Chapter 6

Evolution of a Reusable Design

In order to make the possible uses of reuse contracts more tangible, in this chapter we describe how reuse contracts were used to develop, specify and assess a reusable design in an industrial context.

6.1 Background: Broadcast Planning

The design that was to be set up was part of a bigger system assisting in the planning of television broadcasting [VV96], [CHSV97]. A framework for broadcast planning already exists. It integrates a whole range of aspects, going from seasonal planning to daily planning, as well as video tape management, programme information management and downloading information to specific transmission hardware. The framework also aims at improving the quality of work by automating work flow and reducing administrative overhead.

The framework was developed by a small software company. While the project originally started with a custom-made application, the framework was developed in order to be able to cope with rapidly evolving market opportunities. Television stations are faced with rapidly evolving hardware (e.g. digital video broadcasting) and rapid evolution of their products (e.g. interactive TV). Another good example was when the original client of the custom-made application opened a second channel, announcing this news to its IT department only one month before the start of the channel.

6.2 The Case: Air-Time Sales

In the original framework commercial blocks are treated as units. They can be placed, removed and rescheduled, but nothing is said about the commercials that are part of each block. This is logical, because the rationale behind planning commercials is completely different from the rationale behind planning TV programmes.

Moreover, both tasks are carried out by different departments. Therefore, a new framework needs to be designed, the "air-time sales" (ATS) framework. This framework needs to handle planning of commercials, contracts with advertisers and pricing schemes. Furthermore, air time sales does not only concern commercials, but also sponsoring of programmes, tv sales, etc. In this chapter, we only discuss a small fraction of this framework: planning of commercials.

For a large part, the ATS-framework works independently from the broadcast planning system, but there is also interaction between the two. For example, when a commercial block is moved, deleted or shortened in the general broadcast system, this has repercussions on the commercials that were planned in it. In the other direction, when not enough commercials are found to fill up a commercial block, this has to be notified to the broadcast planning application. This application then has to fill up the remaining time with trailers or move the following programme forward.

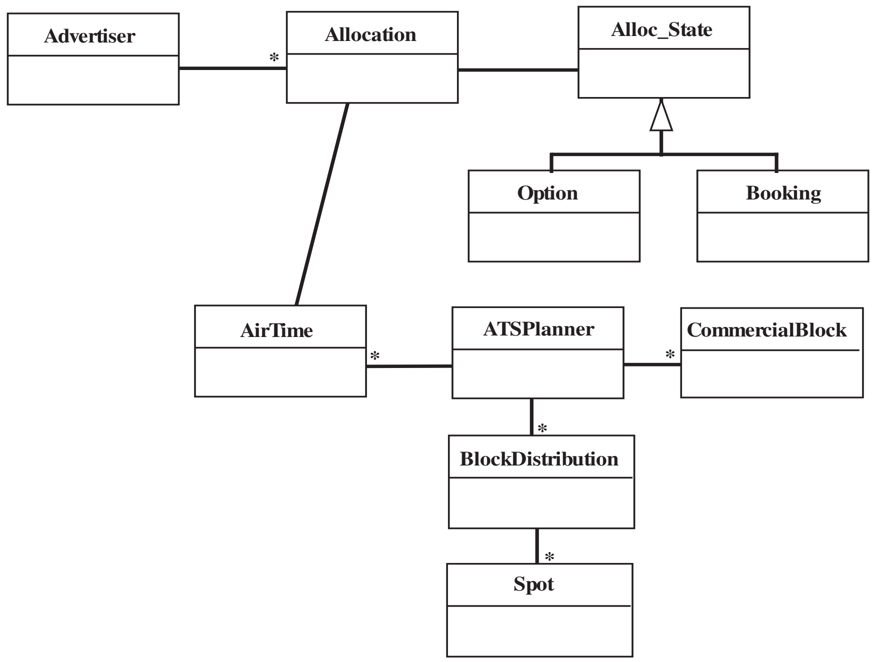

Figure 6.1: Overview of the Air-Time Sales System

The way commercials are planned is quite diverse for different TV stations. The main structure of such systems is however equal; it is depicted in figure 6.1[1]. An advertiser can take an option on air time; this is called an allocation. Air time is

[1]Note that we again use the UML notation. Hence the asterisks to indicate multiplicity zero or more.

the item being sold here. How air time is expressed can be very diverse: advertisers can ask for commercials at particular places in the planning or for a set of spots planned according to a recurring pattern (e.g., every day after the news), etc. After different advertisers have taken options, the planner tries to map these requirements onto actual commercial blocks. When air time is scheduled, the option becomes a booking, a contract can be attached to it, etc. This is depicted by the State design pattern used for the allocations made by advertisers. We will not discuss this part further, since we are most interested in the actual planning.

The heart of the system is the `ATSPlanner`. This planner forms a mapping between `CommercialBlock`s and `BlockDistributions`. `CommercialBlock` is a class from the general broadcast planning application and thus forms the link between both applications. `BlockDistributions` are containers that store which commercials are planned and in what sequence. Therefore a number of `Spots` is attached to a `BlockDistribution`. While an `AirTime` is an abstract description of one or more spots, a `Spot` is a planned commercial.

An ATSPlanner has a number of interesting tasks. The first is to map the air time options to actual commercials, by creating spots in spot distributions. Other behaviour is triggered by the general broadcast planning application. This application can, for example, ask to delete a commercial block, to make it shorter or to move it. How these requests are handled is of course dependent on the kind of air time allocation. We discuss two major cases here: block spot spaces and gross rating points. Moreover, we focus on two basic functionalities: planning of commercials and deletion of commercial blocks. The designs presented below are simplified, but suffice to express the use of reuse contracts.

6.3 A First Design: Block Spot Spaces

A first way of expressing the desired air time is rather basic. An advertiser can take an option on a "block spot space", i.e., a slot in a particular commercial block. The TV station releases weekly plannings describing the programmes and the commercial blocks they have planned. The price of a spot is dependent on its length and on the block the spot is placed in. Commercial blocks right before or after programmes with high viewing figures are more expensive. Based on this information, advertisers can decide where they want to plan commercials. For example, a commercial for a new CD could be planned before or after a music programme.

A reuse contract for the block spot space approach is depicted in figure 6.2. The structure is part of the general structure from figure 6.1. The classes `AirTime`, `ATSPlanner` and `BlockDistribution` are replaced here by specific versions for the block spot space approach, respectively `BlockSpotSpace`, `BSSPlanner` and `BSSDistribution`. Let us look at two main behaviours of the `BSSPlanner` as discussed above: planning and deletion.

A planner can receive the message `planAirTime:` with a particular air time as

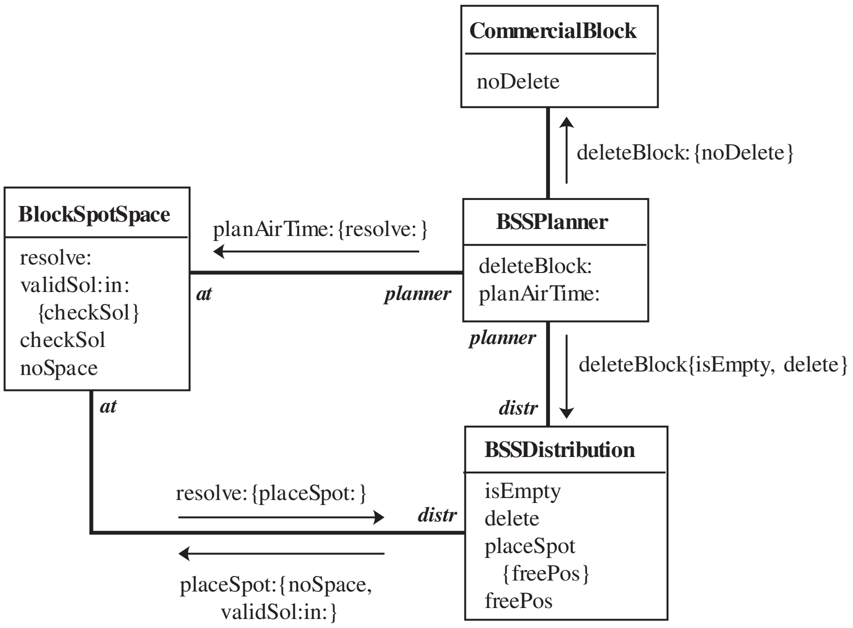

Figure 6.2: Block Spot Spaces

argument[2]. This message expresses the request to make a spot in a distribution for the air time that has been given as argument. As a reaction the planner sends the message `resolve:` to the air time. This message has the planner as argument, because the receiver might need to ask the planner for additional information. As a "block spot space"-allocation knows exactly in which commercial block it wants to be placed, it sends a message `placeSpot:` to the distribution associated with this block. The distribution first checks whether it has a space left for the commercial through `freePos`. When it does not find a free space it sends the message `noSpace` to the block spot space[3]. Otherwise it sends the message `validSol:in:` to the block spot space to check whether the spot it has found is a valid solution. This check is necessary because block spot space allocations might put extra requirements on their request. Different systems might express constraints differently. Possible variations are position constraints (e.g., first commercial in a block), timing constraints (e.g., start within x seconds after the start of the commercial block) or pricing constraints (e.g., the price of this block should not exceed x). We make abstraction of these constraints here and represent them with a call to the message `checkSol`.

Deletion behaviour is straightforward. The policy is that blocks that were already

[2]As this design was set up to be implemented in Smalltalk, we use Smalltalk syntax here.

[3]Note that in a real system one would not send a message `noSpace`, but this would be signalled through the value of the return type. Because we cannot express this with reuse contracts — for now — we use this notation throughout this chapter.

made public and which spots have been appointed to cannot be deleted. Therefore, when receiving the message `delete` a `BSSPlanner` checks whether the associated distribution is empty. If so, it sends the message `delete` to the distribution, if not, it sends the message `noDelete` back to `CommercialBlock`.

6.4 A Second Design: Gross Rating Points

A completely different strategy for air time sales uses so called "gross rating points" (GRPs). Gross rating points are a way to express a desired target audience. An example is that with a certain commercial the advertisers want to reach at least 60% of all males between 16 and 25 years of age. Instead of having advertisers take options on particular commercial blocks, they take options referring to these "gross rating points". It is then up to the broadcast company to distribute the commercials as well as possible, such that the desired audience is reached. The advertisers agree to a certain price for a certain GRP. When, after broadcasting the commercials, viewing figures show that the desired GRP was not reached, the broadcast company has to provide compensation, usually by broadcasting it again for free. Obviously, this working method has a large influence on the implementation of the planning and deletion behaviour. A reuse contract for this version is depicted in figure 6.3.

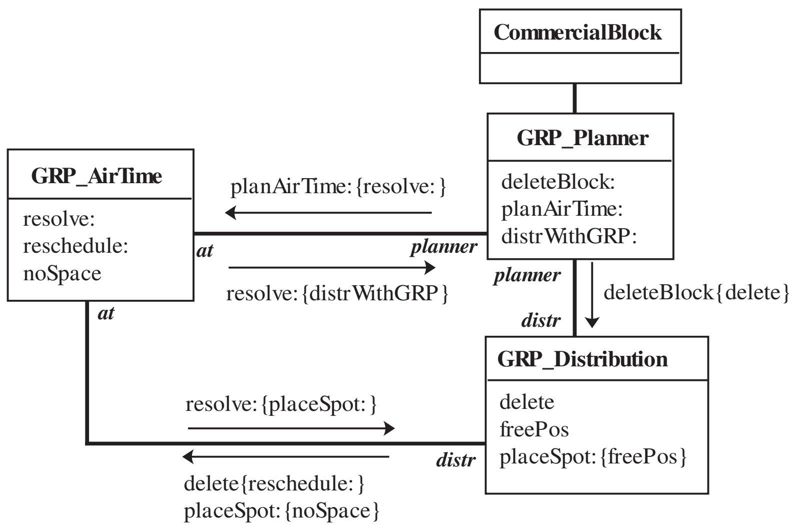

Figure 6.3: Gross Rating Points

As with block spot spaces, when receiving the message `planAirTime:` a `GRP_Planner` sends the message `resolve:` to the air time. A `GRP_Airtime` however first asks the planner to return all distributions with a minimum value on a certain `GRP_category` through the message `distrWithGRP:` and then enumerates over these distributions, asking them to place a spot until the desired GRPs are reached. When asked to place a spot a `GRP_Distribution` checks whether there is a free position left through `freePos`. If there is space left the distribution places the spot, otherwise it returns `noSpace`. It does not need to check the validity of the position where the spot has been placed, as the only requirement of the allocation is to reach the required GRPs and one distribution has a fixed GRP per target group.

Opposed to block spot spaces, deletion of commercial blocks containing spots for `GRP_Airtimes` is possible. If spots were already placed, they can be rescheduled in other commercial blocks amounting for the same GRPs. Therefore, when receiving the message `deleteBlock:` a `GRP_Planner` sends the message `delete` to the associated distribution. If it is not empty, this distribution sends the message `reschedule` to all affected air times before destroying itself.

6.5 Generalisation

Although there are quite some differences between the two approaches, there are also some similarities. It seems logical therefore to abstract these similarities in abstract classes: `ATSPlanner`, `AirTime` and `BlockDistribution`. We can then regard as specialisations the two cases we have so far. A reuse contract of the shared behaviour is depicted in figure 6.4.

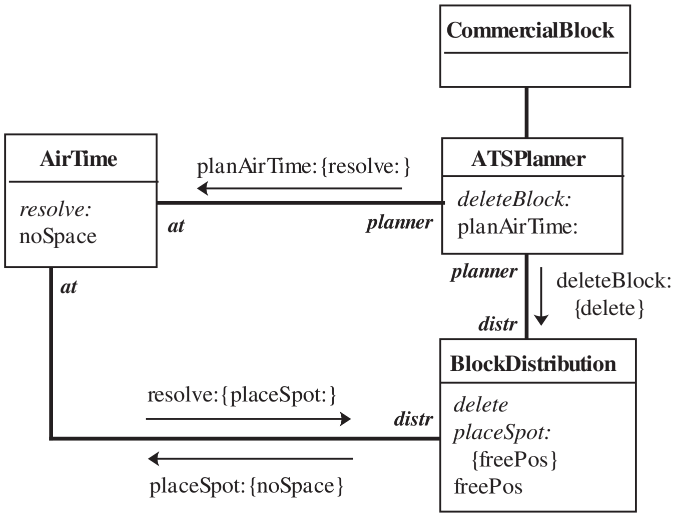

Figure 6.4: Generalised Air Time Sales Behaviour

The methods that have a different implementation in both cases are made abstract here. `planAirTime:` is a concrete method because all it does in both cases is send the message `resolve:` to its argument. Note that although `resolve:` is abstract in both cases it sends `placeSpot:` to the distribution. This is an example of an abstract method with a non-empty specialisation clause. The abstract method `placeSpot:` calling `freePos` and the abstract method `deleteBlock:` calling `delete` are such examples as well.

6.6 Expressing Specialisations through Reuse Operators

Now that we have a general reuse contract we can express how the specific cases are derived from the general one by means of reuse operators. Two approaches can be taken to do this. The first is to take the entire general reuse contract and make two specialisations of this contract. The second is to consider the three hierarchies one by one. Both approaches demonstrate the differences and both cases can be useful, depending on what you want to emphasise.

Figure 6.5 demonstrates the first approach for the GRP case. It demonstrates how the `GRPContract` is derived from the `ATSContract` with the UML notation for refinement. Three operators are used[4]. The combined operator layered concretisation and the two basic operators extension and concretisation. In the modifier, the name of the class is used every time to indicate on which class a change is made. The notation of the specialisation clauses shows along which acquaintance relationships invocations are added.

The other approach is taken in figures 6.6 through 6.8. Figure 6.6 depicts different air times. Both kinds concretise `resolve:`. In the case of GRPs this is a layered concretisation, because simultaneously a refinement is performed. Both kinds of air time also add some new behaviour through an extension.

Figure 6.7 depicts different planners. As `planAirTime:` was already concrete and the implementation is the same in both cases, nothing has to be done for this piece of behaviour. Both cases concretise `deleteBlock:`. `BSSPlanner` first checks whether the block is empty, `GRP_Planner` does not. An extra method `distrWithGRP` is added to `GRP_Planner` to return all distributions with a certain GRP.

Figure 6.8 depicts different distributions. Both cases concretise `placeSpot:` and `delete`, partially with layered concretisations adding some behaviour. `BSSDistribution` also adds an extra method `isEmpty`.

6.7 Evolution

6.7.1 A Combined System

Now imagine wanting to have a system where both kinds of options are offered to advertisers. In some cases they might really want to have a spot in a particular

[4] We left out the renaming for reasons of brevity

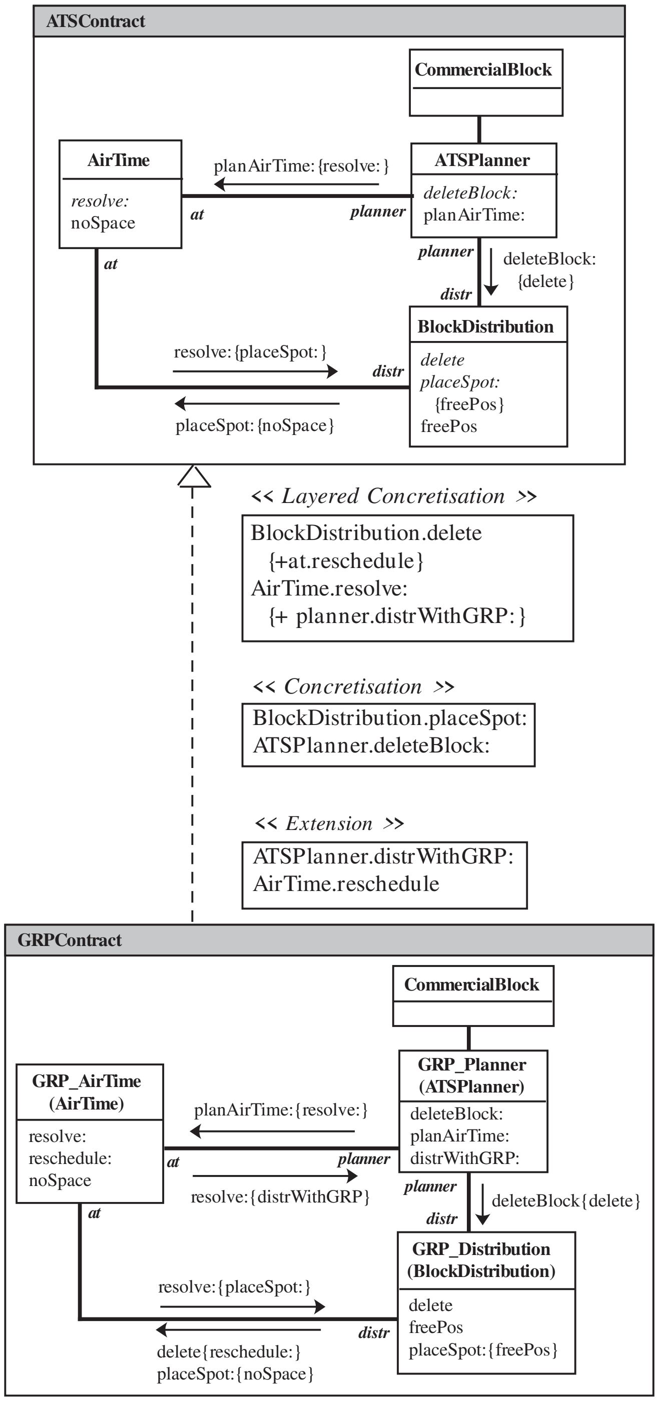

Figure 6.5: Specialising the Air Time Sales Contract

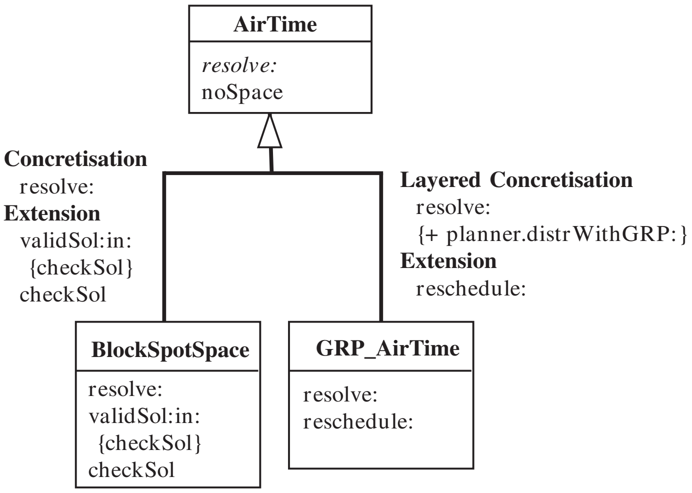

Figure 6.6: Specialising Air Time

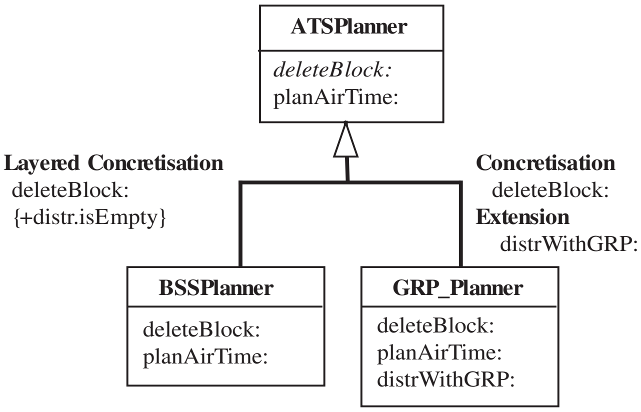

Figure 6.7: Specialising Planner

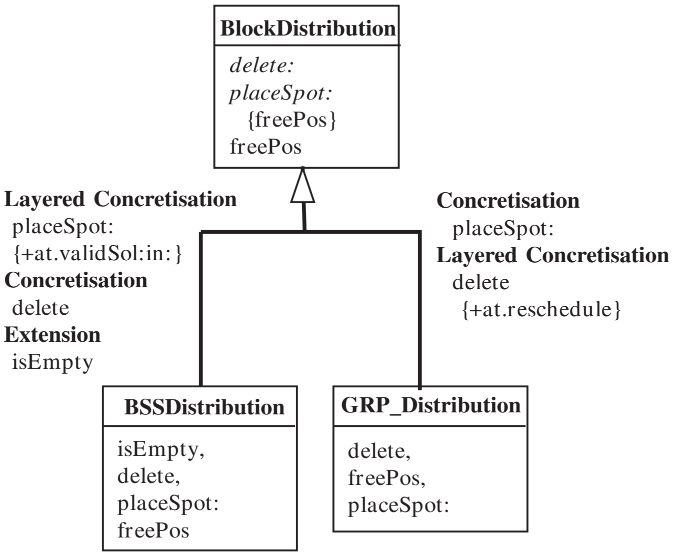

Figure 6.8: Specialising Block Distribution

distribution, for example, during the break of a sports event. In other campaigns they might want to reach a certain audience regardless of how this is achieved. We thus want to keep having different kinds of air time, but want a planner and distribution that can handle both. This is depicted in figure 6.9, where a new kind of planner and distribution are introduced, that communicate with the general `AirTime` class.

By looking at figures 6.7 and 6.8 we can easily detect where the two cases differ. Except for the addition of the method `distrWithGRP:`, no distinction is made on the planner classes concerning the actual planning. Only the deletion behaviour is slightly different. While the `GRP_Planner` just deletes a distribution, the `BSSPlanner` first checks whether the distribution is empty or not. As the behaviour depends on the kind of air time, it seems logical to shift this test to the distribution class and let it query the air time. Therefore we introduce a method `deleteBlock` on `CombinedDistribution` that asks all of its scheduled air times whether they are reschedulable. If they all are, they are asked to reschedule themselves and the distribution is deleted. If at least one of them is not reschedulable `deleteBlock` on `Distribution` reports this to `deleteBlock:` on `Planner` which again notifies the general planning application that deletion was denied. Note that this does require the addition of a method `reschedulable` on all air times. It can be added to the super class `AirTime`, where as a general rule it should return false.

A similar scenario repeats itself concerning the different behaviour of `placeSpot:` on both kinds of distributions. While for `GRP_AirTimes` it only checks whether it

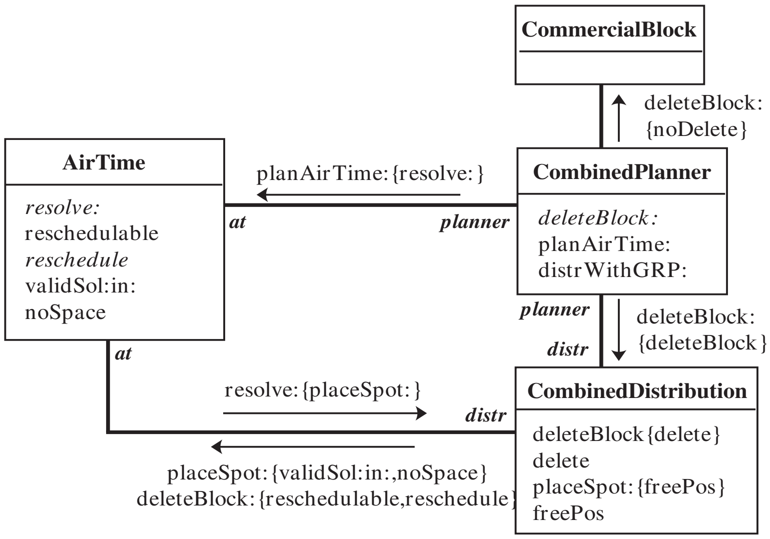

Figure 6.9: A Combined System

has a free spot, for `BSSAirTimes` it also asks whether the suggested solution is valid. We can solve this by always letting the combined distribution send the message `validSol:in:` to its air times and by always letting `validSol:in:` return true for `GRP_AirTimes`. We therefore add a method `validSol:in:` to `AirTime`. This is all depicted in figure 6.9.

While for planning and distribution new classes were introduced, the introduction of the combined system also requires some changes to the class `AirTime`. As two specialisations of `AirTime`, `BlockSpotSpace` and `GRP_AirTime` already exist, the changes to `AirTime` could cause conflicts on the specialisations. The problem is depicted in figure 6.10.

However, now that we have documented by means of reuse operators how these specialisations are derived from `AirTime`, it is easy to verify whether conflicts will occur. We therefore use the tables representing the conflict detection rules as developed in chapters 3 through 5. These tables are repeated in appendix A. Note that the tables do not express that in the situations they depict a conflict will always occur. They express which conflicts possibly occur through which combinations of operators. The corresponding rules then need to be applied to see whether there actually is a conflict in a particular case. So all we need to do here is express each change to the class AirTime by means of one or more reuse operators and then check these operators against the operators representing the specialisations.

We represent this checking in tables 6.1 and 6.2. Three changes were made to the class `AirTime`: three times an extension with a new method. In these tables the

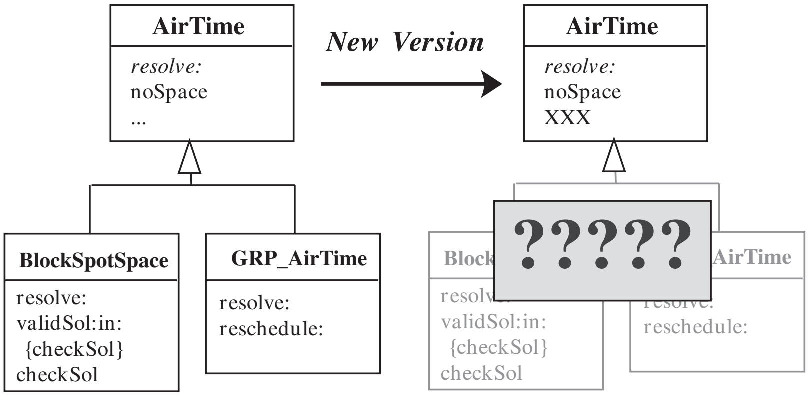

Figure 6.10: Evolution of Air Time

columns depict changes to the parent class, while the rows show the differences between the parent class and its different subclasses. To verify whether conflicts might arise, the different operators are then compared according to the rules of chapters 3 to 5.

We investigate the consequences of the addition of `validSol:in:` to `AirTime` in table 6.1. In this case, the rules signal a problem.

Original Air Time	**New version:** Extension with `validSol:in:`
BlockSpotSpace: Concretisation of `resolve:` and Extension by `validSol:in:`	*method name conflict*
GRP_Airtime: Layered concretisation of `resolve:` and Extension by `reschedule`	*no conflicts*

Table 6.1: Adding `validSol:in:` to `AirTime`

A method named `validSol:in:` already exists on `BSSAirTime`, which gives rise to a method name conflict. As the method on `AirTime` only returns true, the problem can be solved by turning the extension between `AirTime` and `BSSAirTime` into a refinement (as part of an extending refinement), as is depicted in figure 6.11.

We now investigate the consequences of the addition of `reschedulable` and `reschedule` in table 6.2. Again we get a method name conflict, because a method `reschedule:` is also introduced by `GRP_AirTime`. Again this method returns false on the superclass and is overridden on the subclass. We can however not take

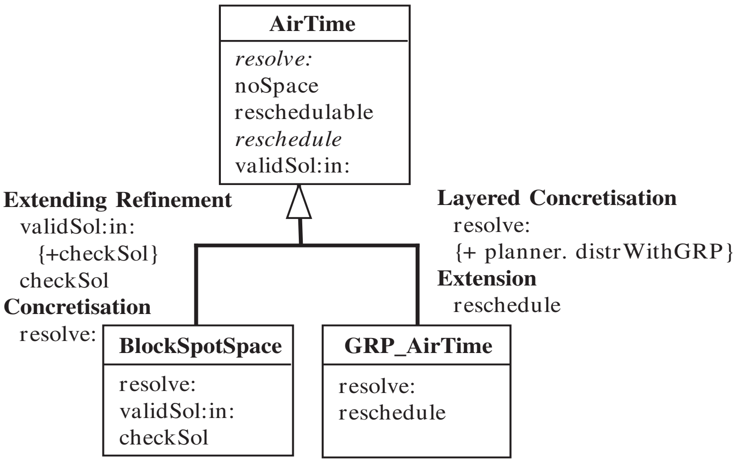

Figure 6.11: Revision of Air Times

the same approach and turn the extension into a refinement as no calls are added (probably there are some calls, but these are not mentioned in this design). Here the current expressiveness of the reuse operators falls short a little. One approach is to use empty refinements[5], another to introduce an extra operator: redefinition. The refinement of the operators is future work.

The addition of `reschedulable` does not cause any conflicts to the existing specialisations of `AirTime`, which is logical since a completely new method is added that does not interact with the other methods. Note that we also have to override the same method on `GRP_AirTime` to return true. We then get the same kind of empty refinement as for `reschedule`.

Original Air Time	**New version:** Extension by `reschedulable` and `reschedule:`
BlockSpotSpace: Concretisation of `resolve:` and Extension by `validSol:in:`	*no conflicts*
GRP_Airtime: Layered concretisation of `resolve:` and Extension by `reschedule`	*method name conflict*

Table 6.2: Adding `reschedulable` to `AirTime`

[5]Note that this is allowed, as specialisation clauses are sets and thus can be empty.

6.7.2 Introducing Clash Codes

Let us now see how to introduce new behaviour. One factor we have not considered yet in the air time sales system is the use of clash codes. Clash codes are associated with the products in each commercial. Some of these products "clash" and should therefore not be broadcasted in the same commercial block. A simple example is that a commercial for washing powder and one for softener should never be advertised in the same block.

In order to add checking of clash code violations we refine the implementation of `validSol:in:` on `AirTime`. The method now checks whether the product that it advertises does not clash with any of the other products in commercials in the distribution. This is achieved by invoking the method `checkClashes:` with the distribution as argument. Apart from the adaptation of `AirTime`, this change also requires all distributions to invoke `validSol:in:` when placing a spot. Until now only `BSSDistribution` and `CombinedDistribution` performed this invocation, `GRP_Distribution` did not. We thus add a call to `validSol:in:` to the specialisation clause of `placeSpot:` on the super class `Distribution`. The new version of the general behaviour is depicted in figure 6.12. Notice that the changes to `AirTime` as discussed in section 6.7.1 were also added.

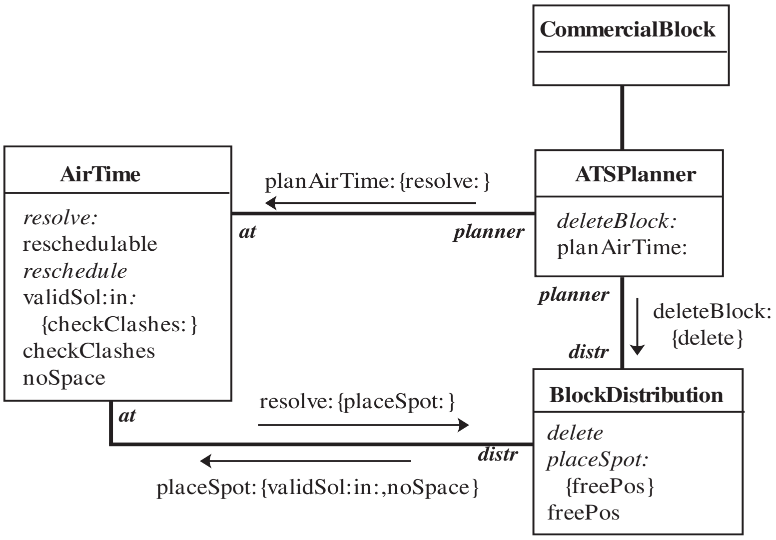

Figure 6.12: Introducing Clash Codes

We now again investigate the consequences of the changes on the existing specialisations. We start with the `Distribution` hierarchy. The only thing that changed on the superclass was the extra call to `validSol:in:` by `placeSpot:`. As table 6.3 demonstrates, the rules notify us that in two of the three specialisations of

`Distribution` an operation invocation conflict occurs, because these specialisations assume they are performing a refinement by adding an invocation to `validSol:in:`[6], while this invocation already exists on the superclass. Note that to keep the table concise, we only show the relevant operators.

Original Distribution	**New version:** Refinement of `placeSpot:` with `validSol:in:`
BSSDistribution: Layered Concretisation of `placeSpot:` with `validSol:in:`	*operation invocation conflict*
GRP_Distribution: Concretisation of `placeSpot:`	*no conflict*
CombinedDistribution: Layered Concretisation of `placeSpot:` with `validSol:in:`	*operation invocation conflict*

Table 6.3: Clash Code Behaviour on `Distribution`

Since there is no difference between the implementation of `placeSpot:` on `Distribution` and those in the subclasses, the solution is to simply omit the refinements between `Distribution` and its two specialisations. In a different situation, the refinement might need to be adapted to add less invocations or else turned into a specialisation.

While we first detected the conflicts in the `Distribution` hierarchy, we now verify whether any conflicts due to the introduction of clash codes might occur on `AirTime`. Two changes have been made. First, the method `validSol:in:` is refined on the superclass `AirTime` to invoke `checkClashes:`. Second, there is an extra call to `validSol:in:` by `placeSpot:`. The operators in table 6.4 reveal two conflicts: a method capture and an operation invocation conflict.

The operation invocation conflict occurs because the refinement of `validSol:in:` on `BlockSpotSpace` does not repeat the invocation of `checkClashes:`. The corresponding method will not call `checkClashes`, which might lead to errors. This problem can be solved by adapting the implementation of `validSol:in:` on `BlockSpotSpace` to first perform a super call, thus changing the refinement between `AirTime` and `BlockSpotSpace` into a specialisation.

The method capture is not really a conflict. It just signals that the call of `validSol:in:` on `AirTime` by `Distribution` will also lead to an invocation of the version of `validSol:in:` of `BlockSpotSpace`, which in this case is desirable. Note that the signalling of a conflict does not always need to cause an actual conflict.

[6] The refinement is part of the layered concretisation.

Original Air Time	**New version:** Refinement of `validSol:in:` with `checkClashes:`	**New version:** Refinement of `placeSpot:` on Distribution with `validSol:in:` on `at`
BlockSpotSpace: Refinement of `validSol:in:` with `checkSol`	*operation invocation conflict*	*method capture*
GRP_Airtime: Layered concretisation of `resolve:` and Extension by `reschedule`	*no conflicts*	*no conflicts*

Table 6.4: Clash Code Behaviour on `AirTime`

The signalling helps however in assessing where changes propagate.

6.7.3 Optimisation

As a last example of evolution, now that we have a number of working systems we might want to see where the performance can be improved. One possible place to improve performance is the checking of clash codes as seen from the distributions. Currently, every time a spot is placed its air time is asked to check whether it does not clash with the other spots in the distribution. It would be better to ask a spot what its clash code is and store this in the distribution. This information can then also be used for other checks and behaviours, for example, to verify that some articles are not advertised before 8 PM. Every time a spot is added, the distribution can then run a check itself and it no longer needs to ask the air time whether it provides a valid solution. The new design is depicted in figure 6.13.

Looking at the operators we see that this leads to inconsistent methods as depicted in table 6.5. While the change on Distribution represents a coarsening and omits the call to `validSol:in:` on `AirTime`, one of the specialisations of `AirTime` refines this method relying on the fact that it is called by `Distribution` whenever a spot is placed.

So if we do make this optimisation for clash code checking we need to ensure that the rest of the checking is still performed. We should therefore not remove the call to `validSol:in:` by `placeSpot:`, but coarsen `validSol:in:` on `AirTime` so that it no longer checks clash codes, but is still invoked on `BlockSpotSpaces` to check constraints. As table 6.6 shows, this coarsening does not cause conflicts on the specialisations of `AirTime`, because `BlockSpotSpace` is a specialisation and thus incorporates changes to the superclass without problems.

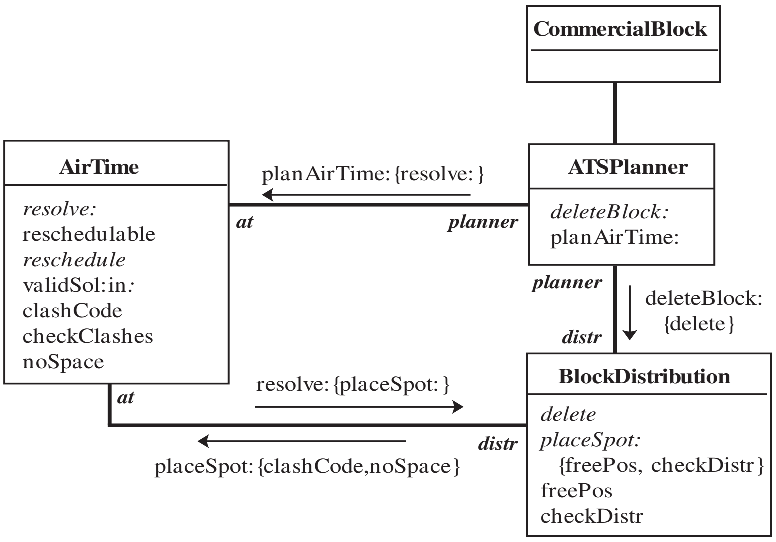

Figure 6.13: Optimising Distributions

Original Air Time	**New version:** Coarsening of `placeSpot:` on Distribution dereferencing `validSol:in:` on `at`
BlockSpotSpace: Specialisation of `validSol:in:` with `checkSol`	*inconsistent methods*
GRP_Airtime: Layered concretisation of `resolve:`	*no conflicts*

Table 6.5: Optimisation of ATSContract - change 1

The changes made to `Distribution` in this optimisation, i.e., the addition of `checkDistr`, the invocation of `checkDistr` by `placeSpot:` and the invocation of `clashCode` by `placeSpot:` do not cause any conflicts. Neither does the addition of `clashCode` on `AirTime`. The verification of these changes can again be performed by checking the tables, but is left to the reader.

6.8 Conclusion

Although we only discussed part of the design of this framework, this chapter has demonstrated how reuse contracts can be helpful in the design and evolution of reusable systems. Reuse contracts help in reasoning about frameworks, they help in assessing which compositions will exhibit desired behaviour and they help during

Original Air Time	**New version:** Coarsening of `validSol:in:` dereferencing `checkClashes:`
BlockSpotSpace: Specialisation of `validSol:in:` with `checkSol`	*no conflicts*
GRP_Airtime: Layered concretisation of `resolve:`	*no conflicts*

Table 6.6: Optimisation of ATSContract - change 2

evolution by signalling places where changes might cause conflicts.

While this experiment demonstrated the usefulness of reuse contracts, regarding its expressive power it also showed some of its weak points. We missed, for example, the possibility to express return types and conditionals. We can express that a method is overridden when the exact same methods are invoked by an empty refinement, but maybe it is better to introduce an extra operator, called redefinition. This was, for example, the case with the methods returning false by default on a root class, which are overridden to return true on subclasses. The extension and refinement of the current notation to increase its expressiveness is future work.

It is also clear that in order to use reuse contracts on larger systems, tool assistance is an absolute necessity. Here we detected conflicts by manually checking all operator combinations, but this rapidly becomes infeasible when the size of the system increases. Therefore, the next chapter discusses how tools can be built to assist in this process.

6.9 Acknowledgements

I thank Wim Codenie, Wilfried Verachtert and Wouter Roose from OOPartners, as well as Tom Mens and Patrick Steyaert for their assistance in this experiment.