Chapter 6. System architecture and planning support prototype

6.1. Introduction
In the previous chapter, we have described and modeled typical planning tasks in the food processing industries. In some respects, the complexity of the planning tasks in food processing industries is not very high. The number of orders is low in comparison to most other industries, there are no or little alternative routings, production lines are often dedicated to product families, and throughput times of production are short. The kind of complexity that can be expressed by the number of alternative plans that exists, is not changed much by changes in the market characteristics we have investigated, the order frequency and lead time. It is aspects that have to do with information availability and allocation of the planning task over multiple persons that makes the planning in the food processing industries intricate, and these aspects are certainly affected by changing the order frequency and lead time. Therefore, our recommendations in Chapter 5 focus on the way that planners deal with events rather than on finding an efficient schedule within the possible alternatives.

In this chapter, we will describe a prototype of a planning support system that enables to continuously react on events. We do not claim that other scheduling support systems do not contain functionality for this. Many planning support systems not only have very strong generation modules, but also have a user-interface that allows all necessary manual manipulations. Still, there are reasons why we propose a prototype whereas existing systems could also be used. First, we have yet to describe how we deal with the reusability requirements that are discussed in Chapter 4. Second, by describing a system without much frills, it is easier to show how the required functionality can be implemented. Third, we show that it is not difficult to make a simple system that implements the most important required functionality from the perspective of task support.

The discussion about the prototype proceeds as follows. Section 6.2 recaptures the reuse requirements that are formulated in Chapter 4. Section 6.3 specifies a system architecture that includes components for reuse. In Section 6.4, we describe a case study and show how the system is configured for this case. Section 6.5 contains the conclusions of this chapter.

6.2. Reuse requirements
In the previous chapters, we have build a list with requirements for planning support from the following perspectives:
• Planning is a task that should be supported (Chapter 3);
• Planning is a hierarchy of constraint setting and assignment decisions (Chapter 4);
• Reaction on events in the food processing industries should take place continuously (Chapter 5).

In Chapter 4, we have also discussed planning support reuse approaches, which resulted in the *transition requirement* for the models that are used. This means that the relations between models must be clear, i.e., the relations between task models, domain models, and support models. For example, an object (which is a component of the domain model) is shown in a window on the screen (which is a component of the support model). The models of the Scheduling Expertise Concept, that are discussed in Section 4.3, satisfy the transition requirement. The combination of (a) reusability of models and (b) the possibility of transition of models to source code, makes it possible to reuse source code. In other words, if we design the source code in a way that transition of domain, task, and application models to source code is possible, then, by virtue of reusability of these models, reuse of source code is also possible.

In Chapter 4, we have chosen to apply a composition based reuse approach, but we also restricted the application domain to the food processing industries. By choosing for composition based reuse, we acknowledge that we deem it impossible to fully describe the application domain. This has implications for the scheduling support system. Somehow, the parts that are expected to be variable should be easily adaptable. Unfortunately, reuse at the programming level has some restrictions. The reuse possibilities are determined by the programming language and development environment. Although (in an object oriented programming language) objects could be programmed that relate to objects from the domain models and subtasks from the task models, this would not make use of libraries of objects that are available in a programming language. The efforts that go into reuse should be balanced with the efforts of constructing a program or parts of a program from scratch. Therefore, the choice is between using existing libraries and forgoing easy transition of models, or emphasizing easy transition, which means a lot of programming effort. The next section, that describes the system architecture, will reckon with this explicitly.

### 6.3. System architecture

#### 6.3.1. Introduction

The system architecture that is depicted in Figure 3.8 (page 55) contains components that relate to *the use of the system*. In this section, we will elaborate the system architecture with components that relate to *reuse aspects of building the system*. The first step towards reuse is to separate code that is stable across applications from code that is specific for an application. We distinguish three layers: a generic layer for all planning systems, a domain layer for systems in a domain (such as the food
processing industries), and a case specific layer (Table 6.1). We will propose a system for which (a) the generic layer source code is usable for all planning problems, (b) the domain layer source code can be used in a domain (such as the food processing industries), and (c) the case specific source code needs to be altered for each case.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Used by</th>
<th>Functionality at the layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>All planning support systems from the perspective of the Scheduling Expertise Concept</td>
<td>Functionality to specify the object model and constraints</td>
</tr>
<tr>
<td>Domain</td>
<td>All planning support systems in a domain (in this research: the food processing industries)</td>
<td>The structure of the object model (e.g., orders, runs, machines, etc); assignment representation (dispatch list, order information interface)</td>
</tr>
<tr>
<td>Case</td>
<td>Planning support system for a specific case</td>
<td>Attributes of objects; calculations of attributes; retrieval of data from external systems; constraints; algorithms</td>
</tr>
</tbody>
</table>

Table 6.1. Layers of specificity

Figure 6.1 shows the components of the system when we take reuse into account.

In comparison to Figure 3.8, three components are added: the domain model specification, class and object administration, and attribute calculation and retrieval. Note that the functionality of these components is of course also available in systems that are based on the architecture in Figure 3.8. But in such systems, this
functionality can be spread across the system and can be hard to change, which impairs reuse. The layer of each component at which we expect it to be stable is listed in Table 6.2.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>Class and object administration; blackboard</td>
</tr>
<tr>
<td>Domain</td>
<td>Domain model specification (structure); editor (user interface components for plan manipulation)</td>
</tr>
<tr>
<td>Case</td>
<td>(attributes); Inspector and evaluator; attribute calculations; generator; external data retrieval</td>
</tr>
</tbody>
</table>

Table 6.2. Components and their layers of specificity

The components that have to do with reusability will be respectively discussed.

6.3.2. Class and object administration and the blackboard

The object oriented modeling paradigm that is used in the SEC provides the opportunity to implement object types in the domain model as classes in object oriented programming languages. The advantage of this would be clarity; the source code is very clear and all kinds of administrative tasks, such as managing object instances, are dealt with by the compiler. There are, however, some major disadvantages. First, a simple change in the object model means that the source code must be recompiled. Second, the domain structure and object characteristics can not be seen by other parts of the program. For example, all parts of the source code where all the attributes are used in a generic way (for example, show all attributes, or save all attributes of an object to the hard disk) should also be modified when a class changes. Because of these disadvantages, we have implemented a set of generic classes that are used to specify the object model and object characteristics during program execution (Table 6.3). These classes and their methods make that the program itself must take care of administration of object instances, but it also means that the program can approach object types and object instances in a generic way. Therefore, these classes satisfy the transition requirements to the implementation level; a domain model can be specified runtime without altering the source code. The data itself is stored in internal memory, and can be saved on and retrieved from disk. The Context class contains functionality to copy schedules. This functionality can be used to make a snapshot of the schedule before each change. In this way, all changes can be tracked back. This implements the blackboard, where multiple (partial) solutions can be stored for later retrieval. How the class and object administration component can be used will be specified next.
<table>
<thead>
<tr>
<th>Class</th>
<th>Function of class</th>
<th>Examples instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseSchedulingClass</td>
<td>Specifies a scheduling class</td>
<td>Order, Machine</td>
</tr>
<tr>
<td>SchedObjectCharacteristic</td>
<td>Specifies a characteristic of a scheduling class</td>
<td>Due date, Capacity</td>
</tr>
<tr>
<td>BaseSchedulingObject</td>
<td>Specifies a scheduling object instance</td>
<td>Order1, Production line 1</td>
</tr>
<tr>
<td>SchedObjectCharacteristicValue</td>
<td>Specifies a value of a characteristic of an object</td>
<td>1-2-2001, 100 kilo/h</td>
</tr>
<tr>
<td>Schedule</td>
<td>Manages the four classes above</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td>Contains a list with schedules; it functions as the blackboard.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.3. Classes of the data model**

### 6.3.3. Domain model specification

With the class and object administration component, a domain model can be specified at run-time. In other words, the structure of the domain model and the attributes of the objects can be altered without changing the parts of the source code that handle the data model. This is done by calling procedures of the classes that are specified in Table 6.3. Figure 6.2 shows a piece of source code that ‘creates’ a domain model.

```plaintext
1 currentContext.addSchedule('food');
2 currentContext.getSchedule('food').newClass('production run');
   currentContext.getSchedule('food').newClass('customer order');
   currentContext.getSchedule('food').newClass('stock order');
3 with currentContext.getSchedule('food').getClass('customer order') do begin
   addCharacteristic('productname');
   addCharacteristic('customer');
   addCharacteristic('amount');
   addCharacteristic('due date');
   addCharacteristic('position in sequence');
   end;
4 currentContext.getSchedule('food').getClass('production run').addClassLink('stock order');
```

**Figure 6.2. Sample code to specify the structure of a domain model**

First, a schedule is added to the context. Then, three object types are added to the schedule. The third excerpt shows how characteristics are added to an object type, and the fourth part shows how two classes are linked, which depicts the structure of a combination object. The excerpts in Figure 6.2 specify the *structure* of the domain.
model. The code in Figure 6.3 shows how an object is created and how values are assigned (the creation of a new order).

```ruby
with currentContext.getSchedule('orderplanning').newObject('order') do
  begin
    setCharacteristicValue('productname', 'chocola reep 2');
    setCharacteristicValue('customer', 'ah');
    setCharacteristicValue('amount', '100');
    setCharacteristicValue('due date', '31-7-2000');
  end;
```

**Figure 6.3.** Sample code to create an object

6.3.4. **Attribute calculation**
The domain model specification determines the attributes or characteristics of objects. From the perspective of the system, there are different kinds of attributes. Many attributes are simply data that belong to an object, for example, the product name. Generic data editing facilities could be implemented for such attributes. Some attributes, however, can be calculated somehow. This is especially the case for combination objects (see Section 4.3.4). For example, the total amount that will be produced in a production run is the sum of the amounts of all orders that are assigned to that run. Such characteristics of objects and the way in which they are calculated are often very specific for a case. This hampers reuse of the data model. For this reason, a specific section in the source code that is case specific handles all calculations and database transactions. A request from anywhere in the program for a characteristic value will trigger procedures in the source code that are case specific. These procedures can be calculations that use other characteristic values, retrieve data from an external database, or a custom data-entry screen. Figure 6.4 shows how the components of the system are used if an attribute value must be shown: the editor needs to show the value of an attribute of an object. It asks the blackboard for that value. The blackboard asks the class and object administration component what kind of attribute it is. If the attribute must be calculated, it calls the function that performs the required calculation. The value is then given to the editor.

The source code in Figure 6.5 shows an example of how an attribute of a combination object is calculated. It calculates the amount of a production order to which a number of (customer or stock) orders are assigned. This function iterates over all linked objects and adds the amounts of each linked object. This code is executed each time this value is needed somewhere in the system, for example, in the user-interface, in another calculation (e.g., to determine the amount of a collection of production orders), or in a constraint checking algorithm. The code “`linkedObject.-getCharacteristicValue('amount');`” can result in calls to other attribute calculations, but these values can also be manually typed in by the user, or be retrieved from a database with orders.
6.3.5. **Inspector and evaluator**

The inspector and evaluator function largely in the same way as the component for attribute calculations. When the system needs to show the constraints, it calls the blackboard, that in turn calls functions that are specifically created for a case. Figure 6.6 shows an example of the determination of a constraint violation (it checks whether the due date of an order is violated or not).
function checkConstraintDueDate(currentObject: TBaseSchedulingObject): String;
var
  plannedDate, dueDate: Date;
begin
  plannedDate := currentObject.getCharacteristicValue('plannedDate');
  dueDate := currentObject.getCharacteristicValue('dueDate');
  if plannedDate <= dueDate
    then result := ''
    else result := 'The order ' + currentObject.getCharacteristicValue('productname') +
        ' is planned on ' + plannedDate +
        ' but the due date is ' + dueDate
  end;

Figure 6.6. Source code to check a constraint (the due date violation of an order)

The evaluator functions in the same way, although it returns a weight on a goal function instead of a yes/no value (for example, the percentage of due date violations of the orders that are assigned in the schedule).

6.3.6. Editor
The interface between the user and the system is established by the editor. The editor contains views; a view shows objects and provides manipulation possibilities on objects. There are four things that can be shown of an object: the attributes, the objects that are assigned to it, the constraint violations, and the goal functions. The most intricate view is how assignments are shown. As discussed in Section 3.4.3 on page 56, there are several views for this. They are highly reusable in a domain, because the structure of the domain model is stable and the variety to show the assignments of a specific domain model is limited (e.g., Gantt-charts, order dispatch list, etc.). Furthermore, most attributes or characteristics of objects can be shown in a generic way (e.g., in a label), so the addition of attributes to a domain model does not alter the way in which the assignments are shown.

6.3.7. Further reuse considerations
The architecture that we proposed implies that the source code must be altered for new applications. Three considerations make the reusability higher and thereby alterations to the source code less likely.

First, the domain structure in Figure 6.2 is created with actual source code. The strength of the class and object administration component is that a domain structure can also be created and altered runtime. The structure can be stored in a configuration file. Then, if the structure alters, the source code does not have to be altered, just the configuration file. This also means that a library of configuration files can be created, so the creation of a new domain structure can be based on selecting a configuration file, altering it, and storing it.

Second, the editor seems highly specific for domains and cases. The variety in domain model structures, however, is likely to be limited. This implies that the
variety in views can also be expected to be limited. A library with views can again delimit the amount of source code that must be altered or added for new applications.

Third, attribute calculations are case specific. But again, a library of calculations can help. Furthermore, attribute calculations are fairly simple and can probably be expressed as mathematical formulas or simple procedures that contain iteration and branching. The system can be equipped with a mathematical parser and scripting language, with which attribute calculations can be specified at run-time instead of in the source code.

The proposed enhancements make the reusability higher, but the initial system becomes more complex. Therefore, these enhancements have not been implemented in the prototype. In addition, we have not dealt with reusability of the generator. On the one hand, the same kind of reasoning (a limited number of domain models and hence a limited number of generators) can be applied to the generator. On the other hand, however, algorithms tend to be highly attuned to case specific details which impairs reuse. Therefore, we have not implemented a generator. The following section will illustrate the various components in a case study.

### 6.4. Case illustration

This section describes a third case study that was performed to demonstrate the components of the prototype. We will first provide a short description of the company. Second, the case will be described with the SEC models. Third, the components that are proposed in the previous section will be made operational by implementing the domain model and the views.

#### 6.4.1. Case description

A manufacturer makes multiple kinds of cookies on a number of different production lines. The factory produces 5 days a week and approximately 10 hours a day. The production process is roughly as follows.

1. Make batter
2. Mold batter (extrude, form, cut)
3. Baking and cooling
4. Package (wrappings, trays, boxes)

There are three kinds of batter for six production lines. At each production line, a number of different (but similar) products are made. Further divergence takes place at the packaging line where a product can be put in many different packaging forms (trays, wrappers, boxes) for different brands. There are two particularities that are important for the planning. First, there are two production lines with moveable machines, so at these lines the configuration can be changed. Second, for some products, the batter must rest for a while. This preprocessing phase can be used as a small buffer (Figure 6.7).
The customers are mainly retail organizations that carry their own brand of cookies. These customers order the same products each week but the amounts differ. The factory also produces some brands of its own for which the products are made to stock. These stock orders are used to fill production runs to an amount that maximizes capacity utilization (i.e., so kettles are filled fully).

There are two people who’s task has to do with production control. Both can carry out all planning tasks but there is some task division; the planners have been assigned their own production lines. Furthermore, one of them does the overall coordination with the production manager. Alongside the planning, the planners deal with many aspects of daily control of the factory, e.g., hiring personnel, labor conditions, and helping in the factory during peaks such as changeovers.

The plan always covers a week (Monday through Friday). This plan is made weekly, and it only takes a few hours to make it. It is made on the Friday prior to the week that the plan applies to. There are approximately one to four orders per production line per day. During the week, all kinds of events take place such as order acceptance and disruptions in production. Orders are accepted by someone from the sales department, and when he suspects problems (e.g., caused by stock positions or capacity considerations), the planners are asked to check if the order can be produced before the due date. Orders that are accepted that do not apply to the current week are not immediately put in a detailed plan but collected until the weekly planning cycle on Friday.

Computer support for the planning task is marginal. A production control system contains the orders and stock positions, which are used as input to make the planning. The planning itself is made in a spreadsheet that is used as editor and with a calculator to convert the number of boxes to weights and the other way around.
6.4.2. Task and domain models

The task model for this case will be based on the model that is depicted in Figure 5.2 on page 99. There are several differences (Figure 6.8).

![Diagram of task model](image)

**Figure 6.8. Task model of planning at the cookie factory**

The determination of capacity and of amounts for production families is not a task of the day-to-day planning. Orders are not grouped to production orders but always assigned directly to production runs. In addition, the assignment of orders to production lines is trivial since the production lines are dedicated to product families without alternatives. In other words, products are always made on the same production line. Sequencing is the most intricate task. It must be determined in what week a run will be made, and then at what day, and then the sequence on the day. The use of patterns in the schedule is high. In other words, many product families are produced on a fixed day in the week. The scheduling task is trivial because the exact starting and ending times of production can be calculated with the sequence and duration of orders. The composition of the run might be changed if it is scheduled and due date violations occur for orders that are in the run. The day is the lowest level of planning for the planner. During the day, the production line supervisors take care that the runs are finished on time. Small deviations from the schedule are handled at the production line, as long as the consequences do not stretch over the day.
Figure 6.9 shows the accompanying domain model. In comparison to Figure 5.3 there are some simplifications because a number of tasks are not carried out. The model does not contain production orders; all orders are directly assigned to production runs.

![Domain model of planning at cookie factory](image)

In the following section, we will lay out the functional design for planning support in this case.

6.4.3. **Functional requirements and functional design**
In this section, we describe for each of the functional requirements for planning support that have been specified in chapters 3, 4, and 5 how it is being applied in the prototype for the case study.

6.4.3.1. **Link to external systems for information collection**
Database transactions are much slower than transactions that are based on the classes in Table 6.3. Still, external systems have the most accurate data. Therefore, there is a trade-off between performance and duplication of data. In our prototype, we only link to a database for information that is likely to change regularly but that is also necessary to check constraints and goals. The most apparent example is stock levels. Other information is only retrieved once, for example the product family that a product belongs to. In this example, a database transaction is needed when a new order is created, but not thereafter.

6.4.3.2. **Show and manipulate the plan**
The editor provides the views on the data; it is used to show and manipulate the plan. It shows attributes of objects and object assignments. The kind of representation that we mostly encountered in the case-studies is a machine dispatch list (Pinedo & Yen, 1997). This list shows for a machine (in this case, a production line) on a given day the orders that must be produced. Because the production lines work independently
and production orders are only assigned to one production line, there is no need to see multiple production lines simultaneously, as is possible in a Gantt-chart. For the prototype, we have chosen to implement the machine dispatch list representation, that shows for a given day and production line the allocated orders, where the orders are grouped in production runs. Furthermore, the editor contains a list of orders that have not yet been assigned. These can be assigned to the chosen day and production line.

6.4.3.3. *Work with multiple alternatives to try solution paths*
Partial solutions can be stored on and retrieved from the blackboard. The blackboard can be used to implement an advanced ‘undo’-function for backtracking. With the use of the blackboard, the planner can work on a solution path, store the partial solution, try a different solution path, conclude that the former path was more promising, and revert to the first. Furthermore, generation algorithms can use the blackboard in the same manner, and the human planner and algorithm can use each others partial plans. We will implement the blackboard as follows. First, the plan is stored on the blackboard after each decision that is made (e.g., changing the size of an order, accepting a customer order, assigning an order to a production run). With this, the planner can undo each decision. Second, the planner can set bookmarks at which a new branch of decisions is started. With this facility, the planner can make his own tree of solution paths. He can always revert to a solution by selecting one on the blackboard, and start a new solution path from there.

6.4.3.4. *Check the plan for errors and evaluate the quality*
The editor shows objects that violate constraints in another color than objects that do not violate constraints. In addition, aggregate objects that have violating objects in their hierarchy are also shown in another color. An explanation of the constraint violation is shown in a separate window. For example, if an order specifies an operation that uses raw material that is not expected to be available at the time of production, the order violates a constraint, and the order, the production run it belongs to, and the day that it is planned at are all shown in another color. This will alert the planner that there is something wrong.

The evaluator provides an assessment of the plan. Evaluation measures are shown real-time to provide feedback to the planner, for example, the number of due date violations and the idle time of production lines. This can help in reaching the goals. Furthermore, if a threshold value on a goal is formulated as a constraint, then real-time feedback on such a goal can avoid constraint violations. Unfortunately, we have not implemented a way to systematically compare plan alternatives on evaluation criteria.

6.4.3.5. *Mixed initiative plan generation support at the subtask level*
The complexity of the scheduling problem in this case study was rather low. Administrative tasks such as collecting information and calculations cost much more time than making the actual assignments. Therefore, generation support does not add
much value in this case. Still, we will provide some guidelines to implement
generation techniques in other cases. We will specify the functional requirements
from the viewpoint of the models in Chapter 4 and 5. According to the functional
requirements that are stated in Section 5.6 on page 121, the planner’s tasks should be
supported by mixed initiative plan generation at the subtask level. In Section 6.4.2,
we have outlined the tasks of the planner. These tasks are described at an aggregate
level. Each of the shown tasks could be further decomposed in smaller subtasks until
the problem solving level. An example of a decomposition of the ‘accept customer
order’ task is shown in Table 6.4.

<table>
<thead>
<tr>
<th>Subtask</th>
<th>Performed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check capacity</td>
<td>Computer</td>
</tr>
<tr>
<td>Calculate the needed raw material</td>
<td>Computer</td>
</tr>
<tr>
<td>Check stock levels of raw material</td>
<td>Computer</td>
</tr>
<tr>
<td>Choose order to postpone if there is enough raw material but not enough capacity</td>
<td>Human / Computer</td>
</tr>
<tr>
<td>Select customer orders that can be postponed without due date violation</td>
<td>Computer</td>
</tr>
<tr>
<td>Select stock orders that can be postponed without the risk of stock-outs</td>
<td>Computer</td>
</tr>
<tr>
<td>Choose the order to postpone</td>
<td>Human</td>
</tr>
</tbody>
</table>

Table 6.4. Example of subtasks of customer order acceptance

In Chapter 3, we discussed three levels that describe the relation between human
controlled versus automated plan generation: the level of human decision making,
the level of solitary search, and the level of mixed initiative. This can be used as a
first categorization to determine the kinds of generation techniques that are
necessary. The first step is to determine for which decisions the human planner does
not need to be bothered with. These decisions (and all decisions that are located
hierarchically lower) can be put in an algorithm that does not need to be able to
communicate about the reasoning process. The second step is to determine for what
decisions the human planner does not want algorithmic support at all. The third step
is to determine the tasks for which the planner wants generative support but also
wants to be able to control the search process. The generation techniques that can be
used at this latter level need to be able to post intermediate results to the blackboard
in a form that is understandable by the planner, so solutions to sub-problems can be
used or altered by the planner. In order to be able to provide adequate generative
support, each subtask should be decomposed until subtasks remain that are either
performed by the human or by the computer. Such a task analysis, however, is
beyond the scope of this research.

6.4.3.6. Support for hierarchical planning
The hierarchical nature of the domain and task models is implemented by making
direct representations of these models in the prototype. Customer and stock orders
are grouped to runs, runs are assigned to a production line, etc. Furthermore, tasks at distinct levels are recognizable as such in the prototype. More specifically, the three main blocks in Figure 6.8 are implemented as shown in the task structure.

6.4.3.7. The effects of a decision on other decisions must be shown real-time
In our decision model, decisions are either imposing constraints on other decisions, or making assignments. A decision can invalidate the constraints that are imposed on it by other decisions. Thus, all planning decisions are somehow related to constraints. One of the functional requirements that has been stated in Chapter 5 is that the effects of a decision on other decisions must be shown real-time. This means that the constraints must be checked and constraint violations must be shown in the editor immediately after a decision, and that an evaluation of the schedule must be made after each change. This is implemented in the prototype without restrictions. The functionality that is described in Section 6.4.3.4 is available continuously. Furthermore, the blackboard allows to navigate through different schedules. If a schedule is chosen, it is immediately checked for errors and evaluated.

6.4.4. Prototype implementation
The three blocks of tasks in Figure 6.8 are separated in two screens. In the first screen, the user can change the maximum amount of production in product families (Figure 6.10).

![Adjust product families](image)

**Figure 6.10.** Determine customer order boundaries

In a similar window, the user can change the minimum and maximum stock levels of product families. The second and third block with tasks are shown in the second screen. There, the user can create customer and stock orders, assign them to production lines and days, and sequence them on days (Figure 6.11). The screen contains three editors (or views on objects): the machine dispatch list, the orders that must be assigned, and an order detail window. Furthermore, the screen shows the blackboard, the inspector, and the schedule evaluation.

Each action of the user (for example dragging an unassigned order to a production run) results in a number of changes on the screen. First, the direct results of the action itself are shown. In the example, the order would disappear from the “unassigned orders” window and appear in the production run it was dropped on. Second, all objects that are currently shown and violate a constraint are drawn in an alternate color. Third, if the selected object (e.g., the production day) violates a
constraint, a message about the constraint violation is shown in the inspector. The constraint violations are calculated after each action. Fourth, the schedule is evaluated again and the new evaluation is shown. Fifth, the action is added to the blackboard.

The screens in Figure 6.10 and Figure 6.11 are of course also linked. If the user changes the maximum amount for a product family, the constraints are directly checked again. In this case, it is an example of a database link. The customer order boundaries are stored externally since they have no representation in our object model. Akin to the example in Figure 6.6, the actual check is not difficult to implement. Because it contains a database transaction, the source code is shown in Figure 6.12. First, the table with product families is read from the database. Second, the amount per product family that is allocated in the schedule is calculated. Third,
the maximum amount is compared to the actual allocated amount and appropriate error messages are created. Such error messages are shown in the inspector in Figure 6.11.

```
function checkConstraintSchedule: String;
var
  curSchedule: TSchedule; curObject2: TBaseSchedulingObject;
i, ind, amount: Integer; error, family: String; prodFamily, prodFamilyMax: TStringList;
begin
  result:=''; error:='';
  if dataModule1.AdoProductFamilies.Active then begin
    prodFamily:=TStringList.Create; prodFamily.Sorted:=True;
    prodFamily.Duplicates:=dupIgnore; prodFamilyMax:=TStringList.Create;
    prodFamilyMax.Sorted:=True; prodFamilyMax.Duplicates:=dupIgnore;
    dataModule1.AdoProductFamilies.First;
    with dataModule1.AdoProductFamilies do
    while not EOF do
    begin
      prodFamilyMax.AddObject(lowercase(fieldByName('productfamily').asString),Pointer(fieldByName('maximumproduction').asInteger));
      Next;
    end;
    curSchedule:=curContext.getSchedule('food');
    for i:=0 to curSchedule.listWithAllObjects.Count-1 do begin
      curObject2:=curSchedule.getObject(curSchedule.listWithAllObjects[i]);
      if (curObject2.myClassName='stock order') or (curObject2.myClassName='customer order') then begin
        amount:=strToIntDef(curObject2.getCharacteristicValue('amount'),0);
        family:=curObject2.getCharacteristicValue('productfamily');
        if prodFamily.indexOf(family)=-1 then prodFamily.addObject(family,pointer(0));
        ind:=prodFamily.indexOf(family);
        prodFamily.Objects[ind]:=Pointer(Integer(prodFamily.Objects[ind])+amount);
      end;
    end;
    for i:=0 to prodFamily.Count-1 do
    begin
      ind:=prodFamilyMax.indexOf(lowercase(prodFamily[i]));
      if ind<>-1 then begin
        if Integer(prodFamily.Objects[i])>Integer(prodFamilyMax.Objects[ind])
        then error:=error+'Product family '+prodFamily[i]+' not within constraints
          ('+inttostr(Integer(prodFamily.Objects[i]))+' allocated; '+inttostr(Integer(prodFamilyMax.Objects[ind]))+' max).'+chr(13)+chr(10);
      end;
    end;
    prodFamily.Free; prodFamilyMax.Free;
  end;
  result:=error;
end;
```

**Figure 6.12.** Source code to check product family boundaries
In Section 6.4.3.1 we mentioned the trade-off between internal and external data storage. An example of duplication of data is shown in Figure 6.13. If a new order is created, the user can choose a product from the database. Instead of retrieving product characteristics from the database each time they are needed, they are duplicated in the domain model.

![Figure 6.13. Add a new order](image)

Note that in this example, the stock level is not duplicated since it changes frequently. Figure 6.14 shows how it is retrieved from the database (and hence is part of the architectural component Attribute Calculation and Retrieval). In this way, the database transactions are shielded from the rest of the program.

```pascal
procedure determineStockorderCurrentStockLevel(curObject: TBaseSchedulingObject);
var
  productName: String;
  curStockLevel: String;
begin
  productName:=curObject.getCharacteristicValue('productname');
  if dataModule1.AdoProducts.Locate('productname',productName,[]) then
    curStockLevel:=dataModule1.AdoProducts.fieldByName('stocklevel').asString
  else curStockLevel:='#not found#';
  curObject.setCharacteristicValue('current stock level',curStockLevel);
end;
```

![Figure 6.14. Retrieve the stock level from the database](image)
6.5. Conclusions

This chapter contains the description of a prototype scheduling support system. The relevance of the prototype system lies in the demonstration of the underlying principles that have been formulated in Chapter 3 (task oriented planning support), Chapter 4 (modeling of planning as hierarchy of constraint setting and assignment decisions), and Chapter 5 (planning and flexibility). These principles can enable an organization to get the most from a planning system by providing guidelines how the organization and task performance can be adjusted to the market requirements, provided adequate planning support. Furthermore, we have shown how reuse can be accomplished by separating components that are stable across cases or application domains. Table 6.5 shows which requirements have been implemented and the source of each requirement.

<table>
<thead>
<tr>
<th>Source of requirement</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application domain (Food processing industries)</td>
<td>Assign orders to production runs and production lines.</td>
</tr>
<tr>
<td></td>
<td>Check the plan for errors</td>
</tr>
<tr>
<td>Task support paradigm</td>
<td>Planning at multiple hierarchical levels.</td>
</tr>
<tr>
<td></td>
<td>Show constraint violations immediately.</td>
</tr>
<tr>
<td>Reuse</td>
<td>Try solution paths with the blackboard.</td>
</tr>
<tr>
<td></td>
<td>A domain independent class and object administration module.</td>
</tr>
<tr>
<td></td>
<td>An architecture with which case specific domain models can be easily implemented.</td>
</tr>
</tbody>
</table>

Table 6.5. Implemented requirements and their source

Many functionalities that could have been implemented have been neglected in the prototype. Most notably, the system can not participate in plan generation, although we have provided the functional requirements for this.

The prototype planning support system that implements our functional requirements is the last part of this research. The following chapter provides a concise retrace of the way in which the research question in Chapter I is answered.