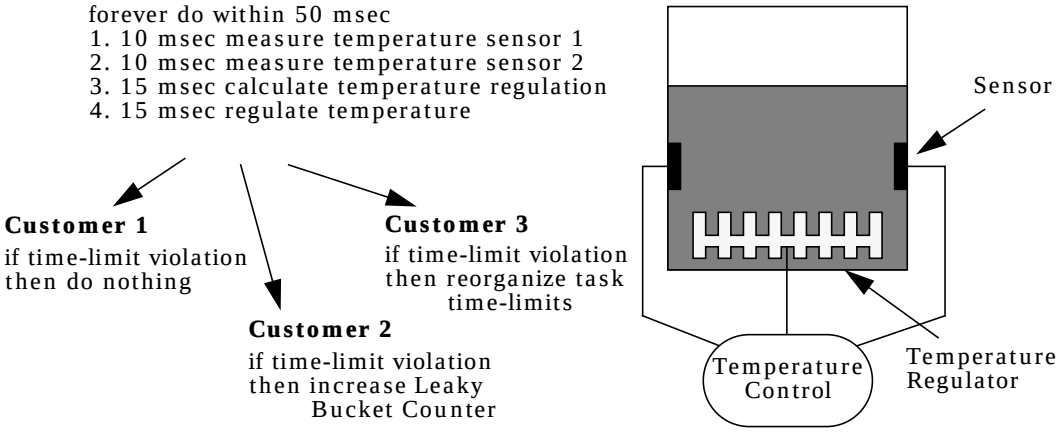
Real-Time Constraints as Strategies

The *Real-time Constraints as Strategies* design pattern decouples real-time specific constraints and behavior from the application service to which they apply. The application service is provided by a service class. Real-time related aspects are delegated to strategies which implement these in a system-specific manner.

Example Consider the temperature control example from the Frame Dispatcher pattern (89). Concrete installations of this system require the integration of different application-specific real-time aspects, like handling deadline misses of the temperature regulation task sequence.



An installation for a household may likely be able to ignore the violation of this time constraint. A more sophisticated system may log deadline misses by increasing Leaky Bucket Counters [PLoP95]. Yet other systems cannot ignore any deadline misses. An example for the latter is an installation for a fermentation tank as part of a pharmaceutical production process at a chemical plant. For a proper fermentation—and thus a successful chemical reaction—it is necessary to keep temperature in the tank on a constant level. Dependent on the product under production, only small deviations can be tolerated. If the temperature regulation sequence processes overly long, however, a constant temperature may not be guaranteed. Action must be tak-

en, ranging from raising an alarm to automatically re-factoring future executions of the task sequence and task-specific deadlines.

Context Configuration of application services with real-time specific aspects.

Problem For many application services in real-time systems it is possible to define implementation skeletons that capture these services' core processing schemes. Different instantiations of the system only differ in their specific real-time constraints and behavior, such as algorithms for task scheduling and handling deadline misses. Three *forces* arise when integrating such real-time specifics:

- The general processing schema of services should not be polluted with system and customer-specific real-time constraints and behavior.
- Varying real-time constraints and behavior should not affect the implementation of a service's general processing schema.
- It should not be possible to accidentally override implementation skeletons of application services when modifying real-time constraints and behavior.

Solution Specialize the Strategy pattern [GHJV95] for separating real-time specifics from the general processing schema of the application service to which they apply. Strategy supports varying real-time aspects independently from the service that depends on them [GHJV95].

Capture the general processing schema for the application service—the context in terms of Strategy—as a template method, according to the Template Method pattern [GHJV95]. This ensures that the general processing schema only contains system-invariant behavior, which also cannot be overridden accidentally [GHJV95].

Define hook methods [Pree95] for all real-time aspects that apply to the service. Use these hook methods in the implementation of the template method—instead of implementing real-time related behavior directly [GHJV95].

Provide the interface for all hook methods in an abstract strategy class, according to the Strategy pattern [GHJV95]. Define specific real-time constraints and behavior by subclassing the strategy class and implementing the hook methods as required. Integrate this

behavior by configuring the template method with the strategy subclass.

Structure A *service class* implements—as a template method [GHJV95]—the invariant skeleton for the real-time sensitive service it offers to clients. Basically, this is the service's *system-invariant* core functionality, abstracting from real-time specifics. Only this invariant behavior is 'hard-coded' in the service class, as if it were no real-time service at all.

For all varying real-time constraints and behavior, the template method uses appropriate hook methods [Pree95]. These hook methods abstract from concrete real-time behavior. By this, the service class becomes independent of application-specific real-time requirements: the execution of concrete real-time behavior is delegated to the hook methods.

Class Service Class	***Collaborator*** • Concrete Strategy
Responsibility • Implements a template method for the service with real-time aspects. • Calls hook methods for real-time specific aspects.	

An *abstract strategy* class defines interfaces for all hook methods that are needed by the service class. These interfaces are commonly shared by all possible implementations of the hook methods.

C*oncrete strategy classes* provide customer-specific implementations of the hook methods as declared by the abstract strategy class. For each different instantiation of the system a separate concrete strategy class exists. At run-time, the service class is configured with a single concrete strategy class, the one that implements the required real-time behavior. When the template method of the service class invokes

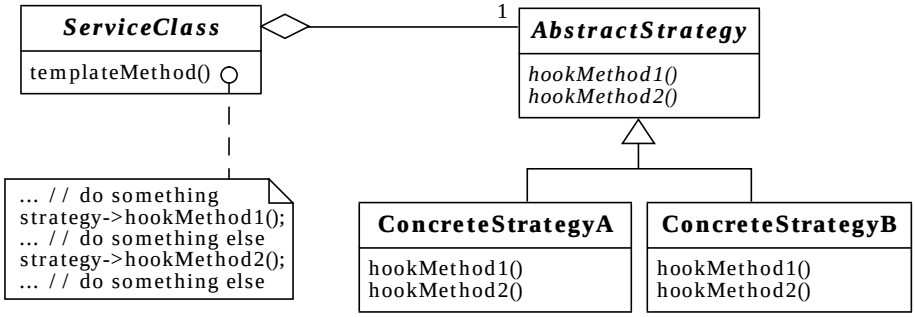
a hook method, the corresponding implementation within the configured concrete strategy class is executed.

Class Abstract Strategy	***Collaborator***
Responsibility • Defines a common interface for hook methods for real-time specific aspects.	

Class Concrete Strategy	***Collaborator***
Responsibility • Implements the hook methods declared by the abstract strategy class in a specific manner.	

Default behavior for hook methods—if any can be specified—can be implemented in a separate concrete strategy class, which is preconfigured with the service class.

The concrete structure of the pattern looks as follows:

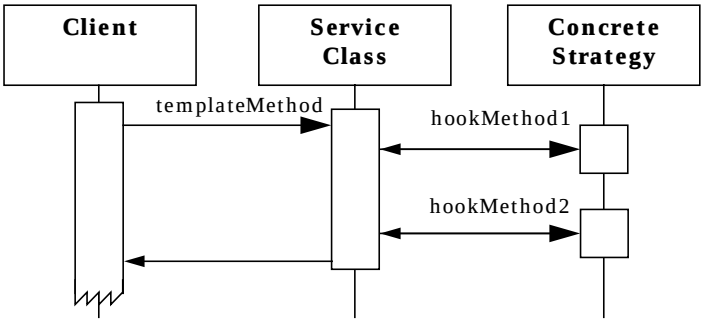


➥ In our example system we specify a service class that offers a single template method. It implements the general processing schema for temperature regulation. The abstract strategy class declares three hook methods. The first hook method provides an interface for handling deadline misses of the whole temperature regulation sequence. The second hook method specifies an interface for handling deadline misses of individual tasks within the sequence. The third hook method, finally, provides an interface for implementing the service's behavior in case the available time for a specific task in the sequence is not completely consumed.

For applications with different real-time requirements for the temperature regulation task sequence we introduce separate concrete strategy classes. Each implements the three hook methods as specified in the system's requirements. ❑

Dynamics The collaborations within the Real-time Constraints as Strategies pattern are relatively simple:

- The client invokes the template method of the service class.
- The template method executes the service it offers.
- For every real-time specific aspect the template method invokes the corresponding hook method. The hook method executes this real-time aspect as implemented in the concrete strategy class that is configured with the service class.



Implementation The implementation of the Real-time Constraints as Strategies pattern comprises 7 steps:

1 *Specify the service's general processing schema*. Identify the invariant behavior of the application service with real-time aspects, according to the guidelines of the Template Method pattern [GHJV95]. In the context of Real-time Constraints as Strategy these are the service's core functionality and the places where application dependent real-time aspects must be considered. The identified general processing schema will form the basis for implementing the service's template method. The places where application-specific real-time aspects must be considered indicate the hook methods.

➥ The functional core of our temperature regulation task sequence comprises three phases. First, the system collects the current temperatures measured by the connected sensors. On basis of these tem-

peratures the system calculates the new setting for the temperature regulator. Finally, the system triggers the temperature regulator to heat or cool accordingly.

While executing the sequence the system measures the time each individual task consumes. If this is more then allowed, or less, we must take a real-time related action. The reason why we handle tasks that finish early is simply that all tasks should begin at a defined point in time. We want all executions of the sequence as equal as possible—at least with respect to task triggering. If the execution of the whole sequence also misses its deadline, we must take another real-time related action.

All three actions described above can vary from instance to instance of the system. Measuring the consumed time is also real-time related, but invariant from the perspective of the service's specification. ❑

2 *Specify the hook methods*. For every real-time specific aspect that can vary in our system—as identified in step 1—we must specify a common interface that fits with all its possible implementations. Each of these interfaces corresponds to a hook method. With this specification at hand, we can declare the abstract strategy class.

➥ For the temperature regulation service in our example system we need three hook methods. Two methods, `handleTaskDeadline()` and `handleSequenceDeadline()`, deal with handling deadline misses for individual tasks in our regulation sequence, or the sequence as a whole respectively. A third hook method, called `idle()`, handles system behavior when an individual task of the sequence consumes less time then potentially allowed. This analysis results in the following `Abstract Strategy` class:

```
class AbstractStrategy {
public:
     // The hook methods
     virtual void handleTaskDeadline(int id) = 0;
     virtual void handleSequenceDeadline() = 0;
     virtual void idle(long idleTime) = 0;
}
```
❑

3 *Define the service class*. This step involves three phases: specifying a mechanism for configuring the class with a concrete strategy object, declaring the interface for the class, and implementing the template methods with help of the hook methods.

➥ In our example system the class `TemperatureRegulation` plays the role of the pattern's service class.

To configure this class with concrete hook methods we provide its constructor with an input parameter of type `AbstractStrategy`. Application programmers must pass the appropriate concrete strategy object as an argument when instantiating a specific `TemperatureRegulation` object.

Class `TemperatureRegulation` further accesses a timer that allows to measure both the execution time for the whole task sequence and the intermediate times for each individual task in the sequence.

Individual task implementations follow the structure introduced by the Frame Dispatcher pattern (89). A common base class `Task` defines a generic interface for task execution. Concrete tasks are implemented by subclassing this base class: a class `TemperatureSensor` for temperature measurement, and a class `TemperatureRegulator` for temperature regulation. Since the temperature regulation task needs the concrete temperatures as its input, both application-specific task classes share a common data repository. For details and a rationale for this design see the implementation section of the Frame Dispatcher pattern (89). For simplicity we assume that there are 10 temperature sensors in the system and one temperature regulator.

From the above discussion we can derive the declaration of class `TemperatureRegulation`. The template method identified in step 1 is called `regulateTemperature()`.

```
const int maxTasks = 11; // 10 sensors, 1 regulator

class TemperatureRegulation {
private:
    // Data members
    Timer*           timer;
    long             limit;
    AbstractStrategy* strategy;

    struct TaskData {
        Task*    task;
        long     limit;
    };
    TaskData tasks[maxTasks];
    int      numTasks;
```

```
public:
    // Constructor and Destructor
    TemperatureRegulation(AbstractStrategy* s);
    ~TemperatureRegulation();

    // Configuring the task sequence
    void setTask(Task* t, int id, long limit);
    void removeTask(int id);

    // The template method
    void regulateTemperature();
}
```

We implement the temperature regulation method according to the Frame Dispatcher pattern (89) and the results from the first implementation step for this pattern. This provides us with a general framework for handling arbitrary periodic task sequences with real-time constraints that follow the schema we identified in step 1.

```
void TemperatureRegulation::regulateTemperature() {
    // run forever
    for (;;) {
        // Reset and start timer for measuring
        // the sequence's exection time
        limit = 0;
        timer->reset();
        timer->start();

        // Execute the tasks
        for (int i = 0; i < numTasks; i++) {
            // Measure task execution time
            timer->startIntermediate();
            tasks[i].task->run();
            timer->stopIntermediate();

            // Calculate the maximum time
            // limit for the partial task sequence
            // executed so far
            limit = limit + tasks[i].limit;

            // RT handling for individual tasks
            if (tasks[i].limit >=
                    timer->getIntermediate())
                // Everything o.k.
                strategy->idle
                    (limit - timer->getTime());
            else
                // OOPS, deadline miss
                strategy->handleTaskDeadline(i);
        };
```

```
            timer->stop();

            // RT handling for the task sequence
            if (limit < timer->getTime())
                // OOPS, deadline miss
                strategy->handleSequenceDeadline();
        };
    };
```
❑

4 *Derive concrete strategy classes*. For every version of the system we derive a separate concrete strategy class from the abstract strategy class. These classes implement the hook methods according to the corresponding application-specific real-time requirements.

➥ For the chemical plant version of our temperature control system we specify a class `ChemicalPlantStrategy`. Let the real-time requirements for the system specify that deadline misses of individual tasks can be ignored as long as they do not occur overly often. Deadline misses of the whole task sequence must be reported and handled immediately.

As a consequence we implement the method `handleTaskDeadline()` such that it increases a Leaky Bucket Counter [PLoP95] for the task that misses its deadline. If the counter reaches its threshold, we raise an alarm. The administrator then can take action, either checking whether the sensor itself causes the faults, or the performance of the software, or whether to re-factor the time limit for the task.

To implement this behavior we define a class `LeakyBucketCounter` as specified in [PLoP95]. The method `handleTaskDeadline()` looks as follows:

```
LeakyBucketCounter* counters[maxTasks];

void ChemicalPlantStrategy::
                    handleTaskDeadline(int id) {
    // Increase counter and check whether
    // it is neccessary to raise an alarm
    if (counters[id]->increase())
        // raise alarm
};
```

The implementation of the hook method `handleSequenceDeadline()` is more straight forward. We always raise an alarm if the whole task sequence misses its deadline. The actions the system administrator can take are similar to those when a deadline miss for an individual task is reported.

For the method `idle()` we also implement task specific Leaky Bucket Counters. The rationale for this is that we want to provide some hints for tuning the performance of the task sequence. If a task finishes early often enough we might reduce its individual time limit. This would allow us to increase time limits of other tasks in the sequence, if they tend to miss their individual deadline often. ❑

5 *Define default behavior for hook methods*, if possible. Default behavior is especially useful when building frameworks: independent of concrete customer-specific specializations the system works in a defined way. Providing default behavior for hook methods involves two steps: specifying the default behavior and pre-configuring it with the service class.

➥ For our example system we can provide default behavior for all three hook methods. In case of the two methods `handleTaskDeadline()` and `handleSequenceDeadline()` we simply log the number of deadline misses that occur for an individual task, or the whole task sequence respectively. The system administrator can check these counts when needed. The method `idle()` idles the time that is passed as an argument. The default behavior is implemented in a class `DefaultStrategy` that is derived from `AbstractStrategy`.

The pre-configuration of class `TemperatureRegulation` with the above default behavior is ensured by modifying its constructor. We change it such that it takes an instance of class `DefaultStrategy` as default argument. If no specific concrete strategy is specified, the default hook method implementations are used.

```
class TemperatureRegulation {
    ...
    TemperatureRegulation(AbstractStrategy* s =
            new DefaultStrategy());
    ...
}
```
❑

If we cannot provide any default behavior, we must *enforce* that application programmers to configure the service class with a concrete strategy. Otherwise the system will crash, since no concrete hook method implementations are available—in our example system there would be a dangling pointer.

➥ The original declaration of class `TemperatureRegulation` already ensures its configuration with a concrete strategy. If no such

strategy is passed as an argument to the constructor, the compiler will report an error. ❑

Another kind of default behavior is to prepare the system such that it can run without considering real-time aspects. In this case we could provide a `NullStrategy`, according to the Null Object pattern [PLoP96], which implements hook methods doing 'nothing'.

6 *Provide support for configuring and changing real-time specific behavior at run-time*. If needed, provide the service class with an additional method for configuring the concrete strategy object with the template method at run-time. Otherwise, the real-time specific behavior can only be configured either at compile-time or even at coding-time, for example to increase performance (see implementation step 7).

Run-time configuration and exchange of real-time specific behavior becomes necessary, for example, if an application supports modes and mode changes. A mode is an operational regime that is defined by a set of real-time operations, such as scheduling parameters and strategies [SRLR89].

➥ The chemical plant version of our system could distinguish between a 'power' mode and a 'regular' mode for temperature regulation. The 'power' mode is used in the beginning of a chemical production process, such as fermentation, when large differences between actual and target temperatures are to overcome. For the 'power' mode we might ignore deadline misses for individual tasks: independently of the current temperature we must heat or cool with high power. When the target temperature is reached, the mode changes to 'regular', introducing a more sensitive handling of deadline misses. ❑

In most real-time systems the configuration or exchange of real-time specific behavior is not in the responsibility of the service component. Rather it is triggered by an external control component, as specified in the Recursive Control architectural pattern [PLoP96]. The reason for this is that often more than one real-time service is affected by such changes. For example, changing modes in a real-time sensitive production process likely requires to re-configure several service components, such as the temperature regulation services installed at different tanks of the plant. Only a central control component is able to maintain all information that is necessary to determine when to change the mode and to perform the change when it is due.

When performing a mode change, the control component calls the reconfiguration methods of all service components that are affected by the change. It passes a new real-time behavior strategy object to the service components which in turn integrate it with their template method. The Service Configurator pattern (77) helps with ensuring that run-time or loading-time changes in real-time behavior are consistent to each other across all affected service components.

7 *Optimize performance*. When implementing the Real-time Constraints as Strategies pattern we also may need to tune its code with respect to performance. This becomes necessary if an application's real-time constraints are so tough that only high-performance code is able to fulfil them.

Note, however, that in general real-time does not mean 'real fast'. It only means that an application must meet certain timing constraints to ensure correct and predictable behavior. Some of these constraints may be easy to meet even by fairly inefficient code. Performance optimization is a completely different issue. It means that some piece of program executes as fast as possible.

How to best tune the performance of the code strongly depends on the programming language used. It is important to note, however, that performance optimizations must not break the core principles of the pattern. Otherwise you will lose its benefits—or you do not implement the pattern at all.

➥ To tune the performance of our example implementation we can use, for example, templates for configuring class `TemperatureRegulation` with a concrete strategy. This allows us to avoid using virtual functions, as in the current implementation. Since virtual functions are expensive, we save execution time. This modification also requires to remove the parameter from the constructor of class `TemperatureRegulation`. We also do not need an abstract strategy class. Instead we provide a set of independent concrete strategies that all implement the same interface.

```
template <class Strategy> class TemperatureRegulation {
private:
    // Data members
    ...
    Strategy strategy;
    ...
```

```
public:
    // Constructor
    TemperatureRegulation();
    ...
    // The template method
    void regulateTemperature();
}
```

The drawback of using templates is, however, that they do not support the re-configuration of the class with a different concrete strategy at run-time. The configuration is carved in stone at coding-time.

Another performance optimization in C++, and thus applicable to our example, is to use inlining. In this case, the code of the inline method replaces all calls to it, thus increasing the performance of the system. The downside of inlining is that the code size grows, because all method calls are replaced by the full method bodies. Also, how well inlining works depends on the compiler you are using. Inlining in C++ is a *hint* for the compiler, not a *must*! Different compilers also vary in their inlining capabilities: a method that is inlined with one compiler may not be accepted by another. ❑

Variants *Real-time Constraints as Hooks*: In this variant, a specialization of the Template Method pattern [GHJV95], the hook methods are declared in the service class, instead of within an abstract strategy. Integration of specific real-time constraints and behavior is achieved by subclassing the service class and implementing the hook methods, instead of providing a hierarchy of strategy classes.

This solution has the advantage that template and hook methods are declared and implemented in one class hierarchy, rather than being separated from each other. This is easier to implement. The drawback is that the Real-time Constraints as Hooks variant allows no run-time re-configuration of real-time specific behavior; at least not without modifications in the clients that use the service class.

If the overall performance of the structure is not a barrier for fulfilling the real-time constraints of the application service, and if no run-time re-configuration of real-time specific behavior is needed, this variant is, however, an option worth considering.

Known Uses **TAO**, **T**he **A**CE **O**RB [POSA3]. The ACE ORB is a configurable high-performance real-time object request broker. It is built on top of the

ACE framework [Sch97]. TAO uses Real-Time Constraints as Strategies to create a scheduling framework which uses the unifying notion of *urgency* to provide a consistent interface to scheduling and dispatching real-time operations. Concrete strategies allow pluggable mappings from real-time operation characteristics into urgency values.

All decisions concerning the pairwise ordering of real-time operations and the off-line and run-time assignment of urgency values are factored out into the individual scheduling strategies. Examples include rate monotonic scheduling (RMS) for statically schedulable rate-based systems, and earliest deadline first (EDF), minimum latency first (MLF), and maximum urgency first (MUF) for systems requiring dynamic scheduling.

```
class ACE_Scheduler_Strategy {
// Abstract Base Class for scheduling strategies: each
// derived class must define an ordering strategy for
// dispatch entries based on a specific
// scheduling algorithm.
public:
     ACE_Scheduler_Strategy
          (ACE_Scheduler_Strategy::Preemption_Priority
           minimum_critical_priority = 0);

     // comparison of two dispatch entries in strategy
     // specific high to low priority
     virtual int priority_comp
          (const Dispatch_Entry &first_entry,
           const Dispatch_Entry &second_entry) = 0;

     // sort the dispatch entry link pointer array
     // according to the specific sort order defined
     // by the strategy
     virtual void sort (Dispatch_Entry **dispatch_entries,
                        u_int count) = 0;

     // determine the minimum critical priority number
     virtual ACE_Scheduler::Preemption_Priority
                       minimum_critical_priority ();

     // comparison of two dispatch entries in strategy
     // specific high to low dynamic subpriority ordering
     virtual int dynamic_subpriority_comp
          (const Dispatch_Entry &first_entry,
           const Dispatch_Entry &second_entry) = 0;
```

```
        // returns a dynamic subpriority value
        // for the given timeline entry at the current time
        virtual long dynamic_subpriority
                (Dispatch_Entry &entry,
                 u_long current_time) = 0;

        // provide a lowest level ordering
        virtual int static_subpriority_comp
            (const Dispatch_Entry &first_entry,
             const Dispatch_Entry &second_entry);

    protected:
        // comparison of two dispatch entries
        int sort_comp (const Dispatch_Entry &first_entry,
                       const Dispatch_Entry &second_entry);

        // the minimum critical priority number
        ACE_Scheduler::Preemption_Priority
                          minimum_critical_priority_;
    };

    class ACE_RMS_Scheduler_Strategy :
            public ACE_Scheduler_Strategy { .... };

    class ACE_MUF_Scheduler_Strategy :
            public ACE_Scheduler_Strategy { .... };

    class ACE_MLF_Scheduler_Strategy :
            public ACE_Scheduler_Strategy { .... };

    class ACE_EDF_Scheduler_Strategy :
            public ACE_Scheduler_Strategy { .... };
```

The scheduling framework then works with the total ordering provided by the urgency values assigned to the real-time operations by the scheduling strategies.

```
    class ACE_Strategy_Scheduler : public ACE_Scheduler {
    // Strategized scheduler implementation.
    public:
        ACE_Strategy_Scheduler
                (ACE_Scheduler_Strategy &strategy);
        virtual ~ACE_Strategy_Scheduler ();
        // assigns priorities to the sorted dispatch
        // schedule, according to the strategy's
        // priority comparison operator.
        status_t assign_priorities
            (Dispatch_Entry **dispatches, u_int count);
```

```
        // assigns dynamic and static sub-priorities to
        // the sorted dispatch schedule, according to
        // the strategy's subpriority comparisons.
        status_t assign_subpriorities
            (Dispatch_Entry **dispatches, u_int count);

        // determine the minimum critical priority number
        virtual Preemption_Priority
            minimum_critical_priority ();

    private:
        // chedules a dispatch entry into the timeline
        // being created
        virtual status_t schedule_timeline_entry
            (Dispatch_Entry &dispatch_entry);

        // sets up the schedule in the order generated
        // by the strategy
        virtual status_t sort_dispatches
            (Dispatch_Entry **dispatches, u_int count);

        // strategy for comparison, sorting of
        // dispatch entries
        ACE_Scheduler_Strategy &strategy_;

        ACE_UNIMPLEMENTED_FUNC (ACE_Strategy_Scheduler
            (const ACE_Strategy_Scheduler &))
        ACE_UNIMPLEMENTED_FUNC (ACE_Strategy_Scheduler
            &operator= (const ACE_Strategy_Scheduler &))
    };
```

REFORM, a framework for hot steel rolling mills [Schu97]. This system uses Real-time Constraints as Strategies to separate functionality from real-time specific behavior. Real-time aspects that are implemented as strategies include setting deadlines and handling deadline misses.

One real-time sensitive task within the system is measurement value acquisition. To ensure a correct material tracking, state data must collected from the rolling mill plant on a periodical basis. If not specified otherwise, the system is configured with default behavior: a time-frame of 200 msec for a complete data update with ignoring deadline misses. For most instantiations of the framework these settings are sufficient, since the real-time requirements for this task are soft, in general. Working with outdated values for one or two time periods doesn't affect the quality of the system's calculations, so they can be tolerated.

Few customers, however, require more stringent real-time behavior, specifically for handling deadline misses. For these customers, specific strategies must be implemented and configured with the system.

Other real-time sensitive tasks within the hot rolling mill framework are material tracking and setpoint transmission. The latter controls the actual stands, rolls, and other devices in the system. In contrast to measurement value acquisition, however, deadlines for these tasks are more stringent, since they control and impact the actual manufacturing process, the latter with respect to reliability, safety, and product quality.

For example, customers require a 99,8% reliability of the automation software *by contract*. Besides sophisticated fault tolerance concepts, Quality of Service with respect to real-time contributes to fulfilling this requirement as well. Only a system that finishes all its computations on time can be reliable. This in turn impacts the selection of strategies for handling deadline misses, for example. They must be configurable and adaptable in order to support system tuning the system for a most optimal behavior under the constraints of the actual plant to be controlled. Real-time Constraints as Strategies provides this flexibility.

For above tasks it is also hard to provide default behavior, since deadlines and deadline miss handling depends heavily on the actual configuration of the plant. For example, the wider the distance between two stands or rolls in the plant, the longer the time-frame for calculating aspects like roll force and screw down position. Thus, customer-specific deadline calculation and deadline miss handling strategies must be defined for these tasks.

MFC, a material flow control system, developed as part of the REBOOT project [Kar95]. This system adapts the Real-time constraints as Hooks variant to provide a task scheduler component with different scheduling strategies, in particular earliest deadline first and maximum urgency first.

R2, a meta-level architecture for soft real-time systems [HT92]. In this architecture, real-time specific behavior, such as setting deadlines and handling deadline misses, as well as real-time related protocols, such as time fence protocols and priority inheritance protocols, are

implemented as metaobjects. These are consulted by base-level objects whenever real-time specific actions are to be performed. The real-time metaobjects are strategies which can be dynamically configured with the base-level objects they 'control'.

Consequences The real-time Constraints as Strategies pattern provides several **benefits**:

Separation of concerns. Core functionality of the application service is separated from its real-time specific behavior. Both can be changed almost independently. Also, due to the encapsulation of the real-time specific behavior in a separate class hierarchy, no accidental overriding of the functional core is possible.

Adaptability. The pattern allows for the integration of different implementations of real-time aspects without modifying the service's core processing schema. As a consequence, the structure introduced by the pattern can be configured easily for different concrete applications that offer the same service with the same kinds of varying real-time aspects.

Support for run-time re-configuration. The structure of the pattern allows an application service to be reconfigured with different hook method implementations at run-time. By this, a system can react to changes in its environment or modes in which it runs. Furthermore clients are shielded from the re-configuration since it affects the service class only.

Efficiency. The pattern's structure is—even though it uses indirection and inheritance—open for performance optimizations. These help to meet tight timing constraints, if the code is otherwise not efficient enough; but they do not break the patterns fundamental principles for solving the original problem.

There is one potential **liability**, however:

Hidden communication costs between the service class and concrete strategies. The pattern assumes that developers can identify and specify a common interface for the hook methods that fits with all their implementations. However, this might be hard to achieve. For example, if different implementations need different input parameters, we need a wide interface comprising all these parameters. On the other hand, the service class is unaware of the type of the con-

crete strategy object attached to it. Thus it must create and pass all parameters, independent if a hook method needs them or not. As a result we may run into overhead in space and time.

Using introspective language features for identifying the type of the concrete strategy object and the parameters that need to be passed, such as Run-time type information, is not a considerable option. We would introduce conditional code in the service class, which sweeps away all the benefits of having the strategies. A better solution is to pass the service class object to the strategy using self delegation. This would allow the strategy to directly access the information it needs from the service class, at the cost that service class and strategy are more tightly coupled.

In general, the interface between the service class and the hook methods should be as small as possible, with all parameters shared by all possible hook method implementations. Otherwise, applying Real-time Constraints as Strategies may be inefficient.

See also Real-time constraints as Strategies builds on two Gang-of-Four patterns [GHJV95]: Strategy and Template Method. Strategy forms the foundation for the pattern. In contrast to the 'pure' Strategy pattern, however, the service class—which corresponds to the context class in Strategy—does not just implement an arbitrary service. Rather it implements the general and invariant schema for a certain kind of service, which factors out all possible variations to strategies. This perspective is covered by the Template Method pattern, which provides the guidelines for providing template methods and their corresponding hook methods. Also, Real-time Constraints as Strategies suggests to use a single strategy hierarchy offering a coherent set of real-time functions. The plain Strategy pattern focuses more on reifying a single algorithm, which would result in several strategy hierarchies if applied to Real-time Constraints as Strategies.

Credits We like to thank our colleague Willi Gruber, who provided us with many details on this pattern. Doug Schmidt and Chris Gill provided detailed information about TAO's scheduling framework. Our colleages Thomas Heimke and Christian Schuderer contributed with information about the hot rolling mill framework. Bob Hanner shepherded the EuroPLoP '98 version of Real-time Constraints as Strategies.