S. Voß and D.L. Woodruff (Eds.), Optimization Software Class Libraries, Kluwer, Boston (2002), pp. 81-154.

4 HOTFRAME: A HEURISTIC OPTIMIZATION FRAMEWORK

Andreas Fink and Stefan Voß

Technische Universität Braunschweig
Institut für Wirtschaftswissenschaften
Abt-Jerusalem-Straße 7, D-38106 Braunschweig, Germany
{a.fink,stefan.voss}@tu-bs.de

Abstract: In this paper we survey the design and application of HOTFRAME, a framework that provides reusable software components in the metaheuristics domain. After a brief introduction and overview we analyze and model metaheuristics with special emphasis on commonalities and variabilities. The resulting model constitutes the basis for the framework design. The framework architecture defines the collaboration among software components (in particular with respect to the interface between generic metaheuristic components and problem-specific complements). The framework is described with respect to its architecture, included components, implementation, and application.

4.1 INTRODUCTION

There are several strong arguments in favor of reusable software components for metaheuristics. First of all, mature scientific knowledge that is aimed at solving practical problems must also be viewed from the point of view of technology transfer. If we consider the main metaheuristic concepts as sufficiently understood, we must strive to facilitate the efficient application of metaheuristics by suitable means. "No systems, no impact!" (Nievergelt (1994)) means that in practice we need easy-to-use application systems that incorporate the results of basic research. Therefore, we also have to deal with the issue of efficiently building such systems to bridge the gap between research and practice. From a research point of view, software reuse may also provide a way for a fair comparison of different heuristics within controlled and unbiased experiments, which conforms to some of the prescriptions in the literature (see, e.g., Barr et al. (1995) and Hooker (1994)).

Metaheuristics are by definition algorithmic concepts that are widely generic with respect to the type of problem. Since algorithms are generally applied in the form of software, adaptable metaheuristic software components are the natural means to incorporate respective scientific knowledge. We have built HOTFRAME, a Heuristic OpTimization FRAMEwork implemented in C++, which provides adaptable components that incorporate different metaheuristics and common problem-specific complements as well as an architectural description of the collaboration among these components (see Fink and Voß (1999b), Fink (2000)).

Others have followed or actively pursue similar research projects, which focus primarily on metaheuristics based on the local search paradigm including tabu search; see the contributions within this volume. Moreover, there is a great variety of software packages for evolutionary algorithms; see Heitkötter and Beasley (2001) as well as Chapter 10 below. On the other hand, there are some approaches to design domain-specific (modeling) languages for local search algorithm, which are partly based on constraint programming; see de Backer et al. (1999), Laburthe and Caseau (1998), Michel and van Hentenryck (1999), Michel and van Hentenryck (2001a), Nareyek (2001), as well as Chapter 9.

The adaptation of metaheuristics to a specific type of problem may concern both the static definition of problem-specific concepts such as the solution space or the neighborhood structure as well as the tuning of run-time parameters (calibration). The latter aspect of designing robust (auto-adaptive) algorithms, though being an important and only partly solved research topic, may be mostly hidden from the user. However, the general requirement to adapt metaheuristics to a specific problem is a more serious obstacle to an easy yet effective application of metaheuristics. Following the “no free lunch theorem” we assume that there can not be a general problem solver (and accordingly no universal software implementation) that is the most effective method for all types of problems (see Culberson (1998), Wolpert and Macready (1997)). This implies that one has to provide implementation mechanisms to do problem-specific adaptations to facilitate the effective application of reusable metaheuristic software components.

In this paper, we survey the design and application of HOTFRAME; for a detailed description in German we refer to Fink (2000). In the next section, we give a brief overview of the framework indicating some of its basic ideas. In Section 4.3, we analyze commonalities and variabilities of metaheuristics to the end that respective formal models provide the basis for reusable software components.

The subsequent sections are organized to follow the classical phases of software development processes. Here we assume that the reader is familiar with basic ideas of object-oriented programming and C++. The framework architecture, which specifies the collaboration among software components, is described in Section 4.4. In Section 4.5, we consider basic aspects of the implementation. In Section 4.6, we sketch the application of the framework and describe an incremental adoption path. Finally, we draw some conclusions.

4.2 A BRIEF OVERVIEW

The scope of HOTFRAME comprises metaheuristic concepts such as (iterated) local search, simulated annealing and variations, different kinds of tabu search methods (e.g., static, strict, reactive) providing the option to flexibly define tabu criteria taking into account information about the search history (attributes of performed moves and traversed solutions), evolutionary algorithms, candidate lists, neighborhood depth variations, and the pilot method.

The primary design objectives of HOTFRAME are run-time efficiency and a high degree of flexibility with respect to adaptations and extensions. The principal effectiveness of the framework regarding competitive results has been demonstrated for different types of problems; see Fink and Voß (1999a), Fink (2000), Fink et al. (2000), Fink and Voß (2001). However, claiming validity of the “no free lunch theorem”, the user must generally supplement or adapt the framework at well-defined adaptation points to exploit problem-specific knowledge, as problems from practice usually embody distinctive characteristics.

The architecture of a software system specifies the collaboration among system elements. The architecture of a framework defines a reusable design of systems in the same domain. Moreover, a framework provides (adaptable) software components (typically in accordance with the object-oriented paradigm), which encapsulate common domain abstractions. Contrary to an ordinary class library, which is some kind of a toolkit with modules that are mainly usable independently from each other, a framework also specifies (some of) the control flow of a system. With respect to the variabilities of different applications in the same domain, frameworks must provide variation points for adaptation and extension. That is, to instantiate a framework for a particular application one generally has to complete certain parts of the implementation, according to some kind of interface definitions.

HOTFRAME aims for a natural representation of variation points identified in the analysis of metaheuristics. Since metaheuristics are generic (abstract) algorithms, which are variable with respect to problem-specific concepts (structures), we follow the generic programming paradigm: A generic algorithm is written by abstracting algorithms on specific types (data structures) so that they apply to arguments whose types are as general as possible (generic algorithm = code + requirements on types); see Musser and Stepanov (1994). HOTFRAME uses parameterization by type as the primary mechanism to make components adaptable. Common behavior of metaheuristics is factored out and grouped in generic templates, applying static type variation. This approach leads to generic metaheuristic components which are parameterized by (mainly problem-specific) concepts such as the solution space, the neighborhood structure, or tabu-criteria. In C++, generic components are implemented as template classes (or functions), which enables achieving abstraction without loss of efficiency. Those templates have type parameters. The architecture defines properties (an interface with syntactic and semantic requirements) to be satisfied by argument types.

For example, steepest descent (greedy local search) is generic regarding the solution space S and the neighborhood N. Accordingly, we have two corresponding type parameters, which results in the following template:

```
SteepestDescent< S, N >
```

To streamline the parameterization of metaheuristic templates, we group type parameters in so-called configuration components, which encapsulate variation points and their fixing. For example, we may define

```
struct myConfiguration
{
  typedef  MCKP_S  S;
  typedef  BV_S_N  N;
};
```

and instantiate a metaheuristic component by

```
SteepestDescent< myConfiguration >.
```

In this case, we define the solution space to be represented by a class `MCKP_S`, which encapsulates solutions for the multi-constrained 0/1-knapsack problem, and we apply a class `BV_S_N`, which represents the classical bit-flip neighborhood for binary vectors. HOTFRAME provides several reusable classes for common solution spaces (e.g., binary vectors, permutations, combined assignment and sequencing) and neighborhood structures (e.g., bit-flip, shift, or swap moves). These classes can be used unchanged (e.g., neighborhoods) or reused by deriving new classes which customize their behavior (e.g., by defining some specific objective function). In cases where one of the pre-defined problem-specific components fits the actual problem, the implementation efforts for applying various metaheuristics can be minor, since the user, essentially, only needs to implement the objective function in a special class that inherits the data structure and the behavior from some appropriate reusable class.

Metaheuristic templates can have a set of type parameters, which refer to both problem-specific concepts as well as strategy concepts. For example, local search strategies may differ regarding the rule to select neighbors (moves). We have hierarchically separated the configuration regarding problem-specific and metaheuristic concepts. Common specializations of general metaheuristic components are pre-defined by configuration components that pass through problem-specific definitions and add strategy definitions. This is exemplified in the following definition:

```
template <class C>
struct CSteepestDescent
{
  typedef  BestPositivNeighbor<C>  NeighborSelection;
  typedef  typename C::S  S;
  typedef  typename C::N  N;
};
```

Then, we may customize a general local search frame, instantiate some metaheuristic object, and use it as shown in the following example:

```
myHeuristic = new IteratedLocalSearch
                  < CSteepestDescent< myConfiguration> >;
myHeuristic->search( initialSolution );
```

Advantages of this design are that it provides a concise and declarative way of system specification, which decreases the conceptual gap between program code and domain concepts (known as achieving high intentionality) and simplifies managing many variants of a component. Moreover, the resulting code can be highly run-time efficient.

4.3 ANALYSIS

Developing a framework requires a solid understanding of the domain under consideration. That is, we must develop a common understanding of the main metaheuristic concepts and the scope of the framework. Therefore, it is important to comprehensively analyze the domain and to develop a concrete and detailed domain model. This domain model comprises the commonalities and variabilities of metaheuristics, laying the foundation for the design of reusable software components with corresponding means for adaptation. In the following, we give a semi-formal domain model of the considered metaheuristics. The concise descriptions presuppose that the reader knows about metaheuristic concepts such as (iterated) local search, simulated annealing and variations, and different kinds of tabu search methods (see, e.g., Reeves (1993), Rayward-Smith et al. (1996), Laporte and Osman (1996), Osman and Kelly (1996), Aarts and Lenstra (1997), Glover and Laguna (1997), Voß et al. (1999), and Ribeiro and Hansen (2002)).

4.3.1 Problem-Specific Concepts

The first step in analysis is the definition of the commonalities (shared features) of different types of problems by an abstract model. Such a model ("domain vocabulary"), which captures problem-specific concepts with the same external meaning, is an essential basis to define metaheuristics, i.e., algorithmic concepts which are widely generic with respect to the type of problem. Eventually, when applying metaheuristics to some specific type of problem, these abstractions have to be instantiated by the specific structures of the problem at hand.

There are different types of

$$\text{problems} \quad P$$

with

$$\text{problem instances} \quad p \in P \ .$$

For every problem, there are one or more

$$\text{solution spaces} \quad S_P(p)$$

with

$$\text{solutions} \quad s \in S_P(p) \ .$$

For ease of notation, we generally restrict to $s \in S$ (where S replaces $S_P(p)$) when there are no ambiguities; this also concerns the subsequent notation. Solutions are evaluated by an

$$\text{objective function} \quad f : S \to \mathsf{R} \; .$$

We generally formulate problems as minimization problems; i.e., we strive for minimization of f.

To efficiently manage information about solutions (in connection with some tabu search method), we may need a function

$$h : S \to S_h$$

which transforms solutions to elements of a suitable set S_h. With respect to efficiency, one mostly uses non-injective ("approximate") functions (e.g., hash-codes).

For every solution space S, there are one or more

$$\text{neighborhoods} \quad N_S \; ,$$

which define for each solution $s \in S$ an ordered set of neighboring solutions

$$N_S(s) = \{n_1(s), \ldots, n_{|N_S(s)|}(s)\} \; .$$

Such a neighborhood is sketched in Figure 4.1.

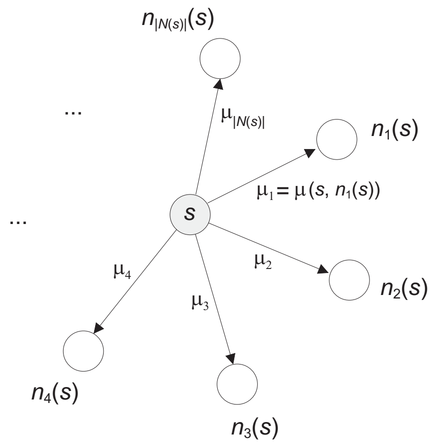


Figure 4.1 Neighborhood

From a transformation point of view every neighbor $n(s) \in N_S(s)$ of a solution $s \in S$ corresponds to a

$$\text{move} \quad \mu(s, n(s)) \; .$$

So we can also define a neighborhood as

$$N_S^{\mu}(s) = \{\mu_1 = \mu(s, n_1(s)), \ldots, \mu_{|N_S(s)|} = \mu(s, n_{|N_S(s)|}(s))\} \; .$$

Moves $\mu(s, n(s)) \in N_S^{\mu}(s)$ are evaluated by a

$$\text{move evaluation} \quad \hat{f}(\mu(s, n(s))) ,$$

which is often defined as $f(s) - f(n(s))$. Other kinds of move evaluations, e.g., based on measures indicating structural differences of solutions, are possible. In general, positive values should indicate "improving" moves. Move evaluations provide the basis for the guidance of the search process.

Both solutions and moves can be decomposed into

$$\text{attributes} \quad \psi \in \Psi ,$$

with Ψ representing some attribute set, which may depend on the neighborhood. A solution s corresponds to a set

$$\psi(s) = \{\psi_1(s), \ldots, \psi_{|\psi(s)|}(s)\} \ \text{ with } \ \psi_j(s) \in \Psi \ \forall j = 1, \ldots, |\psi(s)| .$$

The attributes of a move μ may be classified as plus and minus attributes, which correspond to the characteristics of a solution that are "created" or "destroyed", respectively, when the move is performed (e.g., inserted and deleted edges for a graph-based solution representation). A move μ corresponds to a set of plus and minus attributes:

$$\begin{aligned} \psi(\mu) &= \psi^{+}(\mu) \cup \psi^{-}(\mu) \\ &= \{\psi_1^{+}(\mu), \ldots, \psi^{+}_{|\psi^{+}(\mu)|}(\mu)\} \cup \{\psi_1^{-}(\mu), \ldots, \psi^{-}_{|\psi^{-}(\mu)|}(\mu)\} \\ &= \{\psi_1(\mu), \ldots, \psi_{|\psi(\mu)|}(\mu)\} . \end{aligned}$$

Finally, we denote the inverse attribute of ψ by $\overline{\psi}$.

4.3.2 Metaheuristic Concepts

In this section, we define some of the metaheuristic concepts included in HOTFRAME. We restrict the descriptions to (iterated) local search, simulated annealing (and variations), and tabu search, while neglecting, e.g., candidate lists, evolutionary methods and the pilot method. At places we do not give full definitions of various metaheuristics components (modules) but restrict ourselves to brief sketches. That is, the following descriptions exemplify the analysis of metaheuristics with respect to commonalities and variabilities, without providing a complete model.

There are two kinds of variabilities to be considered for metaheuristics. On the one hand, metaheuristics are generic regarding the type of problem. On the other hand, metaheuristics usually provide specific variation points regarding subordinate algorithms (aspects such as move selection rules or cooling schedules) and simple parameters (aspects such as the termination criterion). Accordingly, a configuration C of a metaheuristic H is composed of a definition of a subset C_P of the problem-specific abstractions (S, N, h, Ψ) discussed in the previous subsection, and of a configuration C_H that is specific to the metaheuristic. Given such a configuration C, a metaheuristic defines a transformation of an initial solution s to a solution s': $H_C : s \rightarrow s'$.

In the following descriptions, we use straightforward pseudo-code (with imperative and declarative constructs) and data structures such as lists or sets, without caring about, e.g., efficiency aspects. That is, such analysis models define meaning but should not necessarily prejudice any implementation aspects. In the pseudo-code description of algorithms, we generally denote a parameterization by "$< \ldots >$" to define algorithmic variation points, and we use "$(\ldots)$" to define simple value parameters (e.g., the initial solution or numeric parameters). When appropriate, we specify standard definitions, which allows using algorithms without explicitly defining all variation points. By the symbol ϕ we denote non-relevance or invalidity. To simplify the presentation, we use Ω to denote an additional termination criterion, which is implicitly assumed to be checked after each iteration of the local search process. By ω we include means to specify external termination, which is useful, e.g., in online settings. Using our notation, the interface of a metaheuristic H may be defined by $H < C > (s, T_{\max}, I_{\max}, \omega)$. Such a metaheuristic with configuration C transforms an initial solution s, given a maximum computation time $T_{\max}$, a maximum iteration number $I_{\max}$, and an external termination criterion ω.

To model the variation points of metaheuristics, we use feature diagrams, which provide a concise means to describe the variation points of concepts in a manner independent from any implementation concerns (see Czarnecki and Eisenecker (2000), Simos and Anthony (1998)).

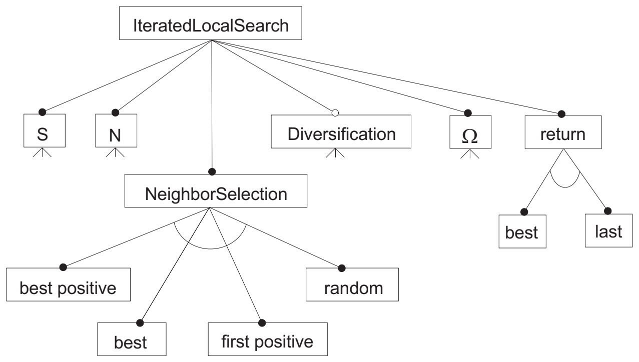


Figure 4.2 Feature Diagram for Simple Local Search Methods

4.3.2.1 Iterated Local Search. Figure 4.2 shows a feature diagram for simple local search methods (IteratedLocalSearch). In principle, all such local search procedures are variable regarding the solution space and the neighborhood structure. That is, S and N are mandatory features (denoted by the filled circles). The crowsfeet indicate that S and N are considered as abstract features, which have to be instantiated by specific definitions/implementations. At the center of any iteration of a local search procedure is the algorithm for selecting a neighbor. The diagram shows that there are four alternatives for instantiating this feature (denoted by the arc): select the best neighbor out of $N(s)$ with a positive move evaluation, select the best neighbor even if its evaluation is non-positive, select the first neighbor with a positive move evaluation, or select a random neighbor. The diversification feature is optional (denoted by the open circle), since we do not need a diversification mechanism, e.g., when we specialize IteratedLocalSearch as a one-time application of a greedy local search procedure. The termination criterion Ω is a mandatory feature, which has to be defined in a general way. Finally, there is a feature that allows to specify whether the search procedure should return the best solution found or the last solution traversed. The latter option is useful, e.g., when we use a local search procedure with a random neighbor selection rule as a subordinate diversification component of another metaheuristic.

The feature diagram defines the basic variation points of the considered concept. On this basis, Algorithm 1 formally defines the specific meaning of IteratedLocalSearch. While not giving formal definitions of all the features, Algorithm 2 exemplifies such a sub-algorithm. By using BestPositiveNeighbor, we may instantiate IteratedLocalSearch to generate SteepestDescent (as shown in Algorithm 3). In a similar manner, we may generate a RandomWalk procedure, which may again be used to generate a more complex procedure as shown in Algorithm 4.

Algorithm 1 IteratedLocalSearch

```
IteratedLocalSearch
    $< S, N, NeighborSelection, Diversification >$
    $(s, T_{\max} = \infty, I_{\max} = \infty, R_{\max} = 1, \omega = \text{false}, returnBest = \text{true})$ :

$\Omega : (t \geq T_{\max})$ or $(\omega)$

$s_{\text{best}} = s$;
for $r = 1$ to $R_{\max}$
    if $r > 1$
        Diversification($s$);
    $i = 0$;
    do
        $i = i + 1$;
        $s'$ = NeighborSelection$< S, N >(s)$;
        if $s'$ is valid
            $s = s'$;
            if $f(s) < f(s_{\text{best}})$
                $s_{\text{best}} = s$;
    while ($s'$ is valid) and ($i < I_{\max}$);
if returnBest
    $s = s_{\text{best}}$;
```

Algorithm 2 BestPositiveNeighbor

```
BestPositiveNeighbor $< S, N > (s)$ :

$j =$ argmax$\{\hat{f}(\mu(s, n_j(s))) \mid j = 1, \ldots, |N(s)|\}$;
**if** $\hat{f}(\mu(s, n_j(s))) > 0$
    **return** $n_j(s)$;
**else**
    **return** $\phi$;
```

Algorithm 3 SteepestDescent

```
SteepestDescent $< S, N > (s, T_{\max}, I_{\max}, \omega)$ :

IteratedLocalSearch $< S, N,$ BestPositiveNeighbor$, \phi >$
                    $(s, T_{\max}, I_{\max}, 1, \omega,$ true$)$;
```

Algorithm 4 IteratedSteepestDescentWithPerturbationRestarts

```
IteratedSteepestDescentWithPerturbationRestarts $< S, N >$
    $(s, T_{\max}, I_{\max}, R_{\max},$ *perturbations*$, \omega)$ :

IteratedLocalSearch
    $< S, N,$ BestPositiveNeighbor,
      RandomWalk $< S, N > (\phi, \infty,$ *perturbations*$, \omega,$ false$) >$
    $(s, T_{\max}, I_{\max}, R_{\max}, \omega,$ true$)$;
```

4.3.2.2 Simulated Annealing and Variations. Figure 4.3 shows the HOTFRAME feature diagram of simulated annealing and variations. Besides S and N, the principal features of such concepts are the acceptance criterion, the cooling schedule, and an optional reheating scheme.

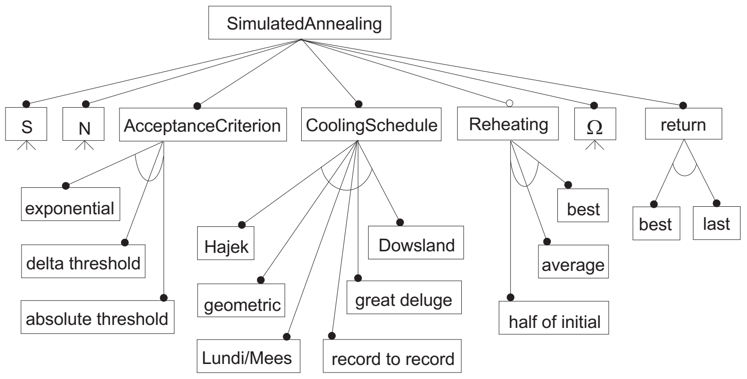


Figure 4.3 Feature Diagram for Simulated Annealing

The algorithmic commonalities of some set of simulated annealing like procedures are captured in Algorithm 5. After defining all the features (not shown here), we may generate a classic simulated annealing heuristic as shown in Algorithm 6. Some straightforward variations of this procedure are defined in Algorithms 7–9.

Nevertheless, it is not reasonable to try to capture all kinds of special simulated annealing procedures by one general simulated annealing scheme. This is exemplified by showing, in Algorithm 10, a simulated annealing procedure which strongly deviates from the general scheme of Algorithm 5. We define such a procedure separately, while we may use the general simulated annealing features as defined above. Given a parameter *initialAcceptanceFraction*, the starting temperature is set so that the initial fraction of accepted moves is approximately *initialAcceptanceFraction*. At each temperature $sizeFactor \times |N|$ move candidates are tested. The parameter *frozenAcceptanceFraction* is used to decide whether the annealing process is *frozen* and should be terminated. Every time a temperature is completed with less than *frozenAcceptanceFraction* of the candidate moves accepted, a counter is increased by one. This counter is reset every time a new best solution is found. The procedure is terminated when the counter reaches *frozenParemeter*. Then it is possible to reheat the temperature to continue the search by performing another annealing process. Algorithm 10 includes default values for the parameters according to the recommendation in Johnson et al. (1989).

Algorithm 5 GeneralSimulatedAnnealing

```
GeneralSimulatedAnnealing
    $< S, N,$ *AcceptanceCriterion*, *CoolingSchedule*, *Reheating* $>$
    $(s, T_{\max} = \infty, I_{\max} = \infty, \omega =$ false, $returnBest =$ true,
     $\tau_0, R_{\max} = 1, deltaRepetitions = 1, numberOfReheatings = 0)$  :
$\Omega : (t \geq T_{\max})$ **or** $(\omega)$

$s_{\text{best}} = s$;
**for** $h = 1$ **to** $numberOfReheatings + 1$
    **if** $h > 1$
        $\tau_0$ = *Reheating*$(\tau_0, \tau_{\text{best}}, \tau)$;
    **else**
        $\tau_{\text{best}}$ = $\tau_0$;
    $\tau$ = $\tau_0$;
    $i = 0$;
    **do**
        **for** $r = 1$ **to** $R_{\max}$
            $i$ = $i + 1$;
            $s'$ = RandomNeighbor$< S, N >(s)$;
            **if** *AcceptanceCriterion*$(\tau, \hat{f}(\mu(s, s')), f(s'))$
                $\tau$ = *CoolingSchedule.moveAccepted*$(\tau, \hat{f}(\mu(s, s')), f(s'))$;
                $s$ = $s'$;
                **if** $f(s) < f(s_{\text{best}})$
                    $s_{\text{best}}$ = $s$;
                    $\tau_{\text{best}}$ = $\tau$;
                    $\tau$ = *CoolingSchedule.newBestObjective*$(f(s))$;
            **else**
                $\tau$ = *CoolingSchedule.moveRejected*$(\tau, \hat{f}(\mu(s, s')), f(s'))$;
        $\tau$ = *CoolingSchedule.repetitionIntervalDone*$(\tau, \tau_0, i)$;
        $R_{\max} = \lfloor deltaRepetitions \cdot R_{\max} \rfloor$;
    **while** $i < I_{\max}$;
**if** *returnBest*
    $s = s_{\text{best}}$;
```

Algorithm 6 ClassicSimulatedAnnealing

```
ClassicSimpleSimulatedAnnealing $< S, N >$ :
    $(s, T_{\max}, I_{\max}, \omega, \tau_0, \alpha, b)$

GeneralSimulatedAnnealing $<S, N,$ ClassicExponentialAcceptanceCriterion,
                              GeometricCooling $< \alpha >, \phi >$
                          $(s, T_{\max}, I_{\max}, \omega,$ true$, \tau_0, 1, b, 0)$;
```

Algorithm 7 ThresholdAccepting

```
ThresholdAccepting $< S, N >$ :
    $(s, T_{\max}, I_{\max}, \omega, \tau_0, \alpha, b)$

GeneralSimulatedAnnealing $<S, N,$ ClassicThresholdAcceptanceCriterion,
                              GeometricCooling $< \alpha >, \phi >$
                          $(s, T_{\max}, I_{\max}, \omega,$ true$, \tau_0, 1, b, 0)$;
```

Algorithm 8 GreatDeluge

```
GreatDeluge $< S, N >$ :
    $(s, T_{\max}, I_{\max}, \omega, \tau_0, \alpha_1, \alpha_2, b)$

GeneralSimulatedAnnealing $<S, N,$ AbsoluteThresholdAcceptanceCriterion,
                              GreatDelugeCooling $< \alpha_1, \alpha_2 >, \phi >$
                          $(s, T_{\max}, I_{\max}, \omega,$ true$, \tau_0, 1, b, 0)$;
```

Algorithm 9 RecordToRecordTravel

```
RecordToRecordTravel $< S, N >$ :
    $(s, T_{\max}, I_{\max}, \omega, \tau_0, \alpha, b)$

GeneralSimulatedAnnealing $<S, N,$ AbsoluteThresholdAcceptanceCriterion,
                              RecordToRecordTravelCooling $< \alpha >, \phi >$
                          $(s, T_{\max}, I_{\max}, \omega,$ true$, \tau_0, 1, b, 0)$;
```

Algorithm 10 SimulatedAnnealingJetal

```
SimulatedAnnealingJetal
        $< S, N, AcceptanceCriterion, CoolingSchedule, Reheating >$
        $(s, T_{\max} = \infty, I_{\max} = \infty, \omega = \mathsf{false}, returnBest = \mathsf{true},$
         $initialAcceptanceFraction = 0.4, frozenAcceptanceFraction = 0.02,$
         $sizeFactor = 16, frozenParameter = 5, numberOfReheatings = 0)$ :
$\Omega : (t \geq T_{\max})$ or $(i \geq I_{\max})$ or $(\omega)$

perform trial annealing run to determine $\tau_0$ such that
        $initialAcceptanceFraction$ of the neighbors are accepted;
$s_{\text{best}} = s$;
for $h = 1$ to $numberOfReheatings + 1$
        if $h > 1$
                $\tau_0$ = Reheating($\tau_0, \tau_{\text{best}}, \tau$);
        else
                $\tau_{\text{best}} = \tau_0$;
        $\tau = \tau_0$;
        frozenCounter = 0;
        $i = 0$;
        do
                $k_1 = k_2 = 0$;
                $R_{\max} = \lfloor sizeFactor \cdot |N(s)| \rfloor$;
                for $r = 1$ to $R_{\max}$
                        $i = i + 1$;
                        $s'$ = RandomNeighbor$< S, N >(s)$;
                        if AcceptanceCriterion($\tau, \hat{f}(\mu(s, s')), f(s')$)
                                if $\hat{f}(\mu(s, s')) > 0$
                                        $k_1 = k_1 + 1$;
                                else
                                        $k_2 = k_2 + 1$;
                                $s = s'$;
                                if $f(s) < f(s_{\text{best}})$
                                        $s_{\text{best}} = s$; $\tau_{\text{best}} = \tau$; frozenCounter = 0;
                $\tau$ = CoolingSchedule.repetitionIntervalDone($\tau, \tau_0, i$);
                if $((k_1 + k_2)/R_{\max} \leq$ frozenAcceptanceFraction$)$ or $(k_1 \equiv 0)$
                        frozenCounter = frozenCounter + 1;
        while frozenCounter < $frozenParameter$;
if returnBest
        $s = s_{\text{best}}$;
```

4.3.2.3 Tabu Search. The HOTFRAME feature diagram of tabu search is shown in Figure 4.4. Besides S and N, the principal features of tabu search are the tabu criterion and the rule to select neighbors. Moreover, there may be an explicit diversification scheme. We explicitly model the strict tabu criterion (i.e., defining a move as tabu if and only if it would lead to an already traversed neighbor solution), the static tabu criterion (i.e., storing attributes of performed moves in a tabu list of a fixed size and prohibiting these attributes from being inverted), and the reactive tabu criterion according to Battiti and Tecchiolli (1994).

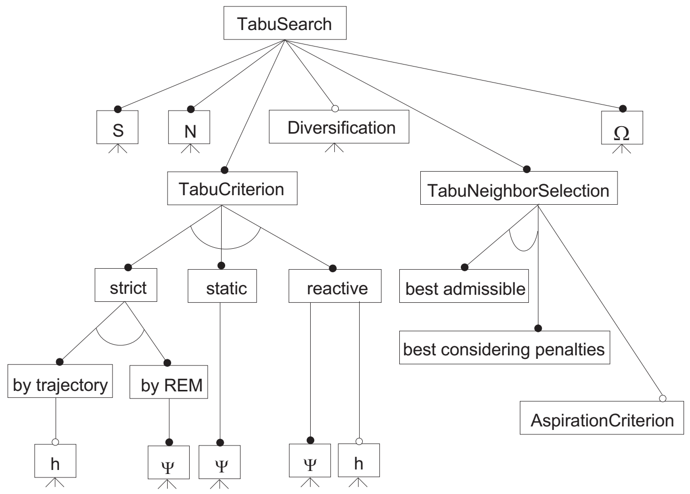


Figure 4.4 Feature Diagram for Tabu Search

The algorithmic commonalities of a tabu search metaheuristic are shown in Algorithm 11. Classic tabu search approaches control the search by dynamically classifying neighbors and corresponding moves as tabu. To implement tabu criteria, one uses information about the search history: traversed solutions and/or attributes of performed moves. Using such information, a tabu criterion defines whether neighbors and corresponding moves are classified as tabu. A move is admissible if it is not tabu or an aspiration criterion is fulfilled. That is, aspiration criteria may invalidate a tabu classification (e.g., if the considered move leads to a neighbor solution with a new best objective function value). The tabu criterion may also signal that an explicit diversification seems to be reasonable. In such a case, a diversification procedure is applied (e.g., a random walk).

The most popular approach to apply the tabu criterion as part of the neighbor selection procedure is by choosing the best admissible neighbor (Algorithm 12). Alternatively, some measure of the tabu degree of a neighbor may be used to compute a penalty value that is added to the move evaluation (Algorithm 13). With regard to the latter option, the tabu criterion provides for each move a tabu degree value (between 0 and 1). Multiplying the tabu degree with a parameter σ results in the penalty value.

The considered tabu criteria are defined in Algorithms 14–17. In each case, the tabu memory is modeled by state variables using simple container data structures such

Algorithm 11 TabuSearch

```
TabuSearch
    $< S, N,$ *TabuCriterion*, *TabuNeighborSelection*, *Diversification* $>$
    $(s, T_{\max} = \infty, I_{\max} = \infty, \omega =$ false$)$ :

$\Omega : (t \geq T_{\max})$ or $(\omega)$

$s_{\text{best}} = s$;
$i = 0$;
while $i < I_{\max}$
    $i = i + 1$;
    $s'$ = TabuNeighborSelection$< S, N, TabuCriterion >(s)$;
    TabuCriterion.add$(\mu(s, s'), i)$;
    $s = s'$;
    TabuCriterion.add$(s, i)$;
    if $f(s) < f(s_{\text{best}})$
        $s_{\text{best}} = s$;
    if TabuCriterion.escape()
        Diversification$(s)$;
$s = s_{\text{best}}$;
```

Algorithm 12 BestAdmissibleNeighbor

```
BestAdmissibleNeighbor $< Aspiration >< S, N, TabuCriterion > (s)$ :

if $\exists j \in \{1, \ldots, |N(s)|\}$ : (not TabuCriterion.tabu$(\mu(s, n_j(s)), n_j(s))$)
                                      or (Aspiration$(\mu(s, n_j(s)), TabuCriterion)$)
    $j =$ argmax$\{\hat{f}(\mu(s, n_j(s))) \mid j = 1, \ldots, |N(s)|,$
                 (not TabuCriterion.tabu$(\mu(s, n_j(s)), n_j(s))$)
                 or (Aspiration$(\mu(s, n_j(s)), TabuCriterion)$)$\}$;
    return $n_j(s)$;
else
    return RandomNeighbor $< S, N >(s)$;
```

as lists or sets, which are parameterized by the type of the respective objects. If lists are defined as having a fixed length, objects are inserted in a first-in first-out manner. Not all tabu criteria implement all functions. For instance, most of the tabu criteria do not possess means to detect and signal situations when an explicit diversification seems to be reasonable.

The strict tabu criterion can be implemented by storing information about all traversed solutions (using the function h). In Algorithm 14, we do not apply frequency

Algorithm 13 BestNeighborConsideringPenalties

```
BestNeighborConsideringPenalties $< Aspiration, \sigma >$
                                 $< S, N, TabuCriterion > (s)$ :

if Aspiration($\mu(s, n_j(s)), TabuCriterion$)
    $j =$ argmax$\{\hat{f}(\mu(s, n_j(s))) \mid j = 1, \ldots, |N(s)|\}$;
else
    $j =$ argmax$\{\hat{f}(\mu(s, n_j(s)))$
                $+ \sigma \cdot$ TabuCriterion.tabuDegree($\mu(s, n_j(s)), n_j(s)$)
                $\mid j = 1, \ldots, |N(s)|\}$;
return $n_j(s)$;
```

Algorithm 14 StrictTabuCriterionByTrajectory

```
StrictTabuCriterionByTrajectory $< S, h >$ :

State: Set$< S_h >$ trajectory;

add$(s, \phi)$ :
    trajectory.insert($h(s)$);

tabu$(\phi, s')$ :
    if $s' \in$ trajectory
        return true;
    else
        return false;

tabuDegree$(\phi, s')$ :
    if $s' \in$ trajectory
        return 1;
    else
        return 0;
```

or recency information to compute a relative tabu degree but simply use an absolute tabu classification. In principle, the strict tabu criterion is a necessary and sufficient condition to prevent cycling in the sense that it classifies exactly those moves as tabu that would lead to an already traversed neighbor. However, as one usually applies a non-injective ("approximate") function h, moves might unnecessarily be set tabu (when "collisions" occur); see Woodruff and Zemel (1993).

As an alternative implementation of the strict tabu criterion, the reverse elimination method (REM, Algorithm 15) exploits logical interdependencies among moves, their attributes and respective solutions (see Glover (1990), Dammeyer and Voß (1993), Voß (1993a), Voß (1995), Voß (1996)). A *running list* stores the sequence of the attributes of performed moves (i.e., the created and destroyed solution attributes). In every iteration, one successively computes a *residual cancellation sequence* (RCS), which includes those attributes that separate the current solution from a formerly traversed solution. Every time when the RCS exactly constitutes a move, the corresponding inverse move must be classified as tabu (for one iteration). It should be noted that it is not quite clear how to generally implement the REM tabu criterion for multi-attribute moves in an efficient way. For this reason, the REM component of HOTFRAME is restricted to single attribute moves that may be coded by natural numbers.

The strict tabu criterion is often too weak to provide a sufficient search diversification. We consider two options to strengthen the tabu criterion of the REM. The first alternative uses the parameter *tabuDuration* to define a tabu duration longer than one iteration. The second uses the parameter *rcsLengthParameter* to define a threshold for the length of the RCS, so that all possibilities to combine (subsets of) the attributes of the RCS of a corresponding maximum length as a move are classified as tabu.

The static tabu criterion is defined in Algorithm 16. The parameter Ψ represents the decomposition of moves in attributes. The parameter κ defines the capacity of the tabu list (as the number of attributes). The parameter ν defines the number of attributes of a move, for which there must be inverse correspondents in the tabu list to classify this move as tabu. Furthermore, ν is also the reference value to define a proportional tabu degree.

Algorithm 17 shows the mechanism of the tabu criterion for reactive tabu search. With regard to the adaptation of the length of the tabu list, a history stores information about traversed moves. This includes the iteration of the last traversal and the frequency. The actual tabu status/degree is defined in the same way as for the static tabu criterion using the parameter ν. The adaptation of the tabu list length is computed in dependence of the parameters δ_1 and δ_2. When a re-traversal of a solution occurs, the list is enlarged considering a maximum length κ. Depending on an exponentially smoothed average iteration number between re-traversals (using a parameter θ), the length of the tabu list is reduced if there has not been any re-traversal for some time. If there are ζ_1 solutions that each have been traversed at least ζ_2 times, the apparent need for an explicit diversification is signalled.

The parameterization of TabuSearch and of the used modules enables various possibilities to build specific tabu search heuristics. For example, Algorithm 18 (StrictTabuSearch) encodes the simplest form of strict tabu search: All traversed solutions are stored explicitly (*id* represents the identity function), which means that they are classified as tabu in the subsequent search process. Algorithm 19 (REMpen) shows the enhanced reversed elimination method in combination with the use of penalty costs. Static tabu search is shown in Algorithm 20. Algorithm 21 defines reactive tabu search in combination with the use of RandomWalk as diversification mechanism (setting most of the parameters to reasonable default values).

Algorithm 15 REMTabuCriterion

```
REMTabuCriterion $< S, N, \Psi >$
                    $($*tabuDuration*$,$ *rcsLengthParameter*$)$ :

State: List$< \Psi >$ *runningList*;
       Set$<$ (Set$< \Psi >$, Integer$) >$ *tabuList*;

add$(\mu, i)$ :
    **for** $j = 1$ **to** $|\psi(\mu)|$
        *runningList.append*$(\psi_j(\mu))$;
    *RCS* $= \emptyset$;
    **for** $j = \overline{|runningList|}$ **downto** $1$
        **if** $\overline{runningList[j]} \in RCS$
            $RCS = RCS \setminus \overline{runningList[j]}$;
        **else**
            *RCS.insert*$(runningList[j])$;
        **if** $|RCS| \leq$ *rcsLengthParameter*
            *tabuList.append*$(\overline{RCS}, i +$ *tabuDuration*$)$;

tabu$(\mu, \phi)$ :
    **if** $\psi(\mu) \in$ *tabuList* **and** $\mu$ was set tabu not longer than
                                    *tabuDuration* iterations before
        **return** true;
    **else**
        **return** false;

tabuDegree$(\mu, \phi)$ :
    **if** $\psi(\mu) \in$ *tabuList* **and** $\mu$ was set tabu not longer than
                                    *tabuDuration* iterations before
        **return** (remaining tabu duration of $\mu)/$*tabuDuration*;
    **else**
        **return** 0;
```

Algorithm 16 StaticTabuCriterion

```
StaticTabuCriterion $< S, N, \Psi > (\nu, \kappa)$ :

State: List$< \Psi, \kappa >$ *tabuList*;

add$(\mu, \phi)$ :
    **for** $j = 1$ **to** $|\psi^{+}(\mu)|$
        *tabuList.append*($\psi_j^{+}(\mu)$);

tabu$(\mu, \phi)$ :
    $k = 0$;
    **for** $j = 1$ **to** $|\psi^{-}(\mu)|$
        **if** $\psi_j^{-}(\mu) \in$ *tabuList*
            $k = k + 1$;
    **if** $k \geq \nu$
        **return** true;
    **else**
        **return** false;

tabuDegree$(\mu, \phi)$ :
    $k = 0$;
    **for** $j = 1$ **to** $|\psi^{-}(\mu)|$
        **if** $\psi_j^{-}(\mu) \in$ *tabuList*
            $k = k + 1$;
    **return** $\min\{k/\nu, 1\}$;
```

Algorithm 17 ReactiveTabuCriterion

```
ReactiveTabuCriterion $< S, N, \Psi, h >$ $(\nu, \delta_1, \delta_2, \kappa, \zeta_1, \zeta_2, \theta)$ :
State: List$< \Psi >$ *tabuList*;
       Set$< (S_h, \text{Integer}, \text{Integer}) >$ *trajectory*;
       *movingAverage* = *lastReaction* = 0;

add$(\mu, \phi)$ :
    **for** $j = 1$ **to** $|\psi^+(\mu)|$
        *tabuList.append*($\psi_j^+(\mu)$);
add$(s, i)$ :
    **if** $h(s) \in$ *trajectory* (with corresponding iteration $k$)
        extend length $l$ of *tabuList* to $\min\{\max\{\lceil l \cdot \delta_1 \rceil, l + \delta_2\}, \kappa\}$;
        *lastReaction* $= i$;
        *movingAverage* $= \theta \cdot (i - k) + (1 - \theta) \cdot$ *movingAverage*;
        update iteration to $i$ and increment frequency of $h(s)$;
        **if** there are $\zeta_1$ solutions in *trajectory*
          that have been traversed $\zeta_2$ times or more
            trigger escape;
    **else**
        *trajectory.insert*($h(s), i, 1$);
        **if** $i -$ *lastReaction* $>$ *movingAverage*
            reduce length $l$ of *tabuList* to $\max\{\min\{\lfloor l/\delta_1 \rfloor, l - \delta_2\}, \delta_2\}$;
            *lastReaction* $= i$;
tabu$(\mu, \phi)$ :
    $k = 0$;
    **for** $j = 1$ **to** $|\psi^-(\mu)|$
        **if** $\psi_j^-(\mu) \in$ *tabuList*
            $k = k + 1$;
    **if** $k \geq \nu$ **return** true;
    **else return** false;
tabuDegree$(\mu, \phi)$ :
    $k = 0$;
    **for** $j = 1$ **to** $|\psi^-(\mu)|$
        **if** $\psi_j^-(\mu) \in$ *tabuList*
            $k = k + 1$;
    **return** $\min\{k/\nu, 1\}$;
```

Algorithm 18 StrictTabuSearch

```
StrictTabuSearch $< S, N >$ $(s, T_{\max}, I_{\max}, \omega)$ :

TabuSearch $<S, N,$ StrictTabuCriterionByTrajectory $< S,$ *id* $>,$
                BestAdmissibleNeighbor $< \phi >, \phi >$
            $(s, T_{\max}, I_{\max}, \omega)$;
```

Algorithm 19 REMpen

```
REMpen $< S, N, \Psi >$
        $(s, T_{\max}, I_{\max}, \omega,$ *tabuDuration*, *rcsLengthParameter*, $\sigma)$ :

TabuSearch $<S, N,$
                REMTabuCriterion
                    $< S, N, \Psi >$ (*tabuDuration*, *rcsLengthParameter*),
                BestNeighborConsideringPenalties $< \phi, \sigma >, \phi >$
            $(s, T_{\max}, I_{\max}, \omega)$;
```

Algorithm 20 StaticTabuSearch

```
StaticTabuSearch $< S, N, \Psi >$ $(s, T_{\max}, I_{\max}, \omega, \nu, \kappa)$ :

TabuSearch $<S, N,$ StaticTabuCriterion $< S, N, \Psi > (\nu, \kappa)$,
                BestAdmissibleNeighbor $< \phi >, \phi >$
            $(s, T_{\max}, I_{\max}, \omega)$;
```

Algorithm 21 ReactiveTabuSearch

```
ReactiveTabuSearch $< S, N, \Psi, h >$
                $(s, T_{\max}, I_{\max}, \omega, \nu, \kappa,$ *perturbations*$)$ :

TabuSearch
    $< S, N,$ ReactiveTabuCriterion $< S, N, \Psi, h > (\nu, 1.2, 2, \kappa, 3, 3, 0.5)$,
    BestAdmissibleNeighbor $< \phi >$,
    RandomWalk$<S, N>(\phi, \infty,$ *perturbations*$, \omega,$ false$) >$
    $(s, T_{\max}, I_{\max}, \omega)$;
```

4.4 DESIGN

Design involves modeling the principal abstractions defined in the domain analysis (Section 4.3) as artefacts, which may more or less be directly implemented as software (modules). This especially concerns the design of a framework architecture, which defines the interplay between software components by means of interface specifications. In particular, such a framework specifies (some of) the control flow of a system. Besides the architecture, a framework defines adaptable software components, which encapsulate common domain abstractions. To be adaptable with respect to the variabilities of different applications in the considered domain, a framework provides variation points, which allow modifying or completing certain parts of the implementation.

In this section, we give an overview of framework architecture. The primary design decisions are about mechanisms that define the interplay between metaheuristics and problem-specific components. These mechanisms involve advanced concepts to adapt and combine components, which requires adequate implementation mechanisms in the programming language employed.

The main constructs to implement adaptable (polymorphic) software components are object-oriented inheritance and genericity by type parameters. Inheritance allows adapting classes by deferring the specific implementation of (some of) the class operations to specialized classes, which inherit the common data structure and functionality from respective general (base) classes. Type parameterization means that methods and classes may have some kind of type "placeholders", which allow for specialization (instantiation) by fixing the type parameters by specific types/classes. In both cases, general classes serve as starting points to modify and extend the common data structures and algorithms identified in the analysis by the specific needs of the application under consideration. The widely used programming language C++ provides both kinds of adaptation constructs (inheritance as well as genericity), as well as enabling run-time efficient software implementations. For a detailed exposition and discussion of software construction, in particular object-oriented programming, inheritance, and genericity, we refer, in general, to Meyer (1997) and, in particular for C++, to Stroustrup (1997).

4.4.1 Basic Design Concepts

In the following, we describe the basic design concepts of HOTFRAME. In some cases, we simplify a little bit to explain our approach. The understanding of the basic design enables a concise description of the architecture and the components later on.

4.4.1.1 Genericity of Metaheuristics. The primary design decision of HOTFRAME is about the interplay between generic metaheuristics components and problem-specific components. The features common to metaheuristics are captured in metaheuristic algorithms (i.e., corresponding software components) as shown in Section 4.3. These algorithms operate on problem-specific structures (C_P, see Section 4.3.1), in particular the solution space and the neighborhood. The natural way to implement this kind of polymorphism is by means of type parameterization. This concept refers to the generic programming paradigm, where algorithms are variable with respect to the structures (types) on which they operate. In C++, type parameterization is implemented in a static manner, as the instantiation, the fixing of type parameters of so-called template classes, is done at compile time.

We illustrate this basic idea by means of the example of a template class SteepestDescent; see the UML class diagram shown in Figure 4.5. (In fact, the steepest descent algorithm is generated as a specialization of an iterated local search software component.) The actual search algorithm is implemented by the operation (member function) search, which transforms an initial solution (passed to the operation). We use classes (instead of singular methods) to represent metaheuristics to enable storing search parameters or the state of a (not yet completed) search process. This allows treating an algorithm as a dynamic object (an instance of a class with state information), which may be constructed, used, and stored. (Furthermore, using classes to represent algorithms allows to adapt these algorithms by means of inheritance. However, to keep the design straightforward, we do not use this feature for the methods implemented in HOTFRAME.)

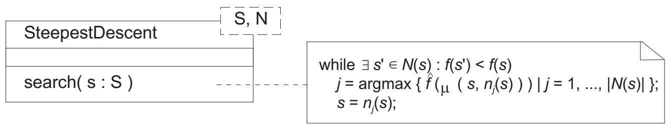

S, N
SteepestDescent
search(s : S)
while $\exists s' \in N(s) : f(s') < f(s)$
$j = \text{argmax} \{ \hat{f} (\mu (s, n_j(s))) \mid j = 1, \dots, |N(s)| \};$
$s = n_j(s);$

Figure 4.5 Class Diagram for SteepestDescent (Simplified)

The template parameters S and N are some sort of placeholders for the solution space and the neighborhood structure. Specific solution spaces and neighborhood structures must be implemented as classes with operations that conform to functional requirements that are derived from the analysis discussed in Section 4.3 (such interfaces are discussed below). Constructing a specialized class from a template class means defining for each template parameter a specific instantiation. This results in a class that is adapted with regard to a problem-specific configuration. That is, we have specialized a metaheuristic (template class) as a problem-specific heuristic (class). Henceforth, by calling some class constructor method (not shown in the figure) with respective parameter values (e.g., termination criteria), we can construct specific heuristic objects, which can be applied to transform an initial solution.

4.4.1.2 Variabilities of Metaheuristics. In addition to the problem-specific adaptation C_P, metaheuristics are variable with respect to the configuration C_H that is specific to the metaheuristic. For example IteratedLocalSearch (see Figure 4.2 or Algorithm 1) is variable regarding the neighbor selection rule and the diversification. These algorithmic abstractions may also be treated as template parameters of the generic metaheuristic class. On the other hand, the termination criterion concerns simple numeric parameters. That is, the termination criterion is not modeled by template parameters but by simple data parameters and corresponding class data elements. The same applies for the parameter that defines whether the algorithm should return the best or the last traversed solution. This results in the UML class diagram as shown in Figure 4.6.

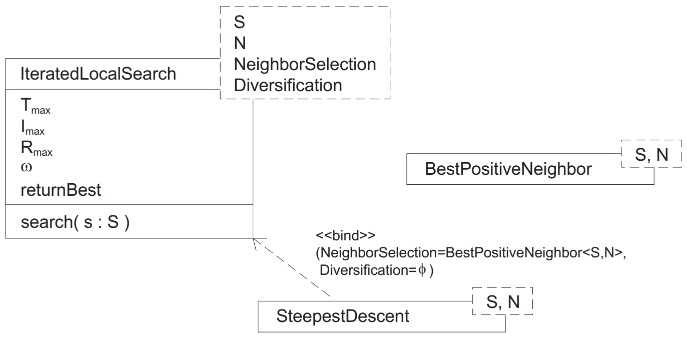


Figure 4.6 Class Diagram for IteratedLocalSearch (Simplified)

Components that are used to instantiate template parameters often have type parameters by themselves. For example, the component BestPositiveNeighbor, which is used to instantiate the template parameter NeighborSelection, is parameterized by the solution space and the neighborhood structure; see Figure 4.6. To denote the partial specialization of the generic class IteratedLocalSearch as (a still generic class) SteepestDescent, we use the UML stereotype bind, which means that we fix two out of four type parameters of IteratedLocalSearch.

Since different metaheuristics certainly possess different (static as well as dynamic) parameters, we have to define and implement for each of the general metaheuristics formulated in Section 4.3.2 (IteratedLocalSearch, GeneralSimulatedAnnealing, TabuSearch) a corresponding metaheuristic component (template class). As discussed for SimulatedAnnealingJetal, there can also be distinct modifications of a metaheuristic that result in specialized components, which are not directly derived from the general metaheuristic. That is, it does not seem reasonable to follow some "one component fits all" approach, since there will always be some distinct modifications, which one has not thought about when defining and implementing the "general" metaheuristic. In particular, one should not try to capture in one large (complicated) component all kinds of variation points. HOTFRAME allows to define such new metaheuristic components, which, of course, may reuse problem-specific components and other elementary components.

4.4.1.3 Neighborhood Traversal. The iterative traversal of the neighborhood of the current solution and the selection of one of these neighbors is the core of metaheuristics that are based on the local search approach. A generic implementation of different neighbor selection rules requires flexible and efficient access to the respective neighborhood. The basic idea of generic programming – algorithms operate on abstract structures – conforms to the neighborhood traversal task and enables an efficient and flexible design.

The different neighbor selection rules imply a few basic requirements, which are illustrated in Figure 4.7 (adaptation of Figure 4.1). The traversal of the neighborhood

$N(s)$ of a solution s conforms to a sequence of moves μ that correspond to neighbor solutions $n_i(s)$, $i = 1, \ldots, |N(s)|$.

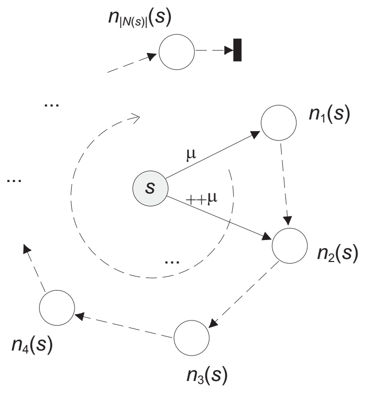


Figure 4.7 Neighborhood Traversal

The implementation of a neighbor selection rule such as BestNeighbor implies the need for the following basic functionality:

- Construction of a move to the first neighbor $n_1(s)$
- Increment of a valid move to the subsequent neighbor (in accordance with C++ represented by an increment operator "$++$")
- Computation of the move evaluation $\hat{f}(\mu)$
- Check for validity of a move

In principle, this functionality is also sufficient to implement the other neighbor selection rules. However, with respect to run-time efficiency we may need to directly construct a move to a random neighbor for metaheuristics such as simulated annealing. Otherwise, to construct a random neighbor, we would have to construct all moves and to select one of these by chance. This is obviously not practical for metaheuristics that require the efficient construction of a random move in each iteration. So we also require the following functionality:

- Direct construction of a move to a random neighbor

However, only suitable neighborhood structures allow an efficient implementation of the selection of a random move, if one requires to apply a uniform probability distribution. There is often a trade-off between the run-time of the random move construction and the quality of the probability distribution. So one generally may need to cope with non-uniform move selection for certain neighborhood structures.

The functionality of the neighborhood traversal largely conforms to the *iterator pattern*, which is about a sequential traversal of some (container) structure without knowing about the specific representation of this structure. That is, the iterator pattern explicitly separates structures from algorithms that operate on these structures; see Gamma et al. (1995). Accordingly, we may refer to respective neighborhood classes as neighborhood iterator classes. The design of these classes is based on the concept of the iterator classes of the *Standard Template Library*; see Musser and Saini (1996). The solution class takes the virtual role of a container class (solution space). Moves as objects of a neighborhood iterator class store a reference to a particular solution and transformational information with respect to the neighbor solution where they point to. For reasons of efficiency, it is apparently not reasonable to physically construct each neighbor solution object but only the neighbor solution that is eventually selected as the new current solution.

To illustrate the neighborhood traversal, we show a (simplified) generic method that implements the selection of the best neighbor (C++ code):

```
template <class S, class N>
N BestNeighbor( const S& s )
{
  N move = N( &s, FirstNeighbor );
  N best = move;
  if ( move.isValid() )
    ++move;
  while ( move.isValid() )
  {
    if ( *best < *move )
        best = move;
    ++move;
  }
  return best;
};
```

This template method is generic with respect to the solution space S and the neighborhood structure N, which are modeled as type parameters. For the dynamic parameter s, the first neighbor of this solution is constructed (i.e., the corresponding move). Then, in each iteration, the next move is constructed (increment operator), checked for validity, and compared with the best move obtained so far.

4.4.1.4 Commonalities and Variabilities of Problem-Specific Abstractions.

The principal problem-specific abstractions (solution space, neighborhood structure, ...) are modeled by corresponding interfaces. These interfaces define the requirements that must be fulfilled by problem-specific components to be applicable as realizations of respective type parameters of metaheuristic components. Such interfaces can be modeled as classes without data elements (by using the UML stereotype interface). Figure 4.8 shows simplified interfaces for solution space template classes and neighborhood template classes. These interfaces are generic: the solution space class depends on a problem class and a neighborhood class; the neighborhood class depends on a solution space class. While the latter dependence is apparent, one may argue that a solution space class should not depend on a neighborhood class. However, for principal reasons – only the solution class knows about the objective function while the neighborhood component defines the transformations – with regard to run-time efficiency, this is indeed a sensible model. This tight relationship between solution classes and neighborhood classes will be discussed on page 116.

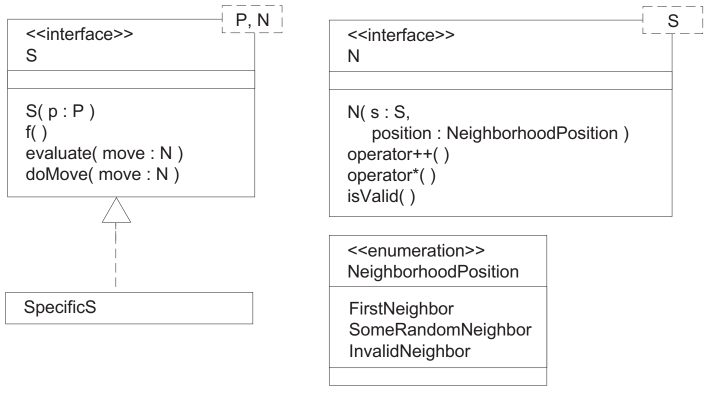


Figure 4.8 Interfaces for Solution Space Classes and Neighborhood Classes (Simplified)

The solution space interface defines the basic functionality that must be implemented by problem-specific classes: construction of a solution (given a problem instance), objective function computation, computation of the evaluation of a given move, modification of the solution according to a given move. The requirements for the neighborhood interface are in accordance with the discussion about the neighborhood traversal. That is, we need operations for the construction of a neighbor (i.e., the corresponding move) in dependence of a parameter that specifies which move is to be constructed (e.g., first versus random), for the increment to the next neighbor, for the evaluation of a move (star/dereference operator), and for the check of the validity of the move.

The relationship between the class SpecificS and the interface S, which is shown in Figure 4.8, denotes that the latter one is an implementation of the requirements set by the former one. HOTFRAME is based on such a clear separation between types (requirements defined by interfaces) and classes, which implement respective requirements.

An analysis of different types of problems shows that there are often quite similar solution spaces and neighborhood structures. So it seems reasonable to model respective commonalities with regard to data structures and algorithms by inheritance hierarchies; see Figure 4.9. This enables an (optional) reuse of the implementation of some common problem-specific structures. The objective function value of the current solution (evaluation) is obviously a common data element of the most general abstract solution space class Generic_S. There are two specializing classes (BV_S for the bit-vector solution space and Perm_S for the permutation solution space) that inherit this data element (as well as the obligation to implement the interface S) from the general class. BV_S and Perm_S, which add some specific data elements and operations, are still abstract (i.e., one cannot instantiate objects of these classes), because they lack an operation that computes the objective function.

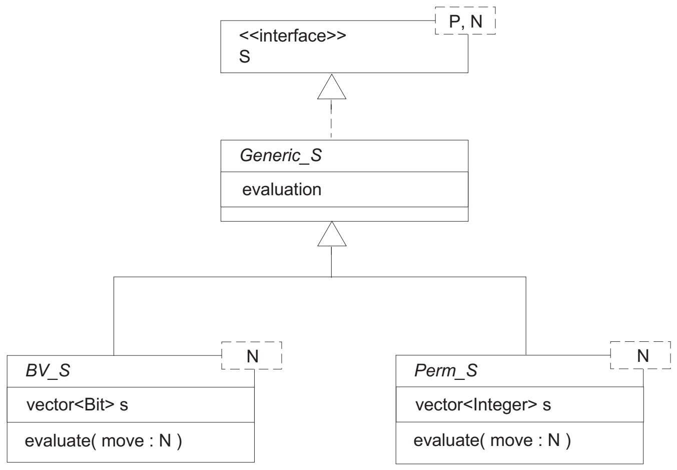


Figure 4.9 Inheritance Hierarchy of Solution Space Classes (Simplified)

4.4.2 Architecture

Building on the above discussion of the basic design ideas, in this section we describe the framework architecture (i.e., the framework components and their interplay) in more detail. For modeling variabilities we use the UML with some extensions. Unfortunately, “UML provides very little support for modelling evolvable or reusable specifications and designs” (Mens et al. (1999), p. 378), so we sporadically extend the notation if necessary.

4.4.2.1 Basic Configuration Mechanisms. Fixing the variable features of a reusable component may be called configuration. In the following, we describe the basic configuration mechanisms of HOTFRAME.

In accordance with generic programming, static configuration means fixing template parameters to define the primary behavior of reusable components (generic classes). Different metaheuristics have different options for configuration, which would lead to heterogeneous template parameter sets for different metaheuristic components. This complication can be avoided by using so-called *configuration components* that encapsulate parameter sets. That is, a configuration component defines, in accordance with respective requirements, a set of static parameters. These parameters mostly constitute type information, modeled as features of class interfaces. Since the UML provides no specific constructs to directly model requirements on such configuration components (the use of the Object Constraint Language (OCL) is not reasonable for our needs), we introduce a stereotype static interface to model respective type requirements (in a sense analogous to the common interface stereotype). By convention, we mark configuration components by a leading C, while denoting requirements by an R.

A basic kind of variability, which may concern all components, regards the numeric data types. We distinguish between a (quasi-)continuous type T (e.g., for cost data)

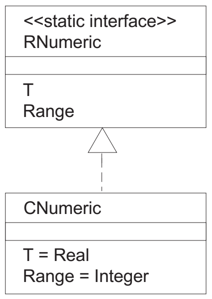


Figure 4.10 Realization of Requirements RNumeric by a Configuration Component CNumeric

and a discrete type Range (e.g., for counting objects). Both data types are required to be symmetric with respect to the representation of positive and negative values. For example, in C++, one typically defines T as float or double, and Range as int. The elementary configuration of components by T and Range is encapsulated in a configuration component with a static interface that has to conform to RNumeric as shown in Figure 4.10. CNumeric is a typical realization of such a configuration component, which may be implemented in C++ as follows:

```
struct CNumeric
{
  typedef float T;
  typedef int Range;
};
```

All problem-specific components require a configuration component that implements RNumeric. On top of this basic condition, the essential use of configuration components is to capture problem-specific type information of metaheuristic components. The specific requirements of metaheuristic components are defined in Section 4.4.2.3. In the following, we exemplify this concept for a component SteepestDescent.

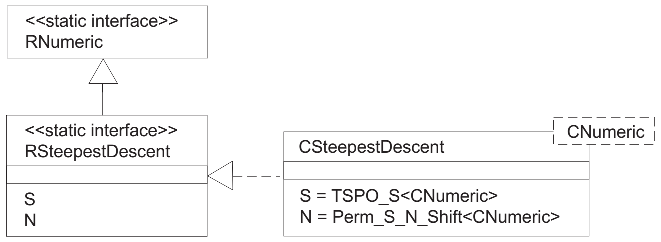


Figure 4.11 Configuration Component CSteepestDescent

Figure 4.11 illustrates modeling the requirements on the respective static interface. RSteepestDescent inherits the requirements of RNumeric, so CSteepestDescent has to define, in addition to S and N, the basic numeric types T and Range. This can be implemented in a modular way by defining CSteepestDescent as a template class where the template parameters model the numeric configuration. That is, CSteepestDescent actually deploys a hierarchical parameterization of configuration components by other configuration components. To simplify the presentation, we assume an implicit transfer of type definitions of configuration components that are used as template parameters. The components that are used in Figure 4.11 to define S and N are described in Section 4.4.2.2 and Section 4.6.2. The actual implementation of CSteepestDescent in C++ requires an explicit "transfer" of the defined types:

```
template <class CNumeric>
struct CSteepestDescent
{
  typedef typename CNumeric::T T;
  typedef typename CNumeric::Range Range;
  typedef TSPO_S<CNumeric> S;
  typedef Perm_S_N_Shift<CNumeric> N;
};
```

We illustrate the hierarchical configuration of metaheuristic components for IteratedLocalSearch (see Algorithm 1, p. 89). As described above, the configuration of metaheuristics can be decomposed in a problem-specific configuration C_P and a metaheuristic-specific configuration C_H. Accordingly, we may define a corresponding requirements hierarchy; see Figure 4.12. RpIteratedLocalSearch defines the requirements for the problem-specific configuration of IteratedLocalSearch. RIteratedLocalSearch defines the additional metaheuristic-specific requirements. The configuration component CpIteratedLocalSearch realizes RpIteratedLocalSearch in the same way as CSteepestDescent realizes RSteepestDescent (see Figure 4.11). On top of this, CIteratedLocalSearch realizes the requirements of RIteratedLocalSearch by using CpIteratedLocalSearch and additionally defining the neighbor selection rule. In Section 4.4.2.3, we shall describe the application of this design approach in more detail for different metaheuristic components.

A disadvantage of the template approach, at least when using C++, is that the template-based configuration is fixed at compile-time. Behavioral variability requirements may also be modeled by defining data elements with class-scope (in C++, such data element are called "static"). This approach is not appropriate if one wants to enable constructing objects with different state information (e.g., heuristics of the same kind with different parameter settings with respect to the termination criterion). However, class-scope is reasonable for configuration options with regard to modules of metaheuristics. Such components can indeed be parameterized by using data elements with class-scope (which are set according to respective data elements of the configuration component). This design, which is illustrated in Figure 4.13 for the cooling schedule GeometricCooling, enables the dynamic variation of respective configuration parameters at run-time. To simplify the presentation, we do not explicitly model the requirements of such components on the configuration component in the diagrams, but we assume respective requirements as implicitly defined. For example, we implic-

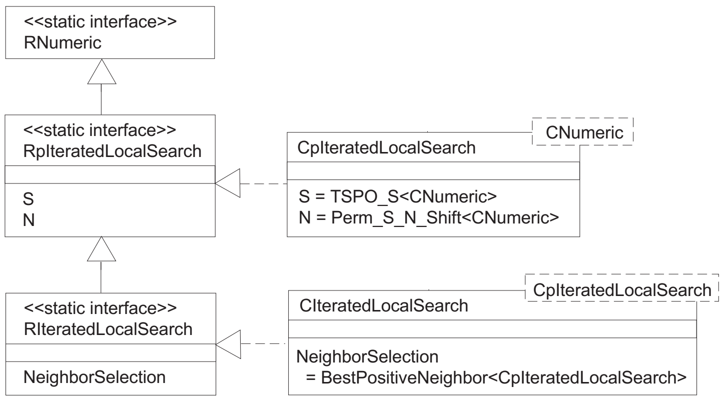


Figure 4.12 Configuration Component CIteratedLocalSearch

itly assume the configuration component CSimulatedAnnealing to be subject to the requirement of defining a numeric data element alpha.

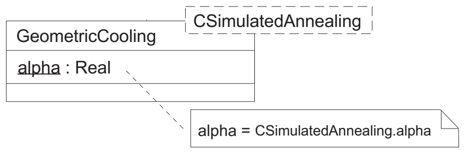


Figure 4.13 Static Parameterization of Components by Numeric Data Elements with Class-Scope

The design provides, from the application point-of-view, a flexible, efficient, and straightforward mechanism to construct components with a special functionality by a composition of adaptable elementary modules (see below).

The natural mechanism to dynamically configure objects is by plain data elements of classes, which are initialized in connection with object construction (in C++, by using constructor parameters). The specification of the termination criterion is the most obvious kind of a dynamic configuration. In accordance with the specification of the metaheuristics IteratedLocalSearch, GeneralSimulatedAnnealing, SimulatedAnnealingJetal, and TabuSearch, the dynamic parameterization conforms to the interfaces of the Algorithms 1, 5, 10, and 11, respectively.

In addition to the usual termination criteria (e.g., iteration number and run-time), we define, for each metaheuristic component, an object parameter that represents the asynchronous termination criterion ω (see p. 88). The plain abstract interface of a corresponding base class, which is called OmegaInterface, is shown in Figure 4.14. The class OmegaInterface is parameterized by the solution space. Metaheuristic components provide the interface with the current solution after each iteration (by calling the operation newSolution(s : S)). Furthermore, metaheuristics call the operation omega() after each iteration – with the metaheuristic being terminated when the operation returns true. For example, one may want to terminate the search process when an "acceptable" solution has been obtained. In general, this design enables the implementation of external termination criteria in online settings.

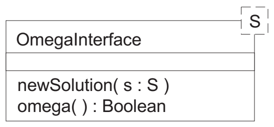


Figure 4.14 Base Class OmegaInterface

Additional parameters of metaheuristics are also modeled as data elements and corresponding dynamic parameters of constructors. This concerns both simple numeric parameters (such as the initial temperature of simulated annealing) and more complex kinds of a configuration. In particular, the tabu criterion of Algorithm 11 (see p. 96) is implemented as an object parameter, which may be explained as follows. The tabu criterion depends on actual state information (search history). The same is true for the diversification method (as a specific instance of some metaheuristic). So both variation points (TabuCriterion, Diversification) are implemented as dynamic object parameters, which allows a flexible use of such objects at run-time. (On the contrary, the static variation point *TabuNeighborSelection* is modeled as a template parameter; see p. 133.)

Variable requirements with regard to the introspection of the search process must also be modeled in a flexible and efficient way. We apply an extensible class hierarchy that represents the different kinds of introspection (e.g., a complete documentation of the move selection in every iteration versus the return of only the final solution). In Section 4.4.2.4, we describe the interface of a base class, which defines a set of operations, which are called by metaheuristic components to convey information about the search process. By deriving classes from this base class and implementing the operations in the needed form, one can model specific introspection requirements. Metaheuristics are called with instances of these classes as dynamic object parameters to implement the introspection interface.

4.4.2.2 Problem-Specific Components. In this section, we define problem-specific components in accordance with the algorithmic specifications from Section 4.3.1. In particular, we describe the component interfaces, which implicitly establishes the basic form of the interplay among components.

We do not need to define (abstract) base classes, since the problem-specific components are used to statically configure metaheuristic components by means of template parameters. Nevertheless, due to implementation reuse, it is possible to model commonalities of problem-specific components by inheritance hierarchies, which allows reuse of common data structures and algorithms (see pp. 122). Such inheritance hierarchies simplify the framework application for suitable problem types, while they do not restrain the possibilities for problem-specific adaptations (due to their optional character).

The following overview especially refers to problem-specific components and their interdependencies. Specific realizations of such components are by convention marked by a leading "X_", where X refers to the problem type that is represented. We assume that the problem-specific abstractions introduced in Section 4.3.1 are implemented by the following components:

- Problem P $\rightarrow$ Problem component X_P
- Solution space S $\rightarrow$ Solution component X_S
- (Hash-)Function h $\rightarrow$ Solution information component X_S_I
- Neighborhood N_S $\rightarrow$ Neighborhood component X_S_N
- Solution attribute out of Ψ $\rightarrow$ Attribute component X_S_A

The intended application of metaheuristics implies the need to implement a respective subset of the problem-specific components according to corresponding interfaces. While the components X_P and X_S must be implemented for all kinds of applications, one needs no neighborhood component for evolutionary methods (without local search hybridization). Components X_S_I and X_S_A are only required for particular tabu search methods.

The class diagram shown in Figure 4.15 models the typical relations between problem-specific components (neglecting data structures, operations, and template parameters). The basic dependencies may be classified as usage relationships (use-stereotype) and derivation relationships (derive-stereotype). Furthermore, there are explicit structural references that represent direct associations between objects of respective classes. The suitability of a problem-specific component as an element of a configuration component only depends on fulfilling the requirements of the interfaces that are described later on. In the class diagram, this is indicated by respective realization relationships.

The solution component X_S uses the problem component X_P to access the problem data. Accordingly, we need a reference from a solution object to the respective problem object. The information that is represented by X_S_I is derived from the data of a respective solution object. Since the state of solution objects is transient we indeed need X_S_I objects to capture such information. The same argument applies to the attribute component X_S_A. In general, attributes can be derived both from solution objects and from move (neighbor) objects. With regard to the latter, we have to distinguish between ("created") *plus attributes* and ("destroyed") *minus attributes*.

The neighborhood component X_S_N models the neighborhood traversal as described in Section 4.4.1.3. In general, the semantics of X_S_N depends on X_S, as one needs to know about the objective function (implemented as part of X_S) to be able to evaluate moves. On the contrary, X_S depends on X_S_N, because the execution of a move to a neighbor alters the solution object (X_S implements the solution data structures while X_S_N models the transformational information).

The tight relationship between, in particular, X_S and X_S_N requires to model respective interdependencies in the types of interface parameters of specific operations. For example, the solution component includes an operation that executes a move; the

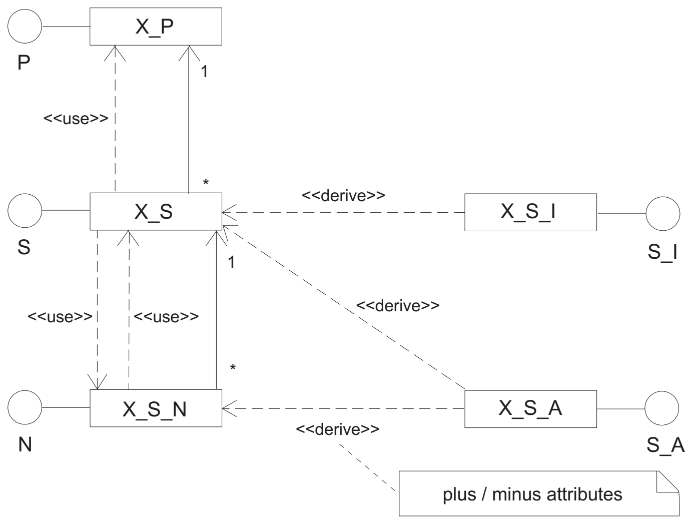


Figure 4.15 Basic Dependencies Between Problem-Specific Components

respective interface must reflect that we may have different types of neighborhoods. Because of the lack of an appropriate UML notation, we call such operations *abstract* (with italicization of respective type identifiers). (Under some restrictions, such a variability requirement could be implemented in C++ by the member template construct (i.e., member functions with separate type parameters), which would provide a modular, efficient and type-safe implementation. As shall be explained below, we actually cannot use this mechanism but have to rely on basic code supplements.)

Problem Component. A problem type P is implemented by a corresponding problem component P. In dependence on the considered type of problem, problem data is modeled by specific data structures (e.g., cost vectors). According to the object-oriented concept of encapsulation of implementation details, the component interface defines the access to the data of problem instance objects. Appropriate constructors must be defined. For example, constructor parameters may point to an input stream, which provides problem data in a specific text format. With regard to a random generation of problem instances, constructor parameters may define the respective probability distribution. The problem component may also serve as an online proxy that connects the metaheuristic to an application system or a data base that comprise the problem instance. In general, respective constructor and access operations depend on the considered application, so we only require one common interface element for all problem components: There should be a serialization of the problem data, which prints to a given output stream; see Figure 4.16.

Solution Component. The basic purposes of a solution component are the construction and representation of solutions from a solution space S, the computation of the objective function, and the modification of solutions according to a given move

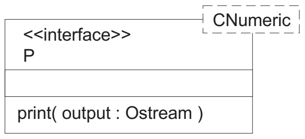


Figure 4.16 General Interface of Problem Components

to a neighbor solution. Figure 4.17 shows the general interface of solution components. (By the stereotype local search, we denote that the subsequent operations are required only if one wants to apply a local search procedure.)

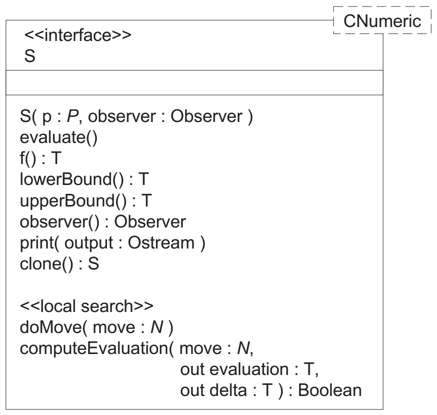


Figure 4.17 General Interface of Solution Components

Realizing the interface S by a corresponding component means implementing the following operations with "algorithmic content":

- S(p : *P*, observer : Observer)
  The constructor builds an initial solution for a given problem p. Additional parameters may be needed with respect to the actual rules of construction (e.g., random construction versus application of a simple construction algorithm).

- evaluate()
  This operation computes (and stores) the objective function value of the actual solution. Calling this operation from the outside is usually not necessary, since the operation f() returns by definition the objective function value of the current solution.

- doMove(move : *N*)
  There are two basic options to implement the modification of a solution:

  1. The neighborhood component transforms the data of the solution component, which requires the neighborhood component to know about the

respective data structures. This enables the introduction of new neighborhood structures without the need to modify solution components.

2. The solution component interprets the modification due to the move and modifies its own data accordingly. In this case, introducing new neighborhood structures requires the adaptation of an existing doMove operation (or the addition of a new doMove operation if the method is statically parameterized by the actual neighborhood type).

Due to reasons of efficiency, it is apparently not possible to dissolve this tight relationship between the solution component and the neighborhood component in a strict object-oriented manner (where each class should fully encapsulate its internal implementation).

- computeEvaluation(move : *N*, out evaluation : T, out delta : T) : Boolean
  Since only the solution component knows about the computation of the objective function, the primary responsibility to actually assess the advantageousness of a move should be assigned to the solution component. This also leads to a tight relationship between the solution component and the neighborhood component. The return parameters evaluation and delta represent the evaluation of a move and the implied change of the objective function value, respectively. These values may equal each other for many applications. In other cases, an appropriate move evaluation may require a special measurement function. In particular, this differentiation is reasonable when there is some kind of min-max objective function, where a single move usually does not directly affect the objective function. Furthermore, for some kinds of problems the exact computation of the implied change of the objective function value may be too costly as to be done for each of the considered moves. In such a case one needs to estimate the "quality" of a move in an approximate manner without computing delta; this must be indicated by returning false (otherwise the operation returns true).

  If moves are evaluated by the implied change of the objective function value, a standard implementation of this operation may be available by actually performing the move (for a copy of the solution object), computing the objective function of the resulting solution from scratch, and finally comparing the respective objective function value with the objective function value of the current solution. However, for reasons of efficiency, one usually has to implement a special, adaptive form of the move evaluation.

The following operations primarily serve for the encapsulation of the data structures of the solution component:

- f() : T
  Return of the objective function value of the solution.

- lowerBound() : T
  Return of the best known lower bound for the optimal objective function value (remember that we consider minimization problems).

- upperBound() : T
  Return of the best known upper bound for the optimal objective function value.
- observer() : Observer
  Return of the observer object.
- print(output : Ostream)
  Print the solution in some defined format to an output stream.

Furthermore, one must implement an operation clone(), which constructs a copy of the actual solution object (in the sense of a *virtual constructor*, see Stroustrup (1997), pp. 424). For some metaheuristics (in particular, evolutionary algorithms), one may need an operation that computes the "difference" between two solutions according to some measurement function (see, e.g., Woodruff (2001)). This requirement might have been specified as a member function of the solution component. However, to simplify the interface of the solution component, we use free template functions distance (with a static template parameter S and both solution objects as dynamic parameters).

Solution Information Component. A solution information component models elements of the set S_h. Respective objects are used to store the search trajectory, which means that the solution information component is only needed for tabu search methods that use such trajectory information. In accordance with this special role, the interface (and the corresponding functionality) of the solution information component is quite simple; see Figure 4.18 and the following description.

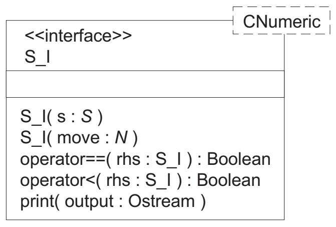

CNumeric
<<interface>>
S_I
S_I(s : S)
S_I(move : N)
operator==(rhs : S_I) : Boolean
operator<(rhs : S_I) : Boolean
print(output : Ostream)

Figure 4.18 General Interface of Solution Information Components

- S_I(s : *S*)
  Construction of the object that corresponds to $h(s)$ by computing the transformation $h : S \to S_h$.
- S_I(move : *N*)
  Due to reasons of efficiency, we require a means to directly construct the solution information of the neighbor solution that corresponds to a given move. (One may also construct the solution information of a neighbor by actually constructing the solution object and deriving the solution information from this object.)
- operator==(rhs : S_I) : Boolean
  The definition of the equivalence relation is needed to implement the trajectory data structures.

- operator$<$(rhs : S_I) : Boolean
  Tree data structures, which may be used to store the trajectory, require the definition of an order operator.
- print(output : Ostream)
  Print the solution information in some defined format to an output stream.

Usually, the solution information component models a hash-function. In this case, the constructors include the respective computations, while the operators are simply implemented as the comparison of integer values.

Neighborhood Component. Neighborhood components (neighborhood iterators) represent neighborhood structures N_S (i.e., respective moves from N_S^μ) for a solution space S. Move object data generally consists of a reference to the actual solution and information about the transformation to a neighbor solution. In accordance with the discussion in Section 4.4.1.3, the basic functionality of neighborhood components is shown in Figure 4.19.

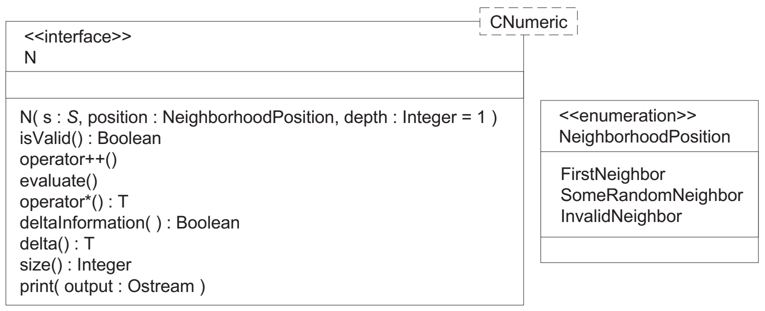


Figure 4.19 General Interface of Neighborhood Components

- N(s : *S*, position : NeighborhoodPosition, depth : Integer = 1)
  To construct a neighbor (i.e., the respective move) for a solution s one specifies by using the parameter position whether the first, a random, or the invalid neighbor is to be constructed. The optional parameter depth may define a neighborhood depth greater than one (i.e., a respective concatenation of elementary moves of the basic neighborhood structure).
- isValid() : Boolean
  This operation returns true if and only if the actual move is valid. The following operations may only be called for valid moves.
- operator++()
  Increment of a valid move to the next neighbor.
- evaluate()
  The evaluation of a move (according to $\hat{f}$) usually occurs when a move is constructed or incremented. The implementation of the evaluation may be delegated to the respective solution class; see the discussion in Section 4.4.2.2.

- operator*() : T
  This operation returns the move evaluation (according to $\hat{f}$).

- deltaInformation() : Boolean
  The value true is returned if and only if one may use the operation delta() to ask for the implied change of the objective function value that is due to the move; see the discussion in Section 4.4.2.2.

- delta() : T
  This operation returns the implied change of the objective function that is due to the actual moves if and only if this information is available (see the operation deltaInformation()).

- size() : Integer
  Some metaheuristics need an estimation of the neighborhood size (e.g., Algorithm 10, p. 94). For reasons of efficiency, we do not require this operation to return the exact number of neighbors of the actual solution, but only a reasonable upper bound.

- print(output : Ostream)
  Print the move in some defined format to an output stream.

The sequence diagram of Figure 4.20 illustrates the interaction between solution and neighborhood components by describing the typical neighborhood traversal of a local search approach. At the beginning, the move to the first neighbor of the actual solution is constructed and evaluated. Iteratively, the move is incremented to the next neighbor until we have reached the invalid neighbor. Finally, the best move found will be executed.

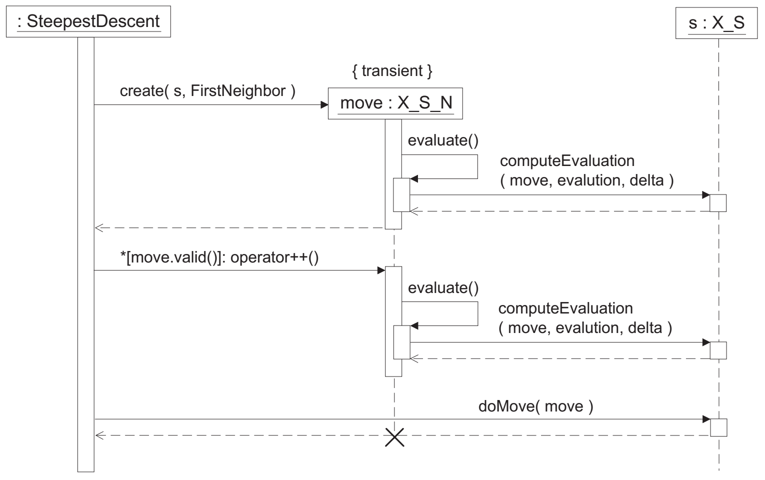

: SteepestDescent
s : X_S
{ transient }
create(s, FirstNeighbor)
move : X_S_N
evaluate()
computeEvaluation (move, evalution, delta)
*[move.valid()]: operator++()
evaluate()
computeEvaluation (move, evalution, delta)
doMove(move)

Figure 4.20 Sequence Diagram of the Interaction Between Solution and Neighborhood Components

Attribute Component. Attribute components represent elementary solution attributes. From the point of view of moves, we have to distinguish between plus and minus attributes. Such information is used by tabu search methods that apply an attribute-based memory. (The decomposition of solutions in elementary moves may be used by tabu search methods that build new solutions by combining attributes from a *vocabulary* of "promising" solution attributes.) The general interface of attribute components is shown in Figure 4.21.

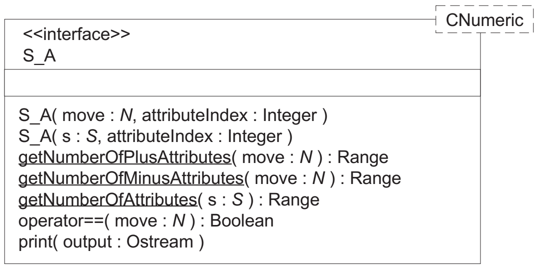

CNumeric
<<interface>>
S_A
S_A(move : N, attributeIndex : Integer)
S_A(s : S, attributeIndex : Integer)
getNumberOfPlusAttributes(move : N) : Range
getNumberOfMinusAttributes(move : N) : Range
getNumberOfAttributes(s : S) : Range
operator==(move : N) : Boolean
print(output : Ostream)

Figure 4.21 General Interface of Attribute Components

- S_A(move : *N*, attributeIndex : Integer)
  Move attributes are constructed in dependence on the parameter attributeIndex. Positive and negative values indicate plus and minus attributes, respectively. For example, the move attribute $\psi_1^-(\mu)$ is constructed when attributeIndex is set to -1.

- S_A(s : *S*, attributeIndex : Integer)
  The operation constructs the attributes of a solution.

- getNumberOfPlusAttributes(move : *N*) : Range
  This operation returns the number of plus attributes of a move ($|\psi^+(\mu)|$). The result of this and the next two operations does not depend on a specific object. (In C++, this leads to an implementation as static class methods.)

- getNumberOfMinusAttributes(move : *N*) : Range
  This operation returns the number of minus attributes of a move ($|\psi^-(\mu)|$).

- getNumberOfAttributes(s : *S*) : Range
  This operation returns the number of attributes of a solution ($|\psi(s)|$).

- operator==(move : *N*) : Boolean
  The equivalence relation between an attribute and a move reflects whether the actual attribute is "destroyed" when the considered move is executed. That is, this operation returns true if and only if the attribute corresponds to a minus attribute of the move. (For reasons of efficiency, we do not rely on the explicit construction and comparison of all minus attributes of a move, but require the implementation of this redundant operation.)

- print(output : Ostream)
  Print the attribute in some defined format to an output stream.

Standard Problem-Specific Components. As discussed in Section 4.4.1.4, different types of problems often possess some commonalities with respect to the solution space. So it seems reasonable to exploit these commonalities by defining, implementing, and (re)using problem-specific components, which provide respective data structures and algorithms. In particular, a factoring of common concepts may be modeled by inheritance hierarchies refering to the problem-specific components/interfaces that have been defined above. The goal is to provide a set of reusable classes, from which a user of the framework can derive special classes according to his application. Then, one has to implement the remaining concerns that are specific to the considered application (e.g., the objective function), while general data structures and algorithms with respect to the solution space, neighborhood structure, etc. may be reused. To summarize, applying HOTFRAME in any case means reuse of the respective architecture and metaheuristics components – if one of the standard problem-specific components that are available fits the considered problem, there is also the option to exploit implementation reuse on the problem side.

In the following, we describe standard problem-specific components for bit-vector and permutation solution spaces. The solution components BV_S and Perm_S represent solutions of the kind $x = (x_1, \ldots, x_n), x_i \in \{0, 1\}, 1 \leq i \leq n$, and $\Pi = (\pi_1, \ldots, \pi_n)$, with Π being a permutation of $(1, \ldots, n)$, respectively. For these solution spaces, we implement commonly used neighborhood structures as respective neighborhood components. (HOTFRAME also includes components for combined assignment and sequencing problems, i.e., a solution is defined by an ordered assignment of some kind of objects to some kind of resources, which are not described in this paper.)

For BV_S, the neighborhood component BV_S_N represents neighbors (i.e., corresponding moves) that result from (one or more) bit inversions. For sequencing problems, there exist several popular neighborhood structures. So we introduce an additional layer in the class hierarchy by defining an abstract component Perm_S_N, from which three specific neighborhood components are derived, which represent three different neighborhood structures with a neighborhood size of $O(n^2)$. The component Perm_S_N_Shift represents neighbors that result from shifting one object to another position (with the option to restrict the maximum "shift distance"). Swaps of two objects are represented by the component Perm_S_N_Swap. The 2-exchange neighborhood (component Perm_S_N_2Exchange) represents those neighbors that result from replacing two predecessor-successor relations (edges when considering the sequence in a graph representation) by two other predecessor-successor relations so that a feasible solution (sequence) results; this move corresponds to an inversion of a partial sequence. There are quite a few additional neighborhood structures for sequencing problems, which may be represented by additional neighborhood components; see, e.g., Tian et al. (1999).

As an extension to Figure 4.9 (see p. 109), Figure 4.22 shows the inheritance relationship of solution components. Figure 4.23 shows the according diagram for neighborhood components. The usual way of reusing these components is by deriving a new solution component and applying some suitable neighborhood component unchanged. (The close relationship between solution components and neighborhood components ("*dual inheritance hierarchies*") leads to some subtle problems, which will be briefly discussed below.)

In the abstract base class Generic_S, we define data elements and operations that are common to all solution components. In particular, every solution component has data elements to store the objective function value, an lower and upper bound, and a reference to an observer object. For all such data elements, we define corresponding access operations. With regard to other methods, we can only define abstract interfaces, which have to be complemented by respective implementations in derived classes. (According to the UML, such operations are shown italicized in the diagram.) To simplify the diagrams, we do not show all access operations for data elements (which are implicitly assumed to exist with the identifier set according to the data element).

In the derived classes BV_S and Perm_S, one needs to implement, based on the neighborhood, the operations for the default evaluation of moves (according to the implied change of the objective function value that is due to the considered move) and the execution of moves for particular neighborhood structures/components. So we must provide respective adaptation mechanisms. Due to subtle technical reasons (see Stroustrup (1997), p. 348), C++ does not allow member template methods to be virtual, which unfortunately prevents us from using member template constructs to implement those operations for specific neighborhoods. So we must refer in the interface declarations of these operations to the most general neighborhood type (Generic_S_N). Implementing those member functions requires dynamic type-casts ("downcasts") for respective neighborhood types, applying run-time-type information (RTTI) to check for type-conformance. Adding new neighborhood structures requires changing these member functions. (See the discussion of this crucial problem of object-oriented design with regard to dual inheritance hierarchies ("codependent domains") by Martin (1997) and Coplien (1999), pp. 210–227.)

Implementing a concrete solution component with a specific objective function for a problem that may be modeled by a bit-vector or permutation solution space means deriving a new solution component from BV_S or Perm_S, respectively. To meet the requirements defined by the solution component interface S one has to implement an appropriate constructor, the clone operation, and the computation of the objective function. With regard to run-time efficiency, one may also need to implement an adaptive move evaluation.

The definition of BV_S includes a specific addition to the solution component. The member functions shown below the variable fixation stereotype enable – in combination with appropriate neighborhood components, which use these operations – restricting the neighborhood by excluding variables of the solution vector from consideration regarding bit inversions. That is, the value of some of the variables may be temporarily fixed by an appropriate implementation of the operations firstVariable, nextVariable, and randomVariable.

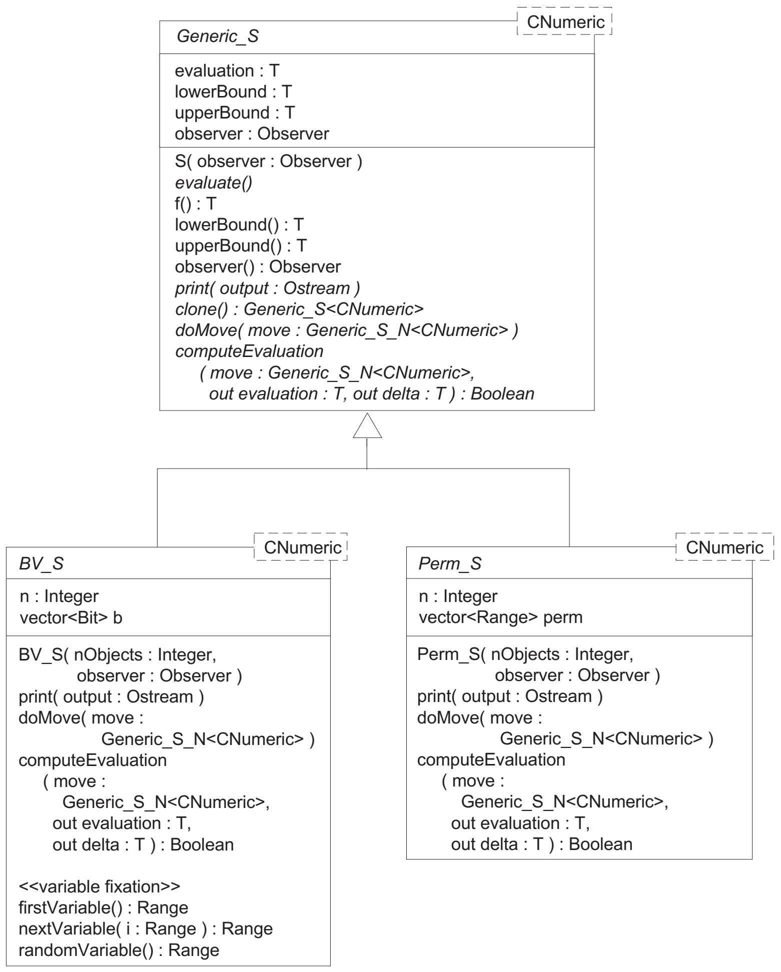

Generic_S
CNumeric
evaluation : T
lowerBound : T
upperBound : T
observer : Observer
S(observer : Observer)
evaluate()
f() : T
lowerBound() : T
upperBound() : T
observer() : Observer
print(output : Ostream)
clone() : Generic_S<CNumeric>
doMove(move : Generic_S_N<CNumeric>)
computeEvaluation (move : Generic_S_N<CNumeric>, out evaluation : T, out delta : T) : Boolean
BV_S
CNumeric
n : Integer
vector<Bit> b
BV_S(nObjects : Integer, observer : Observer)
print(output : Ostream)
doMove(move : Generic_S_N<CNumeric>)
computeEvaluation (move : Generic_S_N<CNumeric>, out evaluation : T, out delta : T) : Boolean
<<variable fixation>>
firstVariable() : Range
nextVariable(i : Range) : Range
randomVariable() : Range
Perm_S
CNumeric
n : Integer
vector<Range> perm
Perm_S(nObjects : Integer, observer : Observer)
print(output : Ostream)
doMove(move : Generic_S_N<CNumeric>)
computeEvaluation (move : Generic_S_N<CNumeric>, out evaluation : T, out delta : T) : Boolean

Figure 4.22 Inheritance Hierarchy for Solution Components

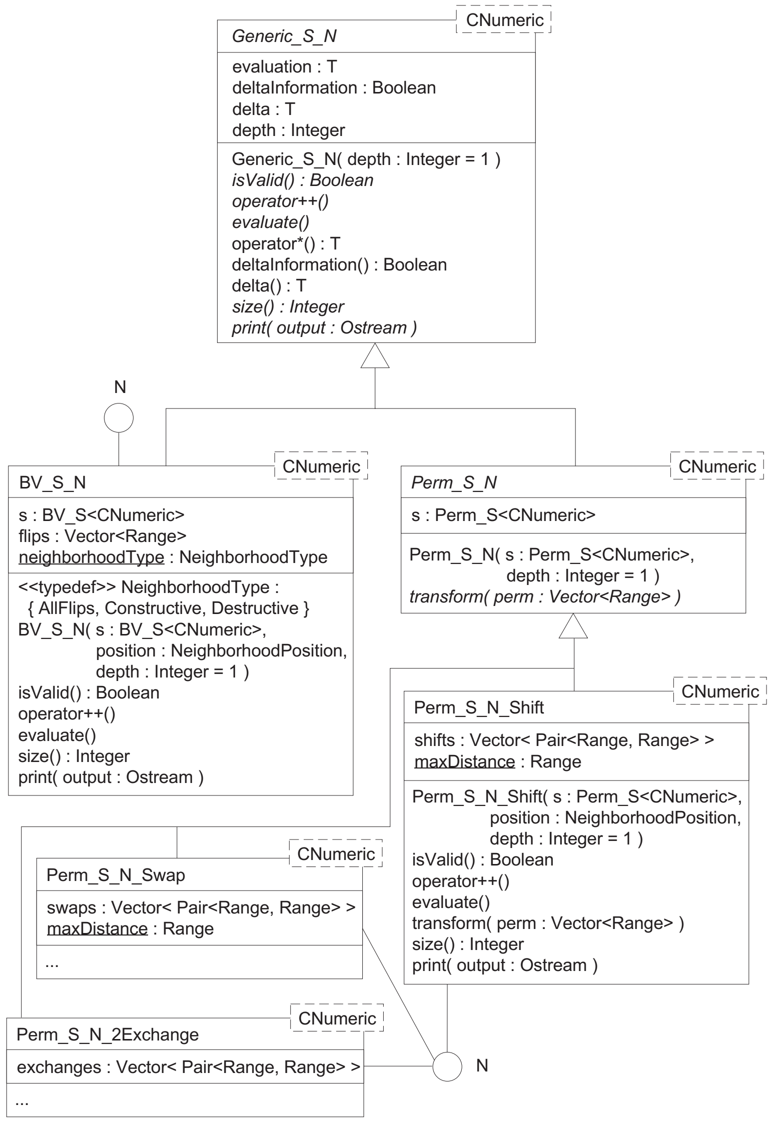

CNumeric
Generic_S_N
evaluation : T
deltaInformation : Boolean
delta : T
depth : Integer
Generic_S_N(depth : Integer = 1)
isValid() : Boolean
operator++()
evaluate()
operator*() : T
deltaInformation() : Boolean
delta() : T
size() : Integer
print(output : Ostream)
N
CNumeric
BV_S_N
s : BV_S<CNumeric>
flips : Vector<Range>
neighborhoodType : NeighborhoodType
<<typedef>> NeighborhoodType : { AllFlips, Constructive, Destructive }
BV_S_N(s : BV_S<CNumeric>, position : NeighborhoodPosition, depth : Integer = 1)
isValid() : Boolean
operator++()
evaluate()
size() : Integer
print(output : Ostream)
CNumeric
Perm_S_N
s : Perm_S<CNumeric>
Perm_S_N(s : Perm_S<CNumeric>, depth : Integer = 1)
transform(perm : Vector<Range>)
CNumeric
Perm_S_N_Shift
shifts : Vector< Pair<Range, Range> >
maxDistance : Range
Perm_S_N_Shift(s : Perm_S<CNumeric>, position : NeighborhoodPosition, depth : Integer = 1)
isValid() : Boolean
operator++()
evaluate()
transform(perm : Vector<Range>)
size() : Integer
print(output : Ostream)
CNumeric
Perm_S_N_Swap
swaps : Vector< Pair<Range, Range> >
maxDistance : Range
...
CNumeric
Perm_S_N_2Exchange
exchanges : Vector< Pair<Range, Range> >
...
N

Figure 4.23 Inheritance Hierarchy for Neighborhood Components

Common data elements and respective access operations of neighborhood components are defined in the abstract base class Generic_S_N. For the bit-vector solution space, the bit inversion neighborhood is implemented by the component BV_S_N, which is directly derived from Generic_S_N. The BV_S_N component defines as data elements a reference to the respective solution object and a vector of integer numbers, which define the variables that are to be flipped. This enables modeling moves that comprise multiple bit inversions (note the depth parameter in Generic_S_N). By the static (class-scope) parameter neighborhoodType one can restrict the neighborhood to "constructive" or "destructive" moves, which means that only bit inversion from 0 to 1 or 1 to 0 are considered, respectively.

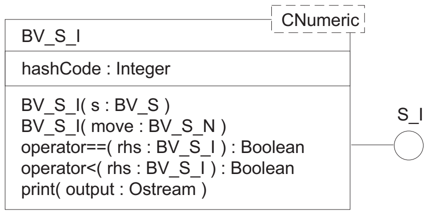
In addition to the constructors, the specific neighborhood components implement those operations that are defined in the base classes as virtual. As discussed above, the evaluation of moves is usually implemented by a delegation to the solution component.

The commonalities of the neighborhood components Perm_S_N_Shift, Perm_S_N_Swap, and Perm_S_N_2Exchange are modeled by an abstract class Perm_S_N. This component defines a reference to a solution object of Perm_S, a constructor, and the abstract member function transform, which represents the move's transformation of the solution data structure according to the implementation in the derived classes. Specialized neighborhood components include data structures that represent the move. For example, in Perm_S_N_Shift, a pair of integer numbers defines which object has to be shifted to which position. Vectors of such pairs define concatenations of moves. For Perm_S_N_Shift and Perm_S_N_Swap, the data element maxDistance can be used to restrict the neighborhood with regard to the shift or swap distance, respectively. The operations of Perm_S_N_Swap and Perm_S_N_2Exchange, which are not shown in Figure 4.23, are defined in the same way as Perm_S_N_Shift.

Solution information components and attribute components are only needed for particular tabu search methods. There is no need to define common base classes, since there are no general reference relations that refer to these components. Moreover, there are no common data structures, which may be modeled in a base class. So we define separate standard components for respective solution spaces. These components fully implement the interfaces S_I and S_A, respectively, and thus can be applied right away.

The solution information components, which are defined in Figure 4.24, represent solutions by applying hash-functions (see p. 119). The respective constructors compute hash-values for solution objects and for neighbor objects. Pre-defined implementations are available for the standard solution spaces and neighborhood structures that have been described above.

The attribute components shown in Figure 4.25 represent attributive information with regard to solutions and neighborhoods. The attributes match the elements of bit-vector and permutation solutions (elementary bit inversions and predecessor-successor-relations). Implementations for the introduced solution spaces and neighborhood structures are available.


CNumeric
BV_S_I
hashCode : Integer
BV_S_I(s : BV_S)
BV_S_I(move : BV_S_N)
operator==(rhs : BV_S_I) : Boolean
operator<(rhs : BV_S_I) : Boolean
print(output : Ostream)
S_I

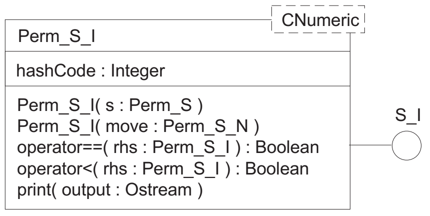
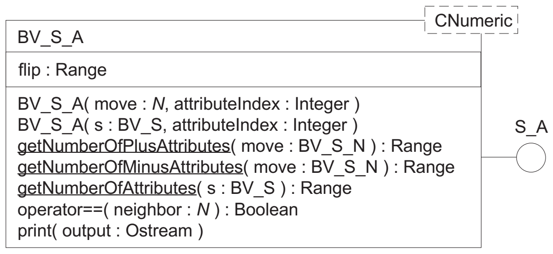

CNumeric
Perm_S_I
hashCode : Integer
Perm_S_I(s : Perm_S)
Perm_S_I(move : Perm_S_N)
operator==(rhs : Perm_S_I) : Boolean
operator<(rhs : Perm_S_I) : Boolean
print(output : Ostream)
S_I

Figure 4.24 Solution Information Components


CNumeric
BV_S_A
flip : Range
BV_S_A(move : N, attributeIndex : Integer)
BV_S_A(s : BV_S, attributeIndex : Integer)
getNumberOfPlusAttributes(move : BV_S_N) : Range
getNumberOfMinusAttributes(move : BV_S_N) : Range
getNumberOfAttributes(s : BV_S) : Range
operator==(neighbor : N) : Boolean
print(output : Ostream)
S_A

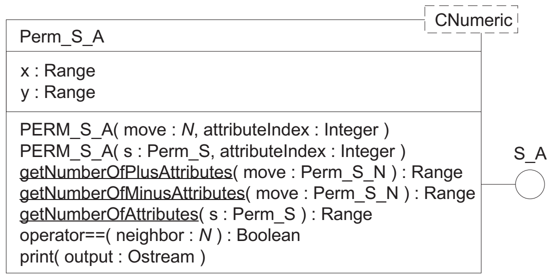

CNumeric
Perm_S_A
x : Range
y : Range
PERM_S_A(move : N, attributeIndex : Integer)
PERM_S_A(s : Perm_S, attributeIndex : Integer)
getNumberOfPlusAttributes(move : Perm_S_N) : Range
getNumberOfMinusAttributes(move : Perm_S_N) : Range
getNumberOfAttributes(s : Perm_S) : Range
operator==(neighbor : N) : Boolean
print(output : Ostream)
S_A

Figure 4.25 Solution Attribute Components

4.4.2.3 Metaheuristic Components. The metaheuristic concepts that have been described in Section 4.3.2 are implemented by corresponding metaheuristic components. In the following, we describe the most important operations of these components, which also includes the requirements with regard to respective configuration components; see Section 4.4.2.1. (To simplify the presentation, we do not explicitly show data elements and corresponding access operations.)

The general interface Heuristic of a metaheuristic is shown in Figure 4.26. To enable a flexible (polymorphic) application of respective algorithmic objects, specific metaheuristic classes of the framework are derived from a common base class (also named Heuristic). This base class is statically parameterized by the solution space.

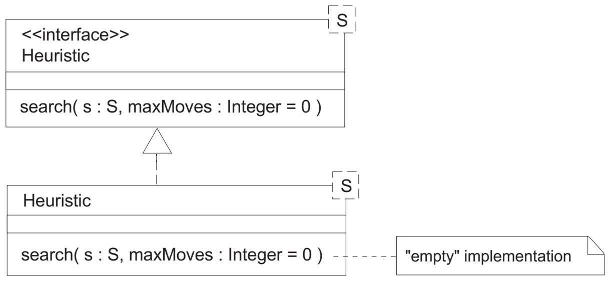

S
<<interface>>
Heuristic
search(s : S, maxMoves : Integer = 0)
S
Heuristic
search(s : S, maxMoves : Integer = 0)
"empty" implementation

Figure 4.26 General Interface and Base Class Heuristic

The derivation of metaheuristic components from Heuristic, which is shown in Figure 4.27, follows the well-known strategy pattern; see Gamma et al. (1995). First of all, metaheuristic components provide an operation search, which transforms a solution. The optional parameter maxMoves facilitates such a modification of the termination criterion, which may be used, e.g., when triggering an adaptive diversification. The operation search of the base class Heuristic is defined as an "empty" method, so that respective objects can be used when one needs a default behavior for some feature (e.g., a non-existent diversification).

Iterated Local Search. Iterated local search and similar methods, which have been discussed in Section 4.3.2, are implemented by the component IteratedLocalSearch; see Figure 4.28. The static configuration of this component as well as the implementation of the features Diversification and Ω as dynamic object parameters have already been discussed in Section 4.4.2.1. The parameter depth determines the neighborhood depth. The parameter returnBest defines whether the algorithm should return the best solution found or the last traversed solution.

The static parameter NeighborSelection, which is an element of the configuration of IteratedLocalSearch, must conform to the interface defined in Figure 4.29, which also shows some components that realize popular neighbor selection rules.

The derivation of specific metaheuristic components is exemplified in Figure 4.30. IteratedSteepestDescent is defined by fixing the type parameter NeighborSelection accordingly.

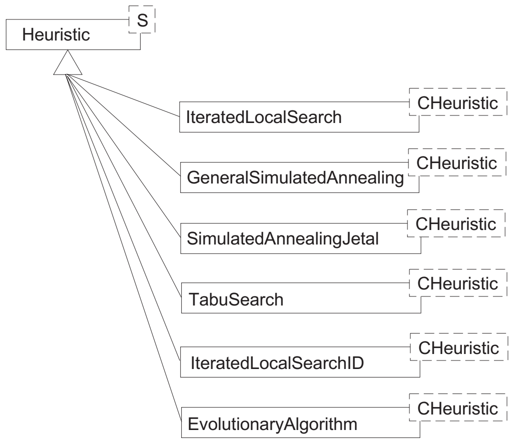


Figure 4.27 Derivation of Specific Metaheuristic Components from Heuristic

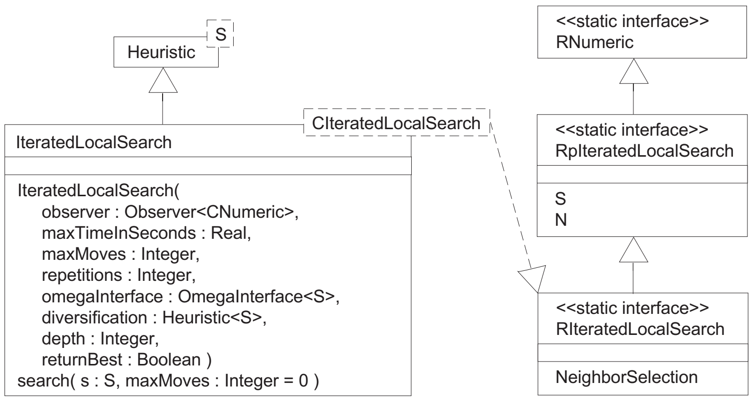


Figure 4.28 Component IteratedLocalSearch

Simulated Annealing and Variations. Simulated annealing and variations, which have been described in Section 4.3.2.2, are implemented by the components GeneralSimulatedAnnealing and SimulatedAnnealingJetal shown in Figure 4.31. With regard to the metaheuristic-specific configuration, one has to specify the features acceptance criterion, cooling schedule, and an optional reheating scheme. In the following, we describe the general interface of respective components and some common instances of these interfaces. With regard to the implementation in C++, these modules are applied by using class-scope parameters (in accordance with the discussion on p. 111).

The interface AcceptanceCriterion is defined in Figure 4.32. The acceptance criterion is implemented by the operation check. In general, the acceptance of a move depends on the current value of the temperature parameter (tau), the evaluation of

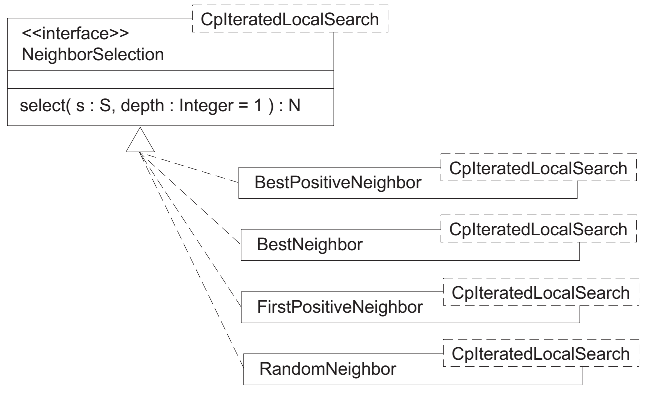


Figure 4.29 Interface NeighborSelection and Respective Components

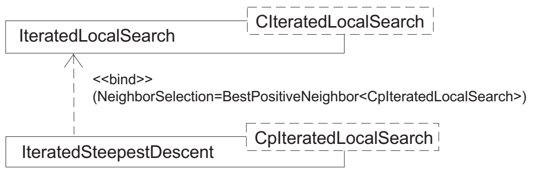


Figure 4.30 Derivation of IteratedSteepestDescent

the considered move (moveEval), and the objective function value of the respective neighbor solution (newSolutionEval).

Figure 4.33 shows the interface CoolingSchedule and some respective components, which implement popular schemes for decreasing the temperature parameter. When specific events occur, respective operations are called; see Algorithms 5 and 10. If an event is not relevant for some cooling schedule, the corresponding operation does not alter the temperature (i.e., “empty” implementation). Specific numeric parameters of the cooling schedule components are implemented as data elements with class-scope. The configuration components are implicitly required to provide corresponding definitions.

Reheating components, which have to conform to the Reheating interface shown in Figure 4.34, re-initialize the temperature parameter when called by a simulated annealing algorithm. The new initial temperature value may depend on the preceding initial value (tinitial), the temperature when the best solution was found (tbest), and the last temperature (tau). We also define an “empty” component NoReheating, which does not affect tinitial (reheating is an optional feature).

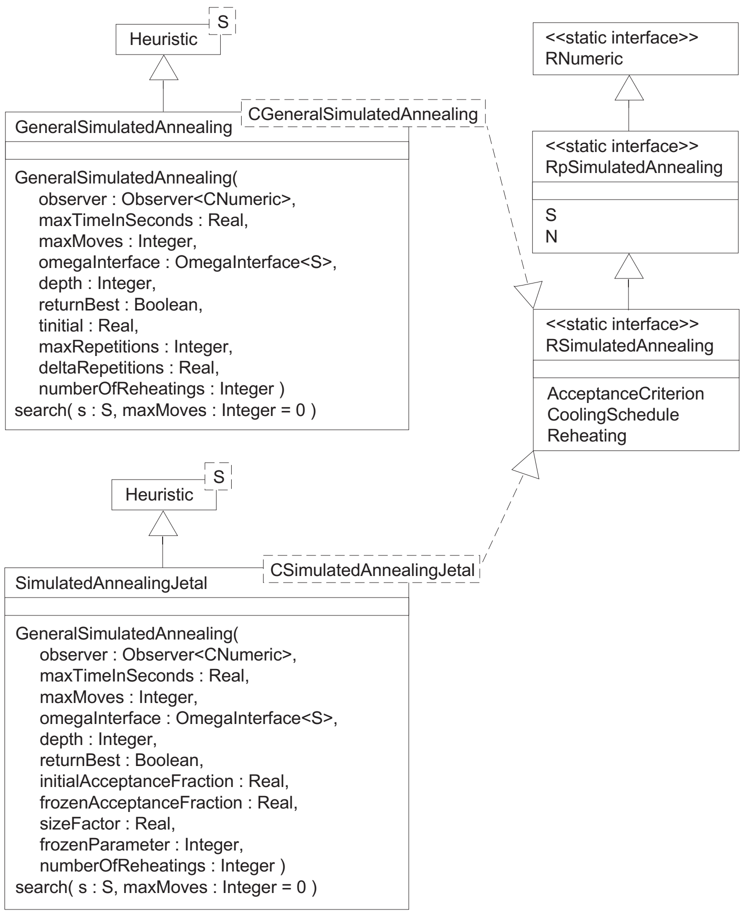

Heuristic
S
GeneralSimulatedAnnealing
CGeneralSimulatedAnnealing
GeneralSimulatedAnnealing(
observer : Observer<CNumeric>,
maxTimeInSeconds : Real,
maxMoves : Integer,
omegaInterface : OmegaInterface<S>,
depth : Integer,
returnBest : Boolean,
tinitial : Real,
maxRepetitions : Integer,
deltaRepetitions : Real,
numberOfReheatings : Integer)
search(s : S, maxMoves : Integer = 0)
<<static interface>>
RNumeric
<<static interface>>
RpSimulatedAnnealing
S
N
<<static interface>>
RSimulatedAnnealing
AcceptanceCriterion
CoolingSchedule
Reheating
Heuristic
S
SimulatedAnnealingJetal
CSimulatedAnnealingJetal
GeneralSimulatedAnnealing(
observer : Observer<CNumeric>,
maxTimeInSeconds : Real,
maxMoves : Integer,
omegaInterface : OmegaInterface<S>,
depth : Integer,
returnBest : Boolean,
initialAcceptanceFraction : Real,
frozenAcceptanceFraction : Real,
sizeFactor : Real,
frozenParameter : Integer,
numberOfReheatings : Integer)
search(s : S, maxMoves : Integer = 0)

Figure 4.31 Components GeneralSimulatedAnnealing and SimulatedAnnealingJetal

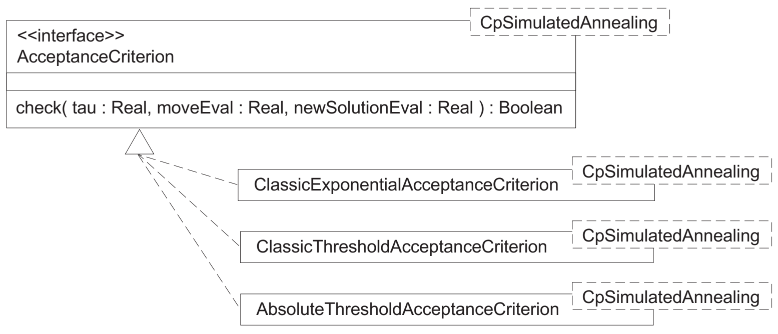

<<interface>>
AcceptanceCriterion
CpSimulatedAnnealing
check(tau : Real, moveEval : Real, newSolutionEval : Real) : Boolean
ClassicExponentialAcceptanceCriterion
CpSimulatedAnnealing
ClassicThresholdAcceptanceCriterion
CpSimulatedAnnealing
AbsoluteThresholdAcceptanceCriterion
CpSimulatedAnnealing

Figure 4.32 Interface AcceptanceCriterion and Respective Components

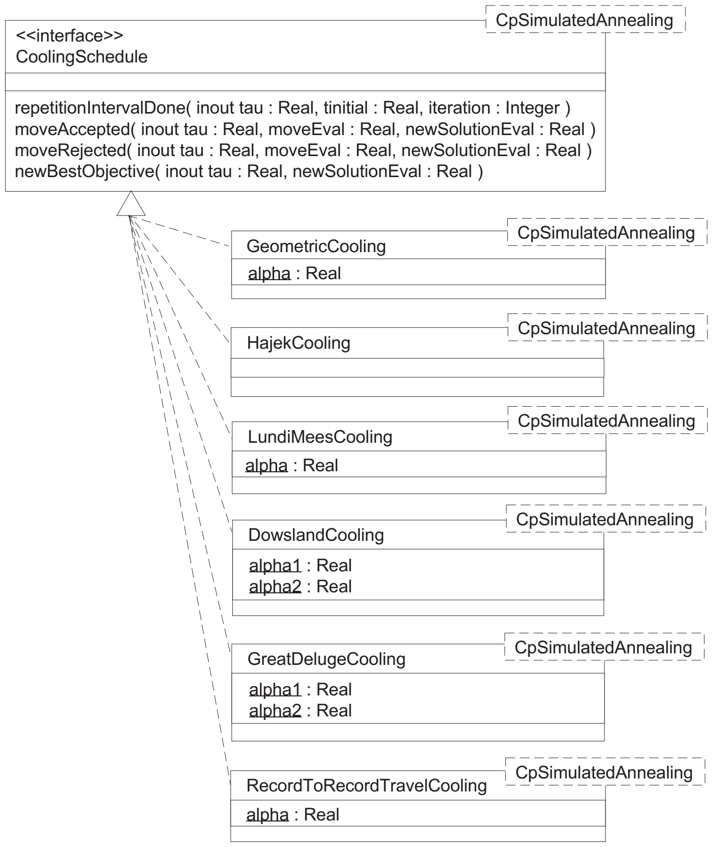

CpSimulatedAnnealing
<<interface>>
CoolingSchedule
repetitionIntervalDone(inout tau : Real, tinitial : Real, iteration : Integer)
moveAccepted(inout tau : Real, moveEval : Real, newSolutionEval : Real)
moveRejected(inout tau : Real, moveEval : Real, newSolutionEval : Real)
newBestObjective(inout tau : Real, newSolutionEval : Real)
GeometricCooling
alpha : Real
HajekCooling
LundiMeesCooling
alpha : Real
DowslandCooling
alpha1 : Real
alpha2 : Real
GreatDelugeCooling
alpha1 : Real
alpha2 : Real
RecordToRecordTravelCooling
alpha : Real

Figure 4.33 Interface CoolingSchedule and Respective Components

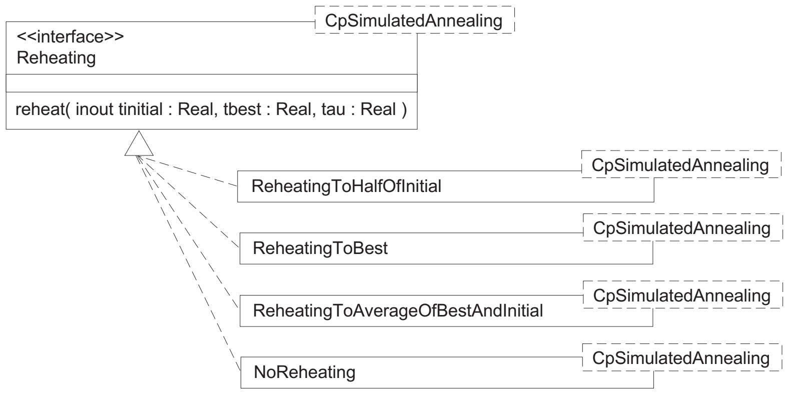


Figure 4.34 Interface Reheating and Respective Components

Tabu Search. The component TabuSearch, shown in Figure 4.35, implements the classic tabu search scheme according to Algorithm 11. A tabu search configuration component in particular defines the neighbor selection rule and an aspiration criterion. The implementation of the features TabuCriterion and Diversification as dynamic object parameters has already been discussed in Section 4.4.2.1.

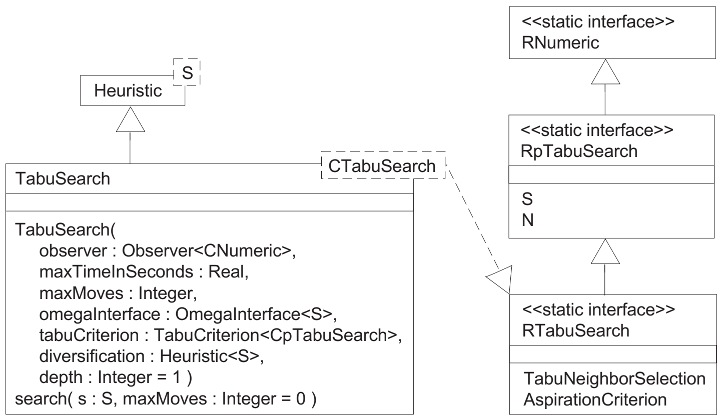


Figure 4.35 Component TabuSearch

The components for different tabu criteria (Algorithms 14–17) are derived from a common base class; see Figure 4.36. The base class defines the general interface of a tabu criterion component according to the discussion in Section 4.3.2.3. Specific requirements of the tabu criteria with regard to the need to define problem-specific components S_I as well as S_A within the configuration component CpTabuSearch conform to the feature diagram shown in Figure 4.4.

The two operations that feed the tabu criterion with applied moves and traversed solutions (addToHistory) provide a return parameter, which is used to indicate whether there is an apparent need for an explicit diversification. By returning a value k greater than zero, the tabu criterion “suggests” to apply some big “escape move” to some solution that is about k moves away from the current solution. Such an explicit diversification is usually accompanied by a call to the operation escape, which re-initializes the tabu memory in an appropriate way. The operation print writes information about the tabu memory in some format to an output stream. The dynamic configuration of the tabu criteria (i.e., the initialization of respective numeric parameter elements such as, e.g., the tabu list length) results from calling constructors, which are not shown in the diagram.

For the component REMTabuCriterion we make two restrictions with regard to an efficient implementation of Algorithm 15. Firstly, we restrict the application of this tabu criterion to single-attribute moves. Secondly, we require the definition of a free template function moveNumber, which computes for each move a unique integer number representative. We identify such free (global) functions in the diagram by using a stereotype free function; see Figure 4.36.

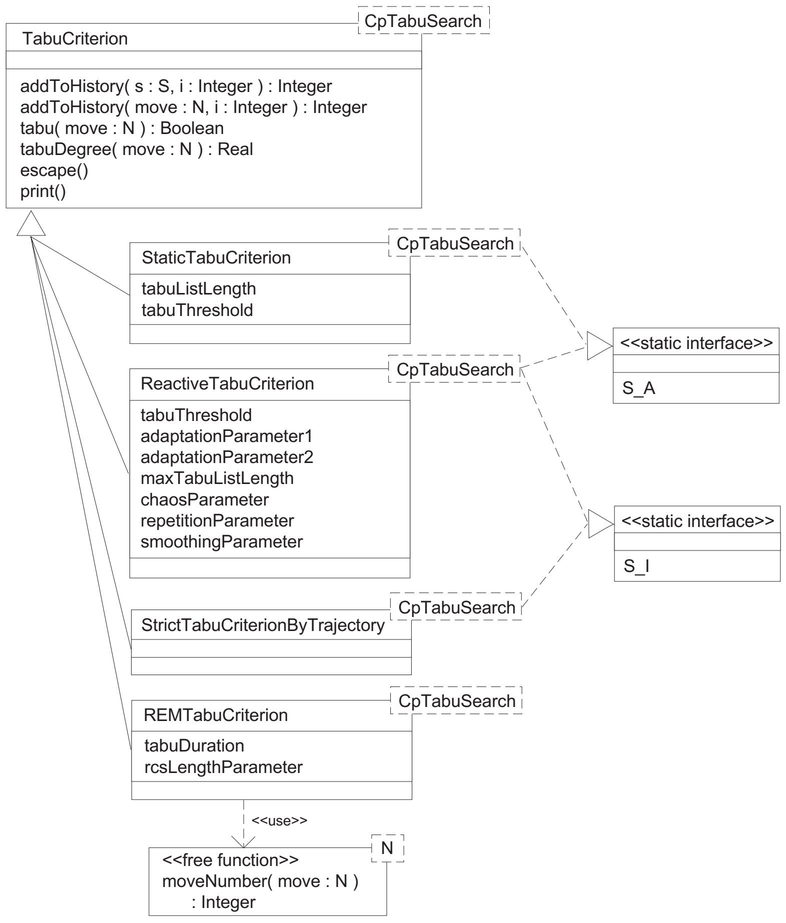


Figure 4.36 Base Class TabuCriterion and Derived Classes

The component TabuSearch requires the definition of a neighbor selection rule (i.e., a corresponding component), which itself depends on the aspiration criterion. Figure 4.37 shows the interface TabuNeighborSelection and respective components. The components BestAdmissibleNeighbor and BestNeighborConsideringPenalties use the aspiration criterion, which is configured in CTabuSearch, in accordance with Algorithms 12 and 13, respectively. The interface of the (optional) aspiration criterion is shown in Figure 4.38. The component NewBestSolutionAspirationCriterion implements the most popular aspiration criterion: A tabu-status is neglected if the move would lead to a new best solution. An efficient implementation of this criterion requires actual information about the implied change of the objective function value that is due to the considered moves (deltaInformation); see p. 119.

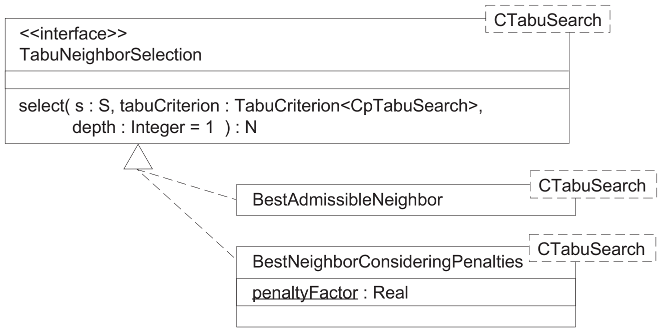

CTabuSearch
<<interface>>
TabuNeighborSelection
select(s : S, tabuCriterion : TabuCriterion<CpTabuSearch>, depth : Integer = 1) : N
CTabuSearch
BestAdmissibleNeighbor
CTabuSearch
BestNeighborConsideringPenalties
penaltyFactor : Real

Figure 4.37 Interface TabuNeighborSelection and Respective Components

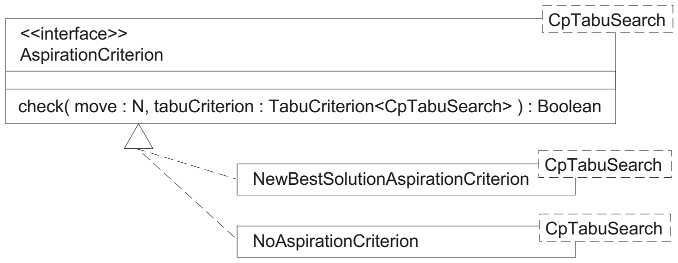

CpTabuSearch
<<interface>>
AspirationCriterion
check(move : N, tabuCriterion : TabuCriterion<CpTabuSearch>) : Boolean
CpTabuSearch
NewBestSolutionAspirationCriterion
CpTabuSearch
NoAspirationCriterion

Figure 4.38 Interface AspirationCriterion and Respective Components

4.4.2.4 Introspection. In Section 4.4.2.1, we argued in favor of flexible introspection means regarding information about the search process. We apply the following design: The interface of a concrete base class Observer represents a comprehensive set of introspection elements; see Figure 4.39. The diagram shows only a subset of the interface. For example, it does not show operations that are specific to a metaheuristic (e.g., an operation that provides access to the contents of the tabu list).

Metaheuristic components and solution components are dynamically parameterized by an object of the class Observer or a derived class. Appropriate object operations are called at respective steps of the algorithms. The member functions of the class Observer are implemented as “empty” functions. So the class Observer defines the maximal introspection interface, yet actually implements a minimal introspection (nothing at all). Special introspection needs are implemented by respective member functions in derived classes, which overwrite the behavior of the base class. Figure 4.39 shows three such components, which all rely on sending data to a stream. The component StreamingObserver simply prints all received data, while the other two components only output the best objective function value and possibly the computation time.

To enable a flexible change of the degree of introspection during the search process, we provide operations to activate or deactivate an observer object. A deactivated object neglects all information sent to it. Moreover, a caller may check for the ac-

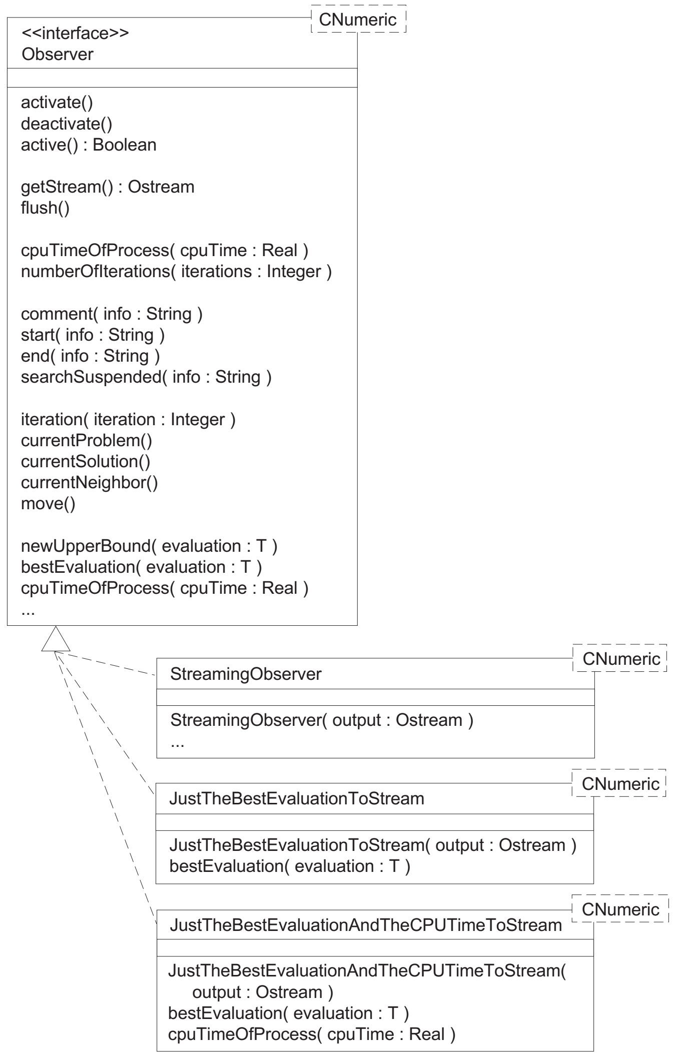

CNumeric
<<interface>>
Observer
activate()
deactivate()
active() : Boolean
getStream() : Ostream
flush()
cpuTimeOfProcess(cpuTime : Real)
numberOfIterations(iterations : Integer)
comment(info : String)
start(info : String)
end(info : String)
searchSuspended(info : String)
iteration(iteration : Integer)
currentProblem()
currentSolution()
currentNeighbor()
move()
newUpperBound(evaluation : T)
bestEvaluation(evaluation : T)
cpuTimeOfProcess(cpuTime : Real)
...
CNumeric
StreamingObserver
StreamingObserver(output : Ostream)
...
CNumeric
JustTheBestEvaluationToStream
JustTheBestEvaluationToStream(output : Ostream)
bestEvaluation(evaluation : T)
CNumeric
JustTheBestEvaluationAndTheCPUTimeToStream
JustTheBestEvaluationAndTheCPUTimeToStream(output : Ostream)
bestEvaluation(evaluation : T)
cpuTimeOfProcess(cpuTime : Real)

Figure 4.39 Derivation of Special Introspection Classes from the Base Class Observer

tivation status of the assigned observer object, which allows to suppress costly data preparations (which would be neglected anyway).

To simplify the transfer of non-atomic information to the observer (e.g., the solution representation in some format), the operation getStream provides the caller with a stream, where respective data may be sent to. With regard to a semantic interpretation of stream data, the caller has to call an appropriate operation before using the stream (e.g., currentSolution). After completing the transfer, the transaction must be finalized by calling the operation flush.

This design, which decouples the search process from the introspection, provides a flexible means for extension. For example, search processes may be coupled to an environment for experimental tests with regard to visualization or statistical analysis of the search process and respective results; see Jones (1994), Jones (1996). Moreover, one may integrate the search process in a decision support system; see Ball and Datta (1997).

4.5 IMPLEMENTATION

As emphasized by Nygaard, *Programming is understanding*. In this sense, implementing metaheuristics as reusable software components – and also eventually using these components – serves for the comprehension of the respective genericity of metaheuristic algorithms. In the following, we briefly describe some essential aspects of the implementation.

4.5.1 Technical Environment and Conventions

In Section 4.4, we argued that C++ is an appropriate programming language in the given context. C++ provides powerful language constructs (in particular with regard to enabling adaptation by type parameterization and inheritance), it facilitates run-time efficient implementations, and it is in wide use in practice. The main argument against C++ is its complexity; see, e.g., the discussion in Gillam (1998), p. 41. However, while this complexity indeed affects the actual developer of a framework, it is mainly hidden from the plain user of a framework.

The implementation is based on Standard C++ (ISO/IEC 14882), which should lead to wide portability. Due to deficiencies of some of the available compilers we decided not to employ member templates, covariant return types, and the exception mechanism. As primary development platform we used *Microsoft Visual C++ 6.0 (Service Pack 3 and higher)*. The code was also successfully tested with the compilers *gcc 3.0* and *MIPSpro C++ 7.3*. A few incompatibilities of these compilers are handled in the source code by relying on preprocessor variables VCC, GCC, and MIPS.

The discussion of the design in the preceding section leads to a broad use of templates. With regard to generic components, the C++ template construct provides a type-safe and run-time efficient means to implement respective adaptation requirements. (On the other hand, the unfamiliarity of most programmers with the template construct and insufficient compiler support with regard to debugging template-intense code may make debugging difficult).

To prevent name clashes all HOTFRAME components are defined in a namespace HotFrame. To apply respective components one may use explicit qualification (e.g., "HotFrame::TabuSearch<...>"), using declarations (e.g., "using HotFrame::TabuSearch<...>"), or using directives (e.g., "using HotFrame").

To streamline the implementation we follow by convention a few coding rules. Class or type identifiers start with a capital letter, while identifiers for global methods, member functions, and local variables start with a lower case letter. Member data identifiers start with an underscore ("_"); static class members start with "_s_". Composite identifiers are partitioned by using capital letters (e.g., computeEvaluation).

Member data access functions are usually named in accordance with the identifier of the member data (e.g., n() provides access to _n). For all problem-specific classes, one has to define appropriate copy constructors, assignment operators, and destructors, if the compiler-generated default versions are not adequate. Member functions that provide a dynamically allocated object for which the caller is responsible (in particular with regard to object destruction) start with create, copy, or clone. Variables that represent iteration numbers are declared as unsigned long; i.e., the maximal iteration number is typically $2^{32} \approx 4.3 \times 10^9$. With x representing an appropriate acronym that identifies the problem type, problem-specific code is usually put into two files x_p.h (problem component) and x_s.h (other components such as the solution space component etc.).

4.5.2 Problem-Specific Standard Components

Applying HOTFRAME involves appropriate problem-specific components that particularly implement the solution space and the neighborhood structure. In Section 4.4.2.3, we described respective requirements in dependence on the metaheuristics that one wants to use. In case the pre-defined problem-specific standard components do not fit for some new type of problem, one may need to implement such components from scratch. HOTFRAME includes the following problem-specific standard components (see pp. 122):

```
template <class C> class Generic_S;
template <class C> class Generic_S_N;

template <class C> class BV_S : public Generic_S<C>;
template <class C> class BV_S_N : public Generic_S_N<C>;
template <class C> class BV_S_A;
template <class C> class BV_S_I;

template <class C> class Perm_S : public Generic_S<C>;
template <class C> class Perm_S_N : public Generic_S_N<C>;
template <class C> class Perm_S_N_Shift : public Perm_S_N<C>;
template <class C> class Perm_S_N_Swap : public Perm_S_N<C>;
template <class C> class Perm_S_N_2Exchange
                         : public Perm_S_N<C>;
template <class C> class Perm_S_A;
template <class C> class Perm_S_I;
```

For the sake of completeness, we also name the components for combined assignment and sequencing problems (without description):

```
template <class C> class AS_S;
template <class C> class AS_S_N_Shift;
template <class C> class AS_S_A1;
template <class C> class AS_S_A2;
template <class C> class AS_S_I;
```

The implementation of BV_S and Perm_S illustrates the alternative strategies to implement the move interpretation as part of the member functions doMove and computeEvaluation (see the discussion on p. 116). On the one hand, BV_S directly interprets and executes the modifications to a solution that are due to a considered move. In contrast, Perm_S_N and derived classes define a member function transform, which realizes respective modifications of the solution data structures and thus can be used by Perm_S. It is important to note that respective default implementations of doMove and computeEvaluation are only valid if the derived solution class for the specific application does not define special data elements that are affected by a move. In such cases, one has to provide specialized implementations of these member functions.

The enumeration NeighborhoodPosition defines the general possibilities to construct neighbor/move objects:

```
enum NeighborhoodPosition { FirstNeighbor,SomeRandomNeighbor,
                            InvalidNeighbor };
```

For all problem-specific classes X, the following stream output operator is defined, which enables a polymorphic call of the respective print member function:

```
template <class C>
  ostream& operator<<( ostream& output, const X<C>& x );
```

4.5.3 Metaheuristic Components

We briefly describe the actual C++ interfaces of the metaheuristic components that have been described in Section 4.4.2.3. The interface of the common base class of metaheuristic components is defined as follows (see Figure 4.26):

```
template <class S>
class Heuristic
{
  public:
    virtual ~Heuristic( )
      { ; }
    virtual void search( S& s, unsigned long maxMoves = 0 )
      { ; }
};
```

4.5.3.1 Iterated Local Search. The interface of the component IteratedLocalSearch is defined as follows (see Figure 4.28):

```
template <class C>
class IteratedLocalSearch
: public Heuristic<typename C::S>
{
  public:
    typedef typename C::T T;
    typedef typename C::Range Range;
    typedef typename C::CNumeric CNumeric;
    typedef typename C::S S;
    typedef typename C::N N;
    typedef typename C::NeighborSelection NeighborSelection;

  protected:
    Observer<CNumeric> *_observer;
    float _maxTimeInSeconds;
    unsigned long _maxMoves;
    unsigned long _repetitions;
    OmegaInterface<S> *_omegaInterface;
    Heuristic<S> *_diversification;
    short _depth;
    bool _returnBest;

  public:
    IteratedLocalSearch( Observer<CNumeric> *observer = 0,
                         float maxTimeInSeconds = 0,
                         unsigned long maxMoves = 0,
                         unsigned long repetitions = 1,
                         OmegaInterface<C> *omegaInterface=0,
                         Heuristic<S> *diversification = 0,
                         short depth = 1,
                         bool returnBest = true );
    virtual void search( S& s, unsigned long maxMoves = 0 );
};
```

The type definitions conform to the requirements on the configuration components, which have been specified in Figure 4.28. The explicit re-definition of these types in the first part of the interface makes requirements explicit and simplifies the use of respective types. The data elements correspond to the method parameters as defined by the constructor interface. (To simplify the presentation, we omit type definitions and data structures from the following descriptions, because these elements can be directly deduced from the requirements formulated in Section 4.4.2.3.)

The constructor interface specifies for all parameters default values. For maxTimeInSeconds and maxMoves, the default value of 0 represents the lack of a corresponding restriction. The constructor definition consists only of the initialization of respective class data elements. The actual iterated local search algorithm is implemented in the member function search.

Components that implement a neighbor selection rule must conform to the following interface NeighborSelection (see Figure 4.29):

```
template <class C>
class NeighborSelection
{
  public:
    static N select( S& s, short depth = 1 );
}
```

The following realizations of this interface are pre-defined:

```
template <class C> class BestPositiveNeighbor;
template <class C> class BestNeighbor;
template <class C> class FirstPositiveNeighbor;
template <class C> class RandomNeighbor;
```

4.5.3.2 Simulated Annealing and Variations. Algorithms 5 and 10 are implemented by the following two components (see Section 4.4.2.3):

```
template <class C>
class GeneralSimulatedAnnealing
: public Heuristic<typename C::S>
{
  public:
    GeneralSimulatedAnnealing(
        Observer<CNumeric> *observer = 0,
        float maxTimeInSeconds = 0,
        unsigned long maxMoves = 0,
        OmegaInterface<S> *omegaInterface = 0,
        short depth = 1,
        bool returnBest = true,
        double tinitial = 100,
        unsigned long maxRepetitions = 1,
        double deltaRepetitions = 1,
        unsigned int numberOfReheatings = 0 );

    virtual void search( S& s, unsigned long maxMoves = 0 );
};

class SimulatedAnnealingJetal
: public Heuristic<typename C::S>
{
  public:
    SimulatedAnnealingJetal(
        Observer<CNumeric> *observer = 0,
        float maxTimeInSeconds = 0,
        unsigned long maxMoves = 0,
        OmegaInterface<S> *omegaInterface = 0,
        short depth = 1,
        bool returnBest = true,
        float initialAcceptanceFraction = 0.4,
        float frozenAcceptanceFraction = 0.02,
        float sizeFactor = 16,
        unsigned int frozenParameter = 5,
        unsigned int numberOfReheatings = 0 );

    virtual void search( S& s, unsigned long maxMoves = 0 );
};
```

The determination of the initial temperature for the latter component is implemented in the following way: Beginning with the initial solution, a trial run of *size-Factor* $\cdot |N(s)|$ iterations is performed, where in each iteration a neighbor solution is randomly generated and accepted if and only if the move evaluation is strictly positive. The observed evaluations of the other moves are stored and sorted. Eventually, the temperature is set so that $initialAcceptanceFraction$ of the observed neighbors would have been accepted.

The interface for components that implement an acceptance criterion is defined as follows (see Figure 4.32):

```
template <class C>
class AcceptanceCriterion
{
  public:
    static bool check( double tau, double moveEval,
                       double newSolutionEval = 0 );
};
```

This interface is realized by three pre-defined generic classes:

```
template<class C> class ClassicExponentialAcceptanceCriterion;
template<class C> class ClassicThresholdAcceptanceCriterion;
template<class C> class AbsoluteThresholdAcceptanceCriterion;
```

In accordance with Figure 4.33, the interface for components that implement a cooling schedule is defined as follows:

```
template <class C>
class CoolingSchedule
{
 public:
   static void repetitionIntervalDone
    ( double& tau, double tinitial, unsigned long iteration );
   static void moveAccepted
    ( double& tau, double moveEval, double newSolutionEval );
   static void moveRejected
    ( double& tau, double moveEval, double newSolutionEval );
   static void newBestObjective
    ( double& tau, double newSolutionEval );
};
```

The following realizations define non-relevant functions by an “empty” implementation:

```
template <class C> class GeometricCooling;
  // double C::alpha;
template <class C> class HajekCooling;
template <class C> class LundiMeesCooling;
  // double C::alpha;
template <class C> class DowslandCooling;
  // double C::alpha1;
  // double C::alpha2;
template <class C> class GreatDelugeCooling;
```

```
    // double C::alpha1;
    // double C::alpha2;
  template <class C> class RecordToRecordTravelCooling;
    // double C::alpha;
```

The commentaries with regard to data elements of the configuration component C indicate the respective parameterization (see Figure 4.33).

In accordance with Figure 4.34, the interface of reheating components is defined as follows:

```
  template <class C>
  class Reheating
  {
    public:
      static void reheat( double& tinitial, double tbest = 0,
                          double tau = 0 );
  };
```

This interface is realized by four pre-defined generic classes:

```
  template <class C> class ReheatingToHalfOfInitial;
  template <class C> class ReheatingToBest;
  template <class C> class ReheatingToAverageOfBestAndInitial;
  template <class C> class NoReheating;
```

4.5.3.3 Tabu Search. The interface of the fundamental tabu criteria is defined as follows (see Figure 4.35):

```
  template <class C>
  class TabuSearch
  : public Heuristic<typename C::S>
  {
    public:
      TabuSearch( Observer<CNumeric> *observer = 0,
                  float maxTimeInSeconds = 0,
                  unsigned long maxMoves = 0,
                  OmegaInterface<S> *omegaInterface = 0,
                  TabuCriterion<Cp> *tabuCriterion = 0,
                  Heuristic<S> *diversification = 0,
                  short depth = 1 );

      virtual void search( S& s, unsigned long maxMoves = 0 );
  };
```

In accordance with Figure 4.36, the base class for the tabu criteria is defined as follows:

```
  template <class C>
  class TabuCriterion
  {
    public:
      virtual ~TabuCriterion( )
        { ; }
      virtual unsigned long addToHistory
          ( const S& s, unsigned long iteration = 0 )
        { return 0; }
      virtual unsigned long addToHistory
          ( const N& move, unsigned long iteration = 0 )
        { return 0; }
      virtual bool tabu( const N& move ) const
        { return false; }
      virtual double tabuDegree( const N& move ) const
        { return tabu( move ); }
      virtual void escape( )
        { ; }
      virtual void print( ) const
        { ; }
  };
```

Different classes are derived from TabuCriterion to implement the tabu criteria according to Algorithms 14–17. Specific requirements of these classes on the configuration component C result from Figure 4.36 (with the addition of an observer object parameter). We restrict the following descriptions of tabu criteria components to the constructor:

```
  template <class C>
  class StrictTabuCriterionByTrajectory
  : public TabuCriterion<typename C::Cp>
  {
    public:
      StrictTabuCriterionByTrajectory
          ( Observer<CNumeric> *observer = 0 );
  }

  template <class C>
  class REMTabuCriterion
  : public TabuCriterion<typename C::Cp>
  {
    public:
      REMTabuCriterion(
        Observer<CNumeric> *observer = 0,
        unsigned long tabuDuration = 1,
        unsigned long rcsLengthParameter = 1,
        unsigned long maximumMoveNumber
                      = numeric_limits<unsigned long>::max());
  }
```

The fourth parameter of the constructor of the REM tabu criterion results from a technical requirement of the implementation, which needs to know about a maximum move number that may occur during the search process.

```
template <class C>
class StaticTabuCriterion
: public TabuCriterion<typename C::Cp>
{
  public:
    StaticTabuCriterion(
        Observer<CNumeric> *observer = 0,
        Range tabuListLength = 7,
        Range tabuThreshold = 1 );
}

template <class C>
class ReactiveTabuCriterion
: public TabuCriterion<typename C::Cp>
{
  public:
    ReactiveTabuCriterion(
        Observer<CNumeric> *observer = 0,
        Range tabuThreshold = 1,
        float adaptationParameter1 = 1.2,
        short adaptationParameter2 = 2,
        Range maxTabuListLength = 50,
        unsigned int chaosParameter = 3,
        unsigned int repetitionParameter = 3,
        float smoothingParameter = 0.5 );
}
```

Components that implement a neighbor selection rule must conform to the following interface (see Figure 4.37):

```
template <class C>
class TabuNeighborSelection
{
  typedef typename C::AspirationCriterion AspirationCriterion;
  public:
    static N select( S& s,
                     TabuCriterion<Cp>& tabuCriterion,
                     short depth = 1 );
}
```

The following two components realize this interface in accordance with Algorithms 12 and 13:

```
template <class C>
    class BestAdmissibleNeighbor;
template <class C>
    class BestNeighborConsideringPenalties;
      // double C::penaltyFactor;
```

In Figure 4.38, the interface of aspiration criteria was specified as follows:

```
template <class C>
class AspirationCriterion
{
  public:
   static bool check( N& move,
                      TabuCriterion<Cp>& tabuCriterion );
};
```

This interface is realized by two pre-defined components:

```
template <class Cp>
    class NewBestSolutionAspirationCriterion;
template <class Cp>
    class NoAspirationCriterion;
```

The latter component represents the waiving of an aspiration criterion (i.e., check always returns false).

4.5.4 *Miscellaneous Components*

In Section 4.4.2.4, we described the use of observer objects to flexibly implement different kinds of introspection needs. The following components correspond to Figure 4.39:

```
template <class C>
  class Observer;
template <class C>
  class StreamingObserver : public Observer<C>;
template <class C>
  class JustTheBestEvaluationToStream : public Observer<C>;
template <class C>
  class JustTheBestEvaluationAndTheCPUTimeToStream
        : public Observer<C>;
```

As an addition to the core functionality, HOTFRAME includes some classes that provide useful functionality such as the computation of hash-codes or the representation of matrices and graphs.

4.6 APPLICATION

In this section, we provide an overview with respect to the actual application of framework components. After illustrating the requirements and procedures of the application of metaheuristics in Section 4.6.1, we discuss an incremental application process in Section 4.6.2. We mostly restrict to exemplary descriptions, which should enable to transfer a respective understanding to other application scenarios.

4.6.1 *Requirements and Procedures*

First of all, to apply a local search procedure one must be able to formulate the problem accordingly (solution representation, scalar objective function, neighborhood structure, etc.). That is, the problem should fit with regard to the problem-specific abstractions introduced in Section 4.3.1, which correspond to components of the framework architecture.

In conformance to the "no-free-lunch theorem" mentioned on p. 82, HOTFRAME enables the implementation of problem-specific components from scratch to fully adapt metaheuristics for the considered problem. In general, if the problem-type necessitates the use of a special solution space and neighborhood structure, the adaptation of metaheuristics may require non-trivial coding. On the contrary, if the problem fits to some typical solution space and neighborhood structure that are available as pre-defined problem-specific components, the application of HOTFRAME may reduce to a few lines of code.

To apply some metaheuristic-component one needs suitable problem-specific components according to the requirements defined in Section 4.4.2.3. Table 4.1 provides a summary of these requirements for the metaheuristics that have been described in this paper. A "+" indicates the need for the respective component; for the REMTabuCriterion one additionally needs a free function moveNumber (see p. 133). Interface requirements for S, N, S_I, and S_A have been described in Section 4.4.2.2.

	S	N	S_I	S_A
IteratedLocalSearch	+	+		
GeneralSimulatedAnnealing	+	+		
SimulatedAnnealingJetal	+	+		
Tabu Search	+	+		
– StrictTabuCriterionByTrajectory			+	
– REMTabuCriterion				
– StaticTabuCriterion				+
– ReactiveTabuCriterion			+	+

Table 4.1 Summary of the Main Problem-Specific Requirements of Metaheuristics

4.6.1.1 Iterated Local Search. To specialize the component IteratedLocalSearch regarding some common uses, there are pre-defined configuration components. In particular, the following configuration components define the neighbor selection rule to be applied:

```
template <class C> struct CSteepestDescent;
template <class C> struct CFirstDescent;
template <class C> struct CRandomWalk;
```

These simple components conform to the requirements RIteratedLocalSearch defined in Figure 4.28. In each case, a problem-specific configuration is extended by the definition of the feature NeighborSelection. For example, CSteepestDescent is implemented as follows:

```
template <class C>
struct CSteepestDescent
```

```
{
  typedef BestPositiveNeighbor< C > NeighborSelection;

  typedef typename C::T T;
  typedef typename C::Range Range;
  typedef typename C::CNumeric CNumeric;
  typedef typename C::S S;
  typedef typename C::N N;
  typedef C Cp;
};
```

Given some problem-specific configuration component Cp, which defines T, Range, CNumeric, S, and N, one can generate a steepest descent heuristic by:

```
IteratedLocalSearch< CSteepestDescent< Cp > >
```

The C++ template construct consequently enables a direct implementation of the abstract design of Figure 4.28. Thus, a typical application of some metaheuristic component is structured as a three-level hierarchical configuration: numeric base types, problem-specific abstractions solution space and neighborhood structure, and neighbor selection rule.

The actual application of a steepest descent heuristic for an initial solution s means that one has to construct a respective object and to call the member function search:

```
Heuristic< Cp::S >
  *steepestDescent
  = new IteratedLocalSearch< CSteepestDescent < Cp > >;
steepestDescent->search( s );
```

As another example, using the dynamic configuration of IteratedLocalSearch as shown in Figure 4.28, Algorithm 4 (IteratedSteepestDescentWithPerturbationRestarts) can be applied as follows:

```
Heuristic< Cp::S >
  *diversification
  = new IteratedLocalSearch< CRandomWalk< Cp > >
        ( 0, 0, 10, 1, 0, 0, 1, false );
Heuristic< Cp::S >
  *iteratedSteepestDescentWithPerturbationRestarts
  = new IteratedLocalSearch< CSteepestDescent < Cp > >
        ( 0, 0, 0, 5, 0, diversification );
iteratedSteepestDescentWithPerturbationRestarts->search( s );
```

In this example, one first constructs an object that represents the diversification (ten random moves, return of the last traversed solution). This object is applied as a parameter to the actual iterated local search procedure.

4.6.1.2 Simulated Annealing and Variations.

The generation of the Algorithms 6–9, which have been defined in Section 4.3.2.2, is based on the component

GeneralSimulatedAnnealing. This component is adapted by using one of the following configuration components:

```
template <class C> struct CClassicSimulatedAnnealing;
template <class C> struct CThresholdAccepting;
template <class C> struct CGreatDeluge;
template <class C> struct CRecordToRecordTravel;
```

These configuration components fix the features cooling schedule, acceptance criterion, and reheating scheme.

The application of a typical simulated annealing procedure for 10,000 moves, using an initial temperature of 100, looks as follows:

```
Heuristic< Cp::S >
  *classicSimulatedAnnealing
  = new GeneralSimulatedAnnealing
          < CClassicSimulatedAnnealing < Cp > >
          ( 0, 0, 10000, 0, 1, true, 100 );
classicSimulatedAnnealing->search( s );
```

The main advantage of the simulated annealing algorithm according to Johnson et al. (1989) is the robustness with respect to the parameter setting. In particular, the user does not need to experiment with the initial temperature. The configuration component CSimulatedAnnealingJetal defines the classic ingredients of simulated annealing as used by Johnson et al. (1989) (exponential acceptance criterion, geometric cooling schedule, no reheating). Such an algorithm, with the default parameter setting, is applied in the following example:

```
Heuristic< Cp::S >
  *simulatedAnnealingJetal
  = new SimulatedAnnealingJetal
          < CSimulatedAnnealingJetal < Cp > >;
  simulatedAnnealingJetal->search( s );
```

4.6.1.3 Tabu Search. The peculiarity of the application of tabu search is that the main variable feature, the tabu criterion, is configured dynamically by an object parameter; see Figure 4.35. This is also the case for the optional explicit diversification procedure, while the neighbor selection rule and the aspiration criterion are defined by configuration components. The pre-defined configuration component CTabuSearch-ByTabuization specifies the mostly used tabu search variant: The tabu criterion is used to dynamically prohibit certain moves, while the aspiration criterion overwrites a tabu status if the move would lead to a new best solution:

```
template <class C>
struct CTabuSearchByTabuization
{
  typedef NewBestSolutionAspirationCriterion< C >
          AspirationCriterion;
  typedef BestAdmissibleNeighbor
```

```
            < CTabuSearchByTabuization< C > >
          TabuNeighborSelection;

    typedef typename C::T T;
    typedef typename C::Range Range;
    typedef typename C::CNumeric CNumeric;
    typedef typename C::S S;
    typedef typename C::N N;
    typedef typename C::S_A S_A;
    typedef typename C::S_I S_I;
    typedef C Cp;
  };
```

In the same way, the configuration component CTabuSearchByPenalties defines the neighbor selection according to Algorithm 13 (without applying an aspiration criterion):

```
  template <class C> struct CTabuSearchByPenalties
  {
    ...

    typedef NoAspirationCriterion< C > AspirationCriterion;
    typedef BestNeighborConsideringPenalties
            < CTabuSearchByPenalties< C > >
          TabuNeighborSelection;

    static double penaltyFactor;
  }
```

The tabu criterion, which is applied as an object parameter to the general tabu search component, is itself statically parameterized with regard to problem-specific aspects. Figure 4.36 and Table 4.1 summarize respective requirements for different tabu criteria; the dynamic parameterization of a tabu criterion object is described in Section 4.5.3.3. The following code example shows the construction of a tabu criterion object and its use as part of a typical application of tabu search:

```
  TabuCriterion< Cp >
    *staticTabuCriterion
    = new StaticTabuCriterion< Cp >( 0, 7, 1 );

  Heuristic< Cp::S >
    *classicTabuSearch
    = new TabuSearch< CTabuSearchByTabuization< Cp > >
                  ( 0, 0, 1000, 0, staticTabuCriterion );
```

4.6.1.4 Combination of Different Algorithms. An appropriate combination of ideas from different methods often leads to high-quality results. Having available a set of flexible metaheuristics software components greatly simplifies building and applying hybrid search strategies. This is illustrated by the following example:

- Reactive tabu search with move penalties on the basis of tabu degree information
- Neighborhood depth of 2 (“quadratic neighborhood”)
- Applying the pilot method (see Duin and Voß (1999)) to evaluate neighbor solutions by, e.g., performing five steepest descent moves
- Explicit diversification by short simulated annealing runs with a different neighborhood than used for the primary search process

Eventually, after writing a few lines of code to construct a corresponding heuristic object, one may even use this object to hybridize an evolutionary algorithm (not described in this paper). That is, a framework provides the user with a powerful toolbox, which can be exploited to easily construct and apply novel algorithms.

4.6.2 Incremental Application

To fully grasp the rules and mechanisms to apply a framework one may have to manage a steep learning curve. Therefore, a framework should enable an incremental application process (“*adoption path*”); see Fink et al. (1999a). That is, the user may start with a simple scenario, which can be successively extended, if needed, after having learned about more complex application mechanisms. Such an evolutionary problem solving approach corresponds to the general tendency of a successive diffusion of knowledge about a new technology and its application; see Allard (1998) and Rogers (1995).

In the following, we describe a typical adoption path for the case that some of the problem-specific standard components are appropriate for the considered application. In this process, we quickly – after completing the first step – arrive at being able to apply several kinds of metaheuristics for the considered problem, while efficiently obtaining high-quality results may require to follow the path to a higher level.

1. **Objective Function:** After selecting an appropriate solution component, one has to derive a new class and to code the computation of the objective function. Of course, one also needs some problem component, which provides problem instance data. All other problem-specific components may be reused without change.

2. **Efficient Neighborhood Evaluation:** In most cases, the system that results from step 1 bears a significant potential with regard to improving run-time efficiency. In particular, one should implement an adaptive computation of the move evaluation (which replaces the default evaluation by computing objective function values for neighbor solutions from scratch). In this context, one may also implement some move evaluation that differs from the default one (implied change of the objective function value).

3. **Problem-Specific Adaptation:** Obtaining high-quality solutions may require the exploitation of problem-specific knowledge. This may refer to the definition (and implementation) of a new neighborhood structure or an adapted tabu criterion by specific solution information or attribute components.

4. Extension of Metaheuristics: While the preceding steps only involve problem-specific adaptations, one may eventually want to extend some metaheuristic or implement a new heuristic from scratch.

We exemplify this incremental adoption path for the "open traveling salesman problem" (TSPO): Given are n "locations" with "distances" c_{ij}, $1 \leq i, j \leq n$, between respective locations. The goal is to obtain a permutation Π that minimizes the sum of distances

$$z(\Pi) = \sum_{i=1}^{n-1} c_{\pi_i, \pi_{i+1}} .$$

By π_i we denote the location that is at position i in the sequence. This problem is obviously suited to the solution component Perm_S. So, in step 1, one can derive TSPO_S from Perm_S:

```
template <class C>
class TSPO_S : public Perm_S<C>
{
 protected:
   TSPO_P<C>& _problem;

 public:
   enum FirstSolution { Identity=0, Random, GivenSolution };
   TSPO_S( TSPO_P<C>& p,
           Observer<C> *observer = 0,
           FirstSolution firstSolution = Identity,
           vector<Range> startPermutation = vector<Range>());
   virtual void evaluate();

   ...
};
```

The constructor implementation includes the determination of the initial permutation according to alternative strategies. The computation of the objective function is implemented in the member function *evaluate*. After implementing a problem component one can immediately apply different metaheuristics by defining the following configuration component (reusing Perm_S_N_Shift, Perm_S_A, and Perm_S_I without change):

```
template <class C>
struct CpTemplateTSPO
{
  typedef typename C::T T;
  typedef typename C::Range Range;
  typedef C CNumeric;

  typedef TSPO_P<C> P;
  typedef TSPO_S<C> S;
  typedef Perm_S_N_Shift<C> N;
```

```
    typedef Perm_S_A<C> S_A;
    typedef Perm_S_I<C> S_I;
  };

  typedef CpTemplateTSPO<CNumeric> Cp;
```

In step 2, the neighborhood evaluation might be implemented in an efficient way in the following move evaluation operation of TSPO_S :

```
    virtual bool computeEvaluation( const Generic_S_N<C>& move,
                                    T& evaluation, T& delta );
```

For the considered problem, this would mean subtracting the length of the deleted edges from the sum of the length of the edges inserted, which provides the implied change of the objective function value (delta). If we use this measure to actually evaluate the advantageousness of a move, evaluation results as −delta.

In case one wants to experiment with, e.g., some new neighborhood structure (such as some kind of a 3-exchange), a respective neighborhood component might be derived from Perm_S_N and implemented (step 3). Eventually, applying ideas with regard to variable depth neighborhoods or ejection chain approaches requires to specially code respective algorithms, which is accompanied with a fluent transition to step 4.

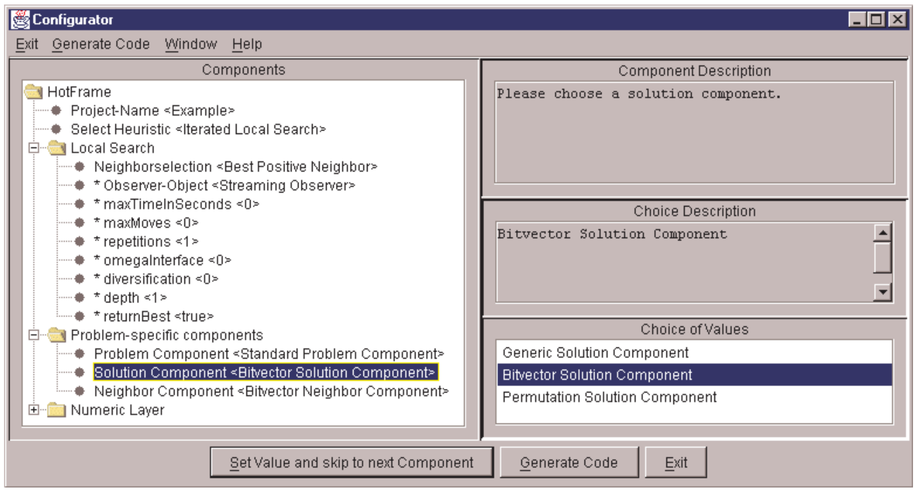
If there are no problem-specific components available that fit for the considered problem type, one has to implement respective components in accordance with the defined requirements for the metaheuristics that one wants to apply (see Table 4.1).

4.7 CONCLUSIONS

The principal effectiveness of HOTFRAME regarding competitive results has been demonstrated for different types of problems; see, e.g., Fink and Voß (1999a), Fink (2000), Fink et al. (2000), Fink and Voß (2001). Moreover, we have used the framework for different practical scenarios in an online setting; see Böse et al. (2000) and Gutenschwager et al. (2001).

HOTFRAME may be extended in various directions. On the one hand, new problem-specific standard components may be added. Ideally, this eventually results in a large set of implemented solution spaces and corresponding components, which enables for many common problem types a straightforward and easy framework application. On the other hand, one may add new metaheuristic components.

However, with regard to the first-time use of HOTFRAME by a new user, even Step 1 of the proposed adoption path requires some easy yet crucial knowledge about the framework. For example, one needs to know which metaheuristic components do exist, how can these metaheuristic components be configured, which problem-specific components are needed, how can problem-specific components be combined with each other, which source code files must be included, and so on. To facilitate the use of the framework we experiment with a software generator with a graphical user interface. A software generator builds customized applications on the basis of high-level specifications (see, e.g., Czarnecki and Eisenecker (2000)). That is, declarative specifications are transformed to specialized code. Our generator, which is based on joint work with



```
HotFrame
  Project-Name <Example>
  Select Heuristic <Iterated Local Search>
  Local Search
    Neighborselection <Best Positive Neighbor>
    * Observer-Object <Streaming Observer>
    * maxTimeInSeconds <0>
    * maxMoves <0>
    * repetitions <1>
    * omegaInterface <0>
    * diversification <0>
    * depth <1>
    * returnBest <true>
  Problem-specific components
    Problem Component <Standard Problem Component>
    Solution Component <Bitvector Solution Component>
    Neighbor Component <Bitvector Neighbor Component>
  Numeric Layer
```

```
Please choose a solution component.
```

```
Bitvector Solution Component
```

```
Generic Solution Component
Bitvector Solution Component
Permutation Solution Component
```

Figure 4.40 Illustration of the User Interface of the Generator

Biskup (2000), is based on a general model which allows to represent framework architectures. Specific frameworks are modeled by using a configuration language. On the basis of the design of HOTFRAME we have defined the framework architecture in this configuration language:

- Metaheuristic components and their static and dynamic parameters
- Problem-specific components and their interdependencies
- Requirements of metaheuristic components regarding problem-specific components
- Associations of components with source code templates and substitution rules regarding the actual configuration

The generator, which is implemented in Java, provides a graphical user interface; see Figure 4.40. The example shows the configuration of an iterated local search component as discussed in Section 4.4. The generator provides an intuitive interface to configure the framework regarding the intended application of some metaheuristic to some problem type. After selection of a metaheuristic, one is provided with customized options to configure dynamic search parameters, problem-specific concepts, and numeric data types. Eventually, the generator produces, in dependence on the specified configuration, customized source code with a few "holes" to be filled by the user. In simple cases, this manual programming is restricted to the coding of the objective function. In general, however, following the argumentation in Section 4.4, one may still have to do considerable parts of the implementation to exploit problem-specific knowledge. Thus, the use of HOTFRAME generally reflects the tradeoff between flexibility and ease-of-use.